

Desarrollo de Sistemas Distribuidos - 4CM5

[Tablero](#) / [Mis cursos](#) / [SISDIS-4CM5](#)

Su progreso



[Avisos](#)



[Lineamientos del curso - Desarrollo de Sistemas Distribuidos](#)



[Calendario académico 20-21](#)



[POLÍTICAS DE USO @alumnoguinda.mx](#)



[Apoyo económico extraordinario para alumnos](#)



Recursos



[Ley de Amdahl y paralelismo](#)



[TOP500 List Statistics](#)



[Time, Clocks, and the Ordering of Events in Distributed Systems, Leslie Lamport, 1978.](#)



[An optimal algorithm for mutual exclusion in computer networks, Ricart y Agrawala, 1981.](#)



[Elections in a Distributed Computing System, Héctor Garcia-Molina, 1982.](#)



[Primer examen parcial](#)



[Segundo examen parcial](#)



El examen es individual y se deberá presentar en forma remota utilizando la plataforma Moodle.

El examen se podrá presentar el miércoles 2 de diciembre a las 12:00 Hrs. con una duración máxima de 90 minutos.

El examen será **50% teórico y 50% práctico**.

Los alumnos y alumnas deberán contar con JDK8 o superior en sus computadoras para desarrollar los programas que serán parte del examen.

Los temas a evaluar son los siguiente:

- Tipos de comunicaciones.
- Fiabilidad de un sistema.
- Clasificación de las fallas de un sistema.
- Sockets stream y sockets datagrama
- Comunicación unicast confiable.
- Comunicación multicast confiable.
- Multicast atómico.
- Paradigma de paso de mensajes.
- Objetos locales.
- Objetos remotos.
- Paradigma de objetos distribuidos.
- Remote Method Invocation.
- Java RMI.
- Cómo usar Java RMI.
- Cómo ejecutar Java RMI en la nube.
- JSON.

- Conceptos básicos de Servicios web.
- Participación en un servicio web.
- Servicios web basados en SOAP.
- Componentes de un mensaje SOAP.
- Web Services Description Language (WSDL).
- Servicios web estilo REST.
- Cómo invocar un método web REST utilizando Java.

1. Introducción

Clase del día - 28/09/2020



Bienvenidos al curso de Desarrollo de Sistemas Distribuidos.

En este curso vamos a estudiar los aspectos teóricos de los sistemas distribuidos y cómo la distribución del cómputo y los datos tienen muchas ventajas sobre los sistemas centralizados.

Vamos a programar en Java

Además de ver teoría, vamos a programar sistemas distribuidos utilizando Java.

Para comunicar los programas vamos a utilizar sockets y programación multi-thread.

Algunos de ustedes habrán cursado ya la asignatura "Aplicaciones para Comunicaciones en Red", en esta materia se explica cómo programar un cliente y un servidor mediante sockets y cómo programar una aplicación multi-thread.

Sin embargo, habrá algunos alumnos y alumnas que no han cursado esta materia debido a que en el mapa curricular 2009 las asignaturas "Desarrollo de Sistemas Distribuidos" y "Aplicaciones para Comunicaciones en Red" dependen de "Redes de Computadoras".

Por esta razón, vamos a iniciar el curso explicando cómo programar un cliente y un servidor en Java utilizando sockets, en dos versiones. La primera versión consistirá en un servidor mono-thread. La segunda versión del servidor será multi-thread.

Los programas cliente/servidor serán la base de la mayoría de los sistemas que desarrollaremos en el curso. Por esta razón es muy importante entender completamente su funcionamiento.

Vamos utilizar el cómputo en la nube

Actualmente todos usamos algún servicio en nube. Por ejemplo Hotmail, Gmail, Youtube, Uber, Netflix y Office 365 son ejemplos de servicios en la nube. También Google Drive y OneDrive son servicios en la nube.

Los primeros son aplicaciones (distribuidas) que ejecutan en la nube y los segundos son servicios de almacenamiento en la nube.

Las empresas están migrando sus sistemas a la nube, por esa razón es muy importante que los egresados de la ESCOM puedan desarrollar, instalar y/o administrar sistemas en la nube.

En nuestro curso vamos a utilizar la nube de Microsoft llamada Azure. Para ello **es necesario que todos** tengan una cuenta de correo institucional del IPN y se inscriban al programa gratuito [Azure for Students](#).

Este programa, Microsoft les regala 100 dólares en servicios de nube de Azure durante un año, sin la necesidad de dar una tarjeta de crédito. Sólo es necesario demostrar su condición de alumno (mediante la cuenta de correo institucional).

Inicialmente vamos a explicar cómo crear máquinas virtuales en la nube (Linux y Windows).

Entonces vamos a utilizar las máquinas virtuales como una red de computadoras donde probaremos los sistemas distribuidos que desarrollaremos durante el curso.

Debido a que 100 dólares no es mucho es términos de servicios en la nube, deberemos tener mucho cuidado en apagar o eliminar las máquinas virtuales tan pronto realicemos alguna prueba o tarea.

Más allá de la teoría "by the book" vamos a aterrizar los temas del curso en la nube. Esto les dará una ventaja competitiva importante al integrarse a la industria.

Vamos a jugar

En nuestro curso vamos a implementar la "gamificación" (game=juego) como apoyo didáctico.

Vamos a jugar [kahoots](#) sobre los temas del curso. A los ganadores de cada kahoot se les otorgará puntos directos a la calificación parcial. Un punto al primer lugar, 1/2 punto al segundo lugar y 1/4 de punto al tercer lugar.

Jugar los kahoots será opcional, pero es conveniente que todos jueguen ya que los exámenes parciales incluirán preguntas parecidas a las de los kahoots.

Los kahoots son más divertidos si se juegan presencialmente, pero en las condiciones actuales vamos a jugar los kahoots en la modalidad de "challenge". En esta modalidad se publicará en la clase del día un enlace al kahoot y cada alumno podrá jugarlo desde cualquier lugar.

Habrà una fecha y hora límite para jugar cada kahoot.

Evaluación parcial

Cada parcial se evaluará de la siguiente forma:

- Tareas (50%)
- Examen teórico (40%)
- Participación en clase (10%)

Las tareas se deberán entregar en tiempo y forma en la plataforma moodle. No habrá extensión en la fecha de entrega de las tareas, salvo causas plenamente justificadas.

Se recomienda realizar las tareas tan pronto se publiquen en moodle, de tal manera que si tienen alguna duda o de plano no corre el programa, puedan consultar con el profesor.

Como pueden ver, las tareas tienen la mayor ponderación en la calificación.

Asistencia a clases

La asistencia a clase será el acceso que hagan a la plataforma moodle para estudiar la "Clase del día" el mismo día en que tenemos clase de acuerdo al horario presencial.

Esto quiere decir que deberán acceder a la plataforma los días: **lunes, miércoles y jueves** a cualquier hora. Los días no laborables no habrá "Clase del día" por lo tanto no es necesario que ingresen a moodle, no obstante la plataforma estará disponible 24x7 durante todo el curso; por cierto, la plataforma moodle ejecuta sobre Azure.

Para obtener el 10% de participación en clase deberán tener el 80% de asistencias en el parcial. Por ejemplo, si en el parcial tenemos 15 clases, será necesario tener 12 asistencias a la "Clase del día" para obtener el 10% de participación. Si no pueden acceder a alguna "Clase del día" por causas justificadas, deberán comunicarlo al profesor.

Actividades individuales a realizar

1. Obtener una cuenta de correo institucional del IPN.
2. Darse de alta en el programa [Azure for Students](#).

Clase del día - 30/09/2020



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 conexión 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 utilizan 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 OutputStream(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 InputStream(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 ByteBuffer 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**:

```
salida.write(a);
```

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 `ByteBuffer` 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();
```


Actividades individuales a realizar

1. Compile los programas [Cliente.java](#) y [Servidor.java](#)
2. Ejecute el programa [Servidor.java](#) en una ventana de comandos de Windows (o terminal de Linux) y ejecute el programa [Cliente.java](#) en otra ventana de comandos de Windows (o terminal de Linux).
3. Modifique el programa cliente para que envíe 10000 números punto flotante utilizando el método `writeDouble` (enviar los números 1.0, 2.0, 3.0 ... 10000.0). Mida el tiempo que tarda el programa cliente en enviar los 10000 números, se sugiere utilizar el método `System.currentTimeMillis()`


4. Modifique el programa servidor para que reciba los 10000 números que envía el programa cliente. Mida el tiempo que tarda el programa servidor en recibir los 10000 números.
5. Ahora modifique el programa cliente para que envíe los 10000 números utilizando ByteBuffer. Mida el tiempo que tarda el programa cliente en enviar los 10000 números.
6. Modifique el programa servidor para que reciba los 10000 números utilizando ByteBuffer. Mida el tiempo que tarda el programa servidor en recibir los 10000 números.
7. ¿Qué resulta más eficiente, enviar los números de manera individual mediante writeDouble o enviarlos empacados mediante ByteBuffer?

 [Cliente.java](#)




 [Servidor.java](#)



 [Cliente2.java](#)



 [Servidor2.java](#)



Clase del día - 01/10/2020

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();
```

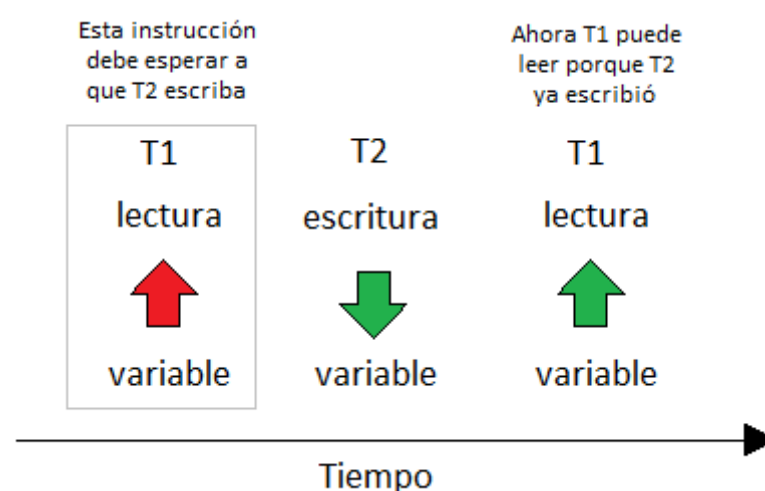
Cuando un thread (en este caso el thread principal) espera la terminación de uno o más threads para continuar su ejecución, se dice que se implementa una **barrera**. En este caso estamos implementando una barrera mediante dos métodos **join**.

Sincronización de threads

Cuando 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

1. Compile los programas [Cliente2.java](#) y [Servidor2.java](#)
2. Ejecute el programa [Cliente2.java](#) en una ventana de comandos de Windows (o terminal de Linux) y ejecute el programa [Servidor2.java](#) en otra ventana de comandos de Windows (o terminal de Linux).
3. Ejecute repetidamente el programa [Cliente2.java](#) en la ventana de comandos, como puede ver el servidor sigue en ejecución recibiendo las conexiones de los clientes y procesando los datos.
4. Compile la clase A que no utiliza sincronización y la clase que utiliza sincronización
 - ¿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?



Tarea 1. Cálculo distribuido de PI



En esta tarea vamos a desarrollar un programa distribuido, el cual calculará una aproximación de PI utilizando la serie de [Gregory-Leibniz](#).

La serie tiene la siguiente forma: $4/1 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + 4/13 - 4/15 \dots$

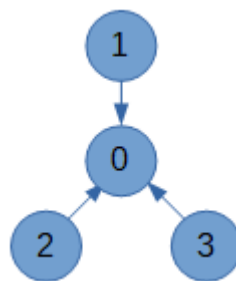
Notar que los denominadores son los números impares: 1,3,5,7,9,11,13 ...

El programa va a ejecutar en forma distribuida sobre cuatro nodos, cada nodo será una computadora diferente.

Por lo pronto, vamos a probar el programa en una sola computadora utilizando cuatro ventanas de comandos de Windows o cuatro terminales de Linux, en cada ventana se ejecutará una instancia del programa.

Cada nodo (incluso el nodo 0) deberá calcular 10 millones de términos de la serie.

Implementaremos la siguiente topología lógica de tipo estrella, cada nodo se ha identificado con un número entero:



El nodo 0 actuará como servidor y los nodos 1, 2 y 3 actuarán como clientes.

Dado que el requerimiento es desarrollar un solo programa, será necesario pasar como parámetro al programa el número de nodo actual, de manera que el programa pueda actuar como servidor o como cliente, según el número de nodo que pasa como parámetro.

Consideremos el siguiente programa:

```

class PI
{
    static Object lock = new Object();
    static double pi = 0;
    static class Worker extends Thread
    {
        Socket conexion;
        Worker(Socket conexion)
        {
            this.conexion = conexion;
        }
        public void run()
        {
            // Algoritmo 1
        }
    }
    public static void main(String[] args) throws Exception
    {
        if (args.length != 1)
        {
            System.err.println("Uso:");
            System.err.println("java PI <nodo>");
            System.exit(0);
        }
        int nodo = Integer.valueOf(args[0]);
        if (nodo == 0)
        {
            // Algoritmo 2
        }
        else
        {
            // Algoritmo 3
        }
    }
}
  
```

Se propone implementar los siguientes algoritmos:

Algoritmo 1

1. Crear los streams de entrada y salida.
2. Declarar la variable "x" de tipo double.
3. Recibir en la variable "x" la suma calculada por el cliente.
4. En un bloque synchronized mediante el objeto "lock":
 - 4.1 Asignar a la variable "pi" la expresión: $x + pi$
5. Cerrar los streams de entrada y salida.
6. Cerrar la conexión "conexion".

Algoritmo 2

1. Declarar una variable "servidor" de tipo ServerSocket.
2. Crear un socket servidor utilizando el puerto 50000 y asignarlo a la variable "servidor".
3. Declarar un vector "w" de tipo Worker con 3 elementos.
4. Declarar una variable entera "i" y asignarle cero.
5. En un ciclo:
 - 5.1 Si la variable "i" es igual a 3, entonces salir del ciclo.
 - 5.2 Declarar una variable "conexion" de tipo Socket.
 - 5.3 Invocar el método servidor.accept() y asignar el resultado a la variable "conexion".
 - 5.4 Crear una instancia de la clase Worker, pasando como parámetro la variable "conexion". Asignar la instancia al elemento w[i].
 - 5.5 Invocar el método w[i].start()
 - 5.6 Incrementar la variable "i".
 - 5.7 Ir al paso 5.1
6. Declarar la variable "suma" de tipo double y asignarle cero.
7. Declarar la variable "i" de tipo entero y asignarle cero.
8. En un ciclo:
 - 8.1 Si la variable "i" es igual a 10000000, entonces salir del ciclo.
 - 8.2 Asignar a la variable "suma" la expresión: $4.0/(8*i+1)+ suma$
 - 8.3 Incrementar la variable "i".
 - 8.4 Ir al paso 8.1
9. En un bloque synchronized mediante el objeto "lock":
 - 9.1 Asignar a la variable "pi" la expresión: $suma+pi$
10. Declarar una variable "i" entera y asignarle cero.
11. En un ciclo:
 - 11.1 Si la variable "i" es igual a 3, entonces salir del ciclo.
 - 11.2 Invocar el método w[i].join()
 - 11.3 Incrementar la variable "i".
 - 11.4 Ir al paso 11.1
12. Desplegar el valor de la variable "pi".

Algoritmo 3

1. Declarar la variable "conexion" de tipo Socket y asignarle null.
2. Realizar la conexión con el servidor implementando re-intento. Asignar el socket a la variable "conexion".
3. Crear los streams de entrada y salida.
4. Declarar la variable "suma" de tipo double y asignarle cero.
5. Declarar una variable "i" de tipo entero y asignarle cero.
6. En un ciclo:
 - 6.1 Si la variable "i" es igual a 10000000, entonces salir del ciclo.
 - 6.2 Asignar a la variable "suma" la expresión: $4.0/(8*i+2*(nodo-1)+3)+suma$
 - 6.3 Incrementar la variable "i".
 - 6.4 Ir al paso 6.1
7. Asignar a la variable "suma" la expresión: $nodo\%2==0?suma:-suma$
8. Enviar al servidor el valor de la variable "suma".
9. Cerrar los streams de entrada y salida.
10. Cerrar la conexión "conexion".

Notar que el algoritmo 1 se deberá ejecutar dentro de un bloque try.

También se debe notar que la variable "lock" debe ser estática para que todos los threads accedan al mismo lock.

Se deberá subir a la plataforma un archivo ZIP que contenga el código fuente del programa desarrollado y un documento PDF con la captura de pantalla de la compilación y ejecución del programa. El archivo PDF deberá incluir una descripción de cada captura de pantalla.

Valor de la tarea: 30% (1.5 punto de la primera evaluación parcial)



Clase del día - 05/10/2020

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.

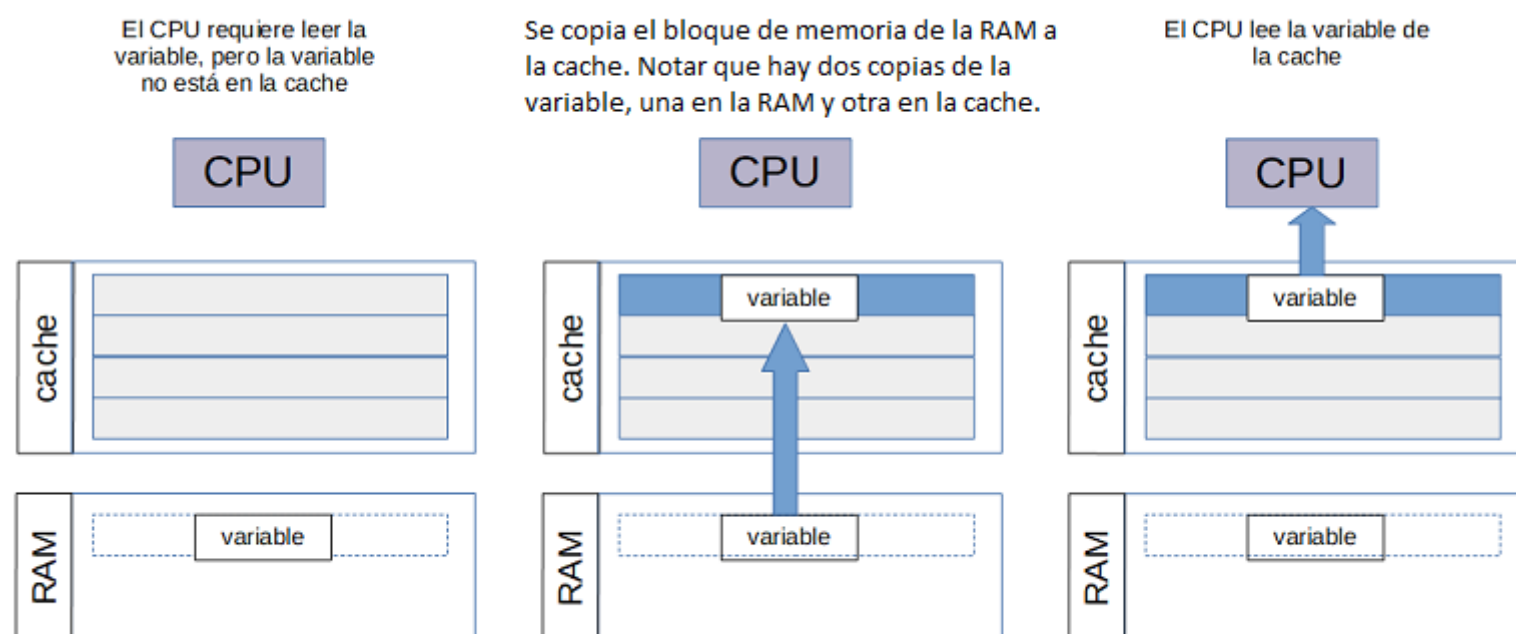


1. El CPU utiliza los **registros** para realizar operaciones aritméticas, lógicas y de control.
2. La memoria **cache** consiste en una memoria asociativa. Las memorias asociativas son muy rápidas, pero como son costosas, suelen ser de poca capacidad. La memoria cache puede estar dividida en varios niveles L1, L2, ...
3. 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.
4. El **disco duro** almacena de manera persistente grandes cantidades de datos.
5. Los **respaldos** pueden ser discos duros de gran capacidad, discos ópticos, cintas, entre otros.

