



# Curso: Desarrollo de Sistemas Distribuidos

- **1. Introducción**

- **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 como 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 la 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 dónde 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; 1/4 de punto al primer lugar, 1/6 de punto al segundo lugar y 1/8 de punto al tercer lugar.

Se agregará a la calificación del parcial, los puntos de kahoots que cada alumno ganó en el mismo parcial. Cada alumno solo podrá aplicar un máximo de 1 punto extra por kahoots cada parcial.

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

Si se sobrepasa la calificación de 10 después de agregar los puntos de kahoot, la calificación que se asentará en el parcial será 10, el excedente no se aplicará a los siguientes parciales. Los puntos de los kahoots no son transferibles a los siguientes parciales.

Es necesario que los alumnos accedan a cada kahoot con su nombre y apellidos, por ejemplo JuanLopezMorales, de manera que sea posible identificar a los ganadores de puntos extra.

## Evaluación parcial

Cada parcial se evaluará de la siguiente forma:

- Tareas (70%)
- Examen teórico (20%)
- Participación en clase (10%)
- Puntos extra

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

Las clases se van a impartir por videoconferencia. Para acceder a las clases se deberá utilizar el enlace “Acceso a la clase” disponible en la sección “[Avisos](#)” de la plataforma.

Deberán acceder a la sesión de videoconferencia con su nombre completo, de manera que sea posible identificarlos y darles acceso a la sesión.

Podrán ingresar a la sesión de videoconferencia en cualquier momento dentro del horario de clase. Se pasará lista de asistencia en la sesión de videoconferencia. La tolerancia para tener asistencia será de 15 minutos.

Los días no laborables no habrá clase, no obstante la plataforma estará disponible 24x7 durante todo el curso. Por cierto, la plataforma moodle ejecuta sobre Microsoft Azure. Se solicitará a los alumnos que presenten su pantalla en la sesión de videoconferencia para revisar el avance en la realización de sus actividades. Para tener asistencia en clase, los alumnos deberán realizar las actividades de la clase.

Los alumnos obtendrán el 10% de participación en clase si tiene al menos el 80% de asistencias en el parcial.

Para poder presentar el examen parcial los alumnos y alumnas deberán tener al menos el 80% de asistencias en el parcial.

## Referencias

1. *Sistemas Distribuidos Principios y Paradigmas*, Tanenbaum, A. Van Steen, M., Ed. Pearson Educación, Segunda edición, 2008.
2. *Sistemas Distribuidos Conceptos y Diseño*, Coulouris, G. Dolimore, J. Kindberg, Ed. Pearson Educación, 2001.
3. *Mastering Cloud Computing: Foundations and Applications Programming*, Buyya, Rajkumar, Vecchiola, Christian, Ed. MK, 2013.
4. *Webservices, Theory and Practice*, Hrushikesh Mohanty, Prasant Kumar, Ed. Springer, 2018.
5. *Java Course*, <http://youtube.com/watch?v=coK4jM5wvko&t=4s>

### Actividades individuales a realizar

1. Obtener una cuenta de correo institucional del IPN.
2. Darse de alta en el programa [Azure for Students](#).
3. Instalar en su computadora el JDK8 o superior.



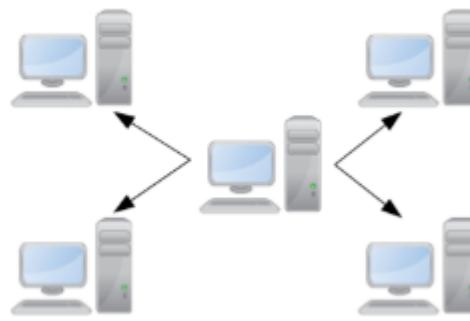
#### ○ Tipos de comunicaciones

Los tipos de comunicaciones entre computadoras son los siguientes:

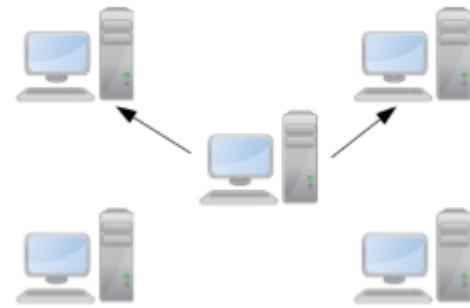
**Unicast.** El unicast es una comunicación punto a punto dónde una computadora envía mensajes a otra computadora.



**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.



## Sockets

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. Los puertos del 0 a 1023 están reservados.

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.

## 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.

La clase D a su vez se divide en tres rangos de acuerdo a su uso:

Dirección inicial	Dirección final	Uso
224.0.0.0	224.0.0.255	Direcciones multicast reservadas
224.0.1.0	238.255.255.255	Direcciones multicast con alcance global (internet)
239.0.0.0	239.255.255.255	Direcciones multicast con alcance local

Fuente: [http://www.tcpipguide.com/free/t\\_IPMulticastAddressing.htm](http://www.tcpipguide.com/free/t_IPMulticastAddressing.htm)

**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. Un protocolo define la estructura de los paquetes de datos.

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 sin conexión)

- 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 sin conexión.

Las características de los sockets sin conexión son las siguientes:

- No requieren un canal dedicado de comunicación.
- No se garantiza la integridad de los datos enviados.
- Los paquetes son enviados y recibidos en diferente orden.
- Los paquetes pueden recibirse duplicados.
- 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.

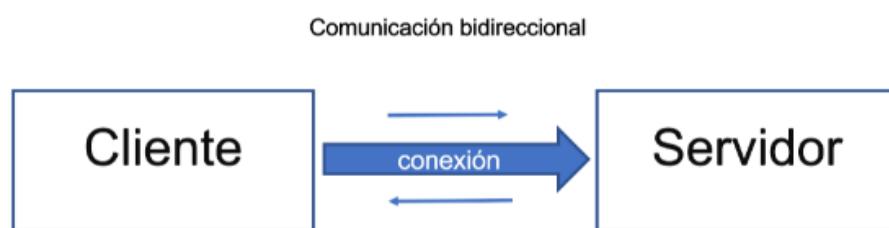
- Se implementan mayormente usando el protocolo UDP.



- **Cliente - Servidor**

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

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



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

En particular, los clientes y servidores que utilizaremos en el curso usan sockets TCP.

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

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

### **Cliente.java**

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

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

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

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

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

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

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

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

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

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

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

```
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**. Por omisión el método **getBytes** utiliza la codificación default (UTF-8), para usar otra codificación se puede pasar como parámetro el nombre de la codificación como string, por ejemplo "UTF-8". 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**. El método **read** tiene tres parámetros, el primer parámetro es un arreglo de bytes con una longitud suficiente para contener los bytes a recibir. El segundo parámetro indica la posición, dentro del arreglo de bytes, donde se pondrán los bytes a recibir, y el tercer parámetro indica el número de bytes a recibir.

El siguiente código crea un arreglo de 4 bytes, invoca el método **read** de la clase **DataInputStream**, crea una instancia de la clase **String** utilizando los bytes recibidos.

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.

```
byte[] buffer = new byte[4];
entrada.read(buffer,0,4);
System.out.println(new String(buffer,"UTF-8"));
```

Sin embargo es necesario considerar que el método **read** podría obtener solo una parte 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"));
```

### Los métodos **writeUTF** y **readUTF**

Para enviar y recibir strings entre programas escritos en Java, se puede utilizar el método **writeUTF** de la clase **DataOutputStream** y el método **readUTF** de la clase **DataInputStream**.

El método **writeUTF** convierte la string a arreglo de bytes utilizando el método **getBytes("UTF-8")** de la clase **String**, escribe al stream de salida la longitud del arreglo de bytes utilizando el método **writeShort** y escribe al stream de salida los bytes utilizando el método **write**.

El método **readUTF** lee del stream de entrada el número de bytes a recibir utilizando el método **readShort**, lee los bytes utilizando el método **read** y crea una instancia de la clase **String** utilizando codificación UTF-8.

En Java una string puede tener una longitud máxima de 2,147,483,647 caracteres, sin embargo los métodos **writeUTF** y **readUTF** solo pueden enviar strings cuya codificación UTF-8 tenga una longitud máxima de 32,767 bytes.

### La clase **ByteBuffer**

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 cerramos la conexión con el servidor (al cerrar el socket se cierran también los streams asociados), en este caso vamos a poner un retardo de un segundo antes de cerrar la conexión, para permitir que el servidor tenga tiempo de recibir los datos:

```
Thread.sleep(1000);
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 la conexión con el cliente:

```
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`?



- Servidor multithread y cliente con re-intentos de conexión

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.

### Orden de las operaciones de lectura y escritura

En las clases de Sistemas Operativos se explica que un thread (hilo) es la ejecución secuencial de las instrucciones de un programa. Un proceso puede crear uno o más threads (hilos de ejecución), los cuales van a ejecutar simultáneamente.

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.

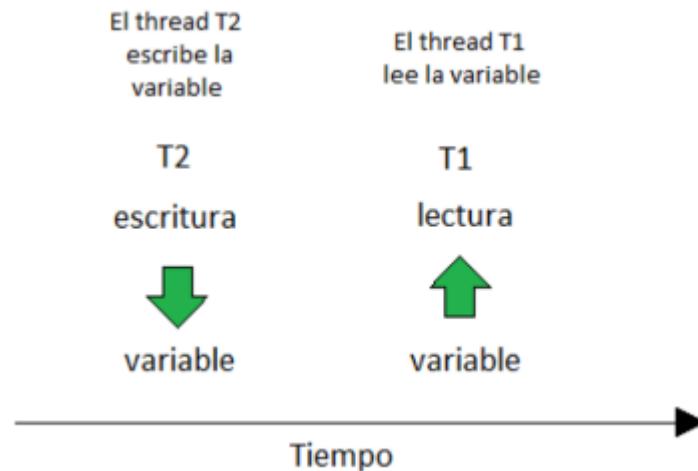
Por otra parte, si un programa crea un número de threads mayor al número de procesadores físicos (*cores*) disponibles en la computadora, entonces los threads ejecutarán en forma concurrente (por turnos).

Los threads dentro de un proceso se comunican entre sí utilizando la memoria. Las operaciones que se realizan sobre la memoria son la lectura y escritura de variables (localidades de memoria).

Para que un thread pueda leer los datos que escribe otro thread en la memoria, es necesario ordenar las operaciones de lectura y escritura que realizan los threads.

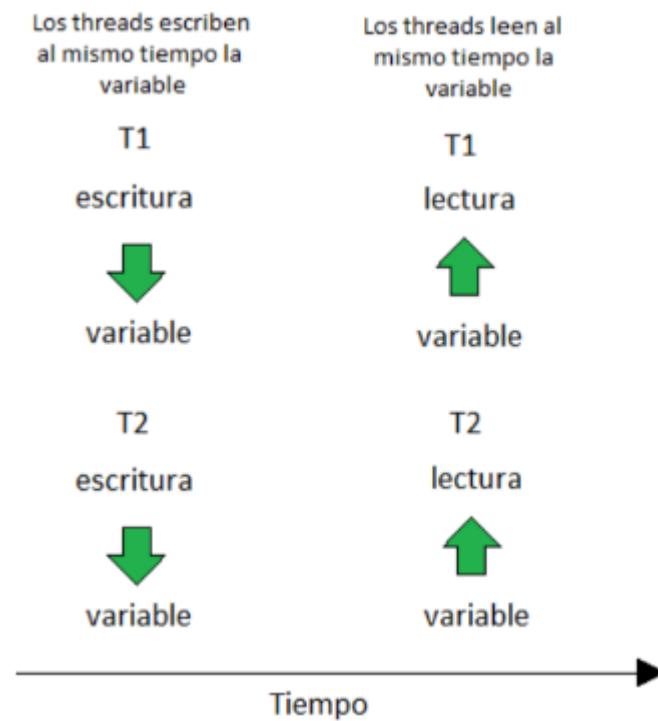
Supongamos que tenemos dos threads, el thread T1 y el thread T2. Si el thread T2 escribe a una variable y posteriormente el

thread T1 lee la variable, el thread T1 tendrá el valor que escribió el thread T2.



### Orden escritura-escritura

Ahora supongamos que los threads T1 y T2 escriben al mismo tiempo una variable y posteriormente los threads leen al mismo tiempo la misma variable. ¿Qué valores leyeron los threads?

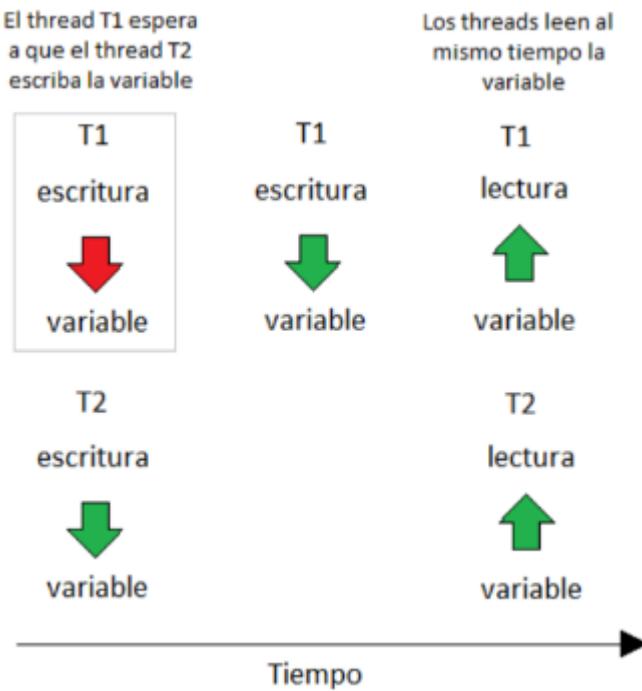


Evidentemente no es posible garantizar que el thread T1 haya leído el valor que escribió el thread T2 o que haya leído el valor que escribió el mismo thread T1. Igualmente, no es posible garantizar que el thread T2 haya leído el valor que escribió el

thread T1 o que el thread T1 haya el valor que escribió el mismo thread T2.

Entonces, para garantizar que un thread lee el valor de una variable escrito por otro thread, es necesario ordenar las operaciones de escritura y de lectura que realizan los threads.

Ahora supongamos que el thread T1 espera a que el thread T2 escriba la variable, entonces después el thread T1 escribe a la variable.

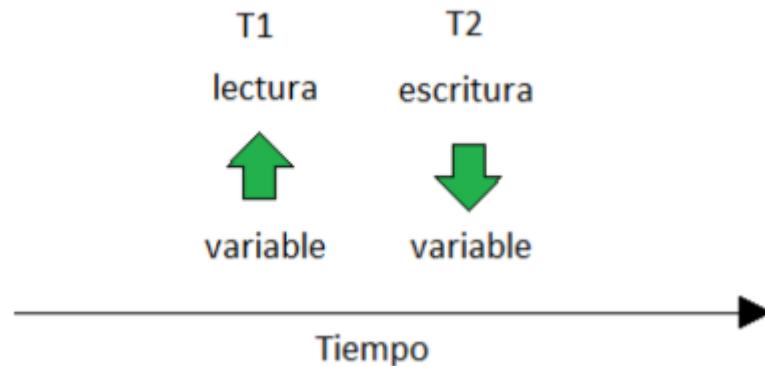


En este caso, el valor que leen los threads T1 Y T2 es el valor que escribió el thread T1.

Notar que dos o más threads pueden leer simultáneamente una variable.

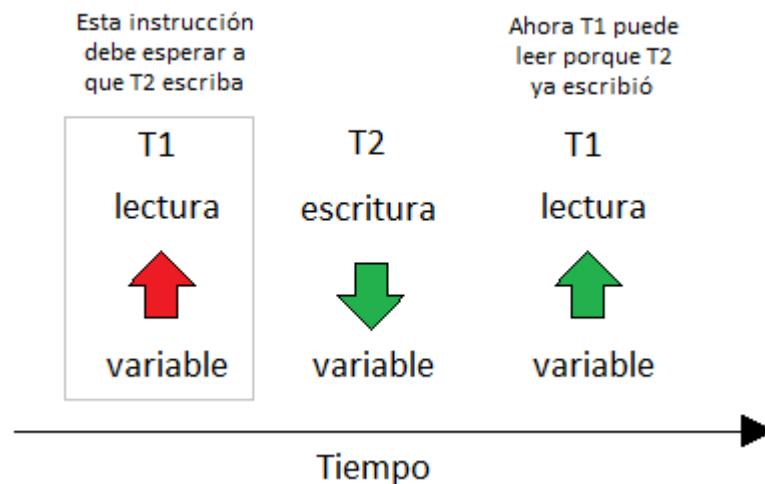
### Orden escritura-lectura

Ahora supongamos que el thread T1 lee la variable y posteriormente el thread T2 escribe la variable ¿qué valor leyó el thread T1?