La memoria cache

Lectura de una variable

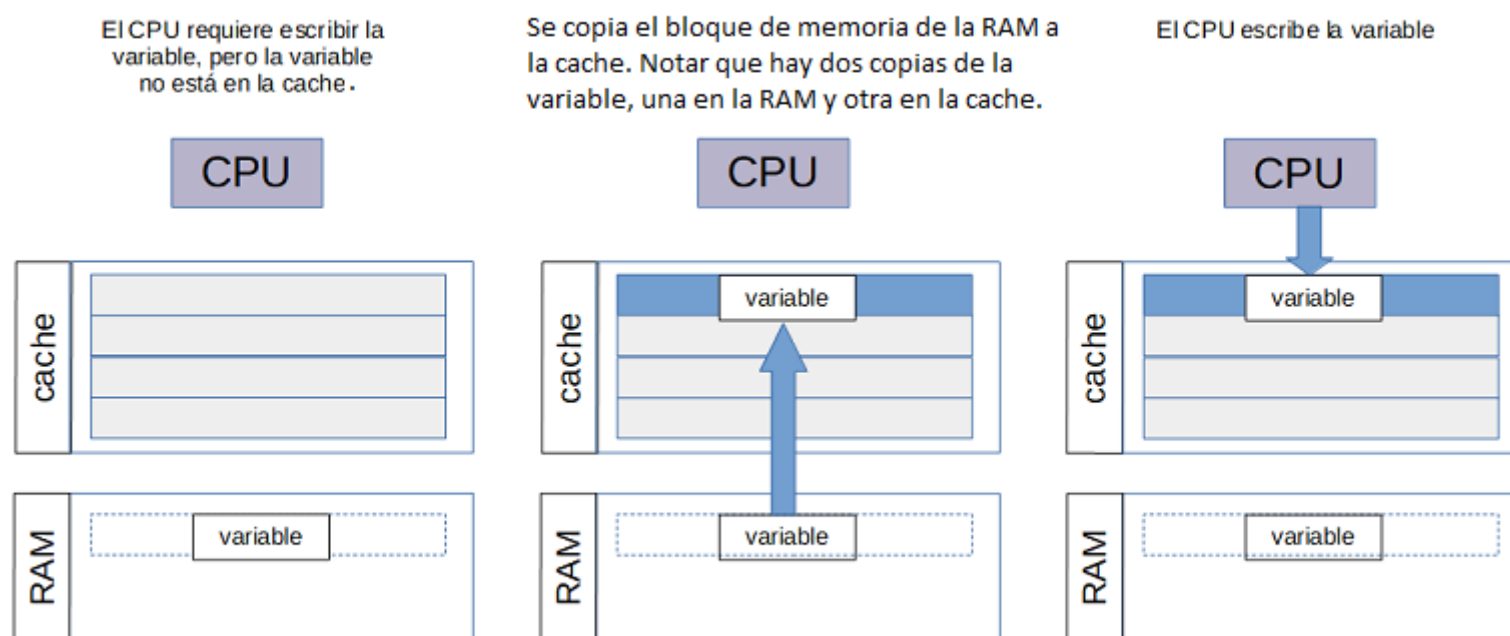
Cuando el CPU requiere leer una variable que se encuentra en la memoria RAM, busca la variable en la cache, si la variable no existe en la cache, copia el bloque de datos (que contiene a la variable) a la cache, entonces el CPU lee la variable de la cache.



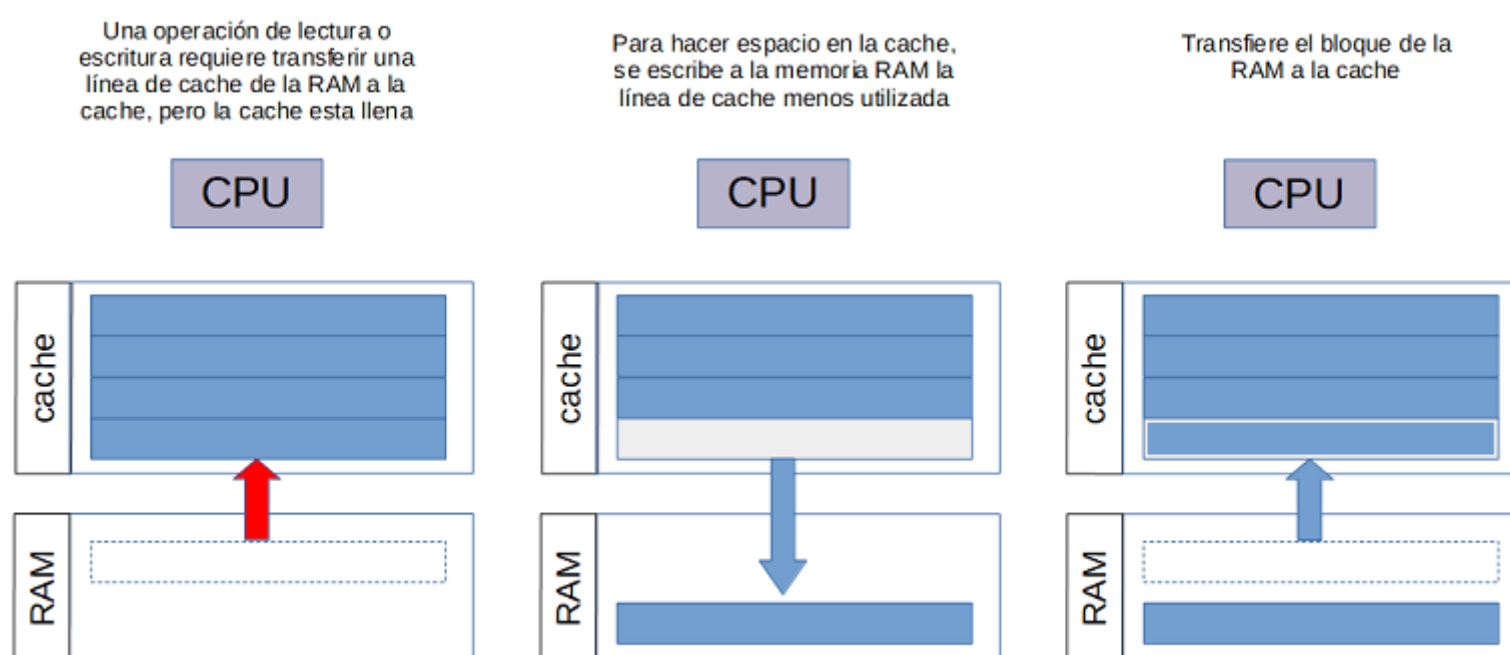
Al bloque de memoria que se transfiere de la RAM a la cache se le llama **línea de cache**. El tamaño de una línea de cache típicamente es de cientos de Kilobytes o Megabytes.

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 los expedientes individuales del archivo.

1. El jefe le pide a su asistente el expediente del Sr. González.
2. La asistente va al archivo a traer el expediente del Sr. González.
3. La asistente le da a su jefe el expediente del Sr. González
4. El jefe le pide a su asistente el expediente del Sr. Gómez.
5. La asistente va al archivo a traer el expediente del Sr. Gómez.
6. La asistente le da a su jefe el expediente del Sr. Gómez.
7. El jefe regresa a su asistente el expediente del Sr. González.
8. La asistente va al archivo a dejar el expediente del Sr. González
9. El jefe le pide a su asistente el expediente del Sr. González.
10. La asistente va al archivo a traer el expediente del Sr. González.
11. 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.

1. El jefe le pide a su asistente el expediente del Sr. González.
2. La asistente va al archivo a traer la caja correspondiente a la letra "G".
3. La asistente le da a su jefe el expediente del Sr. González.
4. El jefe le pide a su asistente el expediente del Sr. Gómez.
5. La asistente le da a su jefe el expediente del Sr. Gómez.
6. El jefe regresa el expediente del Sr. González.
7. El jefe le pide a su asistente el expediente del Sr. González.
8. 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.

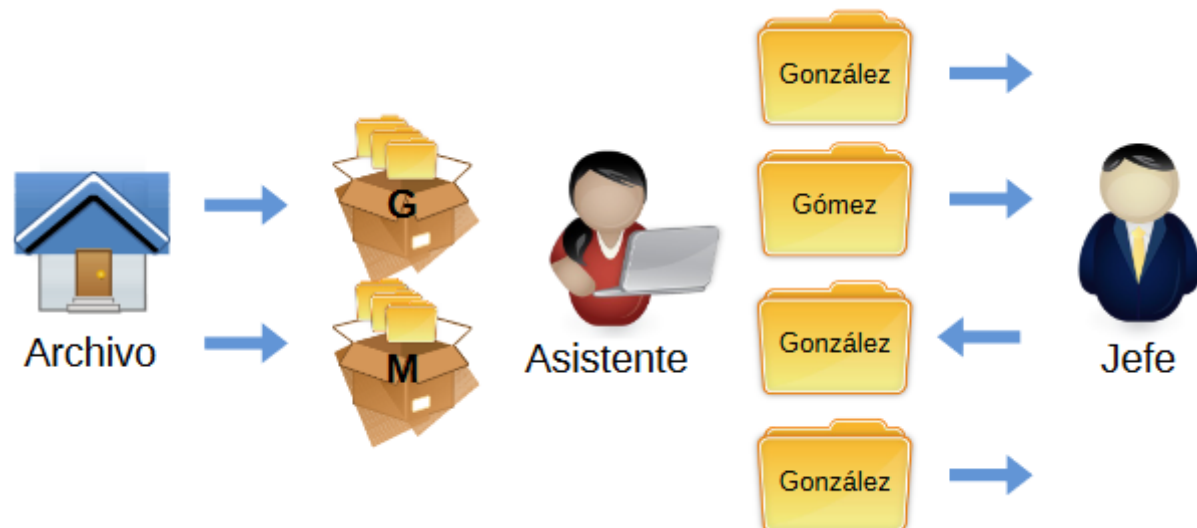
La asistente ha descubierto los conceptos de localidad espacial y localidad temporal.

- 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.

1. El jefe le pide a su asistente el expediente del Sr. González.
2. La asistente va al archivo a traer la caja correspondiente a la letra "G".
3. La asistente le da a su jefe el expediente del Sr. González.
4. El jefe le pide a su asistente el expediente del Sr. Morales.
5. La asistente va al archivo a traer la caja correspondiente a la letra "M".
6. La asistente le da a su jefe el expediente del Sr. Morales.
7. El jefe regresa el expediente del Sr. González.
8. El jefe le pide a su asistente el expediente del Sr. González.
9. 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 accedía 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.

Actividades individuales a realizar

1. Compilar y ejecutar los programas MultiplicaMatriz.java y MultiplicaMatriz_2.java.
2. ¿Por qué el segundo programa es más rápido que el primero?
3. ¿Podría plantear **otro programa** dónde el aumento de la localidad espacial y/o temporal hace más eficiente la ejecución? (enviar por email al profesor el programa original y el programa modificado; 1/2 punto extra en el parcial al primer alumno o alumna que envíe el mismo programa).

En la clase de hoy vamos a jugar un kahoot en la modalidad de "challenge" sobre cliente-servidor multithread. 

Para jugar este kahoot deberán ingresar a la siguiente URL:

https://kahoot.it/challenge/08283790?challenge-id=3882dbfd-4f85-4e86-9fa2-018882d19a9a_1601869345124

Es necesario que los alumnos y alumnas ingresen su "nickname" como su nombre y apellido (por ejemplo JuanLopez), de manera que sea posible identificar a los ganadores de puntos extra.

La hora límite para jugar este kahoot es 11:00 PM del 5 de octubre.



Tarea 2. Uso eficiente de la memoria cache



Compilar y ejecutar los programas MultiplicaMatriz.java y MultiplicaMatriz_2.java que vimos en clase, para los siguientes valores de N: 100, 200, 300, 500, 1000.

Utilizando Excel hacer una gráfica de dispersión (con líneas, sin marcadores) dónde se muestre el tiempo de ejecución de ambos programas con respecto a N (N en el eje X y el tiempo en el eje Y).

Se deberá subir a la plataforma un documento PDF incluyendo el código fuente de los programas, la explicación de cada programa, la gráfica y los siguientes datos:

- Marca y modelo del CPU dónde ejecutó los programas.
- Tamaño de la cache del CPU.
- Tamaño de la RAM.

Valor de la tarea: 10% (1/2 punto de la primera evaluación parcial)



Clase del día - 07/10/2020

En la clase de hoy vamos a jugar un kahoot en la modalidad de "challenge" sobre cache, localidad espacial y localidad temporal.

Para jugar este kahoot deberán ingresar a la siguiente URL:

https://kahoot.it/challenge/07056743?challenge-id=3882dbfd-4f85-4e86-9fa2-018882d19a9a_1602045898005

Es necesario que los alumnos y alumnas ingresen su "nickname" como su nombre y apellido (por ejemplo JuanLopez), de manera que sea posible identificar a los ganadores de puntos extra.

La hora límite para jugar este kahoot es 11:00 PM del 7 de octubre.

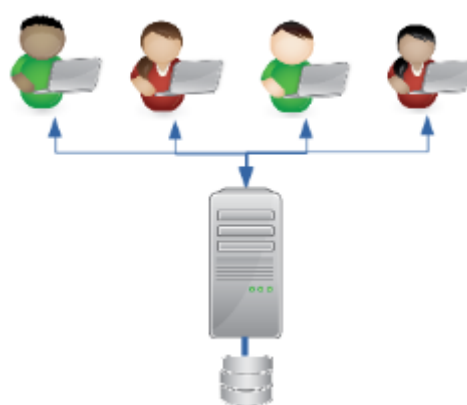


Clase del día - 08/10/2020

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

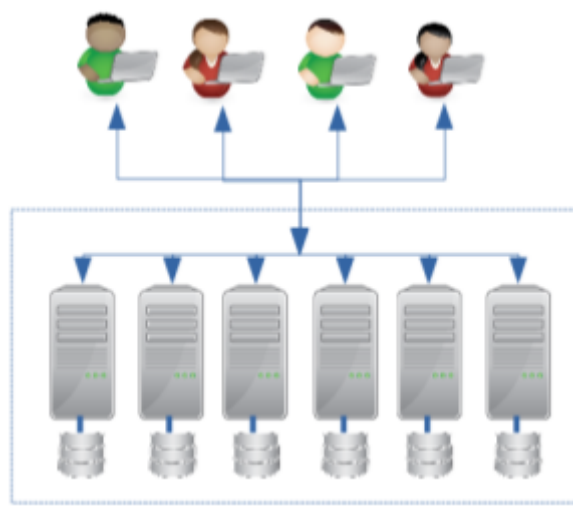
Generalmente los sistemas manejadores de bases de datos (DBMS) son centralizados debido a que resulta más fácil programar sistemas que accedan los datos que se encuentran en una base de datos central. Sin embargo, las plataformas de alcance mundial como Facebook, Twitter, Uber, etc. requieren distribuir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

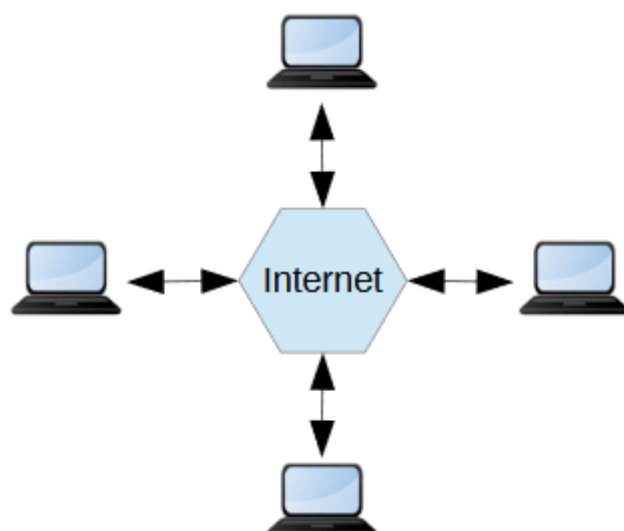
World Wide Web

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

Search 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 (10^{12} operaciones de punto flotante por segundo).

TOP500

Las [500 computadoras](#) más grandes del mundo.

Actualmente la computadora más grande del mundo tiene 7,299,072 procesadores con Linux Red Hat; no se trata de una computadora centralizada sino de un sistema distribuido, alcanzando un rendimiento pico de 513,854.7 TFlop/s

Actividades individuales a realizar

1. Investigar la ley de Amdahl.
2. En la tarea 1 utilizamos cuatro nodos para calcular una aproximación de PI. Suponga que modifica el programa para ejecutar sobre 1, 2, 3 y 4 nodos. Si para cada uno de los casos se obtienen los siguientes tiempos: 20 segundos, 10 segundos, 7 segundos y 5 segundos, obtenga la gráfica de aceleración (speedup) de acuerdo a la ley de Amdahl.
3. Considere la lista del TOP500 ¿En qué lugar aparece la primera computadora con Windows?



Tarea 3. Multiplicación distribuida de matrices



En esta tarea cada alumno deberá desarrollar **un programa en Java**, el cual calculará el producto de dos matrices cuadradas en forma distribuida sobre cinco nodos.

Sea N el tamaño de las matrices, entonces se deberá ejecutar dos casos:

1. N=4, se deberá desplegar las matrices A, B y $C=A \times B$ y el checksum de la matriz C.
2. N=1000, deberá desplegar el checksum de la matriz C.

La matrices se deberán inicializar de la siguiente manera:

$$A[i][j] = 2 * i + j$$

$$B[i][j] = 2 * i - j$$

El checksum de la matrix C es la suma de todos elementos de esta matriz.

El programa deberá ser ejecutado en cinco ventanas de Sistema (Windows) o en cinco terminales (Linux o MacOS).

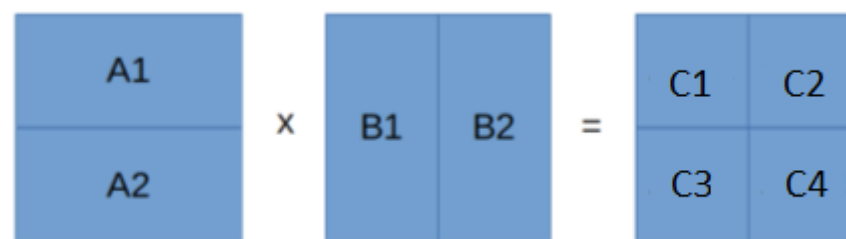
En cada ventana se pasará como parámetro al programa el número de nodo, a saber: 0, 1, 2, 3 y 4.

¿Cómo realizar la multiplicación de matrices en forma distribuida?

Suponga que divide la matriz A en las matrices A1 y A2. El tamaño de las matrices A1 y A2 es N/2 renglones y N columnas.

La matriz B se divide en las matrices B1 y B2. El tamaño de matrices B1 y B2 es N renglones y N/2 columnas.

Entonces la matriz $C=A \times B$ se compone de las matrices C1, C2, C3 y C4, tal como se muestra en la siguiente figura:



Donde:

$$C1 = A1 \times B1$$

$$C2 = A1 \times B2$$

$$C3 = A2 \times B1$$

$$C4 = A2 \times B2$$

Ahora supongamos que tenemos cinco nodos identificados con los números 0, 1, 2, 3 y 4.
Para multiplicar las matrices A y B implementaremos los siguientes procedimientos:

Nodo 0

1. Inicializar las matrices A y B.
2. Enviar la matriz A1 al nodo 1.
3. Enviar la matriz B1 al nodo 1.
4. Enviar la matriz A1 al nodo 2.
5. Enviar la matriz B2 al nodo 2.
6. Enviar la matriz A2 al nodo 3.
7. Enviar la matriz B1 al nodo 3.
8. Enviar la matriz A2 al nodo 4.
9. Enviar la matriz B2 al nodo 4.
10. Recibir la matriz C1 del nodo 1.
11. Recibir la matriz C2 del nodo 2.
12. Recibir la matriz C3 del nodo 3.
13. Recibir la matriz C4 del nodo 4.
14. Calcular el checksum de la matriz C.
15. Desplegar el checksum de la matriz C.
16. Si N=4 entonces desplegar la matriz C

Nodo 1

1. Recibir del nodo 0 la matriz A1.
2. Recibir del nodo 0 la matriz B1.
3. Realizar el producto $C1 = A1 \times B1$.
4. Enviar la matriz C1 al nodo 0.

Nodo 2

1. Recibir del nodo 0 la matriz A1.
2. Recibir del nodo 0 la matriz B2.
3. Realizar el producto $C2 = A1 \times B2$.
4. Enviar la matriz C2 al nodo 0.

Nodo 3

1. Recibir del nodo 0 la matriz A2.
2. Recibir del nodo 0 la matriz B1.
3. Realizar el producto $C3 = A2 \times B1$.
4. Enviar la matriz C3 al nodo 0.

Nodo 4

1. Recibir del nodo 0 la matriz A2.
2. Recibir del nodo 0 la matriz B2.
3. Realizar el producto $C4 = A2 \times B2$.
4. Enviar la matriz C4 al nodo 0.

Sin embargo, debido a que las matrices se guardan en memoria por renglones, es más eficiente transponer la matriz B y dividirla de la siguiente manera:

$$\begin{bmatrix} A1 \\ A2 \end{bmatrix} \times \begin{bmatrix} B1 \\ B2 \end{bmatrix} = \begin{bmatrix} C1 & C2 \\ C3 & C4 \end{bmatrix}$$

Entonces solo se deberá transponer la matriz B después del paso 1 en el procedimiento que ejecuta el nodo 0 y modificar el paso 3 del procedimiento que ejecuta en los nodos 1, 2 y 3, intercambiando los índices de la matriz B, tal como se hizo en la tarea 2.

Se deberá subir a la plataforma un archivo ZIP que contenga el código fuente del programa desarrollado y un documento PDF con la captura de pantalla de la compilación y ejecución del programa. El archivo PDF deberá incluir una descripción de cada captura de pantalla.

Valor de la tarea: 40% (2 puntos de la primera evaluación parcial)



Clase del día - 12/10/2020

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:

1. Facilidad en el acceso a los recursos.
2. Transparencia.
3. Apertura.
4. 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 sincrónica 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.

Actividades individuales a realizar

En esta actividad el alumno va a crear una máquina virtual con Ubuntu en la nube de Microsoft utilizando su cuenta de Azure for Students.

Es muy importante que el alumno elimine la máquina virtual una vez haya terminado de utilizarla, ya que mantener encendida una máquina virtual genera costo, lo que representa una disminución en el crédito que tiene el alumno como parte del programa Azure for Students.