Para que el thread T1 pueda leer el valor que escribe el thread T2, es necesario ordenar las operaciones de escritura y lectura.

Para garantizar que el thread T1 lea el valor escrito por el thread T2, el thread T1 debe esperar a que el thread T2 escriba la variable.



Cuando dos o más threads leen o escriben a una misma variable, y al menos uno de los threads 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.

### Sincronización de threads en Java

Para ordenar las lecturas y escrituras que realizan los threads en Java, se utiliza la instrucción **synchronized**.

```
synchronized(objeto)
{
```

```
instrucciones  
}
```

Debemos recordar que en Java todos los objetos tienen un lock asociado.

En la clase de Sistemas Operativos se explica que un lock (o cerrojo) es un mecanismo que permite limitar el acceso a un recurso compartido por varios procesos o threads, por ejemplo, una variable, un archivo, una impresora. etc.

A la capacidad que tiene un sistema de controlar el acceso a recursos compartidos se le conoce como **exclusión mutua**. Más adelante en el curso veremos cómo implementar la exclusión mutua en sistemas distribuidos.

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

- Primero se verifica si el lock del *objeto* está bloqueado, 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**.

### Programación multithread en Java

Supongamos que tenemos una clase 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 resultado.

### 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**.

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 un 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.

Ahora agregaremos el siguiente código al método **run**:

```
try  
{  
    DataOutputStream salida = new  
    DataOutputStream(conexion.getOutputStream());  
    DataInputStream entrada = new  
    DataInputStream(conexion.getInputStream());  
  
    int n = entrada.readInt();  
    System.out.println(n);  
  
    double x = entrada.readDouble();  
    System.out.println(x);  
  
    byte[] buffer = new byte[4];  
    read(entrada,buffer,0,4);  
    System.out.println(new String(buffer,"UTF-8"));  
  
    salida.write("HOLA".getBytes());  
  
    byte[] a = new byte[5*8];  
    read(entrada,a,0,5*8);
```

```

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

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

```

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.



- 1. ▪ ¿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?
- 2. Compile los programas **Cliente2.java** y **Servidor2.java**
- 3. Ejecute el programa **Cliente2.java** en una ventana de comandos de Windows (o terminal de Linux) y después ejecute el programa **Servidor2.java** en otra ventana de comandos de Windows (o terminal de Linux). El cliente2 debe esperar que el servidor inicie su ejecución.
- 4. 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.



- **Infraestructura de clave pública (PKI)**

La clase de hoy vamos a ver como implementar un cliente y un servidor mediante sockets seguros.

Primeramente vamos a explicar los conceptos básicos de PKI (*Public Key Infrastructure*).

### Criptografía simétrica

La criptografía simétrica es un conjunto de algoritmos que permiten encriptar y desencriptar utilizando la misma clave, conocida como *clave secreta* o *clave privada*.

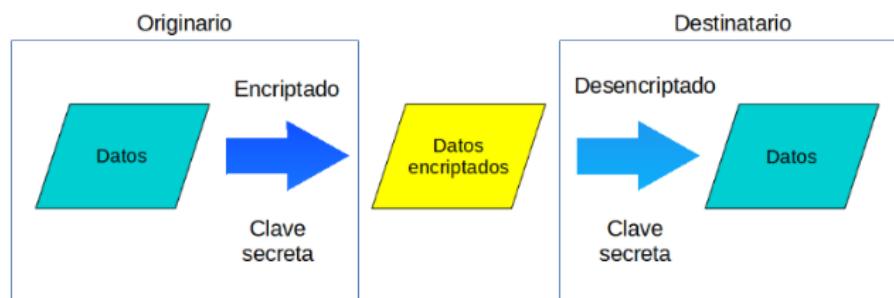
Ejemplos de algoritmos simétricos son el AES-128, AES-256, RC4, RC5, DES, 3DES, entre otros.

Los algoritmos simétricos son muy rápidos, sin embargo resulta complicado intercambiar las claves.

Por ejemplo, supongamos que un amigo se va a estudiar a otro país y en un momento dado te pide le envíes un documento electrónico utilizando email. Desde luego habría que encriptar el documento para evitar que alguna otra persona pudiera hacer mal uso de él.

El problema es que ambos deberían tener la clave para encriptar y desencriptar.

La única solución es que ambos hayan compartido la clave para encriptar y desencriptar antes del viaje, ya que tampoco es seguro enviar la clave por email.



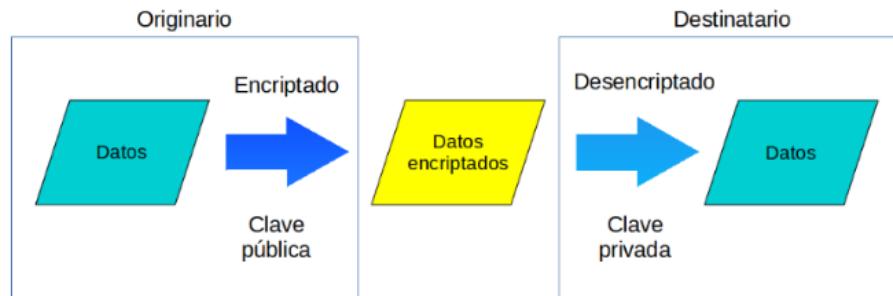
### Criptografía asimétrica

En la criptografía asimétrica el destinatario del mensaje tienen dos claves, una llamada *clave privada* y otra llamada *clave pública*.

La clave privada es una clave que mantiene en secreto el destinatario de los datos, mientras que la clave pública es una clave que puede conocer cualquiera.

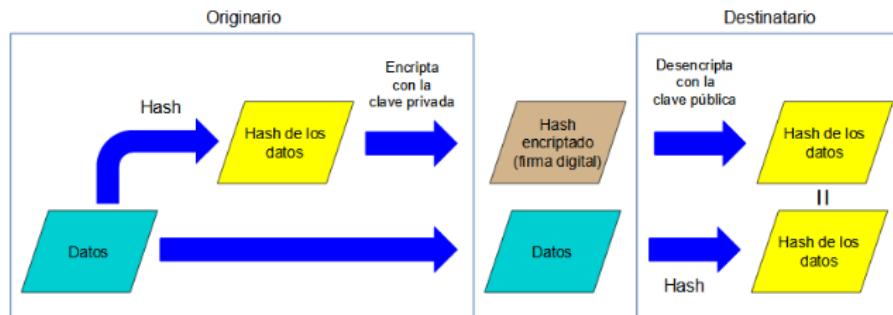
En la criptografía asimétrica (también llamada criptografía de llave pública), el originario utiliza la clave pública del destinatario para encriptar los datos. Entonces el destinatario

utiliza su clave privada para desencriptar los datos encriptados y obtener los datos en claro.



El par de claves pública y privada pueden utilizarse para autenticar un documento electrónico. A esta autenticación se llama **firma digital**.

Supongamos que vamos a enviar un documento (datos) a un destinatario. Para garantizar que el documento procede de un origen determinado, el originario genera un hash de los datos y encripta el hash con su clave privada. Al hash encriptado le llamamos la firma digital del documento.



Entonces el originario envía al destinatario el documento y la firma digital respectiva.

El destinatario desencripta la firma digital utilizando la clave pública del originario, entonces obtiene el hash del documento. Así mismo, el destinatario genera el hash del documento recibido.

Si los dos hashes son iguales, entonces podemos estar seguros que el documento recibido procede del propietario de la clave pública y que el documento no ha sido modificado, debido a que cualquier modificación en el documento cambiaría el hash.

Ahora bien, ¿cómo sabemos que una clave pública pertenece a una persona u organismo determinado?

Si el originario nos envió su clave pública por email, y sabemos con certeza que la dirección de correo electrónico pertenece a ésta persona, entonces concluimos que la clave pública pertenece a ésta persona de tal manera que podemos verificar la firma digital de cualquier documento que nos envíe.

Sin embargo, no siempre podemos conocer a ciencia cierta que una dirección de correo electrónico pertenece a una persona determinada.

Para resolver el problema de la identidad de una clave pública, se utiliza un documento electrónico llamado **certificado digital**.



- Un certificado digital es un documento electrónico que contiene, entre otros datos: la **identidad** de una persona u organización, las fechas de validez del certificado, una **clave pública** de la persona u organización y la **firma digital** de los datos anteriores.

La clave pública que contiene el certificado está asociada a una clave privada que solo conoce la persona u organización propietaria del certificado digital. Para mayor seguridad, la clave privada generalmente se encripta con una clave simétrica.

El certificado digital es firmado por una **autoridad certificadora** (CA) la cual es una organización registrada en el sistema operativo de nuestra computadora como una organización de confianza. El sistema operativo cuenta con un **repositorio de certificados de confianza**; en este repositorio se instalan los certificados de las autoridades certificadoras en las que confiamos.

Es estándar más utilizado para certificados digitales es el X.509

Por default, el sistema operativo incluye en el repositorio de certificados de confianza los certificados de las autoridades certificadoras reconocidas internacionalmente. A estos certificados se le conoce como certificados raíz (*root*).

Existen dos tipos de certificados, los certificados autofirmados y los certificados firmados por una CA.

### Certificado autofirmado

Un certificado autofirmado es aquel que el usuario crea. Al crear un certificado autofirmado se crea un par de claves pública y privada, la clave pública se incluye en el certificado y éste se firma utilizando la clave privada.

Los datos de identidad en el certificado autofirmado los capture el usuario al crear el certificado.

### Certificado firmado por una CA

Un certificado firmado por una CA es un certificado que compramos a un proveedor de certificados digitales (p.e. [cheapsslsecurity.com](https://cheapsslsecurity.com)).

Existen dos tipos de certificados firmados por una CA, aquellos que verifican dominio y aquellos que adicionalmente verifican la empresa.

Para poder tener un certificado con **verificación de dominio**, es necesario ser propietario de un dominio.

Por otra parte, para poder tener un certificado con **verificación de empresa**, es necesario tener una empresa y un dominio.

Cuando se adquiere un certificado firmado por una CA, además del certificado se obtiene un archivo conocido como *bundle*, el cual contiene los certificados de CA que forman una **ruta de certificación** desde el certificado emitido, hasta un certificado raíz pre-instalado en la computadora.

Veamos un ejemplo de certificado digital con verificación de dominio, en este caso el dominio es m4gm.com

The screenshots show the following details:

- General Tab:**
  - Información del certificado:** Este certif. está destinado a los siguientes propósitos:
    - Prueba su identidad ante un equipo remoto
    - Asegura la identidad de un equipo remoto
    - 1.3.6.1.4.1.6449.1.2.2.7
    - 2.23.140.1.2.1
  - Emitido para:** m4gm.com
  - Emitido por:** Sectigo RSA Domain Validation Secure Server CA
  - Válido desde:** 19/01/2021 hasta 24/01/2022
- Details Tab:**

Campo	Valor
Versión	V3
Número de serie	00a0864215fb3315090fc02a...
Algoritmo de firma	sha256RSA
Algoritmo hash de firma	sha256
Emisor	Sectigo RSA Domain Validation...
Válido desde	martes, 19 de enero de 2021...
Válido hasta	lunes, 24 de enero de 2022 06...
Sujeto	m4gm.com
- Certificate Path Tab:**

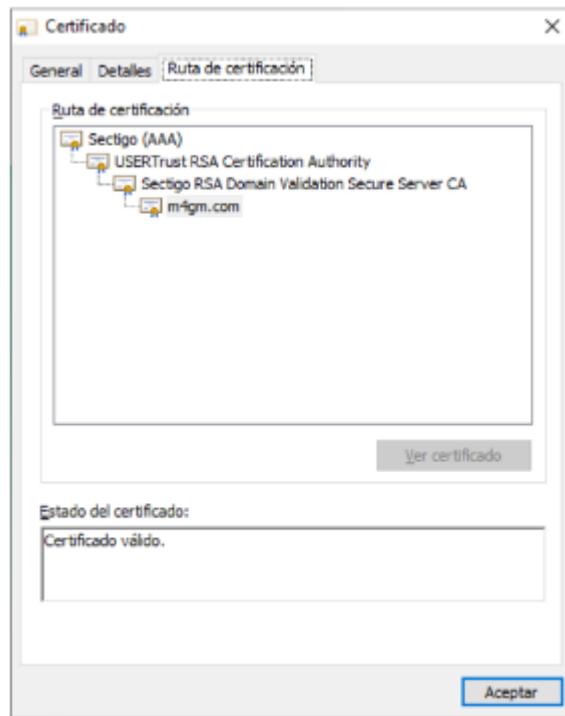
Campo	Valor
Sujeto	m4gm.com
Clave pública	RSA (2048 Bits)
Parámetros de clave pública	05 00
Identificador de clave de en...	Id. de clave =8d8c5ec454ad8a...
Identificador de clave del tit...	e4190355c8778680205f36b89...
Uso mejorado de claves	Autenticación del servidor (1.3...
Directivas del certificado	[!]Directiva de certificado:Id...
Acceso a la información de...	[!]Acceso a la información de...

Hex dump of the certificate content:

```

30 82 01 0a 02 82 01 01 00 c4 c8 f4 19 e9
b5 37 a5 98 5b 5f e6 38 53 7a a0 6d 7a ec
dd c5 93 03 c2 02 0a 1b 47 f4 b3 b6 d7 64
3f 02 e7 f1 0d 38 1b 0a 31 0e c4 93 52
b1 65 32 49 07 40 08 f2 35 0f b3 92 74
07 58 2b 88 65 06 30 89 96 bf da 59 9d 79
73 99 45 8c 54 7b 41 d0 e9 1d ef 52 d9 4f
9a e6 f8 29 72 22 f8 b0 05 b0 80 14 02 08
2a 27 b5 55 cd 04 f9 d4 65 d3 14 a2 9d 00

```



## Cliente - Servidor SSL

Ahora veremos cómo crear un cliente y un servidor los cuales se comuniquen mediante sockets seguros.

Primeramente vamos a crear un certificado autofirmado utilizando el programa keytool incluido en JDK.

```
keytool -genkeypair -keyalg RSA -alias
certificado_servidor -keystore
keystore_servidor.jks -storepass 1234567
```

La opción **genkeypair** genera un par de claves pública y privada. La clave pública se pone en un certificado autofirmado con un solo elemento en la ruta de certificación. El **alias**, en este caso "certificado\_servidor", define un nombre con el cual vamos a identificar el certificado. **Keystore** es un archivo (repositorio) donde se va a almacenar el certificado y la clave privada correspondiente. **Keyalg** es el algoritmo a utilizar para generar el par de claves, en este caso RSA. **Storepass** es la contraseña para el keystore.

Entonces se deberá capturar lo siguiente:

¿Cuáles son su nombre y su apellido?

[Unknown]: **nombre**

¿Cuál es el nombre de su unidad de organización?

[Unknown]: **unidad**

¿Cuál es el nombre de su organización?

[Unknown]: **organizacion**

¿Cuál es el nombre de su ciudad o localidad?

[Unknown]: **CDMX**

¿Cuál es el nombre de su estado o provincia?

[Unknown]: **CDMX**

¿Cuál es el código de país de dos letras de la unidad?

[Unknown]: **MX**

¿Es correcto CN=nombre, OU=unidad,

O=organizacion, L=CDMX, ST=CDMX, C=MX?