Creación de una máquina virtual con Ubuntu

Ingresar al portal de Azure en la siguiente URL:

<https://azure.microsoft.com/es-mx/features/azure-portal/>

1. Dar click al botón "Iniciar sesión".
2. En el portal de Azure seleccionar "Máquinas virtuales".
3. Seleccionar la opción "+Agregar".
4. Seleccionar la opción "+Virtual machine"
5. Seleccionar el grupo de recursos o crear uno nuevo. Un grupo de recursos es similar a una carpeta dónde se pueden colocar los diferentes recursos de nube que se crean en Azure.
6. Ingresar el nombre de la máquina virtual.
7. Seleccionar la región dónde se creará la máquina virtual. Notar que el costo de la máquina virtual depende de la región.
8. Seleccionar la imagen, en este caso vamos a seleccionar Ubuntu Server 18.04 LTS.
9. Dar click en "Seleccionar tamaño" de la máquina virtual, en este caso vamos a seleccionar una máquina virtual con 1 GB de memoria RAM. Dar click en el botón "Seleccionar".
10. En tipo de autenticación seleccionamos "Contraseña".
11. Ingresamos el nombre del usuario, por ejemplo: ubuntu
12. Ingresamos la contraseña y confirmamos la contraseña. La contraseña debe tener al menos 12 caracteres, debe al menos una letra minúscula, una letra mayúscula, un dígito y un carácter especial.
13. En las "Reglas de puerto de entrada" se deberá dejar abierto el puerto 22 para utilizar SSH (la terminal de secure shell).
14. Dar click en el botón "Siguiente: Discos>"

15. Seleccionar el tipo de disco de sistema operativo, en este caso vamos a seleccionar HDD estándar.
16. Dar click en el botón "Siguiente: Redes>"
17. Dar click en el botón "Siguiente: Administración>"
18. En el campo "Diagnóstico de arranque" seleccionar "Desactivado".
19. Dar click en el botón "Revisar y crear".
20. Dar click en el botón "Crear".
21. Dar click a la campana de notificaciones (barra superior de la pantalla) para verificar que la maquina virtual se haya creado.
22. Dar click en el botón "Ir al recurso". En la página de puede ver la dirección IP pública de la máquina virtual. Esta dirección puede cambiar cad vez que se apague y se encienda la máquina virtual.
23. Para conectarnos a la máquina virtual vamos a utilizar el programa putty.exe, el cual se puede encontrar en la siguiente URL: <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>
24. Ejecutar el programa putty.exe
25. Escribir la dirección IP de la máquina virtual en el campo "Host Name". Dar click en el botón "Open". Putty despliega una ventana de alerta de seguridad preguntando si la huella digital del servidor es correcta, dar click al botón "Si".
26. Ingresar el nombre del usuario, en este caso: ubuntu (ver el paso 11). Ingresar la contraseña.

Detener una máquina virtual

Cuando una máquina virtual no se utiliza es conveniente detenerla con el fin de reducir el costo. Para detener una máquina virtual:

1. Dar click en la opción "Detener" en el portal de Azure.
2. Dar click en el botón "Aceptar".

Esperar a que el estado de la máquina virtual sea "Desasignada".

Encender una máquina virtual

Para encender una máquina virtual

1. Seleccionar la opción "Iniciar" en la página de la máquina virtual dentro del portal de Azure.

Esperar a que el estado de la máquina virtual sea "En ejecución".

Eliminar una máquina virtual

Para eliminar una máquina virtual:

1. Seleccionar la opción "Eliminar" en la página de la máquina virtual dentro del portal de Azure.
2. Dar clieck en el botón "Aceptar".

Los recursos asociados (discos, IP pública, interfaz de red, grupo de seguridad de red, etc.) no se eliminarán, para eliminarlos se deberá seleccionar cada recurso y eliminarlos manualmente.

Para eliminar los recursos asociados a una máquina virtual previamente eliminada:

1. Dar click al icono de "hamburguesa" (las tres líneas horizontales) localizado en la parte superior izquierda de

la pantalla.

2. Seleccionar "Todos los recursos".
3. Seleccionar cada recursos (dar click en cada checkbox)
4. Seleccionar "Eliminar".
5. Verificar la lista de recursos a eliminar.
6. Escribir la palabra Sí (con acento en la i).
7. Dar click en el botón "Eliminar".

Ver el video:

Clase del día - 14/10/2020



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 **mall** (*grid*).

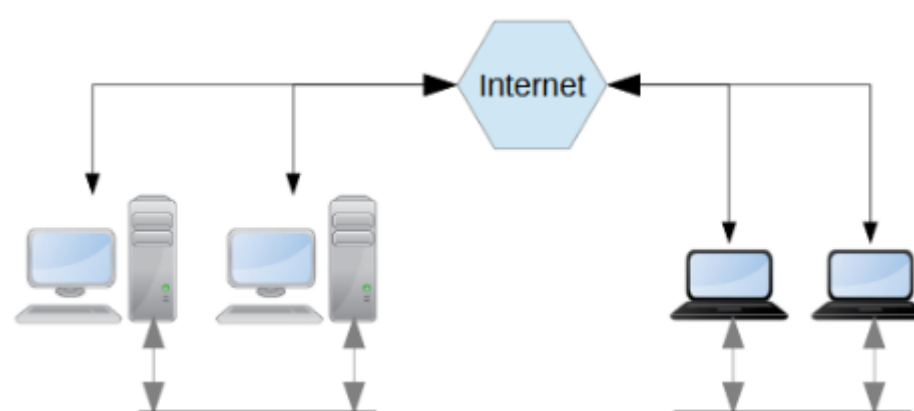
Un cluster es un conjunto de computadoras homogéneas con el mismo sistema operativo conectadas mediante una red local (LAN) de alta velocidad.



Los clusters se utilizan para el cómputo de alto rendimiento, donde los programas se distribuyen entre los diferentes nodos del cluster, con la finalidad de lograr rendimientos superiores.

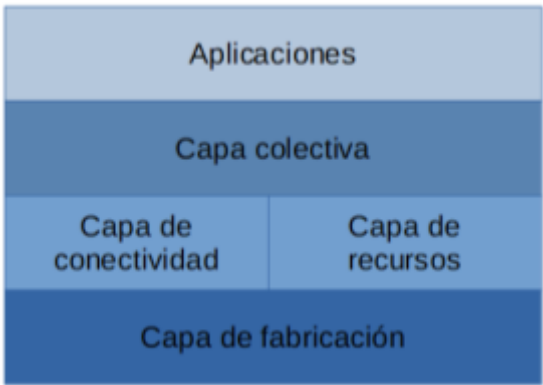
En el TOP500 474 sistemas son clusters, mientras que sólo 40 sistemas son MPP (*Massively Parallel Processing*). Un ejemplo de sistema MPP es la malla (*grid*).

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.

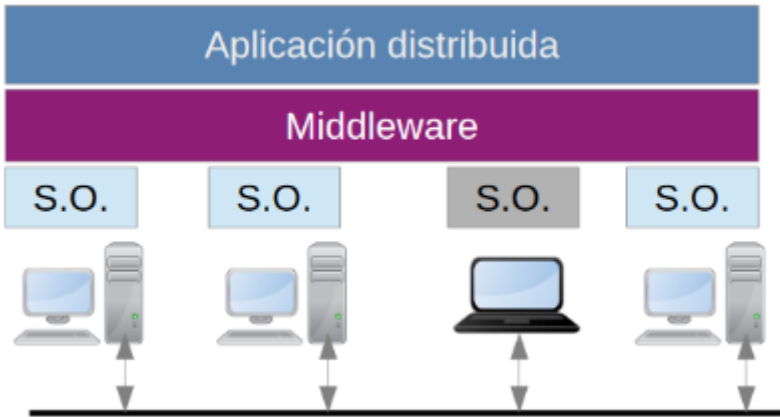
La **capa de aplicaciones** está compuesta por las aplicaciones que ejecutan dentro de la organización virtual.

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.



Actividades individuales a realizar

En esta actividad veremos como crear una máquina virtual con Windows, cómo conectarse a la máquina virtual y cómo transferir archivos.

Es muy importante que el alumno elimine la máquina virtual una vez haya terminado de utilizarla, ya que mantener encendida una máquina virtual genera costo, lo que representa una disminución en el crédito que tiene el alumno como parte del programa Azure for Students.

Creación de una máquina virtual con Windows

1. En el portal de Azure seleccionar "Máquinas virtuales".
2. Seleccionar la opción "+Agregar".
3. Seleccionar el grupo de recursos o crear uno nuevo.
4. Ingresar el nombre de la máquina virtual.
5. Seleccionar la región dónde se creará la máquina virtual. Notar que el costo de la máquina virtual depende de la región.
6. Seleccionar la imagen, en este caso vamos a seleccionar Windows 10 Pro.
7. Seleccionar el tamaño de la máquina virtual, en este caso vamos a seleccionar una máquina virtual con al menos 2 GB de memoria.
8. Ingresar el nombre del usuario administrador y la contraseña.
9. En las "Reglas de puerto de entrada" se deberá dejar abierto el puerto 3389 para utilizar Remote Desktop Protocol (RDP).
10. Dar click en el botón "Siguiente: Discos>"
11. Seleccionar el tipo de disco de sistema operativo, en este caso vamos a seleccionar HDD estándar.
12. Dar click en el botón "Siguiente: Redes>"
13. Dar click en el botón "Siguiente: Administración>"
14. En el campo "Diagnóstico de arranque" seleccionar "Desactivado".
15. Dar click en el botón "Revisar y crear".
16. Dar click en el botón "Crear".
17. Dar click a la campana de notificaciones para verificar que la maquina virtual se haya creado.
18. Dar click en el botón "Ir al recurso".
19. Seleccionar la opción "Conectar". Seleccionar "RDP".
20. Dar click en el botón "Descargar archivo RDP".
21. Ejecutar "cmd" en la computadora local.
22. Vamos a crear un directorio en la computadora local. La máquina virtual recién creada va a ver este directorio como un disco lógico. Por ejemplo, el directorio se llamará "prueba". Ejecutar el siguiente comando en la ventana de Símbolo del sistema:

`mkdir prueba`
23. Ahora vamos a crear un disco lógico como alias del directorio creado. Ejecutar el siguiente comando:

`subst f: prueba`
- Podemos ver que el disco lógico aparece en el explorador de archivos de Windows.
24. Buscar el archivo de conexión en la carpeta de descargas (un archivo con el nombre de la máquina virtual y la extensión ".rdp").
22. Dar click derecho al archivo de conexión y seleccionar "Editar".
23. Seleccionar la pestaña "Recursos locales".
24. Dar click en el botón "Mas..."
25. Abrir la sección "Unidades".
26. Marcar la casilla "Windows (F:)"
27. Dar click en el botón "Aceptar".

28. Dar click en el botón "Conectar" en la pantalla de advertencia.

29. Ingresar el nombre de usuario administrador y la contraseña.

30. Dar click en el botón "Sí" en la ventana de advertencia. Entonces se abrirá una ventana de escritorio remoto, la cual nos dará acceso al escritorio de la máquina virtual.

31. Configurar los parámetros de privacidad y dar click en el botón "Accept".

32. En la ventana "Networks" dar click en el botón "No".

33. Para ver el disco lógico creado en el paso 23, abrir el explorador de Windows de la máquina virtual. Entonces para enviar archivos desde la computadora local a la máquina virtual se deberá colocar los archivos en el directorio creado en el paso 22, y para enviar archivos desde la máquina virtual a la computadora local se deberá colocar los archivos en el disco F de la máquina virtual.

Nota. El teclado local podría no coincidir con la configuración del teclado de la maquina remota.

34. Para desconectarse de la máquina virtual, dar click en el botón "X" del escritorio remoto.

Ver el video:

Clase del día - 15/10/2020



En la clase de hoy vamos a jugar dos kahoots en la modalidad de "challenge" sobre los siguientes temas:

- Sistema centralizado
- Sistema distribuido
- Tipos de distribución
- Objetivos de los sistemas distribuidos
- Técnicas de escalamiento
- Requisitos de diseño
- Tipos de sistemas distribuidos

Para jugar estos kahoots deberán ingresar a los siguientes enlaces:

[Desarrollo de Sistemas Distribuidos - Conceptos básicos - Parte 1](#)

[Desarrollo de Sistemas Distribuidos - Conceptos básicos - Parte 2](#)

Es necesario que los alumnos y alumnas ingresen su "nickname" como su nombre y apellido (por ejemplo JuanLopez), de manera que sea posible identificar a los ganadores de puntos extra.

La hora límite para jugar estos kahoots es 11:00 PM del 15 de octubre.

2. Sincronización y coordinación



Clase del día - 19/10/2020

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:

1. En cada compra se ha registrado la fecha y hora local, y
2. 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).

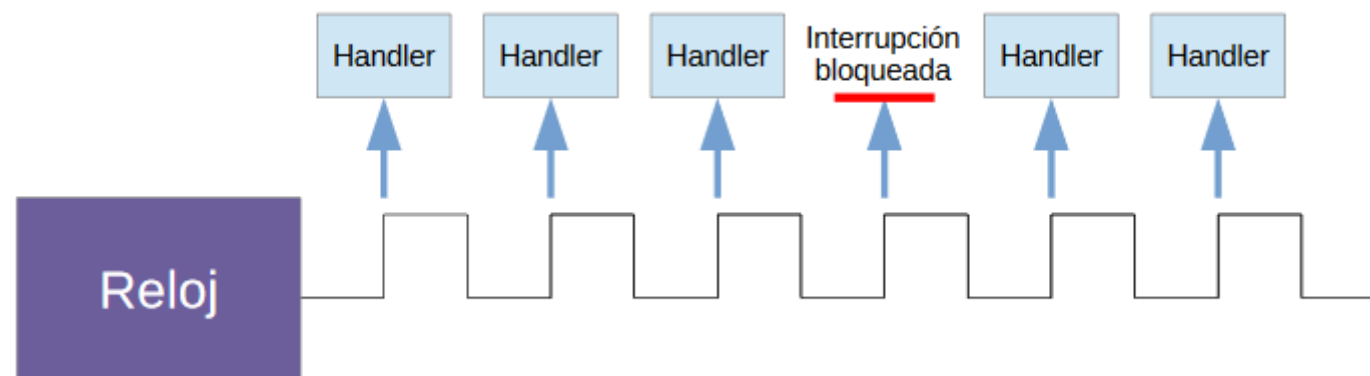
Más adelante veremos que éste orden puede establecerse utilizando relojes físicos (mecanismos que marcan el tiempo real) o bien relojes lógicos (contadores).

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:

1. 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.
2. 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 oscuridad 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.

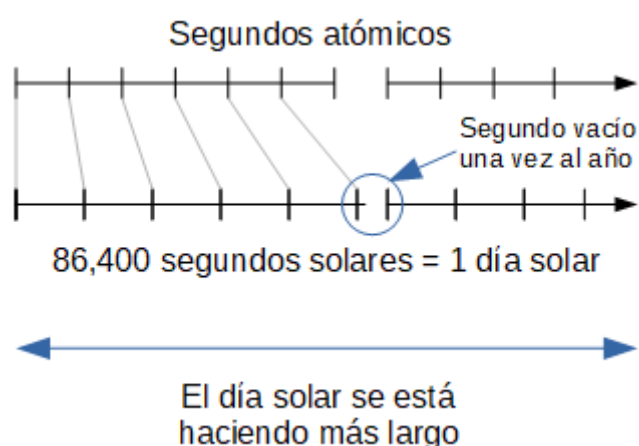
Tiempo universal coordinado UTC

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.

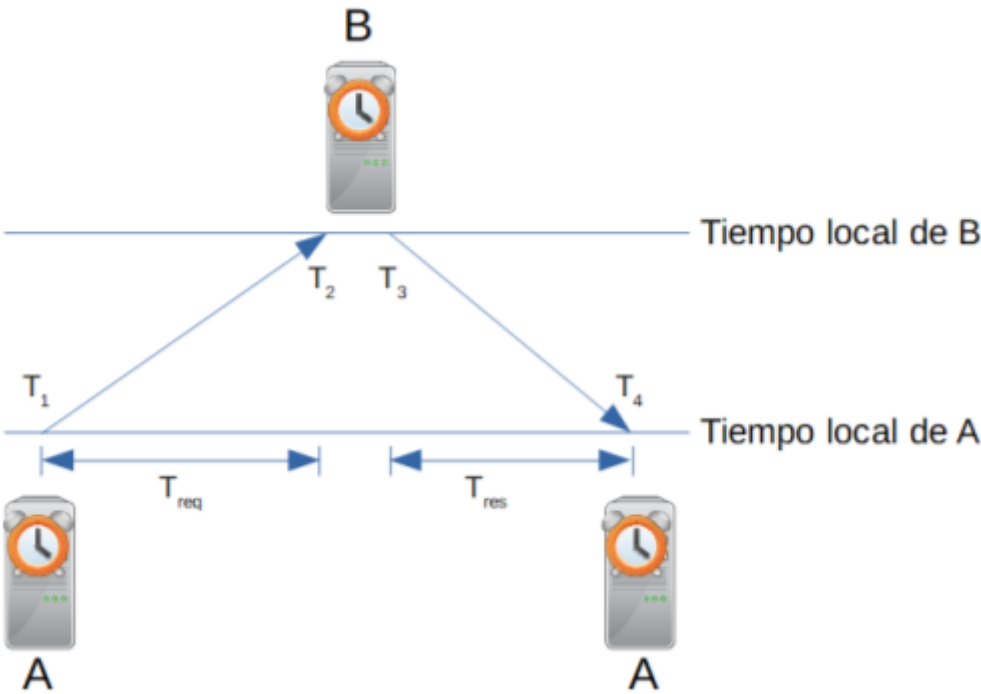
Network Time Protocol NTP

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 sincronizado 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 T_1 el cliente A envía una petición al servidor B, la petición llega al servidor al tiempo local T_2 . El servidor B procesa el requerimiento y al tiempo local T_3 envía la respuesta al cliente A, la respuesta llega al cliente al tiempo local T_4 .

Si el servidor B envía T_3 y T_2 a la computadora A y suponemos $T_{req}=T_{res}$, entonces la computadora A puede estimar T_{res} ya que conoce $T_4-T_1=T_{req}+T_{res}+(T_3-T_2)=2T_{res}+(T_3-T_2)$, por tanto la computadora A podrá modificar su tiempo local a T_3+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 relojes 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*.

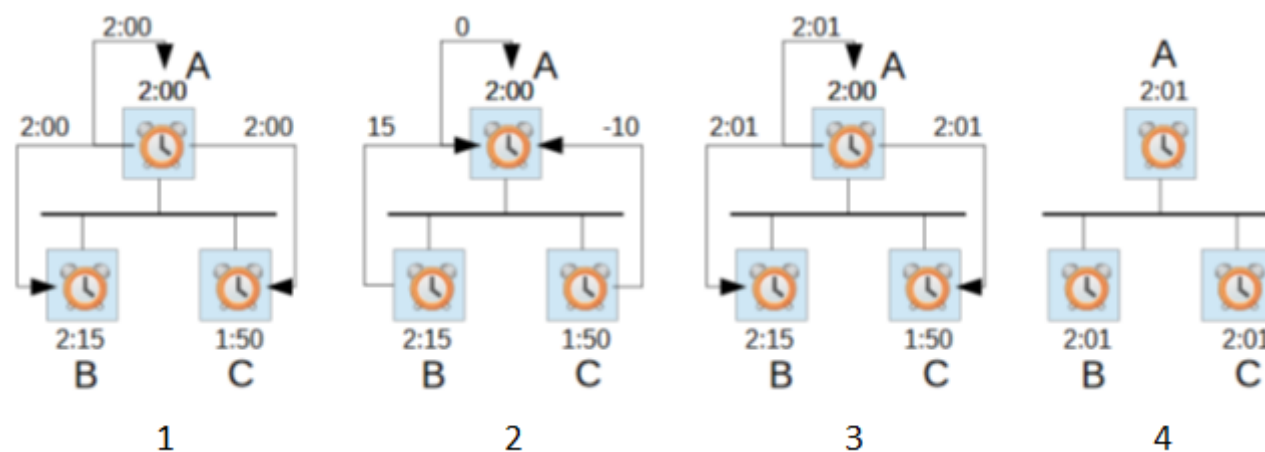
Por tanto, si un grupo de computadoras no se conectan con otras computadoras, es suficiente sincronizar los tiempos de las computadoras en el grupo, aún si el tiempo sincronizado no corresponde al tiempo real (ya que no hay una comunicación con otras computadoras).

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:

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

A continuación se muestra un ejemplo:



Actividades individuales a realizar

1. Considere el ejemplo de la plataforma de comercio electrónico global que planteamos en la clase.
 - 1.1 Si cada servidor es una máquina virtual en la nube ¿las compras se registrarán en tiempo local o en tiempo global?
 - 1.2 Si se instala NTP en cada servidor ¿se puede garantizar que los relojes de los servidores tengan la misma hora?
2. Cree una máquina virtual con Ubuntu en la nube de Azure.
 - 2.1 Ejecute el comando **date** ¿la hora es local o global?
 - 2.2 Instale NTP en la máquina virtual.
 - 2.3 Elimine la máquina virtual y sus recursos asociados.
3. Suponga que tiene 5 computadoras con los siguientes tiempos: 10:20, 13:10, 9:00, 12:15 y 11:30.
 - 3.1 Si la tercera computadora inicia el algoritmo de sincronización de relojes de Berkeley ¿qué hora tendrá cada computadora al terminar el proceso de sincronización?

Clase del día - 21/10/2020



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

En el artículo [Time, Clocks, and the Ordering of Events in Distributed Systems](#) (1978) Leslie Lamport define la relación $A \rightarrow B$ (se lee, A happens-before B) de la siguiente manera:

- 1. Si A y B son eventos del mismo proceso y A ocurre antes que B, entonces $A \rightarrow B$
- 2. Si A es el envío de un mensaje y B la recepción del mensaje, entonces $A \rightarrow B$

La relación happens-before tiene las siguientes propiedades:

- Transitiva: Si $A \rightarrow B$ y $B \rightarrow C$ entonces $A \rightarrow C$
- Anti-simétrica: Si $A \rightarrow B$ entonces no($B \rightarrow A$)
- Irreflexiva: no($A \rightarrow A$) 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 $A \rightarrow B$ 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 happens-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.