[no]: **si**

Introduzca la contraseña de clave para

<certificado\_servidor>

(INTRO si es la misma contraseña que la del almacén de claves):

IMPORTANTE: Para Java, la clave del certificado

<certificado\_servidor> y la clave (storepass) del almacén de

claves (keystore) deben ser las mismas, en este caso: 1234567

Ahora vamos a obtener el certificado contenido en el keystore.

```
keytool -exportcert -keystore
keystore_servidor.jks -alias certificado_servidor
-rfc -file certificado_servidor.pem
```

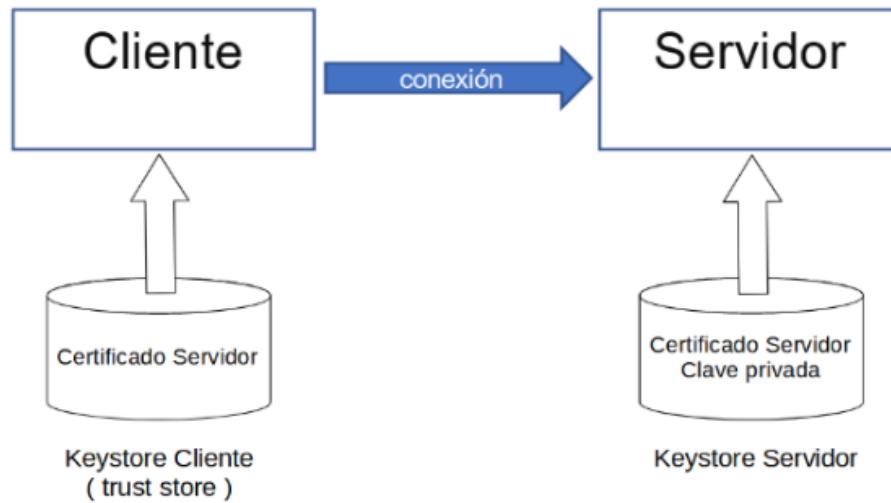
La opción **exportcert** lee del keystore el certificado identificado por el alias y genera un archivo texto que contiene el certificado, en este caso se genera el archivo certificado\_servidor.pem

Entonces vamos a crear un keystore que utilizará el cliente, este keystore deberá contener el certificado del servidor:

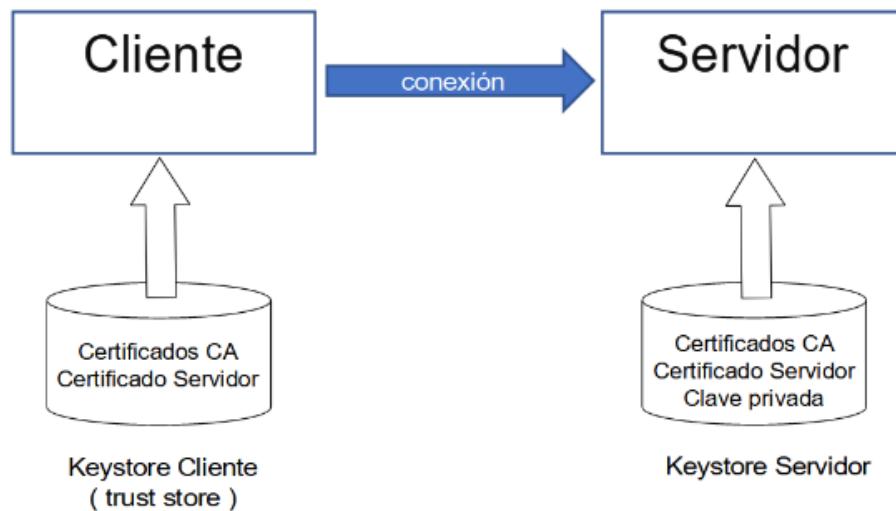
```
keytool -import -alias certificado_servidor -file  
certificado_servidor.pem -keystore  
keystore_cliente.jks -storepass 123456
```

La opción **import** lee el archivo `certificado_servidor.pem` e inserta el certificado en el keystore `keystore_cliente.jks`, identificando el certificado mediante el alias. **Storepass** es la contraseña del keystore.

En la siguiente figura podemos ver que el servidor utilizará el keystore que contiene el certificado del servidor y la clave privada respectiva. El cliente utilizará el keystore que contiene el certificado del servidor.



Por otra parte, también es posible utilizar un certificado firmado por una CA, en este caso será necesario que el keystore que utilizará el servidor contenga los certificados de CA (bundle), el certificado del servidor y la clave privada correspondiente. El keystore que utilizará el cliente deberá contener los certificados de CA (bundle) y el certificado del servidor.



### ClienteSSL.java

El cliente debe crear una instancia de la clase `SSLSocketFactory`:

```
SSLSocketFactory cliente = (SSLSocketFactory)
SSLSocketFactory.getDefault();
```

Entonces vamos a crear un socket que se conectará al servidor invocando el método `createSocket` de la clase `SSLSocketFactory`. En este caso el servidor se llama "localhost" (computadora local) y el puerto abierto en el servidor es el 50000.

```
Socket conexion =
cliente.createSocket("localhost",50000);
```

Abrimos los streams de salida y de entrada como lo hicimos anteriormente.

```
DataOutputStream salida = new
DataOutputStream(conexion.getOutputStream());
DataInputStream entrada = new
DataInputStream(conexion.getInputStream());
```

Ahora podemos enviar datos al servidor, por ejemplo vamos a enviar un double:

```
salida.writeDouble(123456789.123456789);
```

Para terminar el programa cerramos la conexión con el servidor (al cerrar el socket se cierran también los streams asociados), en este caso vamos a poner un retardo de un segundo antes de cerrar la conexión, para permitir que el servidor tenga tiempo de recibir los datos:

```
Thread.sleep(1000);  
conexion.close();
```

### ServidorSSL.java

El servidor debe crear una instancia de la clase `SSLSocketFactory`:

```
SSLSocketFactory socket_factory =  
(SSLSocketFactory)  
SSLSocketFactory.getDefault();
```

Vamos a crear un socket servidor que va a abrir, en este caso, el puerto 50000 utilizando el método `createServerSocket` de la clase `SSLSocketFactory`:

```
ServerSocket socket_servidor =  
socket_factory.createServerSocket(50000);
```

Ahora invocamos el método `accept` de la clase `ServerSocket`. Cuando se recibe la conexión el método `accept` regresa un socket:

```
Socket conexion = socket_servidor.accept();
```

Abrimos los streams de salida y de entrada:

```
DataOutputStream salida = new  
DataOutputStream(conexion.getOutputStream());  
DataInputStream entrada = new  
DataInputStream(conexion.getInputStream());
```

Ahora podemos recibir datos del cliente, en este caso vamos a recibir un double:

```
double x = entrada.readDouble();  
System.out.println(x);
```

Finalmente, cerramos la conexión:

```
conexion.close();
```

Para ejecutar el servidor se debe indicar el nombre del keystore del servidor y la contraseña:

```
java -  
Djavax.net.ssl.keyStore=keystore_servidor.jks -  
Djavax.net.ssl.keyStorePassword=1234567  
ServidorSSL
```

Para ejecutar el cliente se debe indicar el nombre del keystore del cliente (repositorio de confianza) y la contraseña:

```
java -  
Djavax.net.ssl.trustStore=keystore_cliente.jks -  
Djavax.net.ssl.trustStorePassword=123456  
ClienteSSL
```



- Actividades individuales a realizar

En esta actividad vamos a desarrollar un servidor multithread que recibirá un archivo del cliente y lo almacenará en el disco local. La comunicación deberá utilizar sockets seguros.

1. Cuando el servidor reciba una conexión del cliente, deberá crear un thread el cual recibirá el nombre del archivo utilizando el método **readUTF()** de la clase **DataInputStream**, entonces recibirá la longitud del archivo utilizando el método **readInt()** de la clase **DataInputStream** y recibirá el contenido del archivo como arreglo de bytes utilizando el método **read()** estático que se explicó en clase.
2. El servidor deberá escribir el contenido del arreglo de bytes al disco local, utilizando el siguiente método:

```
static void escribe_archivo(String archivo,byte[]  
buffer) throws Exception  
{  
    FileOutputStream f = new  
    FileOutputStream(archivo);  
    try  
    {  
        f.write(buffer);  
    }  
    finally  
    {  
        f.close();  
    }  
}
```

3. Se deberá pasar como parámetro al cliente el nombre del archivo a enviar, entonces el cliente deberá leer el archivo del disco local utilizando el siguiente método:

```
static byte[] lee_archivo(String archivo) throws  
Exception  
{
```

```
FileInputStream f = new  
FileInputStream(archivo);  
byte[] buffer;  
try  
{  
    buffer = new byte[f.available()];  
    f.read(buffer);  
}  
finally  
{  
    f.close();  
}  
return buffer;  
}
```

4. El cliente deberá enviar al servidor el nombre del archivo utilizando el método **writeUTF()** de la clase **DataOutputStream**, deberá enviar la longitud del archivo utilizando el método **writeInt()** de la clase **DataOutputStream** y el contenido del archivo utilizando el método **write()** de la clase **DataOutputStream**.



- **Cliente - Servidor multicast**

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 sin conexión, 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()`. Antes de que el servidor envíe mensajes, necesitamos asignar `true` a la propiedad `java.net.preferIPv4Stack` debido a que nuestro programa utilizará sockets IP v4 y por default Java usa sockets nativos IP v6, si éstos están disponibles en el sistema operativo (<https://docs.oracle.com/javase/8/docs/api/java/net/doc-files/net-properties.html>):

```
System.setProperty("java.net.preferIPv4Stack", "true"); // sugerencia del alumno Jhonatan Jhair Venegas Perez
```

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

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

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

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

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

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

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

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

### 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[] recibir_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 `receive()` de la clase `MulticastSocket`. El método `recibe_mensaje()` regresa el mensaje recibido.

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

Tal como lo hicimos en el servidor, antes de que el cliente reciba mensajes, necesitamos asignar `true` a la propiedad `java.net.preferIPv4Stack`:

```
System.setProperty("java.net.preferIPv4Stack", "true"); // sugerencia del alumno Jhonatan Jhair Venegas Perez
```

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

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

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

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

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

```
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();
```

En el ejemplo que acabamos de ver se utiliza la dirección IP 230.0.0.0 para implementar multicast localmente.

Como vimos en clases anteriores, para implementar multicast entre computadoras diferentes se deberá utilizar el rango de direcciones 224.0.1.0 a 238.255.255.255. También será necesario configurar los ruteadores para soportar multicast IPv4.

Ver: [http://www.ibiblio.org/pub/Linux/docs/howto/other-formats/html\\_single/Multicast-HOWTO.html](http://www.ibiblio.org/pub/Linux/docs/howto/other-formats/html_single/Multicast-HOWTO.html)

Por razones de seguridad, Microsoft Azure ha deshabilitado el multicast entre diferentes máquinas virtuales.

**Actualización a partir de JDK 14**

Debido a que a partir de JDK 14 se deprecó los métodos `joinGroup(InetAddress mcastaddr)` y `leaveGroup(InetAddress mcastaddr)`, el cliente multicast deberá utilizar los métodos `joinGroup(SocketAddress mcastaddr, NetworkInterface netIf)` y `leaveGroup(SocketAddress mcastaddr, NetworkInterface netIf)`, tal como se muestra en el siguiente ejemplo:

```
MulticastSocket socket = new
MulticastSocket(50000);
InetSocketAddress grupo = new
InetSocketAddress(InetAddress.getByName("230.0.0.
0"),50000);
NetworkInterface netInter =
NetworkInterface.getByName("em1");
socket.joinGroup(grupo,netInter);

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

```
byte[] buffer = recibe_mensaje(socket,5*8);
ByteBuffer b = ByteBuffer.wrap(buffer);
for (int i = 0; i < 5; i++)
System.out.println(b.getDouble());

socket.leaveGroup(grupo,netInter);
socket.close();
```

El nombre de una interface de red (*network interface*) indica el driver que usa, por ejemplo las interfaces em0, em1, etc. usan el driver de Intel, mientras que las interfaces bge0, bge1, etc. usan el driver de Broadcom.

Las interfaces em0 y bge0 son utilizadas para WAN y las interfaces em1 y bge1 son utilizadas para LAN.



- **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.



- **Jerarquía de memoria**

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 donde cada nivel representa una capa de hardware que almacena datos.



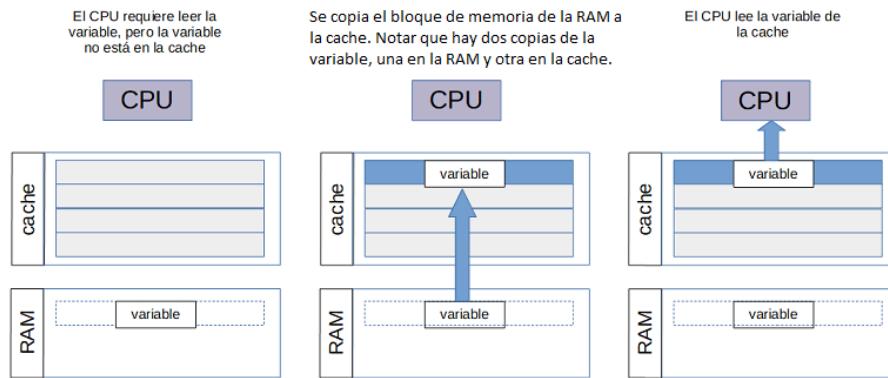
1. El CPU utiliza los **registros** para realizar operaciones aritméticas, lógicas y de control.
2. La memoria **cache** consiste en una memoria asociativa. Las memorias asociativas son muy rápidas, pero como son costosas, suelen ser de poca capacidad. La memoria cache puede estar dividida en varios 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

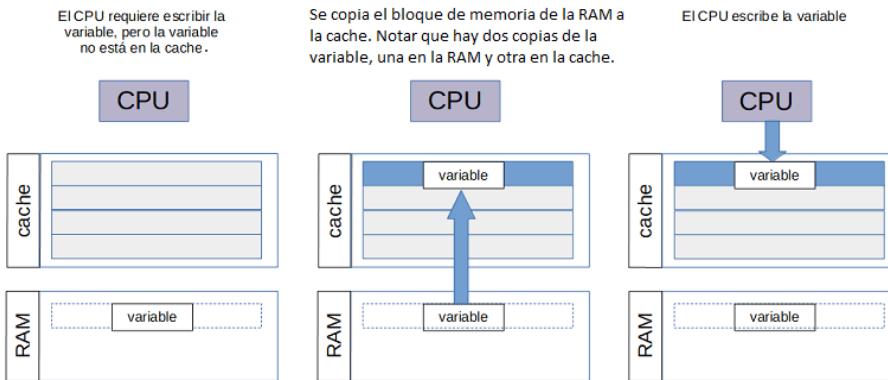
Cuando el CPU requiere leer una variable que se encuentra en la memoria RAM, busca la variable en la cache, si la variable no existe en la cache, copia el bloque de datos (que contiene a la variable) a la cache, entonces el CPU lee la variable de la cache.



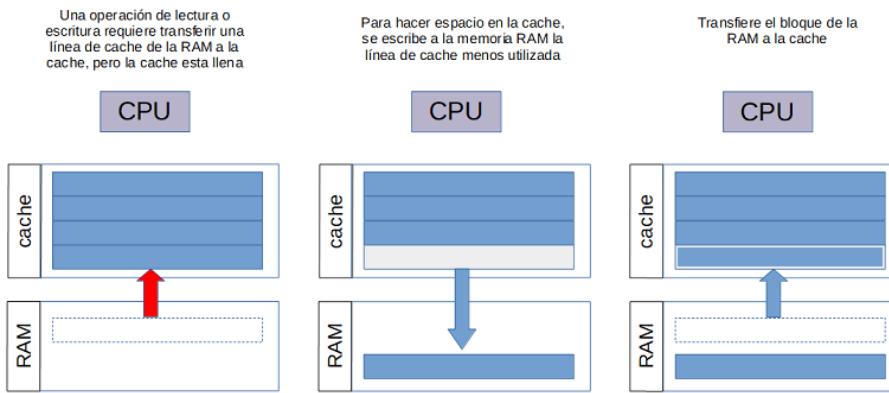
Al bloque de memoria que se transfiere de la RAM a la cache se le llama **línea de cache**. El tamaño de una línea de cache típicamente es de cientos de Kilobytes o Megabytes.

#### 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 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.

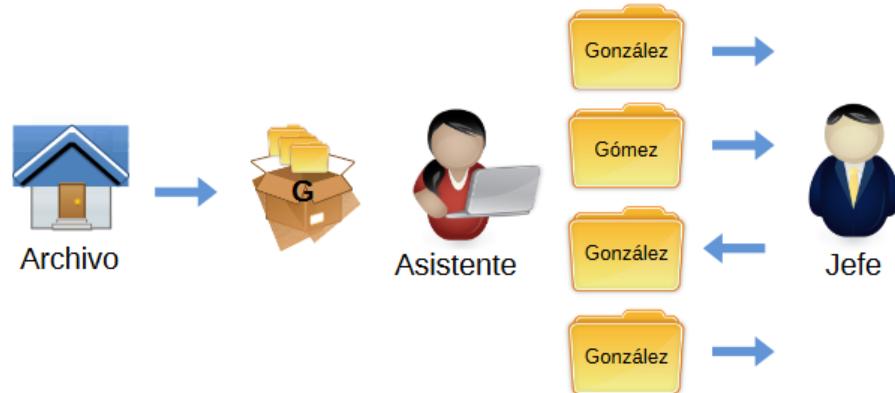


Entonces la asistente tiene que ir cuatro veces al archivo.

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.

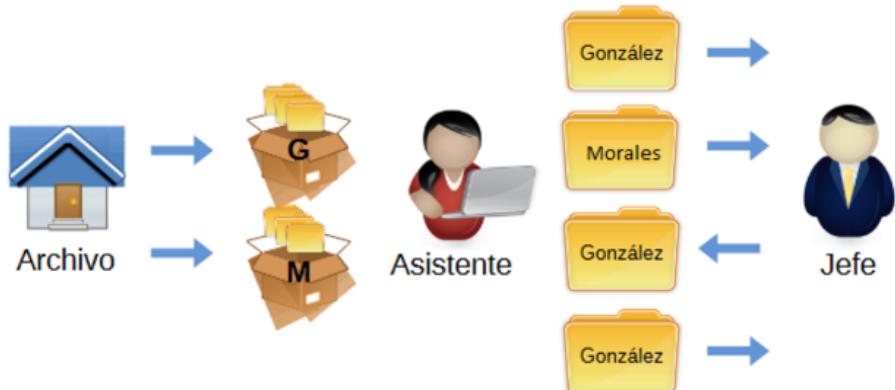
7. El jefe le pide a su asistente el expediente del Sr. González.
8. La asistente le da a su jefe el expediente del Sr. González.



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.

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.

En el caso 2, 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.

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++)
                for (int k = 0; k < N; k++)
                    C[i][j] += A[i][k] * B[k][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](#) que vimos en clase, para los siguientes valores de N: 100, 200, 300, 500, 1000.
2. Utilizando Excel o LibreOffice Calc 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).
3. ¿Por qué el segundo programa es más rápido que el primero?

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

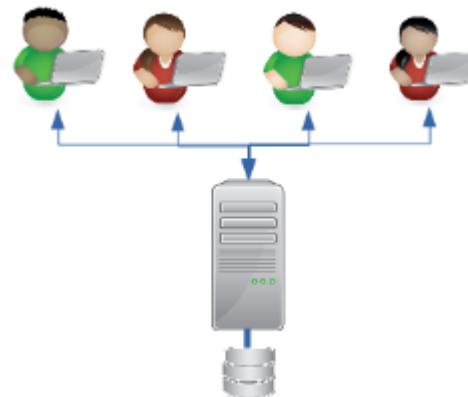


- Sistema centralizado y sistema distribuido

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

### Sistema centralizado

Un sistema centralizado es aquel donde 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.

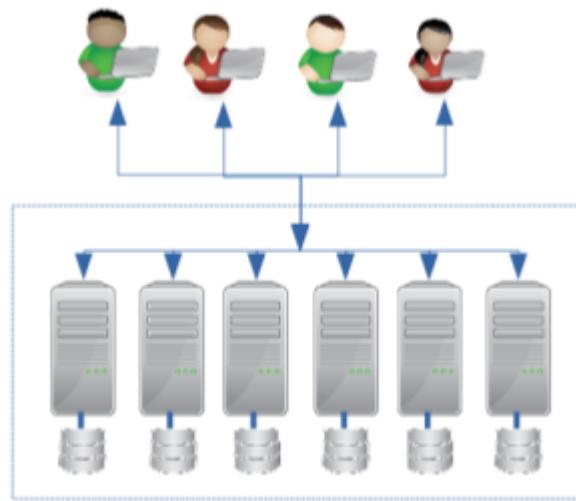
#### Una computadora stand-alone

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 alrededor de 290,000 (2009) computadoras buscando patrones "inteligentes" en señales obtenidas de radiotelescopios. El sistema alcanza los

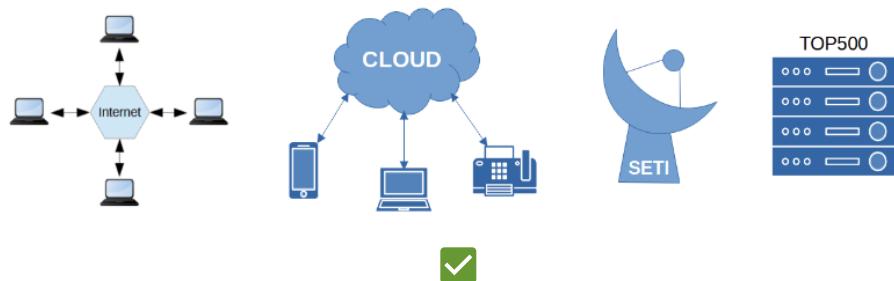
617 TFlop/s (1 TFlop/s =  $10^{12}$  operaciones de punto flotante por segundo).

TOP500

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

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

¿En qué lugar aparece la primera computadora con Windows en TOP500? (ver: [TOP500 List Statistics](#))



- Actividades individuales a realizar

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 "+Crear".
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 cada vez que se apague y se encienda la máquina virtual.

23. Para conectarnos a la máquina virtual vamos a utilizar el programa ssh disponible en Windows, Linux y MacOS.

24. En una ventana de comandos de Windows o una terminal de Linux o MacOS ejecutar el programa ssh así:

```
ssh usuario@ip
```

Donde **usuario** es el usuario que ingresamos en el paso 11, **ip** es la ip pública de la máquina virtual.

25. Para enviar o recibir archivos de la máquina virtual, se puede utilizar el programa sftp disponible en Windows, Linux y MacOS. Se ejecuta así:

```
sftp usuario@ip
```

Para enviar archivos se utiliza el comando put y para recibir archivos se utiliza el comando get.

Para mayor información sobre sftp ver:

<https://www.digitalocean.com/community/tutorials/how-to-use-sftp-to-securely-transfer-files-with-a-remote-server-es>

## Abrir un puerto de entrada

Para que los programas que ejecutan en la máquina virtual pueda recibir conexiones a través de un determinado puerto, es necesario crear una regla de entrada para el puerto.

Por ejemplo, vamos a abrir el puerto 50000 en la máquina virtual que acabamos de crear:

1. Entrar al portal de Azure
2. Seleccionar "Máquinas virtuales".
3. Seleccionar la máquina virtual.
4. Dar clic en "Redes".
5. Dar clic en el botón "Aregar 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" ingresar un nombre para la regla:  
Puerto\_50000

## 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 click 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 los videos:

[Create Ubuntu Linux on Azure using Azure Po...](#)



Reproducir Vídeo

Reproducir Vídeo



- **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 usuarios y aplicaciones acceder a los datos de manera estandarizada. Por ejemplo, un servicio que permite acceder a los archivos que residen en computadoras con diferentes sistemas operativos mediante nombres estandarizados, independientemente del tipos de nomenclatura (la forma de nombrar los archivos) 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 independientemente de su localización física. Por ejemplo,

<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". En este caso "https" indica que se utilizará el protocolo HTTPS para acceder al archivo.

**2.3 Transparencia de migración.** En algunos 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, un sistema que implementa la migración de datos de una computadora a otra de manera transparente, como es el caso de la memoria compartida distribuida (DSM, *Distributed Shared Memory*).

**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, un sistema que permite la migración transparente de procesos en ejecución de una computadora a otra como una estrategia de tolerancia a fallas o balance de carga, sin afectar a los usuarios que ejecutan dichos procesos.

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, finalmente 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 replicación de

los datos como 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 de manera concurrente. 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, si un sistema que se encuentra totalmente replicado, cuando 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 utilizando un administrador de tráfico de red.

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

#### Características de los sistemas abiertos

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 estándares 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 poder crecer mediante la incorporación de componentes fáciles de reemplazar y adaptar, como sería el caso de sistemas basados en OOP (programación orientada a objetos) donde es posible extender la funcionalidad de una clase mediante la herencia. 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, más unidades de almacenamiento o incrementar el ancho de banda de la red. Es decir, el sistema requiere escalar en tamaño.

En relativamente sencillo escalar el tamaño de un sistema distribuido agregando más servidores, en cambio un sistema

centralizado solo puede crecer hasta alcanzar el número máximo de CPUs que soporta la computadora, la cantidad máxima de memoria RAM, el número máximo de controladores de disco duro, el número máximo de controladores de red, etc.

#### 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. Por ejemplo, una empresa global puede instalar un centro de datos en cada región geográfica (América del Norte, América del Sur, Europa, Asia, África). Si la región es de alta demanda (como es el caso de América del Norte y Europa) la empresa puede 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, es decir, 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 también 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 cuando el usuario presiona un botón, si la petición es síncrona 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 (utilizando threads), lo cual garantiza que las aplicaciones seguirán 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 (exhibe localidad temporal), es conveniente tener una copia en una cache local. En el caso de que el archivo sea modificado en el servidor, entonces el servidor 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

Possiblemente 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. El aprovisionamiento es la creación del recurso (por ejemplo una máquina virtual), y el des-aprovisionamiento es la eliminación del recurso.

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

Es muy importante que cada 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 las opciones "+Crear" y "+Máquina virtual".

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 Server 2012.

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").

25. Dar click derecho al archivo de conexión y seleccionar "Modificar".

26. Seleccionar la pestaña "Recursos locales".

27. Dar click en el botón "Mas..."

28. Abrir la sección "Unidades".

29. Marcar la casilla "Windows (F:)"

30. Dar click en el botón "Aceptar".

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

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

33. 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.

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

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

36. 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.

37. Para desconectarse de la máquina virtual, dar click en el botón "X" del escritorio remoto. Notar que al cerrar el escritorio remoto la máquina virtual sigue ejecutando.

Ver el video:

2019 Create an Azure Virtual Machine running...



- **Requisitos de diseño de los sistemas distribuidos y tipos de sistemas distribuidos**

La clase de hoy vamos a ver los requisitos de diseño de los sistemas distribuidos 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 partitiona físicamente el sistema, la organización del sistema en subsistemas 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 deben 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

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

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

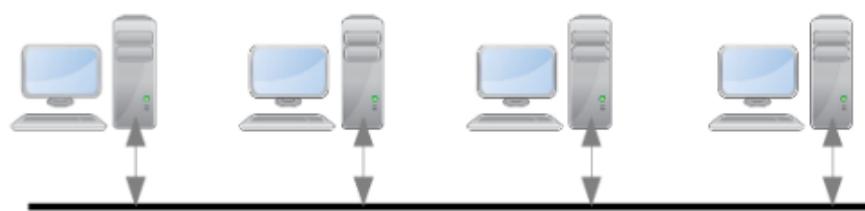
Por tanto, es muy importante definir los **requisitos de seguridad** para el sistema, entre otros: seguridad física del sistema, comunicación encriptada (SSL, TLS, HTTPS), utilización de usuarios no administrativos, configuración detallada de los permisos, programar para la prevención de ataques (p.e.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

### Tipos de sistemas distribuidos

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

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

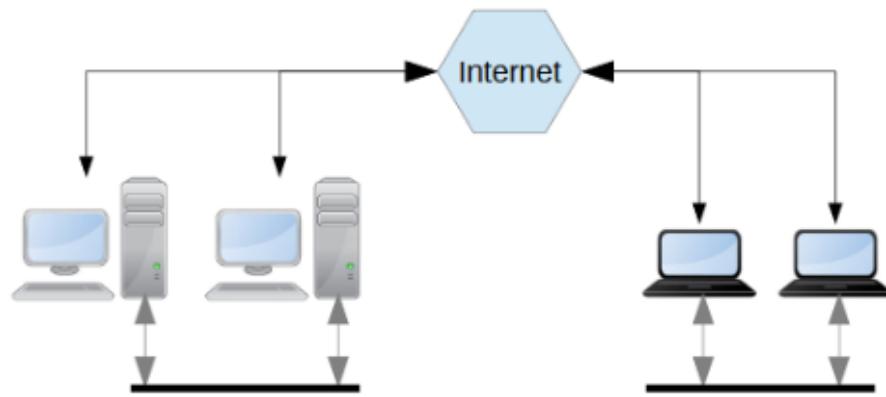
Un cluster es un conjunto de computadoras homogéneas con el mismo sistema operativo conectadas mediante una red local (LAN) de alta velocidad.



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

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

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



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

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



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

funciones administrativas para iniciar el recurso, apagar el recurso o bloquear el recurso.

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

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

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

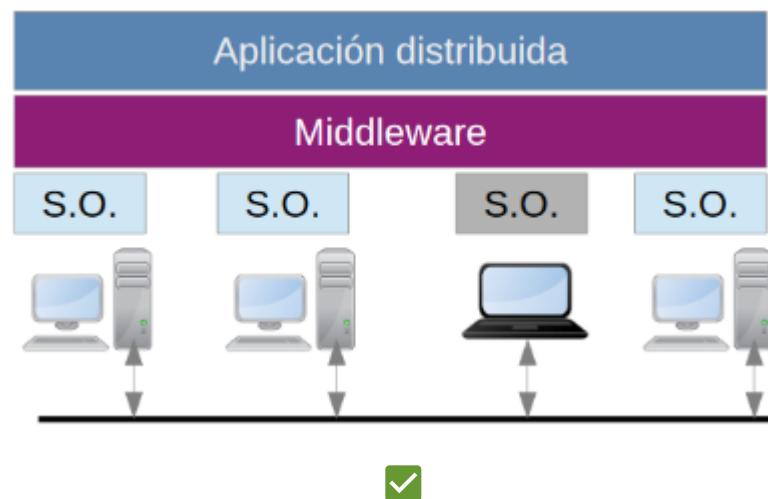
La **capa de aplicaciones** está compuesta por las aplicaciones que ejecutan dentro de la organización virtual.

### Middleware

Un middleware (software en medio) 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 diferentes sistemas operativos.



- **Actividades individuales a realizar**

Vamos a ver cómo crear la imagen de una máquina virtual con Ubuntu en Azure y cómo crear máquinas virtuales a partir de la imagen.

**Una máquina virtual generalizada** es un máquina virtual cuyo sistema operativo se ha despojado de la configuración específica y la configuración de usuarios.

Una imagen generalizada es la captura de un sistema operativo de una máquina virtual generalizada..

### Notas importantes

1. La captura de la imagen de una máquina virtual **inutiliza la máquina virtual** ya que una máquina virtual generalizada no se puede iniciar o modificar.
2. La generalización de una máquina virtual no implica que se borre toda la información confidencial que pudiera existir en la máquina virtual. Es muy importante considerar lo anterior si se va a re-distribuir la imagen de la máquina virtual.
3. La generalización de una máquina virtual no elimina el archivo /etc/resolv.conf (ver: [resolvconf](#))
4. La generalización de una máquina virtual deshabilita la contraseña de root.
5. La opción +user del comando waagent elimina la última cuenta creada en la máquina virtual incluyendo el directorio del usuario. Si se desea conservar el usuario y el directorio, no se deberá utilizar la opción +user al generalizar la máquina virtual mediante el comando waagent.
6. Para generalizar una máquina virtual con Windows se utiliza el programa sysprep.exe, ver: <https://docs.microsoft.com/en-us/azure/virtual-machines/generalize>.

7. Una imagen se cobra de acuerdo al espacio en disco que ocupa, ver: [Precios de Managed Disks](#).

### Crear la imagen de una máquina virtual con Ubuntu

Para generalizar la máquina virtual utilizaremos el agente **waagent** el cual elimina los datos específicos de la máquina virtual.

1. Crear una máquina virtual con Ubuntu.
2. Abrir una ventana cmd de Windows o una terminal de Linux o MacOs.
3. Ejecutar el programa ssh en la ventana, pasando como parámetros el usuario (por ejemplo ubuntu) y la ip pública de la máquina virtual:

ssh usuario@ip

4. Para generalizar la máquina virtual y eliminar la última cuenta de usuario creada incluyendo el directorio del usuario, ejecutar el comando:

sudo waagent -deprovision+user

Si se quiere conservar en la imagen la última cuenta de usuario creada, ejecutar el comando:

sudo waagent -deprovision

5. En el portal de Azure seleccionar la máquina virtual que se quiera capturar como imagen.

6. Seleccionar la opción "Captura".

6.1 En la opción "Compartir imagen con Shared Image Gallery" seleccionar "No, capturar solo una imagen administrada".

7. Marcar la casilla "Eliminar automáticamente esta máquina virtual después de crear la imagen", ya que una máquina virtual generalizada no se puede iniciar o modificar.

8. Ingresar el nombre de la imagen a crear.
9. Dar clic en el botón "Crear".
10. Dar clic en la campana de notificaciones para verificar que se haya creado la imagen de la máquina virtual.

### Crear una máquina virtual a partir de una imagen

1. En la sección "Todos los recursos" en el portal de Azure seleccionar la imagen de la máquina virtual.
2. Seleccionar la opción "+Crear máquina virtual".
3. Seleccionar el grupo de recursos dónde se creará la máquina virtual.
4. Ingresar el nombre de la máquina virtual.
5. Seleccionar el tamaño de la máquina virtual.
6. Seleccionar el tipo de autenticación (Clave pública SSH o Contraseña). En su caso, ingresar el usuario y contraseña.
7. Dar clic en el botón "Siguiente: Discos >"
8. Seleccionar el tipo de disco del sistema operativo (p.e. HDD estándar).
9. Si no hay otra configuración que se quiera realizar, dar clic en el botón "Revisar y crear".
10. Dar clic en el botón "Crear".

Referencias:

[Reproducir Vídeo](#)



- Actividades individuales a realizar

Ahora vamos a ver cómo crear la imagen de una máquina virtual con Windows y cómo crear máquinas virtuales a partir

de la imagen.

### Notas importantes

1. La captura de la imagen de una máquina virtual **inutiliza la máquina virtual** ya que una máquina virtual generalizada no se puede iniciar o modificar.
2. La generalización de una máquina virtual no implica que se borre toda la información confidencial que pudiera existir en la máquina virtual. Es muy importante considerar lo anterior si se va a re-distribuir la imagen de la máquina virtual.
3. La generalización de una máquina virtual elimina las variables de ambiente de sistema y de usuario.
4. Una imagen se cobra de acuerdo al espacio en disco que ocupa, ver: [Precios de Managed Disks](#).

### Crear la imagen de una máquina virtual con Windows

Para generalizar la máquina virtual utilizaremos el programa sysprep.exe el cual elimina los datos específicos de la máquina virtual.

1. Crear una máquina virtual con Windows Server 2012.
2. Conectarse a la máquina virtual utilizando escritorio remoto.
3. Para generalizar la máquina virtual y así eliminar la información de seguridad y las cuentas de usuarios, dar clic derecho en el botón de inicio de Windows. Entonces seleccionar "Command Promt (Admin)" y ejecutar en la ventana el siguiente programa:

\Windows\System32\Sysprep\sysprep.exe

4. Seleccionar "Enter System Out-of-Box Experience (OOBE)", checar la opción "Generalize", seleccionar

"Shutdown" en Shutdown Options y presionar el botón OK.

5. En el portal de Azure seleccionar la máquina virtual que se quiera capturar como imagen.

6. Seleccionar la opción "Captura".

6.1 En la opción "Compartir imagen con Shared Image Gallery" seleccionar "No, capturar solo una imagen administrada".

7. Marcar la casilla "Eliminar automáticamente esta máquina virtual después de crear la imagen", ya que una máquina virtual generalizada no se puede iniciar o modificar.

8. Ingresar el nombre de la imagen a crear.

9. Dar clic en el botón "Revisar y crear".

10. Dar clic en el botón "Crear".

10. Dar clic en la campana de notificaciones para verificar que se haya creado la imagen de la máquina virtual.

11. Eliminar la máquina virtual generalizada.

### Crear una máquina virtual a partir de una imagen

1. En la sección "Todos los recursos" en el portal de Azure seleccionar la imagen de la máquina virtual.

2. Seleccionar la opción "+Crear máquina virtual".

3. Seleccionar el grupo de recursos dónde se creará la máquina virtual.

4. Ingresar el nombre de la máquina virtual.

5. Seleccionar el tamaño de la máquina virtual.

6. Seleccionar el tipo de autenticación (Clave pública SSH o Contraseña). En su caso, ingresar el usuario y contraseña.

7. Dar clic en el botón "Siguiente: Discos >"

8. Seleccionar el tipo de disco del sistema operativo (p.e. HDD estándar).
9. Si no hay otra configuración que se quiera realizar, dar clic en el botón "Revisar y crear".
10. Dar clic en el botón "Crear".

**Reproducir Vídeo**



- **2. Sincronización y coordinación**

- **Sincronización en sistemas distribuidos**

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.

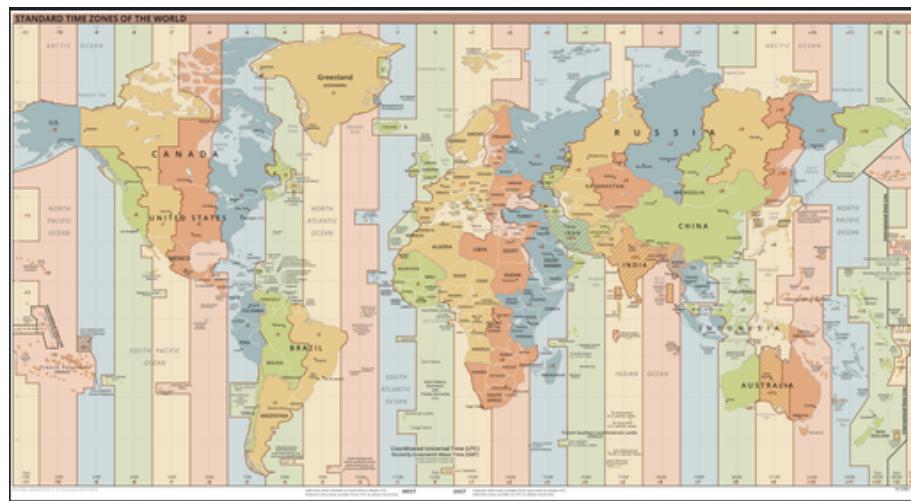
Para consolidar las compras a nivel mundial cada servidor debe enviar los datos a un servidor central. Sin embargo, no es posible ordenar las compras por fecha debido a dos situaciones:

1. Cada compra se ha registrado con 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).

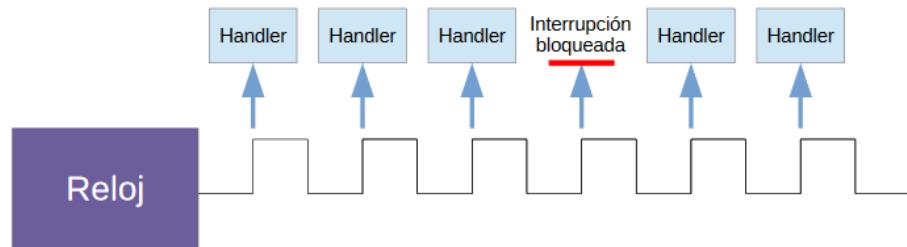
Más adelante veremos que éste orden puede establecerse utilizando relojes físicos (mecanismos que marcan el tiempo real) o bien relojes lógicos (contadores).

### Reloj 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 atrasé.
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 período 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, donde la gravedad es mayor, el tiempo se dilata (transcurre más lentamente) con respecto al tiempo medido en una montaña elevada donde 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 alrededor del mundo.

### Tiempo universal coordinado UTC

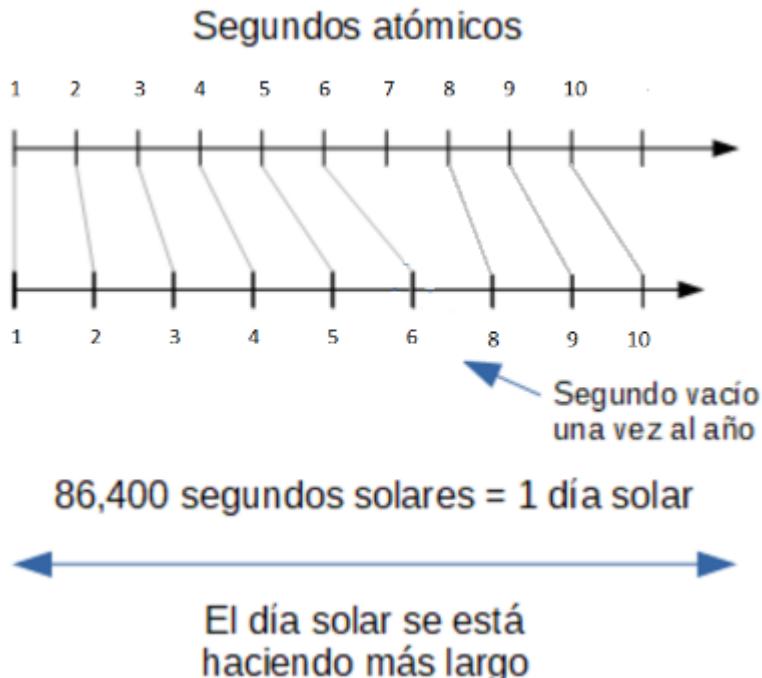
El tiempo universal coordinado UTC (*Coordinated Universal Time, CUT*) es el estándar de tiempo que regula actualmente el tiempo de los relojes a nivel internacional.

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 “adelantar” para alcanzar el 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.

Por ejemplo, en el siguiente diagrama se muestra cómo los segundos solares son más largos que los segundos atómicos, en este caso para sincronizar los segundos solares (UTC) con los segundos atómicos (TAI), se salta del segundo 6 al 8, es decir, el segundo "vacío" es el segundo 7:



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  $T_4$ .

Dado que la computadora A conoce sus tiempos locales  $T_1$  y  $T_4$ , si el servidor B envía sus tiempos locales  $T_3$  y  $T_2$  a la computadora A, y suponemos que  $T_{req}=T_{res}$ , entonces la computadora A puede calcular  $T_{res}$  a partir de las siguientes ecuaciones:

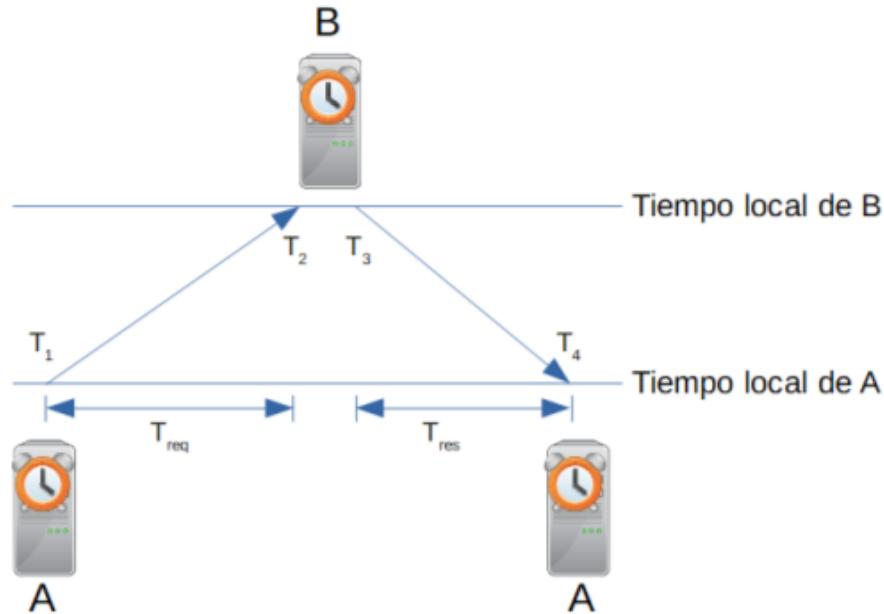
$$T_4-T_1 = T_{req}+T_{res}+(T_3-T_2) = 2T_{res}+(T_3-T_2)$$

Entonces la computadora A puede calcular  $T_{res}$  de la siguiente manera:

$$T_{res} = ((T_4-T_1)-(T_3-T_2))/2$$

Por tanto, cuando la computadora A recibe el mensaje de la computadora B, la computadora A cambiará su tiempo local a  $T_3+T_{res}$ .

Debido a que  $T_{req}$  no necesariamente es igual a  $T_{res}$ , sobre todo cuando la latencia en las comunicaciones es grande, no se puede garantizar que la computadora A tenga la misma hora que la computadora B.



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

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

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