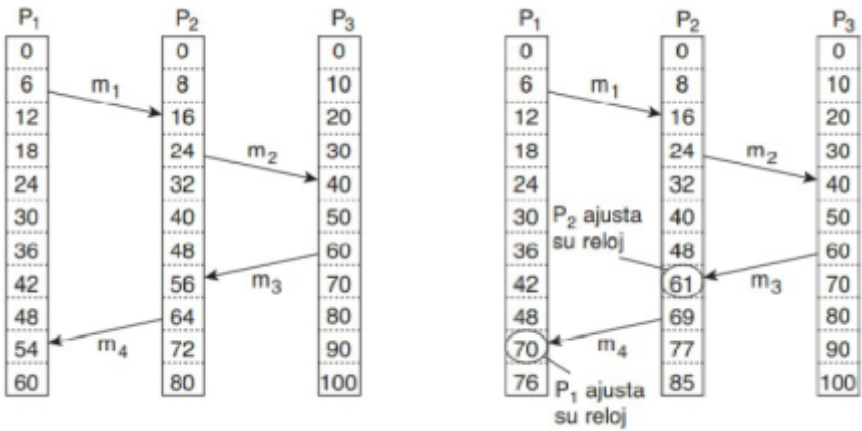
El reloj lógico del procesador P_1 se incrementa en 6, el reloj lógico del procesador P_2 se incrementa en 8 y el reloj lógico del procesador P_3 se incrementa en 10.

¿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 happens-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 happens-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 happens-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 .

Exclusión mutua

Ahora veremos algunos algoritmos que resuelven el problema de exclusión mutua cuando dos o más procesadores requieren acceder simultáneamente un recurso compartido (impresora, memoria, CPU, disco, etc.).

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.

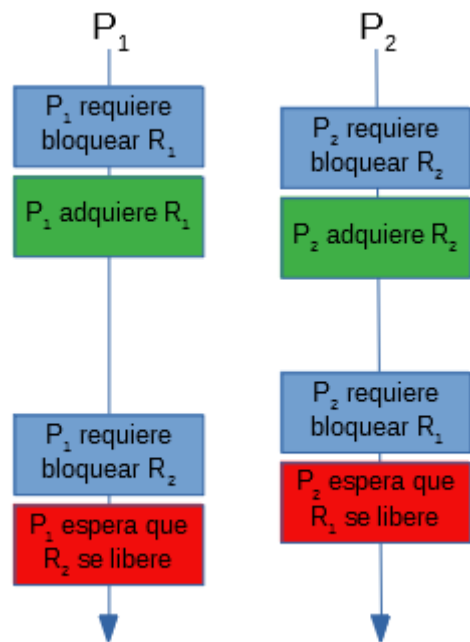
Los bloqueos exclusivos pueden utilizarse para controlar, por ejemplo, el uso de impresoras. Para el acceso a dispositivos de almacenamiento (memorias, discos, etc.), los bloqueos exclusivos se utilizan para proteger operaciones de escritura, mientras que los bloqueos compartidos se utilizan para proteger 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 esta 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 P₁ requiere bloquear el recurso R₁, debido a que R₁ está desbloqueado el procesador P₁ adquiere el recurso R₁. De la misma manera el procesador P₂ requiere bloquear el recurso R₂, debido a que R₂ está desbloqueado el procesador P₂ adquiere el recurso R₂.

Cuando el procesador P₁ requiere bloquear el recurso R₂, no puede hacerlo ya que el procesador P₂ lo tiene bloqueado, por tanto queda esperando a que R₂ se libere. Así mismo, cuando el procesador P₂ requiere bloquear el recurso R₁, no puede hacerlo ya que el procesador P₁ lo tiene bloqueado. Por lo tanto, ambos procesadores quedan bloqueados permanentemente.

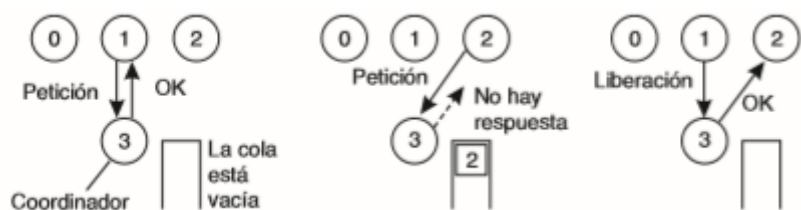
Para evitar que los procesos se bloqueen, los manejadores de bases de datos (p.e. MySQL) detectan el inter-bloqueo como un error, de manera que los programadores puedan controlar la situación.

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.

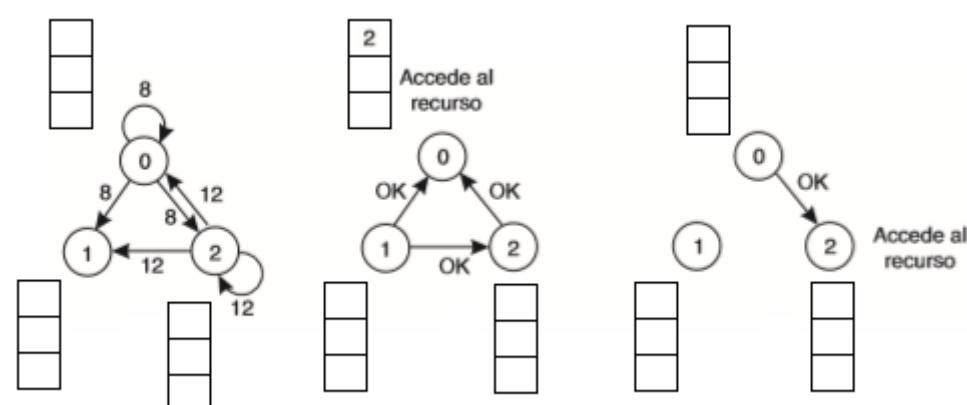
Ricart y Agrawala (1981) desarrollaron el siguiente algoritmo distribuido para exclusión mutua, el cual permite controlar el acceso a múltiples recursos:

- 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 envió 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)

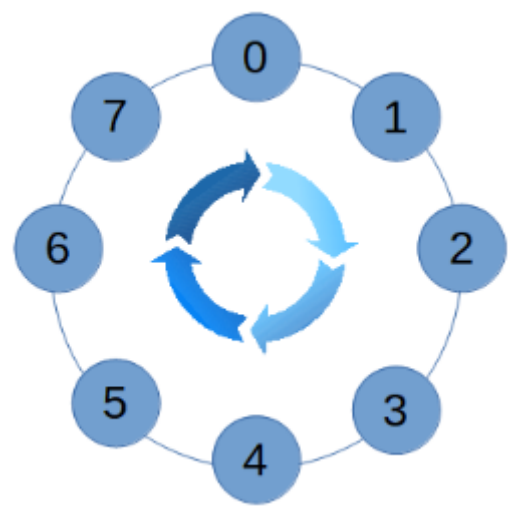
El algoritmo de token en anillo permite implementar la exclusión mutua en un sistema distribuido de manera muy simple, sin embargo tiene la desventaja de que requiere tener una conexión estable entre los nodos, por lo tanto, este algoritmo generalmente se implementa utilizando conexiones físicas.

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

1. Considere el ejemplo que vimos sobre el algoritmo de Lamport. Suponiendo que los relojes lógicos de los tres procesadores inician en cero, y el reloj lógico del procesador P₁ se incrementa en 1, el reloj lógico del procesador P₂ se incrementa en 5 y el reloj lógico del procesador P₃ se incrementa en 10.

Suponga que se envían los siguientes mensajes sin sincronizar los relojes lógicos:

- El procesador P₁ envía el mensaje m₁ al tiempo 1, y el procesador P₂ lo recibe al tiempo 10.
- El procesador P₂ envía el mensaje m₂ al tiempo 15, y el procesador P₃ lo recibe al tiempo 40.
- El procesador P₃ envía el mensaje m₃ al tiempo 60, y el procesador P₂ lo recibe al tiempo 35.
- El procesador P₂ envía el mensaje m₄ al tiempo 40, y el procesador P₁ lo recibe al tiempo 9.

P1	P2	P3
0	0	0
1	5	10
2	10	20
3	15	30
4	20	40
5	25	50
6	30	60
7	35	70
8	40	80
9	45	90
10	50	100
11	55	110

Si aplica el algoritmo de Lamport para sincronizar los relojes lógicos ¿Qué tiempos lógicos tendrán los procesadores P1 y P2 cuando el procesador P3 tenga el tiempo 110?

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

2.1 ¿Este algoritmo puede producir inanición?

2.2 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 - 22/10/2020

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

En el artículo [Elections in a Distributed Computing System](#) (1982), Héctor Garcia-Molina propone un algoritmo de elección para sistemas distribuidos llamado algoritmo del abusón o *bully*.

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:

1. P envía un mensaje de elección a los nodos con mayor número de nodo.
2. Si ningún nodo responde, P se convierte en el coordinador. Entonces P envía un mensaje de coordinador a todos los nodos.
3. 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.

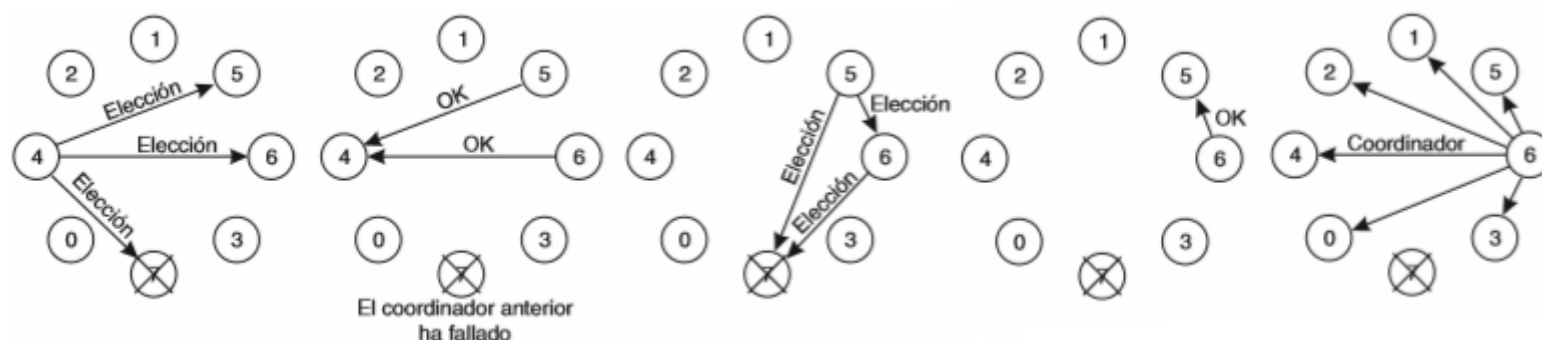
Veamos un ejemplo. Supongamos que tenemos ocho nodos, numerados del 0 al 7.

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.

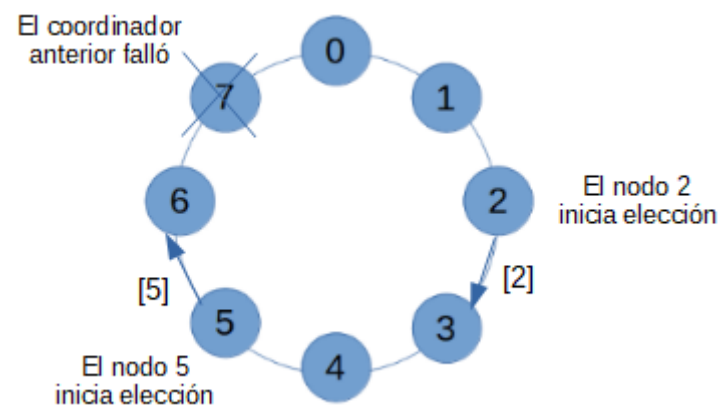
Cuando algún nodo P_n se da cuenta que el coordinador no responde, inicia un proceso de elección:

1. 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.
2. Cuando 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.

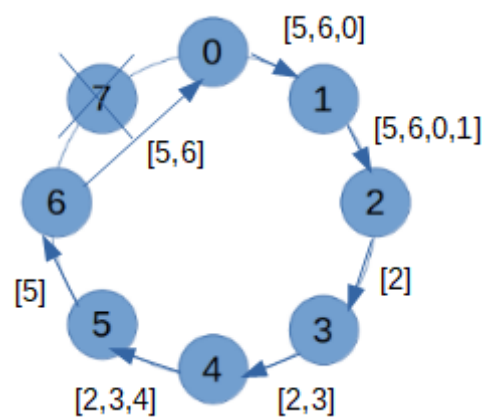
Supongamos que tenemos ocho nodos conectados en una topología de anillo. El nodo 7 es el coordinador actual, pero falló. Los nodos 2 y 5 se comunican con el coordinador, pero éste no responde, por tanto ambos

odos inician un proceso de elecci3n.

El nodo 2 envía un mensaje de elecci3n al nodo 3, incluyendo en el mensaje su n3mero de nodo. El nodo 5 envía un mensaje de elecci3n al nodo 6 incluyendo su n3mero de nodo.

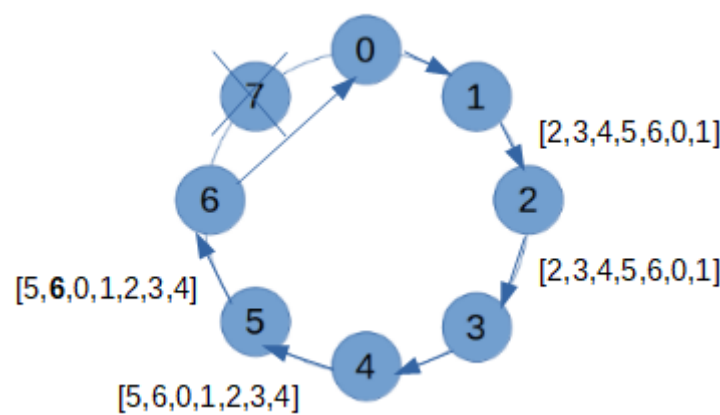


El paso 1 del algoritmo se repite, por tanto cada nodo envía un mensaje de elecci3n al nodo siguiente incluyendo su n3mero de nodo en el mensaje.

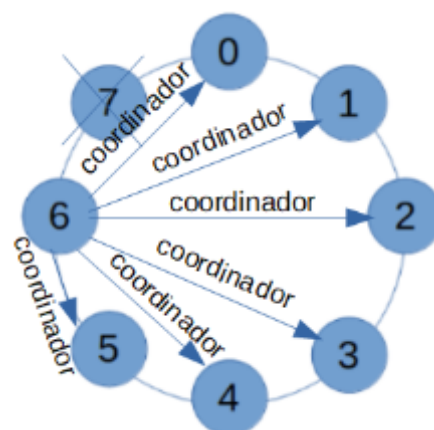


El nodo 2 recibe el mensaje [2,3,4,5,6,0,1], debido a que el nodo 2 no es el mayor nodo en el mensaje, entonces el nodo 2 re-envía el mensaje al nodo 3.

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 abus3n, suponga que tanto el nodo 6 como el nodo 7 no responden, si el nodo 0 y el nodo 1 inician un proceso de elecci3n ¿Qu3 nodo se erigar1 coordinador?

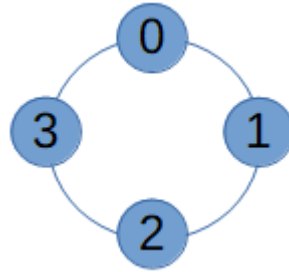
2. Considere el ejemplo que vimos para el algoritmo de elección en anillo, suponga que tanto el nodo 6 como el nodo 7 no responden, si el nodo 0 y el nodo 1 inician un proceso de elección ¿Qué nodo se erigirá coordinador?



Tarea 4. Implementación de un token-ring



Desarrollar un programa en Java, el cual implementará un token que pasará de un nodo a otro nodo, en una topología lógica de anillo. El anillo constará de cuatro nodos, del 0 al 3.



El token será un número entero de 8 bytes. El nodo 0 inicializará el token con cero.

El nodo 0 enviará el token al nodo 1, entonces el nodo 1 incrementará el token y lo enviará al nodo 2. El nodo 2 incrementará el token y lo enviará al nodo 3. El nodo 3 incrementará el token y lo enviará al nodo 0.

Cada nodo deberá desplegar el valor del token cada vez que lo recibe.

Consideremos el siguiente programa (notar que este programa es un cliente y un servidor):

```

class Token
{
    static DataInputStream entrada;
    static DataOutputStream salida;
    static boolean primera_vez = true;
    static String ip;
    static long token = 0;
    static int nodo;

    static class Worker extends Thread
    {
        public void run()
        {
            Algoritmo 1
        }
    }

    public static void main(String[] args) throws Exception
    {
        if (args.length != 2)
        {
            System.err.println("Se debe pasar como parametros el numero de nodo y la IP del siguiente nodo");
            System.exit(1);
        }

        nodo = Integer.valueOf(args[0]); // el primer parametro es el numero de nodo
        ip = args[1]; // el segundo parametro es la IP del siguiente nodo en el anillo

        Algoritmo 2
    }
}
  
```

Se deberá implementar los siguientes algoritmos:

Algoritmo 1

1. Declarar la variable **servidor** de tipo ServerSocket,
2. Asignar a la variable **servidor** el objeto: new ServerSocket(50000)
3. Declarar la variable **conexion** de tipo Socket.
4. Asignar a la variable **conexion** el objeto servidor.accept().
5. Asignar a la variable **entrada** el objeto new DataInputStream(conexion.getInputStream()).

Algoritmo 2

1. Declarar la variable **w** de tipo Worker.
2. Asignar a la variable **w** el objeto new Worker().
3. Invocar el método w.start().
4. Declarar la variable **conexion** de tipo Socket. Asignar null a la variable **conexion**.
5. En un ciclo:
 - 5.1 En un bloque try:
 - 5.1.1 Asignar a la variable **conexion** el objeto Socket(ip,50000).
 - 5.1.2 Ejecutar break para salir del ciclo.
 - 5.2 En el bloque catch:
 - 5.2.1 Invocar el método Thread.sleep(500).
6. Asignar a la variable **salida** el objeto new DataOutputStream(conexion.getOutputStream()).
7. Invocar el método w.join().
8. En un ciclo:
 - 8.1 Si la variable **nodo** es cero:
 - 8.1.1 Si la variable **primera_vez** es true:
 - 8.1.1.1 Asignar false a la variable **primera_vez**.
 - 8.1.2 Si la variable **primera_vez** es false:
 - 8.1.2.1 Asignar a la variable **token** el resultado del método entrada.readLong().
 - 8.2 Si la variable **nodo** no es cero:
 - 8.2.1 Asignar a la variable **token** el resultado del método entrada.readLong().
 - 8.3 Incrementar la variable **token**.
 - 8.4 Desplegar el valor de la variable **token**.
 - 8.5 Invocar el método salida.writeLong(token).

El algoritmo 1 debe implementarse dentro de un bloque try.

Para cada nodo se deberá crear una **máquina virtual en la nube**.

Recuerden que deben **eliminar las máquinas virtuales** cuando no las usen más, con la finalidad de ahorrar el saldo de sus cuentas de Azure.

Para que las máquinas virtuales puedan recibir conexiones a través del puerto 50000, se deberá abrir este puerto en cada máquina virtual.

Para abrir el puerto 50000 en una máquina virtual:

1. Entrar al portal de Azure
2. Seleccionar "Maquinas virtuales".
3. Seleccionar la máquina virtual.
4. Dar click en "Redes".
5. Dar click en el botón "Agregar regla de puerto de entrada".
6. En el campo "Intervalos de puertos de destino" ingresar: 50000
7. Seleccionar el protocolo: TCP
8. En el campo "Nombre" ingrear: Puerto_50000

Se deberá subir a la plataforma un archivo ZIP que contenga el código fuente del programa desarrollado y un documento PDF con las capturas de pantalla. El archivo PDF deberá incluir una descripción de cada captura de pantalla.

Valor de la tarea: 20% (1 puntos de la primera evaluación parcial)

Clase del día - 26/10/2020



En la clase de hoy vamos a jugar un kahoot en la modalidad de "challenge".

Para jugar el kahoot deberán ingresar al siguiente enlace:

[Desarrollo de Sistemas Distribuidos - Sincronización de relojes físicos](#)

Es necesario que los alumnos y alumnas ingresen su "nickname" como su nombre y apellido (por ejemplo JuanLopez), de manera que sea posible identificar a los ganadores de puntos extra.

La hora límite para jugar estos kahoots es 11:00 PM del 26 de octubre.

Clase del día - 28/10/2020



En la clase de hoy vamos a jugar dos kahoots en la modalidad de "challenge".

Para jugar los kahoots deberán ingresar a los siguientes enlaces:

[Desarrollo de Sistemas Distribuidos - Sincronización de relojes lógicos - Exclusión mutua](#)

[Desarrollo de Sistemas Distribuidos - Elección](#)

Es necesario que los alumnos y alumnas ingresen su "nickname" como su nombre y apellido (por ejemplo JuanLopez), de manera que sea posible identificar a los ganadores de puntos extra.

La hora límite para jugar estos kahoots es 11:00 PM del 28 de octubre.