```
sudo apt-get update
```

```
sudo apt-get install ntp
```

### Algoritmo de sincronización de relojes de Berkeley

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

El algoritmo de sincronización de relojoes de Berkeley es un procedimiento descentralizado dónde el servidor tiene una

función activa, ya que cada cierto tiempo inicia la sincronización de un grupo de computadoras.

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

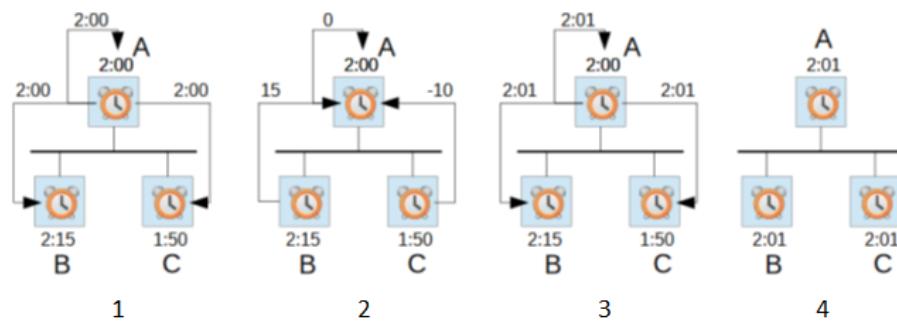
Por tanto, si un grupo de computadoras no se conectan con otras computadoras, es suficiente sincronizar los tiempos de las computadoras en el grupo, aún si el tiempo sincronizado no corresponde al tiempo real (ya que no hay una comunicación con otras computadoras).

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

El algoritmo de Berkeley es el siguiente:

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:



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

### Reloj lógico

Se define un **reloj lógico**  $C_i$  para un procesador (o proceso)

$P_i$  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).

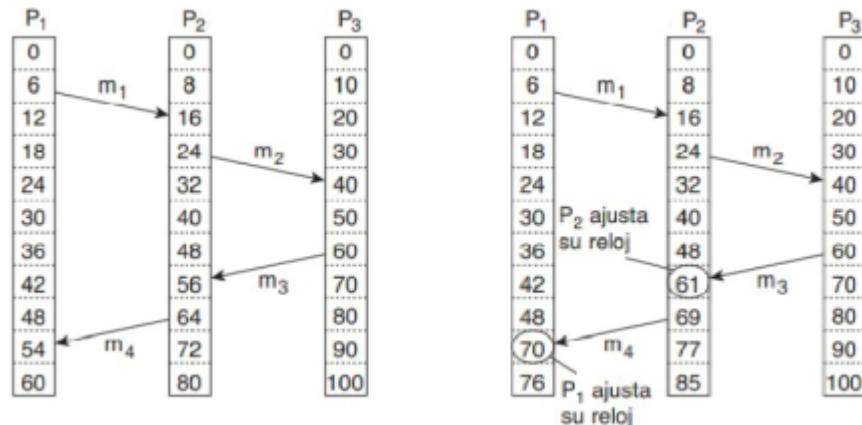
Supongamos que tenemos los procesadores  $P_1$ ,  $P_2$  y  $P_3$ . Cada procesador tiene un reloj lógico (contador) que se incrementa periódicamente mediante un thread.

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 *happends-before*.

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

Nota. Si el tiempo lógico de recepción de un mensaje **es igual** al tiempo lógico de envío, entonces el procesador que recibe el mensaje debe incrementar en uno su tiempo lógico.



- **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?

4. 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  $P_1$  se incrementa en 1, el reloj lógico del procesador  $P_2$  se incrementa en 5 y el reloj lógico del procesador  $P_3$  se incrementa en 10.

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

- El procesador P<sub>1</sub> envía el mensaje m<sub>1</sub> al tiempo 1, y el procesador P<sub>2</sub> lo recibe al tiempo 10.
- El procesador P<sub>2</sub> envía el mensaje m<sub>2</sub> al tiempo 15, y el procesador P<sub>3</sub> lo recibe al tiempo 40.
- El procesador P<sub>3</sub> envía el mensaje m<sub>3</sub> al tiempo 60, y el procesador P<sub>2</sub> lo recibe al tiempo 35.
- El procesador P<sub>2</sub> envía el mensaje m<sub>4</sub> al tiempo 40, y el procesador P<sub>1</sub> 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?



- Exclusión mutua

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

### Tipos de bloqueos

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

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

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

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

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

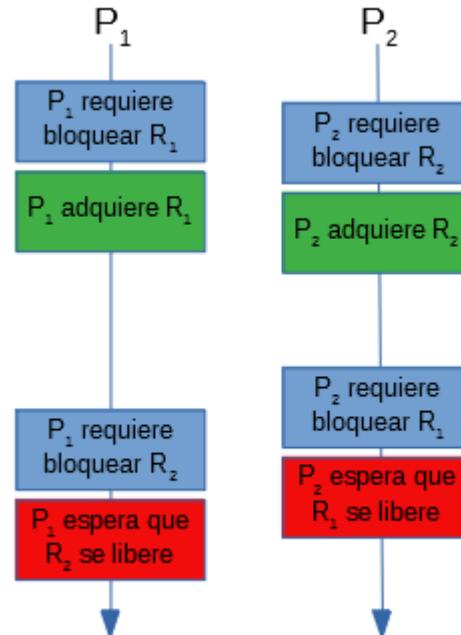
Por ejemplo, un bloqueo compartido sobre un archivo en el disco permite que varios procesadores puedan leer el archivo, pero no permite que ningún procesador escriba el archivo. Por otra parte, un bloqueo exclusivo sobre el archivo garantiza que solo un procesador pueda escribir y ningún otro procesador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

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

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



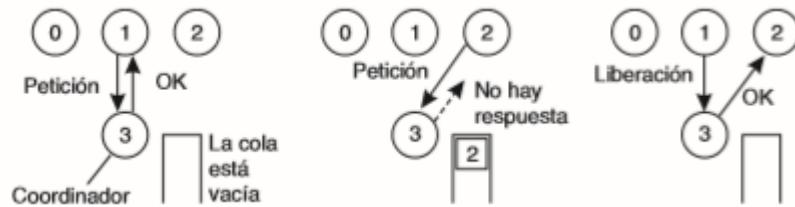
Podemos ver que el procesador  $P_1$  requiere bloquear el recurso  $R_1$ , debido a que  $R_1$  está desbloqueado el procesador  $P_1$  adquiere el recurso  $R_1$ . De la misma manera el procesador  $P_2$  requiere bloquear el recurso  $R_2$ , debido a que  $R_2$  está desbloqueado el procesador  $P_2$  adquiere el recurso  $R_2$ . Cuando el procesador  $P_1$  requiere bloquear el recurso  $R_2$ , no puede hacerlo ya que el procesador  $P_2$  lo tiene bloqueado, por tanto queda esperando a que  $R_2$  se libere. Así mismo, cuando el procesador  $P_2$  requiere bloquear el recurso  $R_1$ , no puede hacerlo ya que el procesador  $P_1$  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 donde 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.

En el artículo [An optimal algorithm for mutual exclusion in computer networks, Ricart y Agrawala](#), 1981, los autores proponen un algoritmo distribuido para exclusión mutua.

Una implementación del algoritmo sería la siguiente:

- **Cuando un nodo requiere acceso al recurso:**

- Envía un mensaje de petición a todos los nodos (incluso a sí mismo), el nodo adquiere el recurso cuando recibe "OK" de todos los nodos. El mensaje de petición incluye el ID del recurso, el número de nodo y el tiempo lógico.

- **Cuando un nodo recibe el mensaje de petición:**

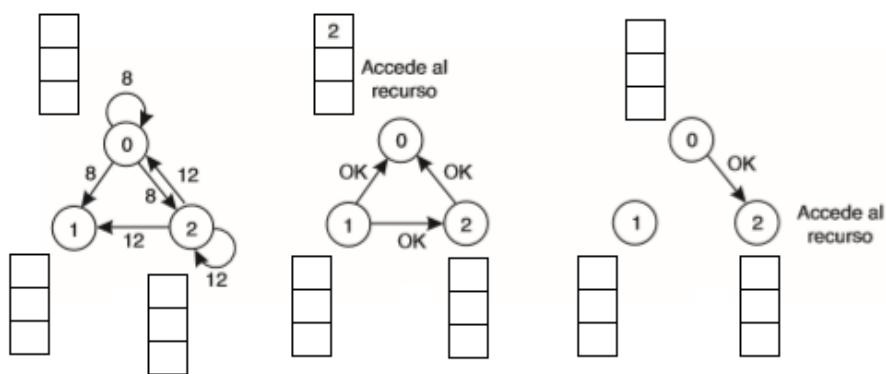
- Si el número de nodo coincide con el nodo actual, se envía a sí mismo un mensaje "OK".
- Si el nodo posee el recurso, coloca en la cola de espera el número de nodo que viene en el mensaje de petición.
- Si el nodo no está esperando por el recurso envía un mensaje "OK" al emisor del mensaje de petición.
- Si el nodo está esperando por el recurso, compara el tiempo lógico (T1) del mensaje de petición que recibió, con el tiempo lógico (T2) del mensaje de petición que previamente envió:
  - Si  $T1 < T2$  entonces envía el mensaje "OK" al nodo que envío el mensaje de petición.
  - Si  $T2 < T1$ , entonces coloca en la cola de espera el número de nodo del mensaje de petición recibido.
  - Si  $T1 = T2$  el nodo menor coloca en la cola de espera al nodo mayor, y el nodo mayor envía el mensaje "OK" al nodo menor.
- **Cuando un nodo libera el recurso:**
  - Envía "OK" a los nodos que están en la cola de espera.

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**.

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)

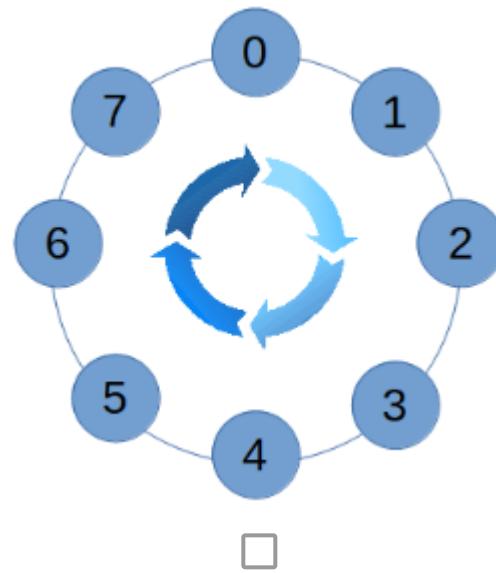
El algoritmo de token en anillo permite implementar la exclusión mutua en un sistema distribuido de manera muy simple, sin embargo tiene la desventaja de que requiere tener una conexión

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. Suponga que tiene tres nodos (0, 1 y 2) los cuales implementan el algoritmo distribuido para exclusión mutua de Ricart.

2. Cuando el tiempo lógico del nodo 0 es 8, el tiempo lógico del nodo 1 es 10 y el tiempo lógico del nodo 2 es 12 los tres nodos requieren bloquear un recurso simultáneamente

3. Describir los pasos a seguir de acuerdo al algoritmo distribuido para exclusión mutua.
4. Desarrollar un servidor multithread de acuerdo a las siguientes especificaciones:
  1. Declarar una variable global estática llamada **hosts** de tipo arreglo de strings.
  2. Declarar una variable global estática llamada **puertos** de tipo arreglo de enteros de 32 bits.
  3. Declarar una variable global estática llamada **num\_nodos** de tipo entero de 32 bits.
  4. Declarar una variable global estática llamada **nodo** de tipo entero de 32 bits.
  5. Escribir una clase llamada **Worker** subclase de la clase **Thread**. En el método **run()** de la clase **Worker**:
    1. Desplegar en la pantalla: "Inició el thread Worker".
  6. Escribir una clase llamada **Servidor** subclase de la clase **Thread**. En el método **run()** de la clase **Servidor**:
    1. Obtener un socket servidor vinculado al puerto **puertos[nodo]**.
    2. En un ciclo infinito:
      1. Esperar una conexión del cliente. Obtener un socket cliente.
      2. Crear un thread **Worker** pasando como parámetro el socket cliente.
      3. Iniciar la ejecución del thread **Worker**.
  7. En la función **main**:
    1. Asignar a la variable **nodo** el número de nodo actual que pasa como parámetro.
    2. Asignar a la variable **num\_nodos** el número de nodos, es decir, el número de parámetros "ip:puerto".
    3. Guardar las ip de los nodos en el arreglo **hosts**.
    4. Guardar los puertos de los nodos en el arreglo **puertos**.
    5. Crear un thread **Servidor**.

## 6. Iniciar la ejecución del thread Servidor.

El programa se ejecutará pasando como parámetros el número del nodo actual y una pareja "ip:puerto" para cada nodo (la ip del nodo y el puerto que abre).

Por ejemplo, si vamos a ejecutar tres nodos en la misma computadora (en diferentes ventanas), entonces al nodo 0 se deberá pasar los siguientes parámetros: 0 localhost:50000 localhost:50001 localhost:50002, al nodo 1 se deberá pasar los siguientes parámetros: 1 localhost:50000 localhost:50001 localhost:50002, y al nodo 2 se deberá pasar los siguientes parámetros: 2 localhost:50000 localhost:50001 localhost:50002. En este caso los puertos que abren los nodos son 50000, 50001 y 50002, respectivamente.

Para probar el programa ejecutarlo en **dos ventanas** de comandos de Windows o dos terminales de Linux o MacOS. En cada ventana ejecutar un nodo (una instancia del programa), en la primera ventana ejecutar el nodo 0 y en la segunda ventana ejecutar el nodo 1.

5. Tomando como base el programa anterior, implementar el algoritmo de Lamport para sincronizar relojes lógicos en tres nodos.

El programa deberá hacer lo siguiente:

1. Declarar una variable global estática llamada **reloj\_logico** de tipo entero de 64 bits (esta variable contiene el valor del reloj lógico en el nodo actual).
2. Escribir una función llamada **envia\_mensaje**, la cual recibirá los siguientes parámetros: **tiempo\_logico** de tipo entero de 64 bits, **host** de tipo string y **puerto** de tipo entero de 32 bits. Esta función hará lo siguiente:
  1. La función **envia\_mensaje** se conectará al servidor en el host a través del puerto. Obtener un socket cliente.

2. La función `envia_mensaje` deberá implementar re-intentos de conexión previendo que el servidor no se encuentre en ejecución.
  3. Una vez establecida la conexión, utilizando el socket cliente se deberá enviar al servidor el `tiempo_logico` que pasó como parámetro a la función `envia_mensaje`.
  4. Se deberá cerrar el socket cliente.
3. Escribir una clase llamada **Reloj** subclase de la clase **Thread**.  
En el método `run()` de la clase **Reloj**:
1. En un ciclo infinito:
    1. Desplegar el valor de la variable `reloj_logico`.
    2. En el nodo 0 sumar 4 a la variable `reloj_logico` cada segundo.
    3. En el nodo 1 sumar 5 a la variable `reloj_logico` cada segundo.
    4. En el nodo 2 sumar 6 a la variable `reloj_logico` cada segundo.
    5. Es necesario utilizar un lock para acceder la variable `reloj_logico` debido a que esta variable se actualiza cada vez que se recibe un mensaje.
  4. Cada vez que se reciba un mensaje (en el método `run` de la clase **Worker**):
    1. Declarar una variable llamada `tiempo_recibido` de tipo entero de 64 bits.
    2. Recibir el tiempo lógico como entero de 64 bits y asignar este tiempo a la variable `tiempo_recibido`.
    3. Implementar el algoritmo de Lamport para sincronizar el reloj lógico (comparar `tiempo_recibido` con `reloj_logico`).
    4. Es necesario un lock para acceder la variable `reloj_logico` debido a que esta variable se actualiza en el thread **Reloj**.
  5. Al final de la función **main** agregar las siguientes instrucciones:
    1. Implementar una barrera que espere que todos los nodos estén en ejecución (no se debe ejecutar la siguiente

instrucción hasta que todos los nodos estén en ejecución).

2. Crear un thread Reloj.
3. Iniciar la ejecución del thread Reloj.
4. Esperar la terminación del thread Servidor.

6. Tomando como base el programa anterior, implementar el algoritmo distribuido de exclusión mutua de Ricart. Al final del método **main** hacer lo siguiente:

1. Esperar un segundo.
2. Bloquear el recurso (enviar petición a todos los nodos).
3. Esperar a que el nodo actual adquiera el recurso.
4. Esperar tres segundos
5. Desbloquear el recurso (enviar OK a los nodos en la cola de espera).



- Coordinación

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 García-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.

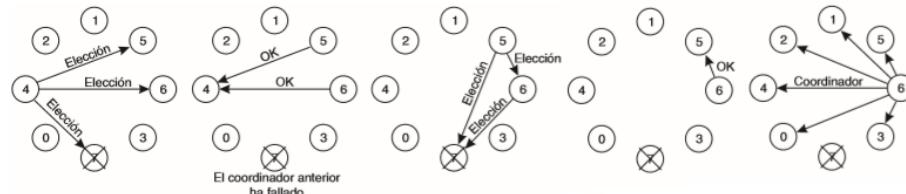
El nodo 4 envía un mensaje de elección a los nodos 5, 6 y 7.

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**

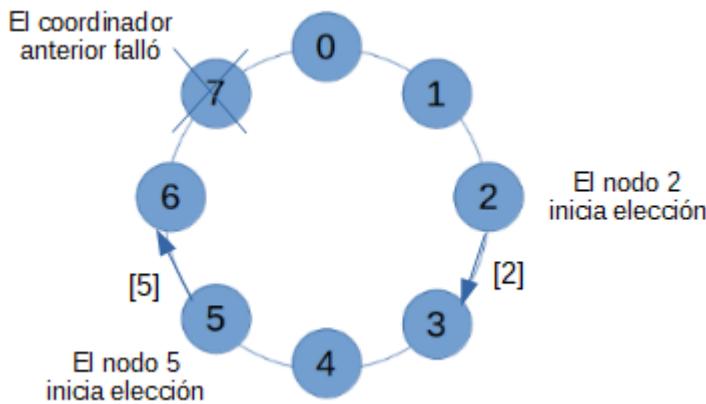
En el algoritmo de elección en anillo se supone que los nodos están conectados en una **topología lógica de anillo** ordenados por número de nodo, de menor a mayor.

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

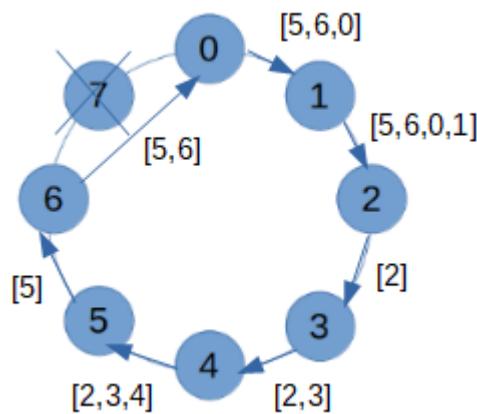
1.  $P_n$  envía un mensaje de elección al nodo  $P_{n+1}$  agregando al mensaje su número de nodo  $n$ . Si el nodo  $P_{n+1}$  no responde, entonces el nodo  $P_n$  envía el mensaje al nodo  $P_{n+2}$  y así sucesivamente hasta encontrar un nodo que responda.
2. Cuándo un nodo  $P_m$  recibe un mensaje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

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

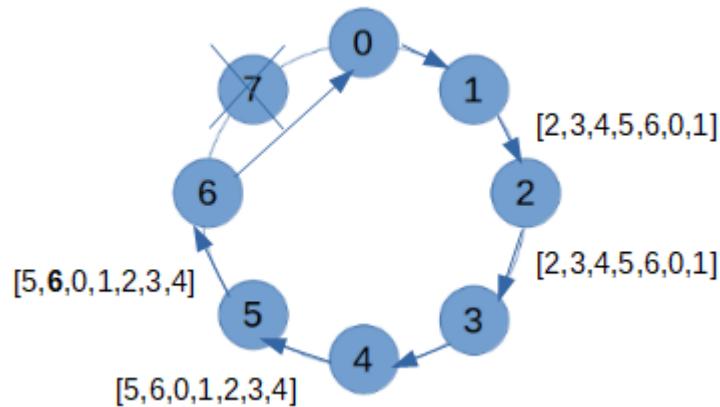
El nodo 2 envía un mensaje de elección al nodo 3, incluyendo en el mensaje su número de nodo. El nodo 5 envía un mensaje de elección al nodo 6 incluyendo su número de nodo.



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

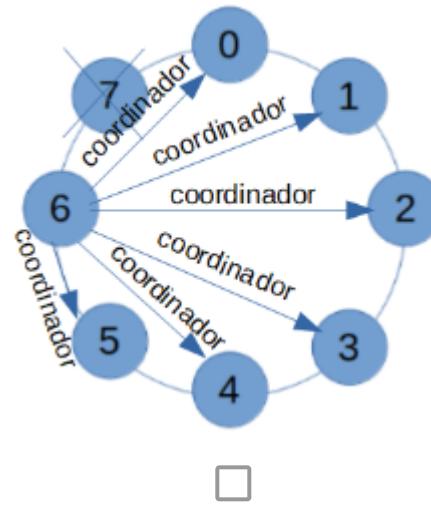


Eventualmente el nodo 2 recibirá el mensaje  $[2,3,4,5,6,0,1]$ , debido a que el nodo 2 no es el mayor nodo en el mensaje, entonces el nodo 2 re-envía el mensaje al nodo 3.



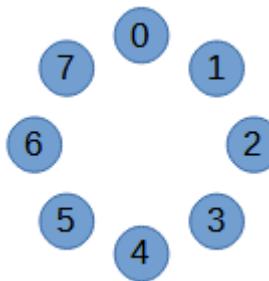
Por otra parte, eventualmente el nodo 5 recibirá el mensaje  $[5,6,0,1,2,3,4]$ , debido a que el nodo 5 no es el mayor nodo en el mensaje, entonces el nodo 5 envía el mensaje de elección al nodo 6. Cuando el nodo 6 recibe el mensaje  $[5,6,0,1,2,3,4]$  encuentra que es el mayor nodo, por tanto se erige como coordinador.

Finalmente, el nodo 6 envía un mensaje de coordinador a todos los nodos.



- Actividades individuales a realizar

A partir del servidor multithread desarrollado en la actividad anterior, escribir un programa en Java que muestre el uso del algoritmo del abusón (bully) para la elección de un coordinador en ocho nodos:



El programa deberá hacer lo siguiente:

- Declarar una variable global estática llamada **coordinador\_actual** de tipo entero de 32 bits.
- En el método run de la clase Worker:
  - Declarar una variable llamada **mensaje** de tipo string.
  - Recibir del cliente una string y asignarla a la variable **mensaje**.
  - Si el mensaje recibido es "ELECCION":
    - Enviar al cliente la string "OK".
    - Invocar la función eleccion(nodo).
- Si el mensaje recibido es "COORDINADOR":
  - Recibir del cliente un número entero de 32 bits y asignarlo a la variable **coordinador\_actual**

3. Escribir una función llamada **envia\_mensaje\_elección**, la cual recibirá los siguientes parámetros: **host** de tipo string y **puerto** de tipo entero de 32 bits. La función regresará una string. Esta función hará lo siguiente:
  1. La función intentará conectarse al host a través del puerto.
  2. Si se establece la conexión con el servidor:
    1. Enviar al servidor la string "ELECCION"
    2. Recibir del servidor una string.
    3. Cerrar el socket cliente.
    4. La función regresará la string que recibió del servidor.
  3. Si no se establece la conexión con el servidor:
    1. La función regresará una string vacía ("").
4. Escribir una función llamada **envia\_mensaje\_coordinador**, la cual recibirá los siguientes parámetros: **host** de tipo string y **puerto** de tipo entero de 32 bits. Esta función hará lo siguiente:
  1. La función intentará conectarse al host a través del puerto.
  2. Si se establece la conexión con el servidor:
    1. Enviar al servidor la string "COORDINADOR"
    2. Enviar al servidor el valor de la variable **nodo**.
    3. Cerrar el socket cliente.
    4. Terminar la función.
  3. Si no se establece la conexión con el servidor:
    1. Terminar la función.
5. Escribir una función llamada **elección**, la cual recibirá el parámetro **nodo** de tipo entero de 32 bits (el número de nodo que inicia un proceso de elección):
  1. Esta función deberá implementar el algoritmo de elección del abusón utilizando las funciones **envia\_mensaje\_elección** y **envia\_mensaje\_coordinador**.

6. Al final de la función **main** agregar las siguientes instrucciones:
  1. Implementar una barrera que permita esperar que todos los nodos estén en ejecución (no se debe ejecutar la siguiente instrucción hasta que todos los nodos estén en ejecución).
  2. Esperar 3 segundos.
  3. Si el nodo actual es el 7, entonces terminar el programa (esto simula que el nodo 7 deja de funcionar después de 3 segundos de iniciado).
  4. Si el nodo actual es el 4, invocar la función elección(4) (el nodo 4 inicia el proceso de elección).
  5. Esperar la terminación del thread Servidor.

Ejecutar el programa en ocho ventanas de comandos de Windows, terminales de Linux o MacOS. En cada ventana ejecutar una instancia del programa, en la primera ventana ejecutar el nodo 0, en la segunda ventana ejecutar el nodo 1, en la tercera ventana ejecutar el nodo 2, etc.



- **Comunicación en grupo confiable**

### 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 sistema 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 cuando 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.

### Comunicación unicast confiable

Para establecer la comunicación unicast confiable se utiliza generalmente el protocolo TCP, el cual implementa la retransmisión de mensajes para ocultar las fallas por omisión. Las fallas por congelación (cuando se produce la desconexión) no son ocultadas por el protocolo TCP, sin embargo el cliente es informado de la falla de manera que pueda re-conectarse (ver el programa [Cliente2.java](#)).

### Comunicación multicast confiable

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

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

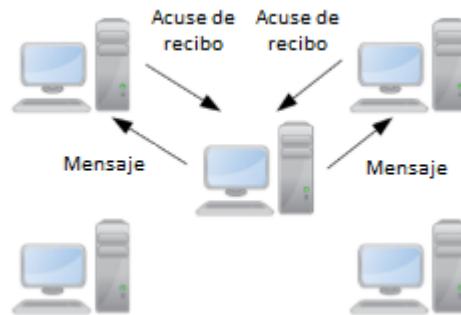
La comunicación punto a punto confiable se implementa con relativa facilidad mediante TCP, sin embargo la comunicación multicast confiable resulta mucho más complicada.

Una primera aproximación para implementar la comunicación multicast confiable es la utilización de múltiples conexiones

punto a punto, sin embargo esta solución resulta poco eficiente debido a las características del protocolo TCP.

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

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



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

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



### Multicast atómico

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

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

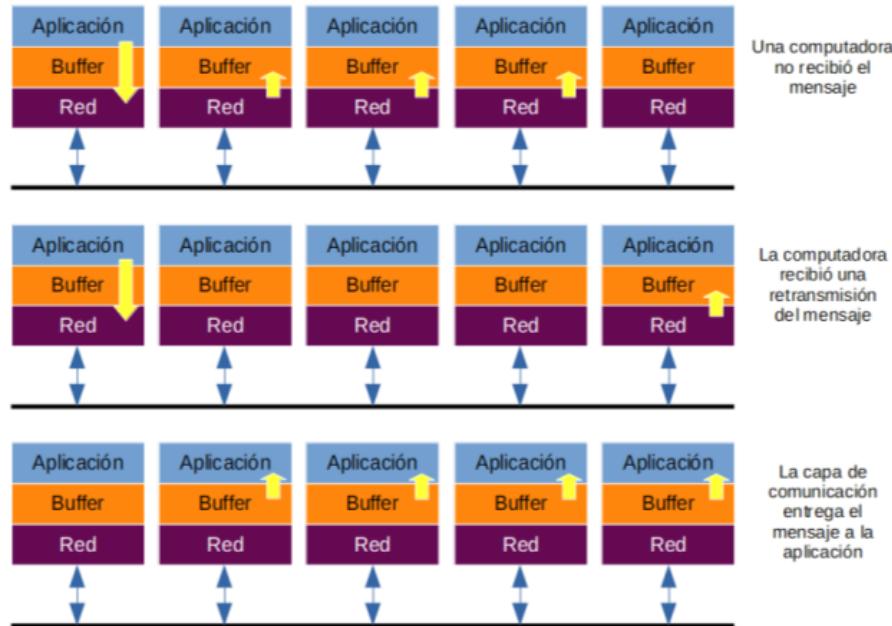
Por ejemplo, si un archivo es replicado en un grupo de computadoras, los cambios que se realizan al archivo deben ser replicados en **todas** las computadoras que forman parte del grupo. Si bien la consistencia del archivo es un requerimiento no funcional del sistema de archivos distribuido, este requerimiento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

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

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

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

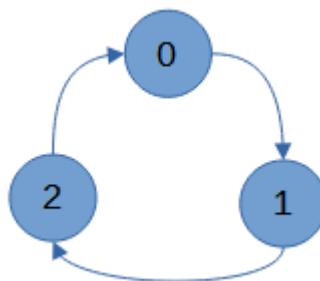
Entonces los mensajes entrantes deberán permanecer en un almacén temporal (buffer) en la capa de comunicaciones, y solo en el caso de que todas las computadoras confirmen la recepción del mensaje, entonces y solo entonces el mensaje será entregado a las aplicaciones.



- Actividades individuales a realizar

A partir del servidor multthead desarrollado anteriormente, escribir un programa en Java que implemente el algoritmo de exclusión mutua mediante token en anillo.

El programa funcionará en tres nodos:



El programa deberá hacer lo siguiente:

1. En el método run() de la clase Worker:

1. Declarar una variable llamada **token** de tipo entero de 64 bits.
  2. Recibir del cliente un número entero de 64 bits y asignarlo a la variable **token**.
  3. Desplegar el valor de la variable **token**.
  4. Enviar el valor de la variable **token** al siguiente nodo en el anillo.
- 
2. Implementar el algoritmo de exclusión mutua de token en anillo. Al final de la función **main** agregar las siguientes instrucciones:
    1. Si el nodo actual es cero, enviar 1 al nodo 1.
    2. Esperar 3 segundos.
    3. Adquirir el bloqueo.
    4. Desplegar un letrero que indique que el nodo adquirió el bloqueo.
    5. Esperar 3 segundos.
    6. Desbloquear.
    7. Desplegar un letrero que indique que el nodo liberó el bloqueo.

Ejecutar el programa en tres ventanas de comandos de Windows, tres terminales de Linux o MacOS. En cada ventana ejecutar una instancia del programa, en la primera ventana ejecutar el nodo 0, en la segunda ventana ejecutar el nodo 1 y en la tercera ventana ejecutar el nodo 2.



- **3. Sistemas basados en objetos distribuidos**

- 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 recibirlós.

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.

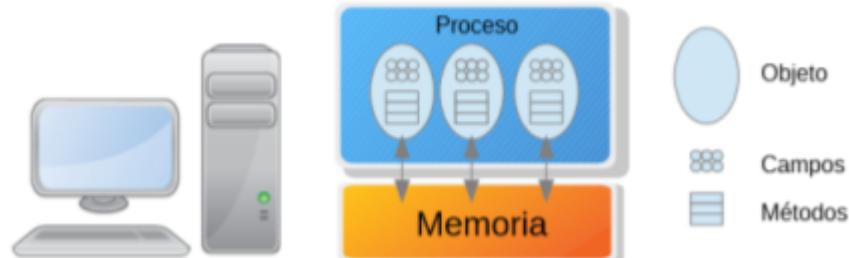
El paradigma de paso de mensajes es **orientado a datos**.

### 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 donde 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.



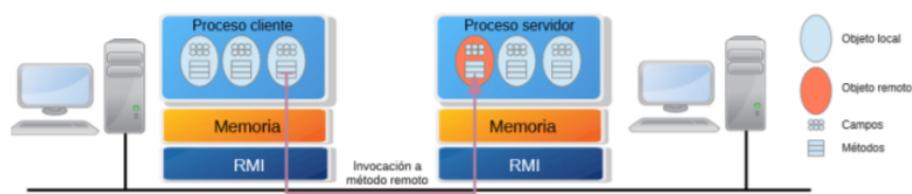
### Objetos remotos

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 deberá 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 siguientes 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) throws
RemoteException
{
    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

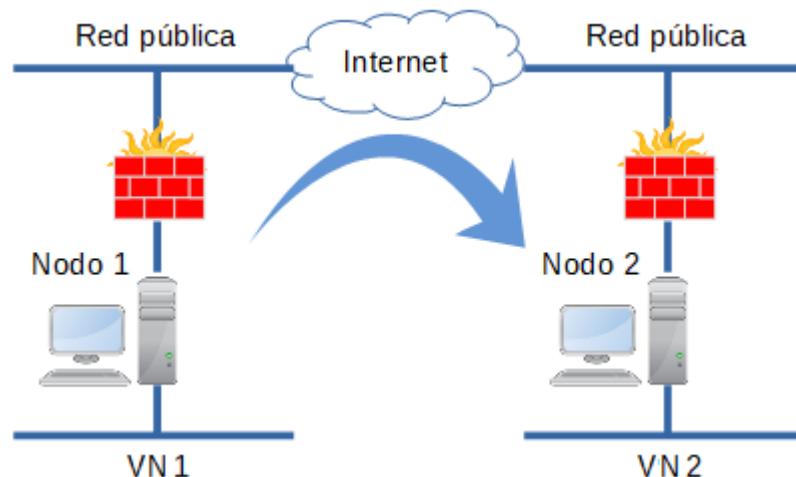
1. Compilar la interface InterfaceRMI.java, la clase ClaseRMI.java y los programas ClienteRMI.java y ServidorRMI.java.
2. En una ventana de comandos de Windows (o una terminal de Linux) ejecutar el programa rmiregistry.
3. En una ventana de comandos de Windows (o una terminal de Linux) ejecutar el programa 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.



- El día de hoy vamos a explicar cómo ejecutar una aplicación Java RMI en la nube.

### 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

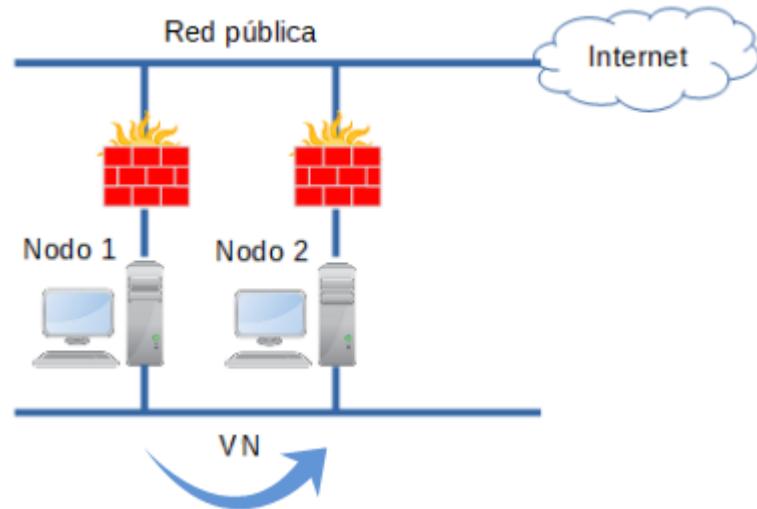
definen qué direcciones públicas y qué puertos se pueden conectar a la máquina virtual, mientras que las reglas de salida definen a qué direcciones públicas y a qué puertos se puede conectar la máquina virtual.

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.



- 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

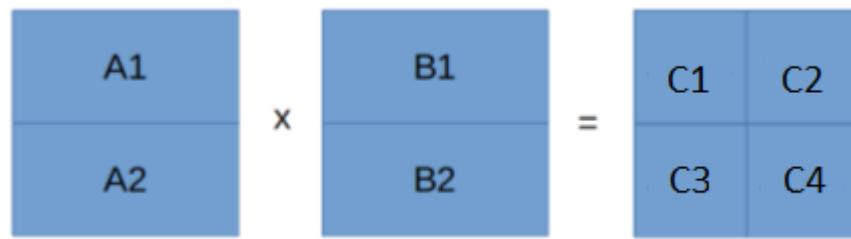
controlar explícitamente la serialización, el envío y la des-serialización de los datos, además de la lógica particular del sistema.

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  $C = A \times B$  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):