Clase del día - 04/11/2020



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

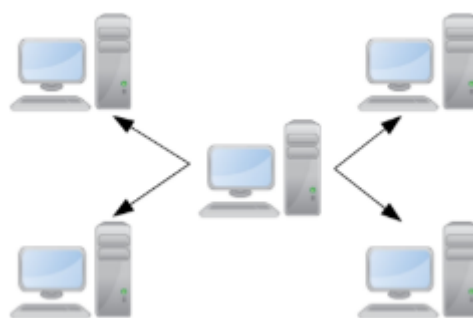
Tipos de comunicaciones

Los tipos de comunicaciones entre computadoras son los siguientes:

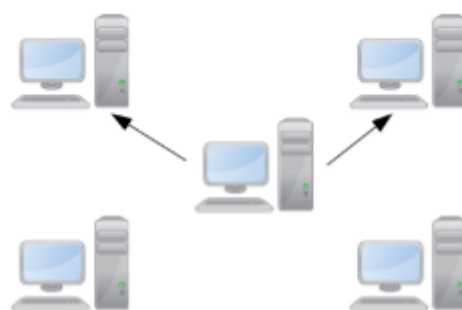
Unicast. El unicast es una comunicación punto a punto dónde una computadora envía mensajes a otra computadora.



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 $1 - 1/(24 \times 60 \times 60) = 99.998\%$, 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 cuando produce respuestas arbitrarias, en cualquier momento y con tiempos de respuesta 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.

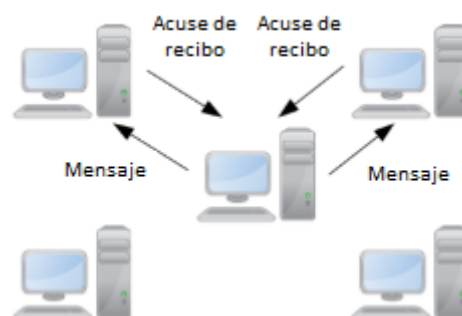
Definimos la comunicación multicast confiable como los mecanismos que garantizan que todos los miembros de un grupo reciben los mensajes transmitidos, sin importar el orden en que reciben los mensajes.

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 conectados. 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.

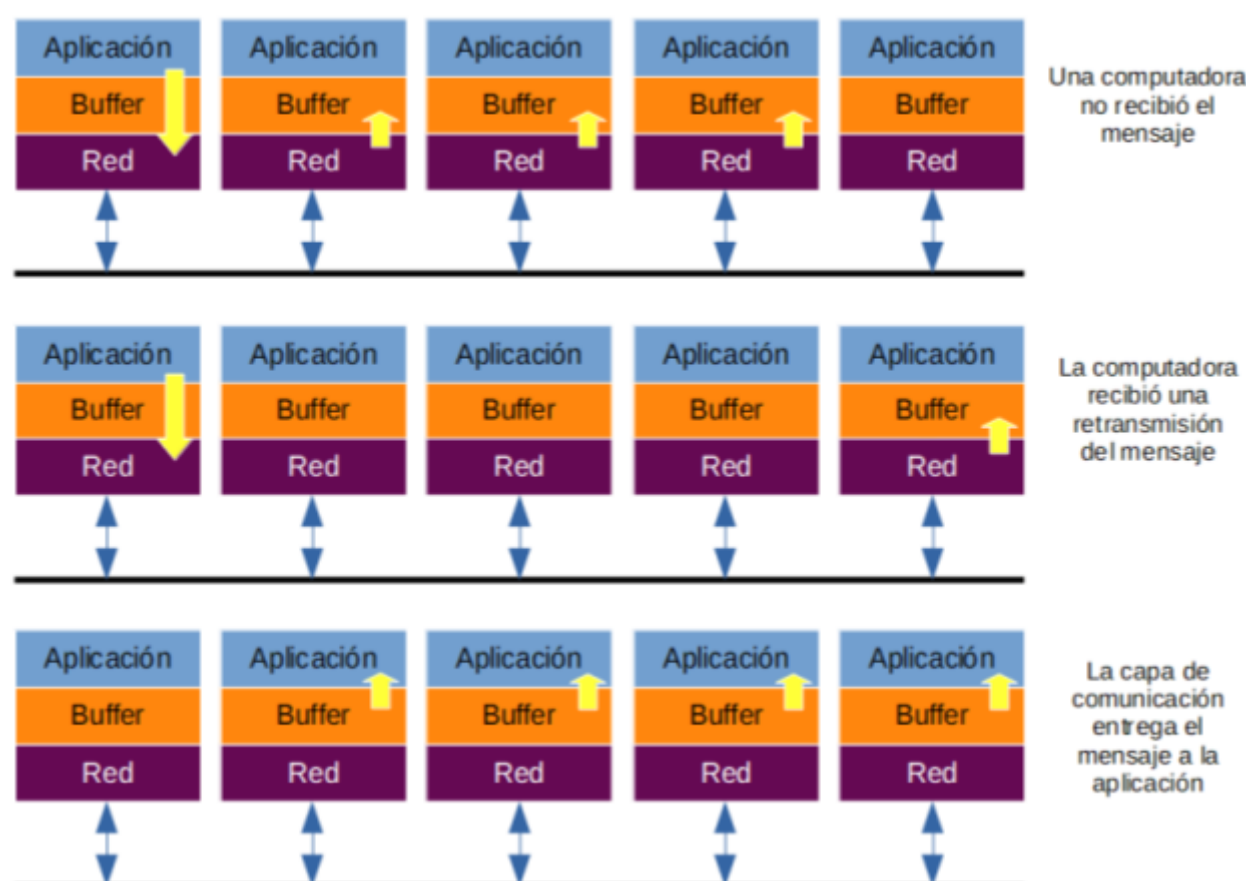
Por ejemplo, si un archivo es replicado en un grupo de computadoras, los cambios que se realizan al archivo deben ser replicados en **todas** las computadoras que forman parte del grupo. Si bien la consistencia del archivo es un requerimiento no funcional del sistema de archivos distribuido, este requerimiento 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.

ServidorMulticast.java

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":

```
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 `ByteBuffer` 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);
```

ClienteMulticast.java

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`:

```
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`:

```
socket.joinGroup(grupo);
```

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();
```

Actividades individuales a realizar

1. Compile los programas [ServidorMulticast.java](#) y [ClienteMulticast.java](#)
2. Ejecute el programa [ClienteMulticast.java](#) en tres ventanas de comandos de Windows (o terminales de Linux) y ejecute el programa [ServidorMulticast.java](#) en otra ventana de comandos de Windows (o terminal de Linux). Notar que primero se debe ejecutar los clientes y después se ejecuta el servidor.

 [ServidorMulticast.java](#)



 [ClienteMulticast.java](#)



Clase del día - 05/11/2020

En la clase de hoy vamos a jugar un kahoot en la modalidad de "challenge".

Para jugar el kahoot deberán ingresar al siguiente enlace:

[Desarrollo de Sistemas Distribuidos - Comunicación en grupo confiable](#)

Es necesario que los alumnos y alumnas ingresen su "nickname" como su nombre y apellido (por ejemplo JuanLopez), de manera que sea posible identificar a los ganadores de puntos extra.

La hora límite para jugar este kahoot es 11:00 PM del 5 de noviembre.



Tarea 5. Chat multicast



Desarrollar un programa en Java que implemente un chat utilizando comunicación multicast mediante datagramas, tal como vimos en clase.

Se **deberá** ejecutar el programa en cuatro ventanas de comandos de Windows o cuatro terminales de Linux. Solo se admitirá la tarea si se trata de un programa en modo consola de caracteres (no se admitirá el programa en modo gráfico).

Se **deberá** pasar como parámetro al programa el nombre del usuario que va escribir en el chat. Para demostrar el programa se **deberá** utilizar los siguientes usuarios: donald, hugo, paco y luis (no usar otros usuarios).

El programa **deberá** utilizar las funciones `envia_mensaje()` y `recibe_mensaje()` que vimos en clase.

El funcionamiento general del programa es el siguiente:

- El programa creará un thread que actuará como cliente multicast, el cual recibirá los mensajes del resto de los nodos. Cada mensaje recibido será desplegado en la pantalla. Notar que el thread desplegará el mensaje que envía el mismo nodo.
- En el método `main()`, dentro de un ciclo infinito se desplegará el siguiente prompt: "**Escribe el mensaje:** " (sin las comillas), se leerá una string utilizando el método `readLine()` de la clase `BufferedReader`. Entonces se **deberá** enviar el mensaje a los nodos que pertenecen al grupo identificado por la IP 230.0.0.0 a través del puerto 50000. El mensaje **deberá** tener la siguiente forma: *nombre_usuario:mensaje_ingresado*, donde *nombre_usuario* es el nombre del usuario que pasó como parámetro al programa (donald, hugo, paco o luis) y *mensaje_ingresado* el mensaje que el usuario ingresó por el teclado.

Se **deberá** completar el siguiente programa:

```
class Chat
{
    static class Worker extends Thread
    {
        public void run()
        {
            // En un ciclo infinito se recibirán los mensajes enviados al grupo
            // 230.0.0.0 a través del puerto 50000 y se desplegarán en la pantalla.
        }
    }
    public static void main(String[] args) throws Exception
    {
        Worker w = new Worker();
        w.start();

        String nombre = args[0];
        BufferedReader b = new BufferedReader(new InputStreamReader(System.in));

        // En un ciclo infinito se leerá los mensajes del teclado y se enviarán
        // al grupo 230.0.0.0 a través del puerto 50000.
    }
}
```

Para probar el programa, se **deberá** ejecutar la siguiente conversación (en el nodo 0 escribe donald, en el nodo 1 escribe hugo, en el nodo 2 escribe paco y en el nodo 3 escribe luis):

-donald escribe: hola
-hugo escribe: hola donald
-paco escribe: hola donald
-luis escribe: hola donald
-donald escribe: ¿alguien sabe dónde será la fiesta el sábado?
-paco escribe: será en la casa de tío mac pato
-hugo escribe: ¿a qué hora?
-luis escribe: a las 8 PM
-donald escribe: adios
-hugo escribe: adios donald

Se deberá subir a la plataforma un archivo texto con el código fuente del programa desarrollado y un reporte de la tarea en formato PDF con portada, desarrollo y conclusiones como mínimo. El archivo PDF deberá incluir las capturas de pantalla de la compilación y ejecución del programa así mismo, se deberá incluir una descripción de cada pantalla.

No se admitirá la tarea si se envían archivos en formato RAR o en formato WORD.

Nota importante. "**deberá**" significa que se trata de un requerimiento que no es opcional.

Valor de la tarea: 20% (1 puntos de la segunda evaluación parcial)

3. Sistemas basados en objetos distribuidos



Clase del día - 09/11/2020

La clase de hoy vamos a iniciar con el tema Sistemas basados en objetos distribuidos.

Paradigma de paso de mensajes

Hasta ahora hemos desarrollado programas distribuidos utilizando paso de mensajes.

El paradigma de **paso de mensajes** es el modelo natural para el desarrollo de sistemas distribuidos, ya que reproduce la comunicación entre las personas.

En el paradigma de paso de mensajes, las computadoras comparten los datos utilizando mensajes. El programador debe serializar los datos antes de enviarlos, y des-serializar los datos después de recibirlos.

El desarrollo de sistemas basados en paso de mensajes es complejo debido a que el programador debe controlar el intercambio de los mensajes, además de desarrollar la funcionalidad propia del sistema.

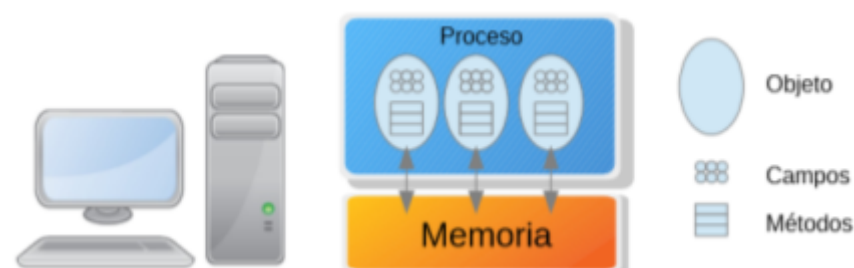
El paradigma de paso de mensajes es **orientado a datos**.

Objetos locales

Un objeto encapsula variables (campos) y funciones (métodos). Las variables guardan el estado del objeto y los métodos permiten modificar y acceder el estado del objeto.

Un objeto local es aquel cuyos métodos son invocados por un proceso local, es decir, un proceso que ejecuta en la misma computadora dónde reside el objeto.

Los objetos locales comparten el espacio de direcciones, en otras palabras, los objetos locales son objetos que residen en la misma memoria.



Objetos remotos

Un objeto remoto es aquel cuyos métodos son invocados por procesos remotos, es decir, procesos que ejecutan en una computadora remota conectada mediante una red.

Los objetos que se encuentran en diferentes computadoras no comparten el espacio de direcciones, por tanto, solo comparten valores pero no referencias.

La siguiente figura muestra un proceso cliente y un proceso servidor que ejecutan en diferentes computadoras. En el proceso cliente un método local invoca un método remoto, el cual forma parte de un objeto contenido en el proceso servidor.

En este caso, la invocación de los métodos remotos se realiza mediante una capa llamada RMI (*Remote Method Invocation*).



Paradigma de objetos distribuidos

El paradigma de objetos distribuido combina objetos locales y objetos remotos. La ventaja que tiene, comparado con el paradigma de paso de mensajes, es que el paradigma de objetos distribuidos representa una abstracción sobre el paso de mensajes, por tanto el programador no debe preocuparse por controlar el paso de mensajes entre los nodos.

El paradigma de objetos distribuidos es **orientado a la acción**, ya que se basa en la acción que realiza el método remoto invocado.

Remote Method Invocation

En un sistema que utiliza RMI existe un proceso llamado *registry* el cual hace las funciones de servidor de nombres.

En cada nodo, hay un proceso servidor el cual registra en el servidor de nombres los objetos que exportará. Cada objeto exportado por el servidor será identificado mediante una URL.

Para acceder a un objeto remoto, el proceso cliente consulta el servidor de nombres utilizando la URL, si el objeto es encontrado, entonces el servidor de nombres regresa al cliente una referencia que apunta al objeto remoto. Entonces el proceso cliente utiliza la referencia para invocar los métodos del objeto remoto, los cuales se ejecutan en el servidor.

El paso de parámetros y regreso de resultado es manejado automáticamente por la capa RMI.

Java RMI

Java RMI es un API que implementa la invocación de métodos remotos. JDK incluye un servidor de nombres llamado **rmiregistry**, esta aplicación se encuentra en el directorio bin del JDK

¿Cómo usar Java RMI?

Para utilizar Java RMI se debe seguir los siguientes pasos:

1. Para cada objeto remoto se debe crear una interface **I** que defina el prototipo de cada método a exportar. Es necesario declarar que los métodos remotos pueden producir la excepción `java.rmi.RemoteException`. La interface **I** debe heredar de `java.rmi.Remote`.
2. El código de los métodos remotos se debe escribir en una clase **C** que implemente la interface **I**. La clase **C** debe ser una subclase de `java.rmi.server.UnicastRemoteObject`. El constructor default de la clase **C** debe invocar el constructor de la superclase. Es necesario declarar que los métodos remotos pueden producir la excepción `java.rmi.RemoteException`.
3. El proceso servidor deberá registrar la clase **C** invocando el método `bind()` o el método `rebind()` de la clase `java.rmi.Naming`. A los métodos `bind()` y `rebind()` se les pasa como parámetros la URL correspondiente al objeto remoto y una instancia de la clase **C**. La URL tiene la siguiente forma: **rmi://ip:puerto/nombre**, donde *ip* es la dirección IP de la computadora dónde ejecuta el programa **rmiregistry**, *puerto* es el número de puerto utilizado por **rmiregistry** (se puede omitir si **rmiregistry** utiliza el puerto default 1099) y *nombre* es el nombre con el que identificaremos el objeto.
4. El proceso cliente deber invocar el método `lookup()` de la clase `java.rmi.Naming` para obtener una referencia al objeto remoto. El método `lookup()` regresa una instancia de la clase `Remote`, la cual se debe convertir al tipo de la interface **I** mediante casting. Utilizando la referencia, el proceso cliente invocará los métodos remotos de la clase **C**.

Por razones de seguridad, la aplicación rmiregistry se debe ejecutar en la misma computadora dónde ejecuta el servidor.

Por default la aplicación rmiregistry utiliza el puerto 1099, si se utiliza otro puerto, se deberá pasar el número de puerto como argumento al ejecutar rmiregistry.

Se puede notar que el proceso servidor permanece en ejecución debido a que los métodos bind() y rebind() crean threads que no terminan.

Ejemplo de Java RMI

Como vimos anteriormente, para crear una aplicación que utilice Java RMI es necesario crear una interface, una clase, y dos programas (un cliente y un servidor).

En este caso, vamos a crear un objeto remoto que exportará los siguiente métodos:

- mayusculas(), recibe como parámetro una cadena de caracteres y regresa la misma cadena convertida a mayúsculas.
- suma(), recibe como parámetros dos enteros y regresa la suma.
- checksum(), recibe como parámetro una matriz de enteros y regresa la suma de todos los elementos de la matriz.

Primeramente creamos una interface que incluya los prototipos de los métodos a exportar:

```
public interface InterfaceRMI extends Remote
{
    public String mayusculas(String name) throws RemoteException;
    public int suma(int a,int b) throws RemoteException;
    public long checksum(int[][] m) throws RemoteException;
}
```

Ahora escribimos la clase **ClaseRMI** la cual va a contener el código de los métodos definidos en la interface **InterfaceRMI**. Notar que la clase **ClaseRMI** es subclase de UnicastRemoteObject e implementa la interface **InterfaceRMI**.

```
public class ClaseRMI extends UnicastRemoteObject implements InterfaceRMI
{
    // es necesario que el constructor ClaseRMI() invoque el constructor de la superclase
    public ClaseRMI() throws RemoteException
    {
        super( );
    }
    public String mayusculas(String s) throws RemoteException
    {
        return s.toUpperCase();
    }
    public int suma(int a,int b)
    {
        return a + b;
    }
    public long checksum(int[][] m) throws RemoteException
    {
        long s = 0;
        for (int i = 0; i < m.length; i++)
            for (int j = 0; j < m[0].length; j++)
                s += m[i][j];
        return s;
    }
}
```

La clase **ServidorRMI** registra en el rmiregistry una instancia de la clase **ClaseRMI** utilizando el método rebind().

```
public class ServidorRMI
{
    public static void main(String[] args) throws Exception
    {
        String url = "rmi://localhost/prueba";
        ClaseRMI obj = new ClaseRMI();

        // registra la instancia en el rmiregistry
        Naming.rebind(url,obj);
    }
}
```

El cliente **ClienteRMI** obtiene una referencia al objeto remoto utilizando el método lookup(), esta referencia es utilizada para invocar los métodos remotos.

```
public class ClienteRMI
{
    public static void main(String args[]) throws Exception
    {
        // en este caso el objeto remoto se llama "prueba", notar que se utiliza el puerto default 1099
        String url = "rmi://localhost/prueba";

        // obtiene una referencia que "apunta" al objeto remoto asociado a la URL
        InterfaceRMI r = (InterfaceRMI)Naming.lookup(url);

        System.out.println(r.mayusculas("hola"));
        System.out.println("suma=" + r.suma(10,20));

        int[][] m = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
        System.out.println("checksum=" + r.checksum(m));
    }
}
```

Actividades individuales a realizar

- 1. Compilar la interface InterfaceRMI.java, la clase ClaseRMI.java y los programas ClienteRMI.java y ServidorRMI.java.
- 2. En una ventana de comandos de Windows (o una terminal de Linux) ejecutar el programa **rmiregistry**.
- 3. En una ventana de comandos de Windows (o una terminal de Linux) ejecutar el programa ServidorRMI. Notar que el servidor queda en ejecución.
- 4. En una ventana de comandos de Windows (o una terminal de Linux) ejecutar el programa ClienteRMI. El cliente invoca el método lookup() para obtener del rmiregistry una referencia al objeto remoto, entonces invoca los métodos del objeto remoto los cuales se ejecutan en el servidor.

<

>

</>

InterfaceRMI.java

✓

</>

ClaseRMI.java

✓

</>

ServidorRMI.java

✓

</>

ClienteRMI.java

✓

✓

Clase del día – 11/11/2020

En la tarea 3 desarrollamos un programa que multiplica matrices cuadradas en forma distribuida usando paso de mensajes.

Como pudimos ver, la programación de un sistema distribuido utilizando el paso de mensajes es complicada, ya que se debe controlar explícitamente la serialización, el envío y la des-serialización de los datos, además de la lógica particular del sistema.

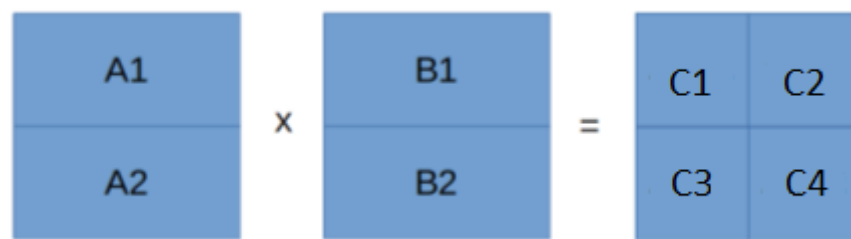
En la clase de hoy vamos a ver cómo desarrollar un programa distribuido que calcule el producto de matrices cuadradas utilizando Java RMI.

Partición de los datos

Dadas las matrices A y B de tamaño NxN, el producto C=AxB se obtiene dividiendo la matriz A en las matrices A1 y A2, y dividiendo la **transpuesta** de la matriz B en las matrices B1 y B2.

El tamaño de las matrices A1, A2, B1 y B2 es N/2 renglones y N columnas

Entonces, la matriz C se compone de las matrices C1, C2, C3 y C4, tal como se muestra en la siguiente figura:



Las matrices C1, C2, C3 y C4 se calculan de la siguiente manera:

$C1 = A1 \times B1$

$C2 = A1 \times B2$

$C3 = A2 \times B1$

$C4 = A2 \times B2$

Multiplicación de matrices utilizando objetos distribuidos

Para multiplicar matrices utilizando objetos distribuidos, escribiremos un servidor RMI que ejecute un método remoto llamado `multiplica_matrices()`, este método recibe como parámetros dos matrices de tamaño $(N/2) \times N$ y regresa como resultado una matriz cuadrada de tamaño $(N/2) \times (N/2)$.

Ahora debemos escribir un cliente RMI que inicialice las matrices, transponga la matriz B, invoque el método `multiplica_matrices()` y acomode las matrices C1, C2, C3 y C4 para formar la matriz C.

Consideremos el método `multiplica_matrices()` (este método ejecutará en el servidor RMI):

```
public int[][] multiplica_matrices(int[][] A,int[][] B) throws RemoteException
{
    int[][] C = new int[N/2][N/2];
    for (int i = 0; i < N/2; i++)
        for (int j = 0; j < N/2; j++)
            for (int k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[j][k];
    return C;
}
```

Cuando el método `multiplica_matrices()` se invoca localmente, recibe como parámetros las referencias a las matrices A y B y regresa una referencia a la matriz C.

Cuando el método es invocado en forma remota, entonces la capa RMI serializa las matrices A y B en el cliente y las des-serializa en el servidor. De la misma forma, la capa RMI serializa la matriz C en el servidor y la des-serializa en el cliente.

Ahora veamos el método `parte_matriz()` el cual utilizaremos para obtener las matrices A1, A2, B1 y B2:

```
static int[][] parte_matriz(int[][] A,int inicio)
{
    int[][] M = new int[N/2][N];
    for (int i = 0; i < N/2; i++)
        for (int j = 0; j < N; j++)
            M[i][j] = A[i + inicio][j];
    return M;
}
```

El método `parte_matriz()` recibe como parámetros la matriz a dividir y el renglón inicial. El método regresará una matriz de tamaño $(N/2) \times N$.

Entonces, podemos obtener las matrices A1, A2, B1 y B2 de la siguiente manera:

```
int[][] A1 = parte_matriz(A,0);
int[][] A2 = parte_matriz(A,N/2);
int[][] B1 = parte_matriz(B,0);
int[][] B2 = parte_matriz(B,N/2);
```

Dadas las matrices A1, A2, B1 y B2, podemos obtener las matrices C1, C2, C3 y C4 utilizando el método `multiplica_matrices()`:

```
int[][] C1 = multiplica_matrices(A1,B1);
int[][] C2 = multiplica_matrices(A1,B2);
int[][] C3 = multiplica_matrices(A2,B1);
int[][] C4 = multiplica_matrices(A2,B2);
```

Veamos ahora el método `acomoda_matriz()`, el cual permite construir la matriz C a partir de las matrices C1, C2, C3 y C4:

```
static void acomoda_matriz(int[][] C,int[][] A,int renglon,int columna)
{
    for (int i = 0; i < N/2; i++)
        for (int j = 0; j < N/2; j++)
            C[i + renglon][j + columna] = A[i][j];
}
```

El método `acomoda_matriz()` recibe como parámetros la matriz C, la sub-matriz a acomodar, y la posición (renglón,columna) en la matriz C donde se va a colocar la sub-matriz.

Finalmente para obtener la matriz C podemos hacer lo siguiente:

```
int[][] C = new int[N][N];
acomoda_matriz(C,C1,0,0);
acomoda_matriz(C,C2,0,N/2);
acomoda_matriz(C,C3,N/2,0);
acomoda_matriz(C,C4,N/2,N/2);
```

Actividades individuales a realizar

Desarrollar un programa en Java que multiplique dos matrices cuadradas de tamaño NxN en forma local, utilizando las funciones `parte_matriz()`, `multiplica_matrices()` y `acomoda_matriz()`.



Tarea 6. Multiplicación de matrices utilizando objetos distribuidos



Cada alumno deberá desarrollar un sistema que calcule el producto de dos matrices cuadradas utilizando Java RMI, tal como se explicó en clase.

Se deberá ejecutar dos casos:

- N=4, se deberá desplegar las matrices A, B y C y el checksum de la matriz C.
- N=500, deberá desplegar el checksum de la matriz C.

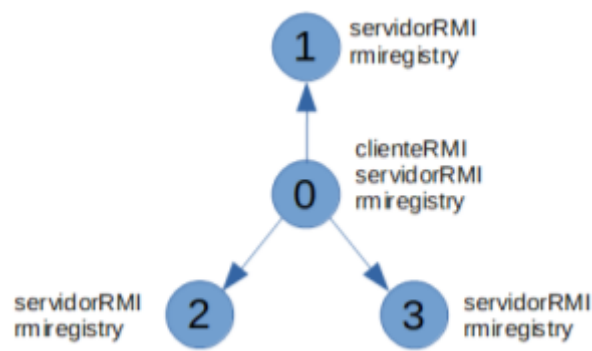
Los elementos de las matrices serán de tipo `int` y el checksum será de tipo `long`.

Se deberá inicializar las matrices A y B de la siguiente manera (notar que la inicialización es diferente a la que se realizó en la tarea 3):

```
A[i][j]=2 * i - j
B[i][j] = 2 * i + j
```

El servidor RMI ejecutará en cuatro máquinas virtuales (nodo 0, nodo 1, nodo 2 y nodo 3) con **Ubuntu** en Azure (no se admitirá esta tarea si se usan maquinas virtuales con Windows). El programa `rmiregistry` ejecutará en cada nodo donde ejecute el servidor RMI.

El cliente RMI, el cual ejecutará en el nodo 0, inicializará las matrices A y B, obtendrá la transpuesta de la matriz B, invocará el método remoto `multiplica_matrices()`, calculará el checksum de la matriz C, y en su caso (N=4) desplegará las matrices A, B y C.



Los alumnos tomarán como base el programa que desarrollaron previamente como actividad individual. Se deberá utilizar las funciones que vimos en clase: `parte_matriz()`, `multiplica_matrices()` y `acomoda_matriz()`.

Se deberá subir a la plataforma un archivo ZIP que contenga el código fuente de los programas desarrollados y el reporte de la tarea en formato PDF. El reporte de la tarea **deberá incluir** portada, desarrollo, captura de pantallas de la compilación y ejecución, y conclusiones.

No se admitirán tareas que incluyan archivos en formato RAR o WORD.

Valor de la tarea: 30% (1.5 puntos de la segunda evaluación parcial)

Clase del día - 12/11/2020

✓

En la clase de hoy vamos a jugar un kahoot en la modalidad de "challenge".

Para jugar el kahoot deberán ingresar al siguiente enlace:

Desarrollo de sistemas distribuidos - Objetos distribuidos con Java RMI

Es necesario que los alumnos y alumnas ingresen su "nickname" como su nombre y apellido (por ejemplo JuanLopez), de manera que sea posible identificar a los ganadores de puntos extra.

La hora límite para jugar este kahoot es 11:00 PM del 12 de noviembre.

Clase del día - 16/11/2020

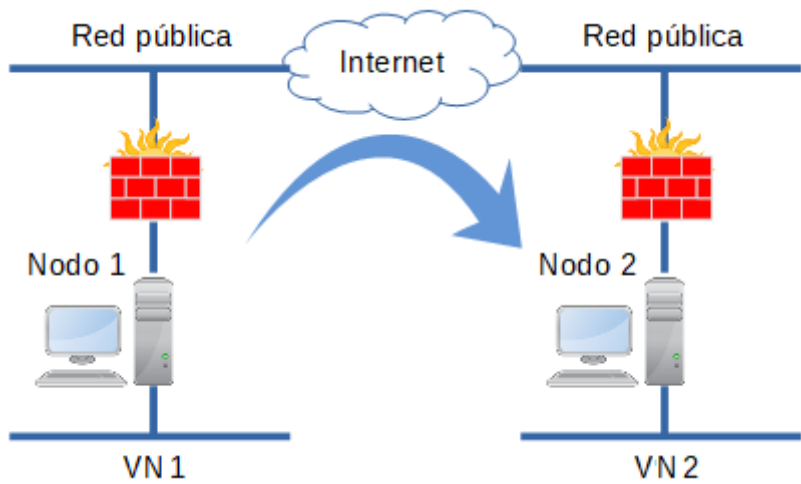
✓

El día de hoy vamos a explicar algunos conceptos que serán de utilidad para realizar la tarea 6.

Debido a que hoy es inhábil, ésta clase no contará como asistencia o inasistencia.

Red privada y red pública

Supongamos que creamos dos máquinas virtuales en Azure, cada máquina virtual en un grupo de recursos diferente, esto implica que cada máquina virtual estará conectada a una red virtual (VN) diferente, tal como se muestra en la siguiente figura:



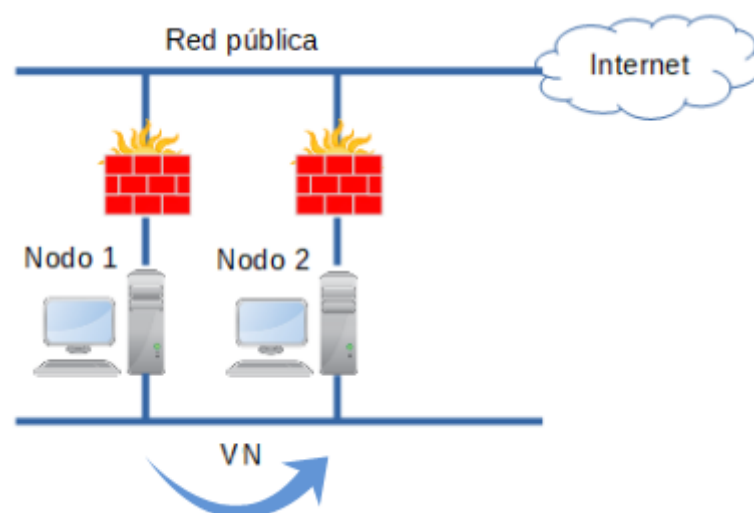
El firewall de una máquina virtual se puede configurar con reglas de entrada y reglas de salida, las reglas de entrada definen qué direcciones públicas y qué puertos se pueden conectar a la máquina virtual, mientras que las reglas de salida definen a qué direcciones públicas y a qué puertos se puede conectar la máquina virtual.

Por seguridad de la máquina virtual, las reglas de entrada suelen ser más restrictivas que las reglas de salida.

En este caso, para que el Nodo-1 se pueda conectar al Nodo-2, solo necesitamos crear una regla de entrada que permita que el Nodo-1 se conecte a través de un puerto específico.

Por otra parte, debido a que las redes virtuales VN1 y VN2 están desconectadas, no es posible conectar el Nodo-1 y el Nodo-2 utilizando las direcciones IP privadas.

Ahora supongamos que **creamos dos máquinas virtuales en el mismo grupo de recursos**. En este caso las dos máquinas virtuales comparten la misma red virtual (VN).



Si el Nodo-1 requiere comunicarse con el Nodo-2 no es necesario crear una regla en el firewall del Nodo-2 ya que ambos nodos están conectados a través de la misma red virtual.

Notar que la comunicación entre las máquinas virtuales mediante la VN se realiza utilizando las direcciones IP privadas de las máquinas virtuales.

¿Cómo ejecutar Java RMI en la nube?

Para registrar un objeto remoto en el rmiregistry utilizamos el método rebind().

Debido a que el servidor RMI debe ejecutar en la misma computadora donde ejecuta rmiregistry, la URL que pasa como parámetro al método rebind() deberá incluir el dominio **localhost**, tal como se muestra en el siguiente ejemplo:

```
public class ServidorRMI
{
    public static void main(String[] args) throws Exception
    {
        String url = "rmi://localhost/prueba";
        ClaseRMI obj = new ClaseRMI();

        // registra la instancia en el rmiregistry
        Naming.rebind(url,obj);
    }
}
```

Para que el cliente RMI pueda invocar los métodos del objeto remoto registrado por el servidor RMI, se debe obtener una referencia al objeto remoto utilizando el método lookup(). Entonces la URL que pasa como parámetro al método lookup() **deberá definir la IP privada** del nodo dónde ejecuta el servidor RMI.

Supongamos que la dirección IP privada donde ejecuta el servidor RMI, es **10.0.2.4**:

```
public class ClienteRMI
{
    public static void main(String args[]) throws Exception
    {
        // en este caso el objeto remoto se llama "prueba", notar que se utiliza el puerto default 1099
        String url = "rmi://10.0.2.4/prueba";
    }
}
```

```
// obtiene una referencia que "apunta" al objeto remoto asociado a la URL
InterfaceRMI r = (InterfaceRMI)Naming.lookup(url);

System.out.println(r.mayusculas("hola"));
System.out.println("suma=" + r.suma(10,20));

int[][] m = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
System.out.println("checksum=" + r.checksum(m));
}
}
```

Actividades individuales a realizar

1. Crear dos máquinas virtuales (Nodo-1 y Nodo-2) en el mismo grupo de recursos.
2. Compilar el programa ClienteRMI.java en el Nodo-1. Utilizar la IP privada del Nodo-2 en la URL.
3. Compilar el programa ServidorRMI.java en el Nodo-2. Utilizar localhost en la URL.
4. Ejecutar en el Nodo-2: rmiregistry&
5. Ejecutar en el Nodo-2: java ServidorRMI&
6. Ejecutar en el Nodo-1: java ClienteRMI&

IMPORTANTE: se debe eliminar las máquinas virtuales y todos sus recursos lo más pronto posible, ya que se deberá ahorrar saldo para poder realizar las tareas siguientes.



Clase del día – 18/11/2020

En la clase de hoy vamos a explicar cómo utilizar **JSON** para serializar y des-serializar objetos.

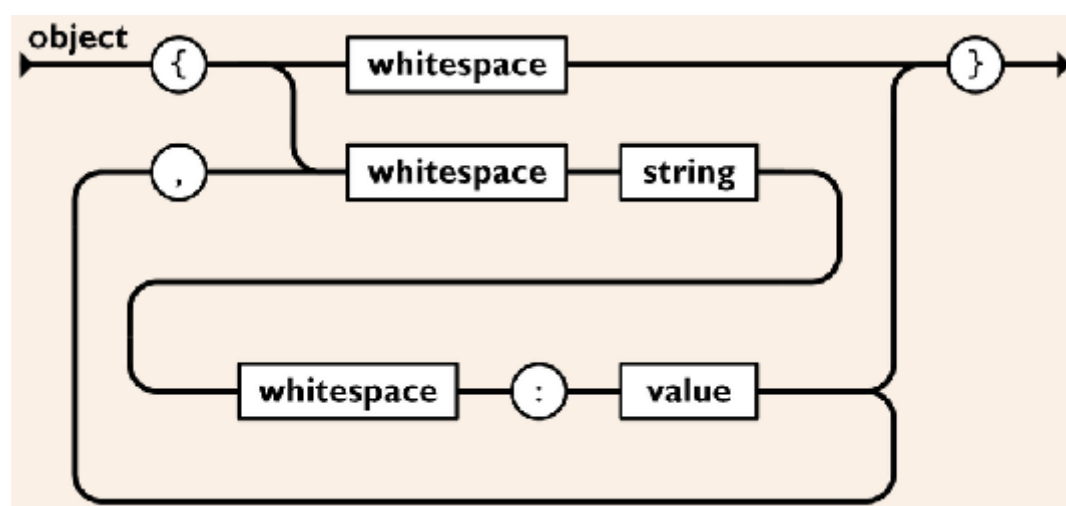
JSON (JavaScript Object Notation) es un formato texto para el intercambio de datos. JSON corresponde a la sintaxis utilizada en Javascript para escribir objetos.

JSON es un formato independiente del lenguaje de programación, de manera que es posible escribir fácilmente programas en cualquier lenguaje que creen mensajes en formato JSON así como programas que lean mensajes en formato JSON.

En JSON es posible crear dos estructuras: objetos y arreglos.

Un **objeto** es una colección no ordenada de parejas nombre:valor separadas por coma. Un objeto comienza con una llave que abre "{" y termina con una llave que cierra "}".

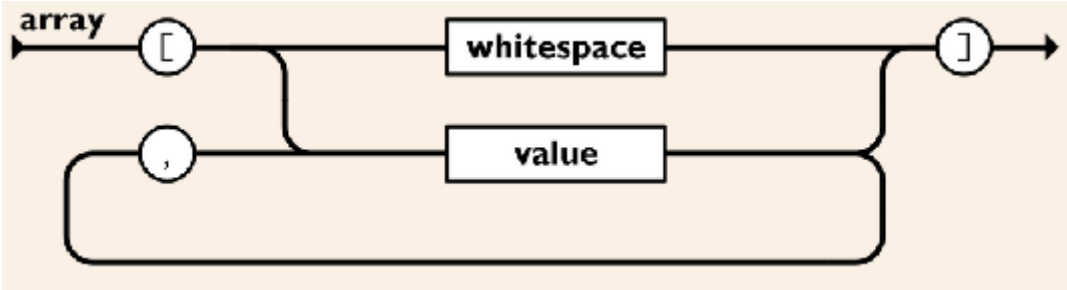
La sintaxis de un objeto es la siguiente:



Fuente: www.json.org

Un **arreglo** es una colección ordenada de valores separados por coma. Un arreglo comienza con un corchete que abre "[" y termina con un corchete que cierra "]".

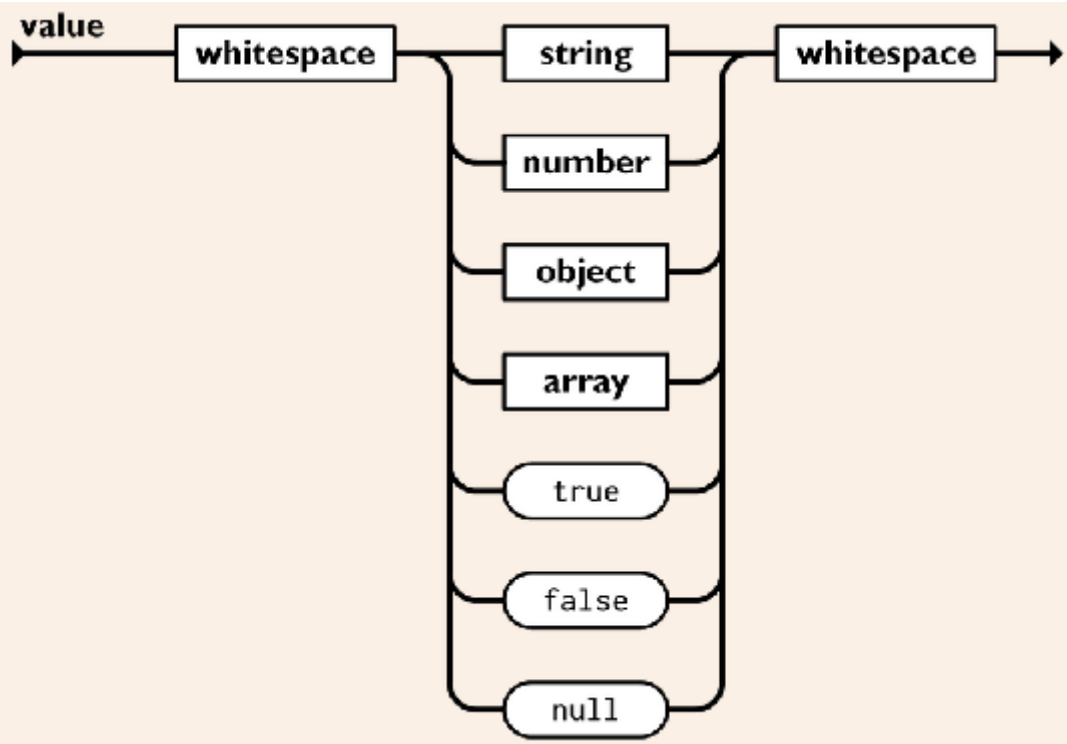
La sintaxis de un arreglo es la siguiente:



Fuente: www.json.org

Un **valor** puede ser una cadena de caracteres entre comillas, o un número, o un objeto, o un arreglo, o las constantes true, false o null.

La sintaxis de un valor es la siguiente:



Fuente: www.json.org

Una **cadena de caracteres** (string) es una secuencia de cero o más caracteres Unicode encerrados entre comillas.

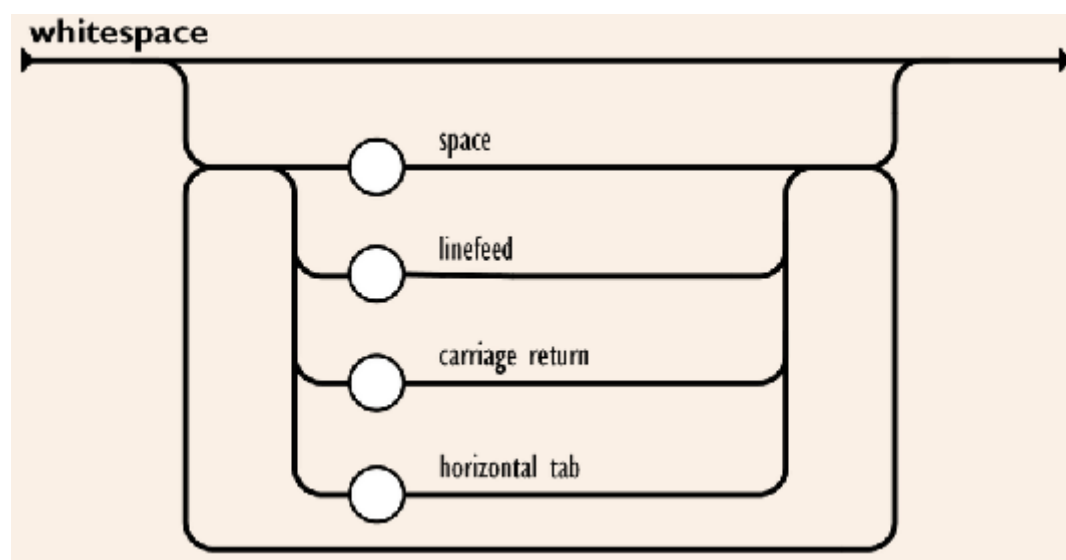
Una cadena de caracteres puede contener las siguientes secuencias de escape:

Secuencia de escape	Descripción

\"	comillas
\\	diagonal inversa
\	diagonal
\b	back space
\f	form feed
\n	line feed
\r	carriage return
\t	tabulador
\uxxxx	caracter unicode con código hexadecimal xxxx

Un **número** sigue la sintaxis de los números decimales en lenguaje C.

Un **whitespace** es un separador de tokens, de acuerdo a la siguiente sintaxis:



Fuente: www.json.org

Ahora veremos un ejemplo, utilizando GSON (implementación de JSON desarrollada por Google).

Se requiere descargar el archivo gson-2.8.6.jar de la siguiente URL:

<https://repo1.maven.org/maven2/com/google/code/gson/gson/2.8.6/gson-2.8.6.jar>

Descargar de la plataforma el programa [EjemploGSON.java](#) y colocarlo en el mismo directorio dónde está el archivo gson-2.8.6.jar que se descargó anteriormente.

Para compilar el programa [EjemploGSON.java](#) se debe ejecutar el siguiente comando:

```
javac -cp gson-2.8.6.jar EjemploGSON.java
```

Para ejecutar el programa en Windows:

```
java -cp gson-2.8.6.jar;. EjemploGSON
```

Para ejecutar el programa en Linux:

```
java -cp gson-2.8.6.jar:. EjemploGSON
```

Ahora se explicará como funciona este programa.

Primeramente se declaran los imports de las clases que se utilizarán: Gson, GsonBuilder y Timestamp.

Se define una clase Empleado que contiene los campos: nombre, edad, sueldo y fecha_ingreso. Por conveniencia se define un constructor para inicializar los campos al crear una instancia.

Notar que la fecha se maneja como fecha-hora debido a que en los sistemas globales (Internet) la fecha no indica qué sucede antes y que sucede después a nivel mundial; recordar lo que se vimos sobre tiempo UTC y tiempo local.

En la función main se crea un arreglo de 3 empleados, cada elemento se inicializa con una instancia de la clase Empleado, con diferentes datos.

Entonces se crea una instancia de la clase Gson. Notar que se utiliza el método setDateFormat para utilizar el formato de fecha ISO8601, el cual es el formato que utiliza Javascript para manejar fecha-hora.

Después se utiliza el método **toJson** de la clase Gson para serializar el arreglo de empleados. Esta clase produce la siguiente string:

```
[{"nombre":"Hugo","edad":20,"sueldo":1000.0,"fecha_ingreso":"2020-01-01T20:10:00.000"},
{"nombre":"Paco","edad":21,"sueldo":2000.0,"fecha_ingreso":"2019-10-01T10:15:00.000"},
{"nombre":"Luis","edad":22,"sueldo":3000.0,"fecha_ingreso":"2018-11-01T00:00:00.000"}]
```

Finalmente se utiliza el método **fromJson** de la clase Gson para deserializar la string anterior y producir un nuevo arreglo de empleados. Entonces se despliegan los datos de los empleados:

Hugo 20 1000.0 2020-01-01 20:10:00.0

Paco 21 2000.0 2019-10-01 10:15:00.0

Luis 22 3000.0 2018-11-01 00:00:00.0

Actividades individuales a realizar

1. Compilar y ejecutar el programa [EjemploGSON.java](#)
2. Agregar un nuevo campo a la clase Empleado, el campo "jefe" de tipo Empleado.
3. Modificar el constructor de la clase Empleado para incluir la inicialización del campo "jefe", tal como se muestra:

```
static class Empleado
{
    String nombre;
    int edad;
    float sueldo;
    Timestamp fecha_ingreso;
    Empleado jefe;
    Empleado(String nombre,int edad,float sueldo,Timestamp fecha_ingreso,Empleado jefe)
    {
        this.nombre = nombre;
        this.edad = edad;
        this.sueldo = sueldo;
        this.fecha_ingreso = fecha_ingreso;
        this.jefe = jefe != null ? jefe : this;
    }
}
```

4. Crear los tres empleados de manera que Hugo sea jefe de si mismo (Hugo es el jefe máximo), Hugo sea jefe de Paco, y Paco sea jefe de Luis, tal como se muestra:

```
Empleado[] e = new Empleado[3];
e[0] = new Empleado("Hugo",20,1000,Timestamp.valueOf("2020-01-01 20:10:00"),null);
e[1] = new Empleado("Paco",21,2000,Timestamp.valueOf("2019-10-01 10:15:00"),e[0]);
e[2] = new Empleado("Luis",22,3000,Timestamp.valueOf("2018-11-01 00:00:00"),e[1]);
```

5. Cambiar el despliegue del arreglo deserializado, de la siguiente manera:

```
for (int i = 0; i < v.length; i++)
    System.out.println(v[i].nombre + " " + v[i].edad + " " + v[i].sueldo + " " + v[i].fecha_ingreso + " jefe:" +
(v[i].jefe != null ? v[i].jefe.nombre : null));
```

6. Compilar y ejecutar el programa.
 - 6.1 ¿Qué despliega la serialización?
 - 6.2 Al serializar el empleado Hugo ¿Se muestra la auto-referencia que hace el empleado Hugo a sí mismo?
 - 6.3 ¿Hay alguna redundancia en los objetos serializados?
 - 6.4 ¿Qué despliega la deserialización?
 - 6.5 ¿Qué concluye sobre GSON y la forma en que serializa y deserializa las referencias a objetos?

7. Ver el video:

 [EjemploGSON.java](#)

Clase del día - 19/11/2020

En la clase de hoy vamos a jugar un kahoot en la modalidad de "challenge".

Para jugar el kahoot deberán ingresar al siguiente enlace:

[Desarrollo de Sistemas Distribuidos - JSON](#)

Es necesario que los alumnos y alumnas ingresen su "nickname" como su nombre y apellido (por ejemplo JuanLopez), de manera que sea posible identificar a los ganadores de puntos extra.

La hora límite para jugar este kahoot es 11:00 PM del 19 de noviembre.



Clase del día 23/11/2020

En la clase de hoy veremos los conceptos básicos de los servicios web (Web Services) así como los elementos de servicios web SOAP y REST.

Conceptos básicos de Servicios web

En el documento [Web Services Architecture](#) del World Wide Web Consortium (W2C) define un servicio web como:

“Un sistema de software diseñado para soportar la interacción interoperable de maquina-a-máquina sobre una red. Este cuenta con una interface descrita en un formato el cual puede ser procesado por una computadora (específicamente WSDL). Otros sistemas interactúan con el servicio web en una manera prescrita por su descripción usando mensajes SOAP, típicamente transportados usando HTTP con una serialización XML en conjunción con otros estándares relativos a la Web”.

Un servicio web es un concepto abstracto que debe ser implementado mediante un agente concreto.

Un **agente** es el software o hardware que envía y recibe mensajes. El **servicio** es el recurso caracterizado por un conjunto abstracto de la funcionalidad que se provee. Un servicio web no cambia aún cuando cambie el agente, es decir, la funcionalidad es independiente de la implementación de ésta.

El propósito de servicio web es proveer cierta funcionalidad a nombre de su propietario (una persona o una organización). La **entidad proveedora** es aquella persona u organización que provee un agente que implementa un determinado servicio.

Una **entidad solicitante** es una persona u organización que desea hacer uso del servicio mediante un **agente solicitante** (también llamado *solicitante del servicio*) que intercambia mensajes con el **agente proveedor** (también llamado *proveedor del servicio*).

En la mayoría de los casos el agente solicitante es el que inicia la comunicación con el agente proveedor, aunque no siempre es así, no obstante se sigue llamando agente solicitante aunque no sea el que inicia la comunicación.

La **semántica** de un servicio web es la expectativa compartida sobre el comportamiento del servicio, en particular el comportamiento en respuesta a los mensajes que recibe.

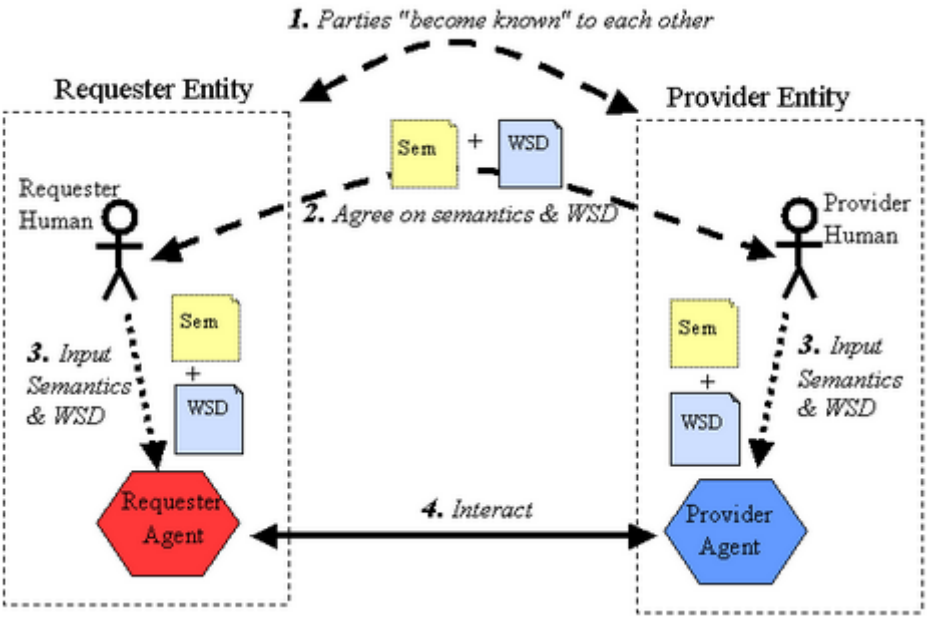
Se le llama **contrato** al acuerdo entre la entidad solicitante y la entidad proveedora. Un contrato puede ser explícito o implícito, escrito u oral, establecido entre las personas y/o las computadoras, legal o informal.

Hay dos tipos de contratos: 1) la *descripción del servicio* es el contrato que gobierna la mecánica de interacción con un servicio en particular y 2) la *semántica* del servicio es el contrato que gobierna el significado y propósito de la interacción. Sin embargo, puede haber contratos “híbridos” que incluyan elementos de descripción y elementos de semántica.

Participación en un servicio web

Una entidad solicitante puede participar de un servicio web de diferentes maneras. La siguiente figura muestra el proceso general de participación en un servicio web.

- 1. Las entidades solicitante y proveedora se conocen una a la otra, o por lo menos una conoce a la otra.
- 2. Las entidades acuerdan la descripción (WSD: Web Service Description) y semántica del servicio.
- 3. La descripción y la semántica son implementadas por el agente solicitante y el agente proveedor.
- 4. Los agentes solicitante y proveedor intercambian mensajes.



Fuente: Web Services Architecture, W3C

Servicios web basados en SOAP

SOAP (Simple Object Access Protocol) define un protocolo de RPC (Remote Procedure Call) basado en XML, para la interacción cliente-servidor a través de la red utilizando: 1) HTTP como la base de transporte, y 2) documentos XML para la codificación de requerimientos y respuestas.

SOAP permite la comunicación entre aplicaciones ejecutando en diferentes sistemas operativos, con diferentes tecnologías y lenguajes de programación.

Componentes de un mensaje SOAP

Un mensaje SOAP es un documento XML compuesto por los siguientes elementos:

- Envelope (sobre). Identifica el documento XML como un mensaje SOAP.
- Header (encabezado). Contiene información de encabezado.
- Body (cuerpo). Contiene información del requerimiento y la respuesta.
- Fault (falla). Contiene errores e información de estatus.

Para implementar servicios web en Java se puede utilizar la API JAX-WS.

Web Services Description Language (WSDL)

Un documento WSDL es un documento XML que contiene la descripción de un servicio web SOAP. Este especifica la localización del servicio y los métodos del servicio.

Un cliente puede hacer un requerimiento HTTP a un servicio web SOAP para obtener el WSDL que describe el servicio web.

Los elementos de un documento WSDL son los siguientes:

Elemento	Descripción
<types>	Define los tipos de dato usados por el servicio web
<message>	Define los elementos de datos para cada operación
<servicioportType>	Describe las operaciones que pueden ser ejecutadas y los mensajes involucrados
<binding>	Define el protocolo y el formato de los datos para cada portType

Un ejemplo de WSDL es el siguiente (fuente www.w3schools.com/xml/xml_wSDL.asp):

```

<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
<binding type="glossaryTerms" name="b1">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http" />
  <operation>
    <soap:operation soapAction="http://example.com/getTerm"/>
    <input> <soap:body use="literal"/> </input>
    <output> <soap:body use="literal"/> </output>
  </operation>
</binding>

```

El siguiente código, es un ejemplo de un servicio web escrito en Java utilizando el API JAX-WS:

```
package negocio;
```

```

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebService;
import javax.naming.InitialContext;
import javax.sql.DataSource;
import java.sql.Connection;

```

```
@WebService
```

```
public class ServicioSOAP
```

```
{
```

```
    static DataSource pool;
```

```
    static
```

```
    {
```

```
        pool = null;
```

```
        try
```

```
        {
```

```
            pool = (DataSource)new InitialContext().lookup("java:comp/env/jdbc/prueba");
```

```
        }
```

```
        catch (Exception e)
```

```
        {
```

```
            e.printStackTrace();
```

```
        }
```

```
    }
```

```
    @WebMethod
```

```
    public Integer suma(@WebParam(name = "a") Integer a, @WebParam(name = "b") Integer b) throws
```

```
Exception
```

```
    {
```

```
        return a + b;
```

```
    }
```

```
    @WebMethod
```

```
    public String mayusculas(@WebParam(name = "s") String s) throws Exception
```

```
    {
```

```
        return s.toUpperCase();
```

```
    }
```

```
    @WebMethod
```

```
    public void prueba_conexion_bd() throws Exception
```

```
{  
    Connection conexion = pool.getConnection();  
    conexion.close();  
}  
}
```

En este ejemplo podemos ver que la anotación `@WebService` define la clase correspondiente al servicio web. Cada operación del servicio web se implementa como un método de Java incluyendo la anotación `@WebMethod`. La anotación `@WebParam` se utiliza para definir los parámetros de cada operación del servicio web.

En este caso el servicio web accede a una base de datos llamada "prueba" mediante JDBC.

Servicios web estilo REST

REST define un conjunto de principios arquitectónicos para la creación de servicios web. REST fue presentado por Roy Fielding el año 2000 en su disertación doctoral "Architectural Styles and the Design of Network-based Software Architectures".

El diseño de servicios web estilo REST sigue cuatro principios:

- Utilizar métodos HTTP de forma explícita.
 - Un servicio web utiliza los métodos de HTTP para crear un recurso (POST), leer (GET), cambiar el estado o actualizar un recurso (PUT), y borrar un recurso (DELETE).
- Los servicios son sin estado (stateless).
 - Los clientes de servicios web estilo REST deben enviar peticiones completas e independientes, es decir, las peticiones deben incluir todos los datos que permitan completar el servicio, sin la necesidad de guardar un estado entre peticiones.
- Los URIs representan una estructura de directorios.
 - Los URIs (Uniform Resource Identifier) deben ser intuitivos y auto-explicados. Un URI es una jerarquía que corresponde a la estructura de los servicios web definidos en la empresa.
- Se transfiere XML, JSON o ambos.
 - Los recursos que provee un servicio web pueden ser documentos, imágenes, videos y en general objetos. La representación de objetos mediante XML o JSON es fácil e independiente de la plataforma.

Actividades individuales a realizar

Ver el video:

Clase del día - 25/11/2020



La clase de hoy vamos realizar una práctica la cual se entregará como tarea 7.

Veremos cómo crear un servicio web estilo REST utilizando el API de Java JAX-RS sobre el servidor de aplicaciones Tomcat.

Primeramente instalaremos Tomcat y las bibliotecas necesarias para la implementación de servicios web estilo REST los cuales podrán acceder una base de datos MySQL.

Instalación de Tomcat con soporte REST

1. Crear una máquina virtual con Ubuntu 18 con al menos 1GB de memoria RAM. Abrir el puerto 8080 para el protocolo TCP.

2. Instalar JDK8 ejecutando los siguientes comandos en la máquina virtual:

```
sudo apt update

sudo apt install openjdk-8-jdk-headless
```

3. Descargar la distribución binaria de Tomcat 8 de la siguiente URL (descargar la opción Core "zip"):
<https://tomcat.apache.org/download-80.cgi>

4. Copiar a la máquina virtual el archivo ZIP descargado anteriormente y desempacarlo utilizando el comando unzip.

5. Eliminar el directorio webapps el cual se encuentra dentro del directorio de Tomcat. Crear un nuevo directorio webapps y dentro de éste se deberá crear el directorio ROOT.

NOTA DE SEGURIDAD: Lo anterior se recomienda debido a que se han detectado vulnerabilidades en algunas aplicaciones que vienen con Tomcat, estas aplicaciones se encuentran originalmente instaladas en los directorios webapps y webapps/ROOT.

6. Descargar la biblioteca "Jersey" de la siguiente URL. Jersey es una implementación de JAX-RS lo cual permite ejecutar servicios web estilo REST sobre Tomcat:

<https://repo1.maven.org/maven2/org/glassfish/jersey/bundles/jaxrs-ri/2.24/jaxrs-ri-2.24.zip>

7. Copiar a la máquina virtual el archivo descargado anteriormente, desempacarlo y **copiar todos los archivos** con extensión ".jar" de **todos los directorios** desempacados, al directorio "lib" de Tomcat.

8. Borrar el archivo javax.servlet-api-3.0.1.jar del directorio "lib" de Tomcat (esto debe hacerse ya que existe una incompatibilidad entre Tomcat y Jersey 2).

9. Descargar el archivo gson-2.3.1.jar de la URL:

<https://repo1.maven.org/maven2/com/google/code/gson/gson/2.3.1/gson-2.3.1.jar>

10. Copiar el archivo gson-2.3.1.jar al directorio "lib" de Tomcat.

11. Ahora vamos a instalar el driver de JDBC para MySQL. Ingresar a la siguiente URL:

<https://dev.mysql.com/downloads/connector/j/>

Seleccionar "Platform independent" y descargar el archivo ZIP.

12. Copiar el archivo descargado a la máquina virtual, desempacarlo y copiar el archivo mysql-connector...jar al directorio "lib" de Tomcat.

Iniciar/detener el servidor Tomcat

1. Para iniciar el servidor Tomcat es **necesario** definir las siguientes variables de entorno:

```
export CATALINA_HOME=aquí va la ruta del directorio de Tomcat 8

export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
```

2. Iniciar la ejecución de Tomcat ejecutando el siguiente comando:

```
sh $CATALINA_HOME/bin/catalina.sh start
```

3. Para detener la ejecución de Tomcat se deberá ejecutar el siguiente comando:

```
sh $CATALINA_HOME/bin/catalina.sh stop
```

Notar que Tomcat se ejecuta sin permisos de administrador (no se usa "sudo"), lo cual es muy importante para prevenir que algún atacante pueda entrar a nuestro sistema con permisos de super-usuario.

Instalación de MySQL

1. Actualizar los paquetes en la máquina virtual ejecutando el siguiente comando:

```
sudo apt update
```

2. Instalar el paquete default de MySQL:

```
sudo apt install mysql-server
```

3. Ejecutar el script de seguridad:

```
sudo mysql_secure_installation
```

Press y|Y for Yes, any other key for No: **N**
New password: *contraseña-de-root-en-mysql*
Re-enter new password: *contraseña-de-root-en-mysql*
Remove anonymous users? (Press y|Y for Yes, any other key for No) : **Y**
Disallow root login remotely? (Press y|Y for Yes, any other key for No) : **Y**
Remove test database and access to it? (Press y|Y for Yes, any other key for No) : **Y**
Reload privilege tables now? (Press y|Y for Yes, any other key for No) : **Y**

4. Ejecutar el monitor de MySQL:

```
sudo mysql
```

5. Ejecutar el siguiente comando SQL para modificar la contraseña de root:

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'contraseña-de-root-en-mysql';
```

6. Actualizar los privilegios:

```
FLUSH PRIVILEGES;
```

7. Ejecutar el siguiente comando para salir del monitor de MySQL:

```
quit
```

Crear un usuario en MySQL

1. Ejecutar el monitor de MySQL:

```
mysql -u root -p
```

2. Crea el usuario "hugo":

```
create user hugo@localhost identified by 'contraseña-del-usuario-hugo';
```

3. Otorgar todos los permisos al usuario "hugo" sobre la base de datos "servicio_web":

```
grant all on servicio_web.* to hugo@localhost;
```

4. Ejecutar el siguiente comando para salir del monitor de MySQL:

```
quit
```

Crear la base de datos

1. Ejecutar el monitor de MySQL (notar que ahora se utiliza el usuario "hugo"):

```
mysql -u hugo -p
```

2. Crear la base de datos "servicio_web":

```
create database servicio_web;
```

3. Conectar a la base de datos creada anteriormente:

use servicio_web;

4. Crear las tablas "usuarios" y "fotos_usuarios", así mismo, se crea una regla de integridad referencial y un índice único:

```
create table usuarios
(
  id_usuario integer auto_increment primary key,
  email varchar(256) not null,
  nombre varchar(100) not null,
  apellido_paterno varchar(100) not null,
  apellido_materno varchar(100),
  fecha_nacimiento date not null,
  telefono varchar(20),
  genero char(1)
);
create table fotos_usuarios
(
  id_foto integer auto_increment primary key,
  foto longblob,
  id_usuario integer not null
);
alter table fotos_usuarios add foreign key (id_usuario) references usuarios(id_usuario);
create unique index usuarios_1 on usuarios(email);
```

5. Salir del monitor de MySQL:

```
quit
```

Compilar, empacar y desplegar el servicio web

1. Descargar de la plataforma y desempacar el archivo [Servicio.zip](#).

2. Definir la variable de ambiente CATALINA_HOME:

```
export CATALINA_HOME=aquí va la ruta completa del directorio de Tomcat 8
```

3. Cambiar al directorio dónde se desempacó el archivo [Servicio.zip](#) (en ese directorio se encuentra el directorio "negocio").

4. Compilar la clase Servicio.java:

```
javac -cp $CATALINA_HOME/lib/javax.ws.rs-api-2.0.1.jar:$CATALINA_HOME/lib/gson-2.3.1.jar:.
negocio/Servicio.java
```

5. Editar el archivo "context.xml" que está en el directorio "META-INF" y definir el username de la base de datos y el password correspondiente. El usuario "hugo" fue creado en el paso 2 de la sección **Crear un usuario en MySQL**.

6. Ejecutar los siguientes comandos para crear el servicio web para Tomcat (notar que los servicios web para Tomcat son archivos JAR con la extensión .war):

```
rm WEB-INF/classes/negocio/*
```

```
cp negocio/*.class WEB-INF/classes/negocio/.
```

```
jar cvf Servicio.war WEB-INF META-INF
```

7. Para desplegar (*deploy*) el servicio web, copiar el archivo **Servicio.war** al directorio "webapps" de Tomcat. Notar que Tomcat desempaca automáticamente los archivos con extensión .war que se encuentran en el directorio webapps de Tomcat.

Para eliminar el servicio web se deberá eliminar el archivo "Servicio.war" y el directorio "Servicio", en éste orden.

Cada vez que se modifique el archivo Servicio.java se deberá compilar, generar el archivo Servicio.war, borrar el archivo Servicio.war y el directorio Servicio del directorio webapps de Tomcat, y copiar el archivo Servicio.war al directorio webapps de Tomcat.

Probar el servicio web utilizando HTML-Javascript

1. Copiar el archivo [usuario_sin_foto.png](#) al subdirectorio webapps/ROOT de Tomcat.

Notar que todos los archivos que se encuentran en el directorio webapps/ROOT de Tomcat son accesibles públicamente.

Para probar que Tomcat esté en línea y el puerto 8080 esté abierto, ingresar la siguiente URL en un navegador:

```
http://ip-de-la-máquina-virtual:8080/usuario_sin_foto.png
```

2. Copiar el archivo [WSClient.js](#) al directorio webapps/ROOT de Tomcat.

3. Copiar el archivo [prueba.html](#) al directorio webapps/ROOT de Tomcat.

4. Ingresar la siguiente URL en un navegador:

```
http://ip-de-la-máquina-virtual:8080/prueba.html
```

5. Dar clic en el botón “Alta usuario” para dar de alta un nuevo usuario. Capturar los campos y dar clic en el botón “Alta”.

6. Intentar dar de alta otro usuario con el mismo email (se deberá mostrar una ventana de error indicando que el email ya existe)

7. Dar clic en el botón “Consulta usuario” para consultar el usuario dado de alta en el paso 5. Capturar el email y dar clic en el botón “Consulta”,

8. Modificar algún dato del usuario y dar clic en el botón “Modifica”:

9. Recargar la página actual y consultar el usuario modificado, para verificar que la modificación se realizó.

10. Dar clic en el botón “Borra usuario” para borrar el usuario. Capturar el email del usuario a borrar y dar clic en el botón “Consulta”.

Actividades individuales a realizar

Utilizando un teléfono inteligente y/o una tableta, probar el servicio web accediendo a la siguiente URL en un navegador (Chrome, Firefox, Opera, Safari, etc):

```
http://ip-de-la-máquina-virtual:8080/prueba.html
```

	Servicio.zip	
	WSClient.js	
	usuario_sin_foto.png	
	prueba.html	
	Tarea 7. Implementación de un servicio web estilo REST	

Cada alumno ejecutará el procedimiento que vimos en clase, dónde instalamos Tomcat, instalamos MySQL y creamos un servicio web estilo REST.

Se deberá probar el servicio web utilizando la aplicación web [prueba.html](#) tal como se explicó en clase.

Se deberá entregar un reporte en formato PDF que incluya la descripción de cada paso y la captura de pantalla correspondiente. Además, el reporte deberá tener portada y conclusiones.

Valor de la tarea: 20% (1 punto de la segunda evaluación parcial)



La clase de hoy explicaremos cómo funciona el servicio estilo REST que creamos la clase anterior.

En la plataforma se publicó los siguientes archivos:

- [Servicio.zip](#) contiene el código Java y archivos de configuración de un servicio web estilo REST.
- [prueba.html](#) contiene una aplicación web que invoca el servicio web mediante Javascript.
- [WSClient.js](#) funciones para invocar el servicio web mediante AJAX
- [usuario_sin_foto.png](#) imagen que se despliega cuando el usuario no tiene foto.

El archivo [Servicio.zip](#) contiene los siguientes directorios:

- META-INF
- negocio
- WEB-INF

El directorio META-INF contiene el archivo **context.xml**, en el cual se configura lo siguiente:

El atributo **name** de la etiqueta **Resource**, define el nombre del datasource, en este caso el datasource se llama "jdbc/datasource_Servicio".

Un datasource permite configurar las conexiones que realiza el servicio web, sin tener que escribir estos parámetros de configuración en el código (por ejemplo el nombre y contraseña del usuario de la base de datos).

El atributo **url** define el nombre de la base de datos, en este caso la base de datos se llama "servicio_web", el atributo **username** define el nombre del usuario de la base de datos y el atributo **password** define la contraseña del usuario.

El directorio WEB-INF contiene el directorio **classes** y el archivo **web.xml**.

El archivo **web.xml** configura lo siguiente:

La etiqueta **load-on-startup** igual a 1 indica que el servicio web se debe cargar cuando inicie el servidor de aplicaciones Tomcat.

La etiqueta **url-pattern** indica la ruta del servicio web. La URL del servicio web se explicará más adelante.

La etiqueta **resource-ref**, indica el nombre del datasource y la etiqueta **res-type** indica el tipo de recurso javax.sql.DataSource.

En el directorio **classes** se colocarán las clases compiladas del servicio web (archivos .class). En este caso el directorio incluye un subdirectorio llamado "negocio", debido a que las clases del servicio web se agrupan en un paquete llamado "negocio".

Al mismo nivel de los directorios META-INF y WEB-INF se encuentra un directorio llamado **negocio** dónde se puede encontrar los archivos que contienen el código fuente del servicio web.

Los archivos incluidos en el directorio **negocio** son los siguientes:

AdaptadorGsonBase64.java

La clase AdaptadorGsonBase64 permite modificar la forma en que GSON serializa los campos con tipo byte[].

Por omisión GSON convierte un campo de tipo byte[] en una lista de números separados por comas, lo cual ocupa mucho espacio (y tiempo en la comunicación).

La clase AdaptadorGsonBase64 permite convertir los campos de tipo byte[] a base 64, lo cual produce un texto más compacto.

Notar que jax-rs reemplaza cada "+" por espacio, sin embargo el decodificador Base64 no reconoce el espacio, por lo que es necesario reemplazar los espacios por "+".

Error.java

La clase Error va a permitir regresar un mensaje de error dentro de un objeto.

Foto.java

La clase Foto encapsula un arreglo de bytes y una clave numérica. El servicio web utiliza esta clase para enviar y recibir imágenes.

Usuario.java

La clase Usuario encapsula los datos del usuario. El servicio web utiliza esta clase para recibir los datos del usuario como un objeto.

Es muy importante notar que la anotación `@RequestParam` requiere un método que convierta una String a objeto de tipo Usuario. En este caso se implementa el método **valueOf** para este propósito.

Servicio.java

La clase Servicio implementa los métodos del servicio web.

URL del servicio web

La URL del servicio web se compone de cuatro elementos:

1. Dominio y el puerto, por ejemplo **localhost:8080**
2. Nombre del archivo .war, en este caso **Servicio**
3. Ruta definida en la etiqueta url-pattern en el archivo web.xml, en este caso la ruta: **rest**
4. Ruta definida en la anotación `@Path` de la clase Servicio, en este caso se define: **ws**

En este caso la URL completa sería: `http://localhost:8080/Servicio/rest/ws`

Pool de conexiones

Con frecuencia cuándo un servicio web recibe una petición (requerimiento), abre una conexión a la base de datos. Sin embargo realizar una conexión a la base de datos es lento. Por tanto, es conveniente que los servicio web utilicen un pool de conexiones mediante un datasource.

Como se dijo anteriormente, un datasource permite configurar los parámetros de conexión a la base de datos fuera del código del servicio web. No es buena práctica escribir el nombre de usuario de base de datos y la contraseña en el código del servicio web, ya que habría que editar y recompilar el código cada vez que se cambie la contraseña del usuario.

Así mismo, el uso de datasource y pool de conexiones permite establecer un número de conexiones a la base de datos y mantenerlas abiertas siempre. Cada vez que un método del servicio web requiera conectarse a la base de datos, solicita una conexión disponible en el pool.

Cuándo el método termina deberá liberar la conexión invocando el método "close", no obstante, este método no cierra la conexión sino que la regresa al pool de conexiones, para su posterior re-uso.

Es muy importante considerar que cualquier configuración que el método haga a la conexión (por ejemplo activar o desactivar el autocommit) quedará establecida en la conexión, de manera que si otro método re-usa la conexión, esta conexión tendrá una configuración previa.

Debido a lo anterior, un método que obtiene una conexión del pool de conexiones deberá re-configurar la conexión de acuerdo a sus propósitos, y no suponer que la conexión tiene una configuración determinada.

En el caso de la clase Servicio, el pool de conexiones se crea en la inicialización estática de la clase. Notar que el datasource se identifica con el nombre que se le dio en el archivo context.xml

El método POST

Como vimos en la clase anterior, un servicio web de estilo REST utiliza los métodos GET, POST, DELETE y PUT para realizar operaciones determinadas. En este caso, el servicio web que implementamos se aparta un poco del estándar ya que TODOS los métodos web utilizan el método POST, por esta razón todos los métodos de la clase Servicio incluyen la anotación **@POST**.

El servicio web produce JSON

El el código de la clase Servicio puede observarse que cada método incluye una anotación **@Produces**(MediaType.APPLICATION_JSON). Esto significa que el método regresará los resultados en formato JSON.

Para poder serializar y des-serializar JSON utilizaremos GSON (la implementación de JSON de Google), se declara una variable estática de tipo Gson y se inicializa con una instancia creada mediante el método create de la clase GsonBuilder:

```
static Gson j = new GsonBuilder()
    .registerTypeAdapter(byte[].class,new AdaptadorGsonBase64())
    .setDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS")
    .create();
```

Notar que en este caso se configura la instancia de Gson para utilizar el adaptador AdaptadorGsonBase64 para serializar y des-serializar arreglos de bytes, y se define el formato de fecha como ISO8601.

El servicio web consume URL encoded

Cada método incluye la anotación **@Consumes**(MediaType.APPLICATION_FORM_URLENCODED) la cual indica que los parámetros del método se recibirán codificados como URL. Esta codificación reemplaza algunos caracteres especiales como ampersam &, igual =, espacio, etc. los cuales son caracteres reservados cuando se construye una URL.

El nombre del método web

Cada método incluye la anotación **@Path**, esta anotación define el nombre del método web el cual puede ser diferente al nombre del método. Por ejemplo, el método "alta_usuario" se invocará como el método web "alta".

Los nombres de los parámetros del método web

Para indicar el nombre de cada parámetro del método web se utiliza la anotación **@FormParam**. El nombre del parámetro de método web puede ser diferente al nombre del parámetro del método Java.

El método web regresa Response

El método web regresa una instancia de la clase Response. Básicamente hay tres tipos de respuesta:

El método no regresa resultados. El método termina sin error pero no regresa resultados al cliente (por ejemplo el método "alta_usuario"). El método deberá regresar al cliente la siguiente instancia de la clase Response: Response.ok().build() esta instancia incluye el código HTTP 200 indicando que no hubo error.

El método regresa resultados. El método termina sin error y regresa algún resultado al cliente. Debido a que se definió en la anotación @Produces que el método regresa JSON, el método deberá regresar al cliente la siguiente instancia de la clase Response: Response.ok().entity(j.toJson(r)).build() en este caso r sería un objeto o un arreglo, el cual será serializado mediante GSON. El cliente recibe el código HTTP 200 indicando que no hubo error.

El método regresa un mensaje de error. El método termina con error. Para regresar un mensaje de error al cliente se utiliza la clase Error. Debido a que se definió en la anotación @Produces que el método regresa JSON, el método deberá regresar al cliente la siguiente instancia de la clase Response: Response.status(400).entity(j.toJson(new Error("Mensaje de error"))).build() el mensaje de error será de acuerdo al error que se produjo. El cliente recibe el código HTTP 400 indicando que hubo error. Adicionalmente, si así se desea se podría agregar a la clase Error un código numérico.

El método web debe cerrar la conexión a la base de datos

Un error frecuente de los programadores es que no cierran las conexiones a la base de datos. En general, los DBMS cuentan con un número limitado de conexiones. Si una aplicación no cierra las conexiones que no utiliza, se agotarán las conexiones en el DBMS y el sistema dejará de funcionar.

Para evitar esta situación, es necesario que el programador cierre las conexiones a la base de datos, archivos, sockets y cualquier otro recurso que haya abierto el método web. Esto debe hacerse mediante un cuidadoso

diseño del manejo de excepciones mediante try-finally-catch.

El archivo **WSClient.js**

Este archivo contiene código Javascript utilizado por [prueba.html](#) para consumir el servicio web mediante AJAX (XMLHttpRequest).

La función WSClient incluye la función "post" la cual recibe los siguientes parámetros: el nombre del método web a invocar, un arreglo con los argumento del método web y una función callback que se invocará cuándo el método web termine.

El archivo **prueba.html**

Este archivo contiene una Single-Page Application (SPA) escrita en HTML-Javascript, la cual invoca los métodos web del servicio web "Servicio".

Actividades individuales a realizar

1. Agregar un método web llamado "borrar_usuarios" al archivo Servicio.java, este método deberá borrar todos los usuarios de la tabla "usuarios". El método deberá regresar un mensaje de error si no pudo borrar los usuarios.
2. Agregar un botón al archivo [prueba.html](#). Al dar clic a este botón se deberá invocar el método web "borrar_usuarios".
3. Compilar el archivo Servicio.java
4. Construir el archivo Servicio.war
5. Borrar el archivo Servicio.war y el subdirectorio Servicio los cuales se encuentran en el directorio webapps de Tomcat.
6. Copiar el archivo Servicio.war al directorio webapps de Tomcat.
7. Utilizando un navegador probar el nuevo botón que invoca el método web "borrar_usuarios".



Tarea 8. Desarrollo de un cliente para un servicio web REST



Cada alumno deberá desarrollar un programa Java que consuma el servicio web que creamos en la tarea 7.

El programa deberá desplegar el siguiente menú:

MENU

- a. Alta usuario
- b. Consulta usuario
- c. Borra usuario
- d. Borra todos los usuarios
- e. Salir

Opción: _

Las opciones deberán implementar la siguiente funcionalidad:

- La opción "Alta usuario" leerá del teclado el email, el nombre del usuario, el apellido paterno, el apellido materno, la fecha de nacimiento, el teléfono y el género ("M" o "F"). Entonces se invocará el método "alta" del servicio web. Se deberá desplegar "OK" si se pudo dar de alta el usuario, o bien, el mensaje de error que regresa el servicio web. Notar que el método web "alta" recibe como parámetro una instancia de la clase Usuario (ver el archivo Servicio.java), recordemos que esta clase se define de la siguiente manera:

```
class Usuario
{
    String email;
    String nombre;
    String apellido_paterno;
    String apellido_materno;
    String fecha_nacimiento;
    String telefono;
    String genero;
```

```
byte[] foto;
}
```

Para invocar el método web "alta" es necesario crear un objeto de tipo Usuario y asignar los valores a los campos (en este caso el campo "foto" será null). Una vez que se tenga el objeto de tipo Usuario se deberá utilizar **GSON** para convertir el objeto a una string JSON, entonces se deberá utilizar esta string como valor del parámetro (ver más adelante cómo se enviarán los parámetros a los métodos web).

- La opción "Consulta usuario" leerá del teclado el email de un usuario previamente dado de alta. Entonces se invocará el método "consulta" del servicio web. Si el usuario existe se desplegará en pantalla el nombre del usuario, el apellido paterno, el apellido materno, la fecha de nacimiento, el teléfono y el género. La foto del usuario se ignorará. Notar que el método web "consulta" regresa una string JSON la cual representa un objeto de tipo Usuario (ver el archivo Servicio.java), por tanto será necesario utilizar **GSON** para convertir la string JSON a un objeto de tipo Usuario y posteriormente desplegar los campos del objeto (excepto el campo "foto"). Si hubo error, se desplegará el mensaje que regresa el servicio web.
- La opción "Borra usuario" leerá del teclado el email de un usuario previamente dado de alta. Entonces se invocará el método "borra" del servicio web. Se deberá desplegar "OK" si se pudo borrar el usuario, o bien, el mensaje de error que regresa el servicio web.
- La opción "Borra todos los usuarios" invocará el método "borrar_usuarios" creado en la actividad individual de la clase pasada. Se deberá desplegar "OK" si se pudo borrar el usuario, o bien, el mensaje de error que regresa el servicio web.
- La opción "Salir" terminará el programa.

¿Cómo invocar un método web REST utilizando Java?

A continuación se muestra un ejemplo de código Java que permite invocar el método "consulta" del servicio web Servicio.war, el cual creamos en la tarea 7.

La URL que utilizaremos para acceder al método "consulta" es la siguiente (notar que el nombre del método se escribe al final de la URL):

```
http://ip-de-la-máquina-virtual:8080/Servicio/rest/ws/consulta
```

Para que el método web pueda recibir los parámetros, cada parámetro se codifica de la siguiente manera: **nombre=valor**. Los parámetros se deben separar con un carácter **&**

El **valor** debe codificarse utilizando el método `URLEncoder.encode()`. La codificación URL reemplaza los caracteres reservados por un % y un valor hexadecimal, ver: <https://developers.google.com/maps/documentation/urls/url-encoding>.

Por ejemplo, si un método web tiene tres parámetros a, b y c, y los parámetros tienen los valores 10, 20 y 30, entonces se deberá enviar al método web los parámetros de la siguiente manera: "a=10&b=20&c=30", en este caso no es necesario utilizar el método `URLEncoder.encode()` para codificar el valor ya que se trata de un número.

El código que permite invocar el método "consulta" del servicio web Servicio.war es el siguiente:

```
URL url = new URL("http://ip-de-la-máquina-virtual:8080/Servicio/rest/ws/consulta");
```

```
HttpURLConnection conexion = (HttpURLConnection) url.openConnection();
```

```
conexion.setDoOutput(true);
```

```
// se utiliza el método HTTP POST (ver el método en la clase Servicio.java)
conexion.setRequestMethod("POST");
```

```
// indica que la petición estará codificada como URL
conexion.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
```

```
// el método web "consulta" recibe como parámetro el email de un usuario, en este caso el email es
"c@com"
String parametros = "email=" + URLEncoder.encode("c@com", "UTF-8");
```

```
OutputStream os = conexion.getOutputStream();
```

```
os.write(parametros.getBytes());
```

```
os.flush();

// se debe verificar si hubo error
if (conexion.getResponseCode() != HttpURLConnection.HTTP_OK)

    throw new RuntimeException("Codigo de error HTTP: " + conexion.getResponseCode());

BufferedReader br = new BufferedReader(new InputStreamReader((conexion.getInputStream())));

String respuesta;

// el método web regresa una string en formato JSON
while ((respuesta = br.readLine()) != null) System.out.println(respuesta);

conexion.disconnect();
```

Notar que el código anterior se puede usar para desarrollar aplicaciones Android que requieran consumir servicios web REST.

Valor de la tarea: 30% (1.5 puntos de la segunda evaluación parcial)



Clase del día - 30/11/2020

En la clase de hoy vamos a jugar un kahoot en la modalidad de "challenge".

Para jugar el kahoot deberán ingresar al siguiente enlace:

[Desarrollo de Sistemas Distribuidos - Servicios Web](#)

Es necesario que los alumnos y alumnas ingresen su "nickname" como su nombre y apellido (por ejemplo JuanLopez), de manera que sea posible identificar a los ganadores de puntos extra.

La hora límite para jugar este kahoot es 11:00 PM del 30 de noviembre.

4. Servicios de nombres, archivos y replicación

5. Cómputo en la nube

© Carlos Pineda Guerrero, 2020.

- Usted está ingresado como Luis Enrique Rojas Alvarado (Salir)
- Reiniciar tour para usuario en esta página
- Página Principal (home)
- Resumen de conservación de datos
- Obtener la App Mobile