```
static int[][] multiplica_matrices(int[][] A, int[][] B) {
    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 separa\_matriz() el cual utilizaremos para obtener las matrices A1, A2, B1 y B2:

```
static int[][] separa_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 separa\_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 = separa_matriz(A,0);
int[][] A2 = separa_matriz(A,N/2);
int[][] B1 = separa_matriz(B,0);
int[][] B2 = separa_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**

1. Desarrollar un programa en Java que multiplique dos matrices cuadradas de tamaño NxN (N=6), utilizando las funciones separa\_matriz(), multiplica\_matrices() y acomoda\_matriz() en forma local.
2. Probar el programa utilizando una ventana de cmd en Windows o una terminal de Linux o MacOS.
3. Tomando como base el programa anterior, desarrollar un sistema (interface, clase, servidor y cliente) que multiplique dos matrices cuadradas tamaño NxN (N=6) utilizando RMI, se deberá ejecutar el método multiplica\_matrices() en forma remota, es decir, la interface deberá incluir el prototipo de esta función, así mismo, el programa servidor deberá incluir el código del método multiplica\_matrices().
4. Probar el programa utilizando tres ventanas de cmd en Windows o tres terminales de Linux o MacOS, en una ventana ejecutará rmiregistry, en otra ventana ejecutará el servidor y en otra ventana va a ejecutar el cliente.



- 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.

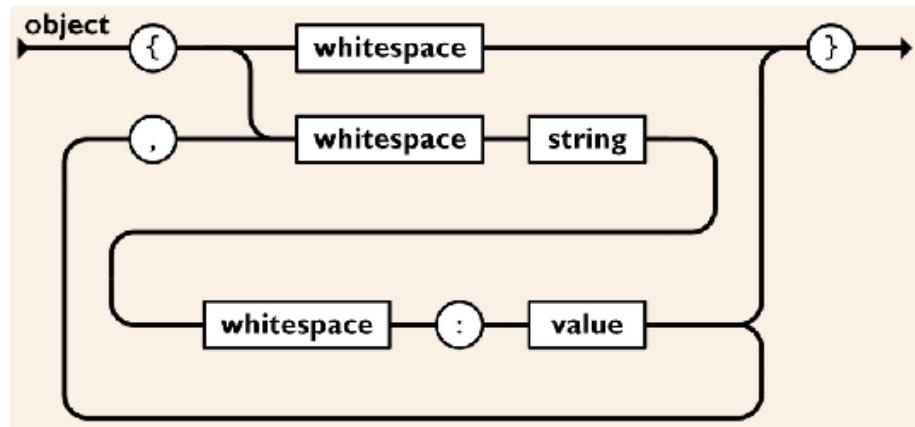
JSON es un formato independiente del lenguaje de programación, de manera que es posible escribir fácilmente programas en cualquier lenguaje que creen mensajes en formato JSON así como programas que lean mensajes en formato JSON.

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 “}”.

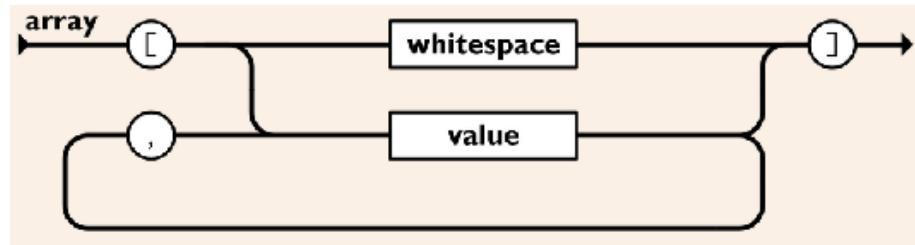
La sintaxis de un objeto es la siguiente:



Fuente: [www.json.org](http://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 ”]”.

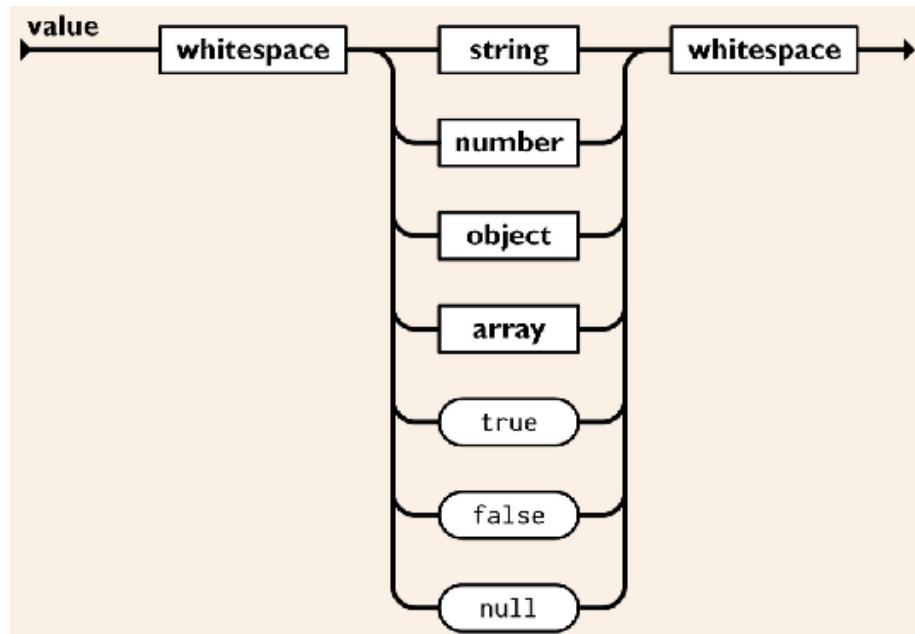
La sintaxis de un arreglo es la siguiente:



Fuente: [www.json.org](http://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](http://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
-----------	-----------	-------------

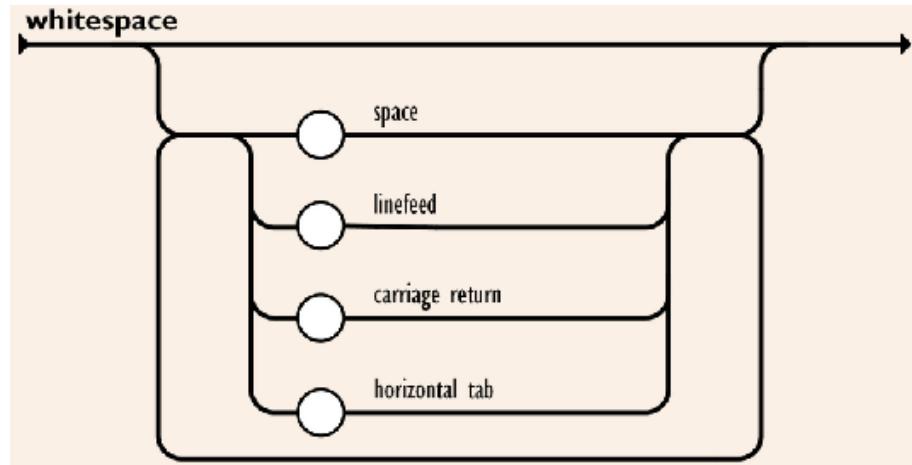
---

\"		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](http://www.json.org)

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

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

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

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

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

Para ejecutar el programa en Windows:

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

Para ejecutar el programa en Linux:

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

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 des-serializar 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



- **4. Servicios de nombres, archivos y replicación**

- **Nombre, identificador y dirección**

Un **nombre**, en el contexto de un sistema distribuido, es una cadena de caracteres que hace referencia a una entidad o recurso (servidor, impresora, archivo, disco, página web, etc).

Se entiende como **punto de acceso** a una entidad como el dispositivo desde el cual se tiene acceso a la entidad. Por ejemplo, una computadora es el punto de acceso a los archivos que contiene. Al nombre de un punto de acceso se le llama **dirección**.

Por ejemplo, la **dirección** de un servicio que ofrece un servidor es el *end point* del servicio (dirección IP y puerto).

Un **identificador** es un nombre que tiene las siguientes propiedades:

1. El identificador hace referencia a una entidad como máximo.
2. Cada entidad tiene un identificador.
3. Un identificador siempre hace referencia a la misma entidad.

En general una dirección no es un identificador, ya que la dirección de una entidad puede cambiar (por ejemplo las direcciones IP dinámicas son modificadas por los proveedores de servicios de Internet).

Por otra parte, el nombre de dominio es en efecto un identificador.

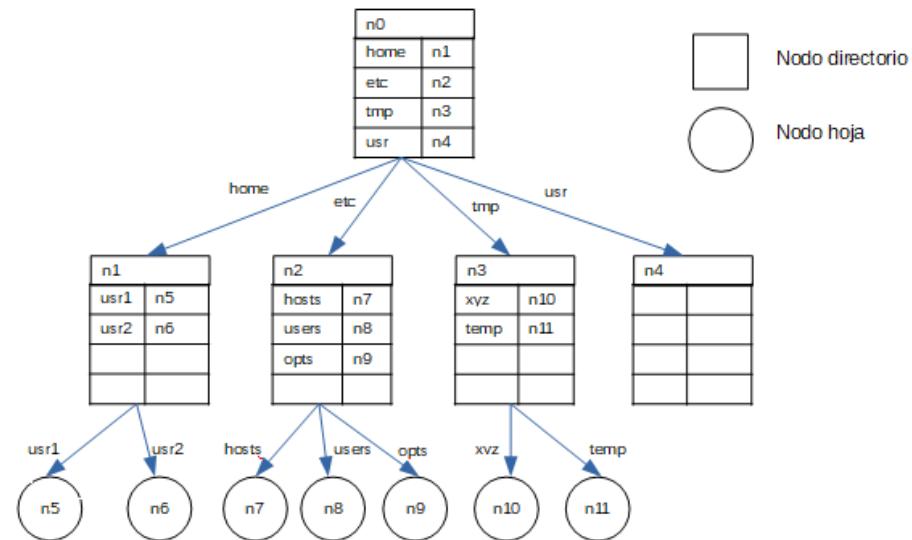
### Espacios de nombres

Los nombres se organizan en una estructura llamada **espacio de nombres**.

Un espacio de nombres es un grafo etiquetado dirigido con dos tipos de nodos: nodos hoja y nodos directorio.

Un grafo etiquetado dirigido está compuesto por nodos conectados por arcos (o aristas). Los arcos tienen una dirección (cada arco tiene una flecha) y una etiqueta. Un arco es de **salida** si apunta afuera del nodo y es de **entrada** si apunta adentro del nodo.

Por ejemplo, el siguiente grafo representa un sistema de archivos:



Un **nodo directorio** contiene una tabla llamada **tabla de directorio** la cual contiene pares (etiqueta, identificador de nodo), cada etiqueta corresponde a un arco de salida del nodo directorio y cada identificador corresponde al nombre del nodo al que conecta.

Un **nodo hoja** representa una entidad con un nombre, se trata de un nodo hoja ya que no tiene arcos de salida. Este tipo de nodo guarda información sobre la entidad.

Un nodo directorio puede conectar a otro nodo directorio o a un nodo hoja. Al nodo que sólo tiene arcos de salida se le llama **nodo raíz**. Un grafo que representa un espacio de nombres puede tener más de un nodo raíz, aunque por simplicidad los espacios de nombres tienen un sólo nodo raíz.

En un grafo de nombres cada nodo hoja corresponde a una entidad, en el ejemplo anterior cada entidad es un archivo.

El nombre completo de un nodo hoja en un espacio de nombres se compone de la secuencia de etiquetas de los arcos, iniciando con el nombre del nodo raíz:

N: etiqueta<sub>1</sub>, etiqueta<sub>2</sub>, ..., etiqueta<sub>n</sub>

Donde N es el primer nodo de la ruta. Por ejemplo, el archivo "xyz" tiene la siguiente ruta desde el nodo n0:

n0: tmp, xyz

A esta secuencia se le llama **nombre de ruta**.

Si el primer nodo de la ruta es el nodo raíz se le llama **nombre de ruta absoluto**, de otra manera, se le llama **nombre de ruta relativo**.

En un sistema de archivos, al primer nodo se le llama generalmente disco lógico y la secuencia de etiquetas se separa por una diagonal "/", entonces el nombre de ruta es una cadena de caracteres que corresponde a la secuencia de etiquetas.

En general, los recursos tales como procesos, dispositivos de E/S, memoria, etc. forman parte de un espacio de nombres, por tanto, también se puede aplicar cadenas de caracteres como nombres de ruta.

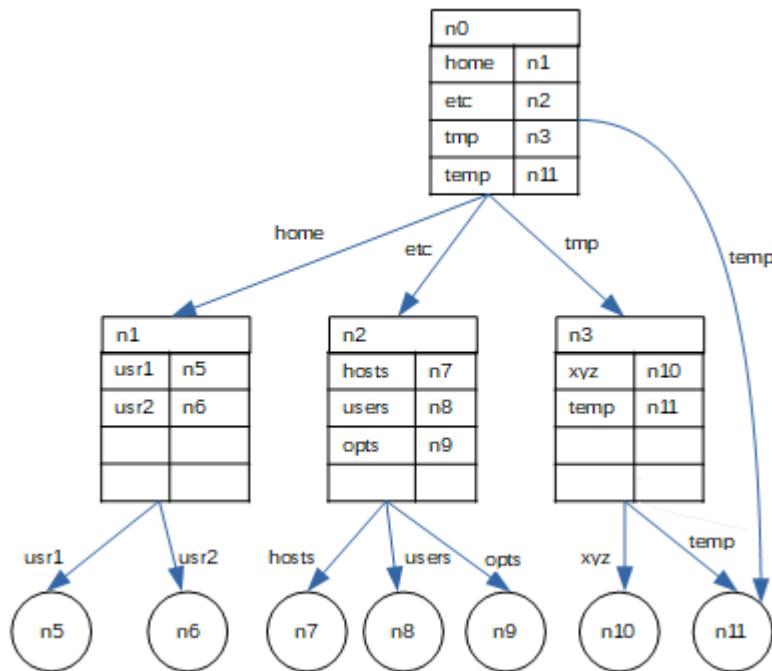
## Árbol de nombres

Un árbol de nombres es un grafo de nombres dónde cada nodo tiene exactamente un arco de entrada (un nodo padre), excepto el nodo raíz el cual no tiene arco de entrada.

El ejemplo mostrado anteriormente es un árbol de nombres.

## Grafo dirigido no cíclico

En un grafo dirigido no cíclico un nodo puede tener más un arco de entrada, pero no se permite que haya ciclos.



En este grafo podemos ver que el nodo 11 puede ser accedido utilizando dos rutas. Este es el caso de los archivos con vínculos (*links*) en los sistemas de archivos modernos.

Podemos ver que no se trata de un árbol de nombres debido a que en este caso hay nodos que pueden tener más de un arco de entrada (más de un nodo padre).

Ahora vamos a ver cómo se resuelven los nombres en un espacio de nombres y cómo se implementa un espacio de nombres distribuido.

### Resolución de nombres

Una de las aplicaciones de los espacios de nombres es el almacenamiento y recuperación de recursos mediante nombres.

En general, dado el nombre de ruta deberá ser posible encontrar el recurso asociado al nombre. Al proceso de búsqueda de un nombre en un espacio de nombres se le llama **resolución de nombre**.

Supongamos que tenemos un nombre de ruta de la forma:

N: etiqueta<sub>1</sub>, etiqueta<sub>2</sub>, etiqueta<sub>2</sub>, ... ,etiqueta<sub>n</sub>

La resolución de nombre inicia en el nodo N, entonces se busca la etiqueta<sub>1</sub> en la tabla de directorio del nodo N, si existe, se obtiene el identificador del nodo siguiente. Ahora se busca la etiqueta<sub>2</sub> en la tabla de directorio del nodo actual, si existe, se obtiene el identificador del nodo siguiente.

El proceso continúa hasta encontrar el nodo correspondiente a la etiqueta<sub>n</sub>

### Mecanismo de clausura

Se le llama **mecanismo de clausura** a la selección del nodo inicial dentro de un espacio de nombres en el cual comienza la resolución de nombre.

Un mecanismo de clausura es implícito al proceso de resolución de nombre, lo cual significa que el mecanismo de clausura debe estar definido e implementado antes de iniciar el proceso de resolución.

Por ejemplo, si el primer nodo de un nombre de ruta es el nodo raíz, entonces sabemos que el nodo raíz será un nodo directorio donde se realizará la búsqueda de la primera etiqueta.

Si el primer nodo del nombre de ruta no es el nodo raíz, entonces deberá existir una forma de saber cómo encontrar ese nodo inicial. En un sistema de archivos de tipo Unix, si se quiere resolver utilizando un nombre de ruta relativo, el mecanismo de clausura queda definido por el directorio actual (*working directory*) el cual se obtiene mediante el comando `pwd`.

### Vínculo absoluto y vínculo simbólico

Un **alias** es un segundo nombre para una misma entidad en un espacio de nombres.

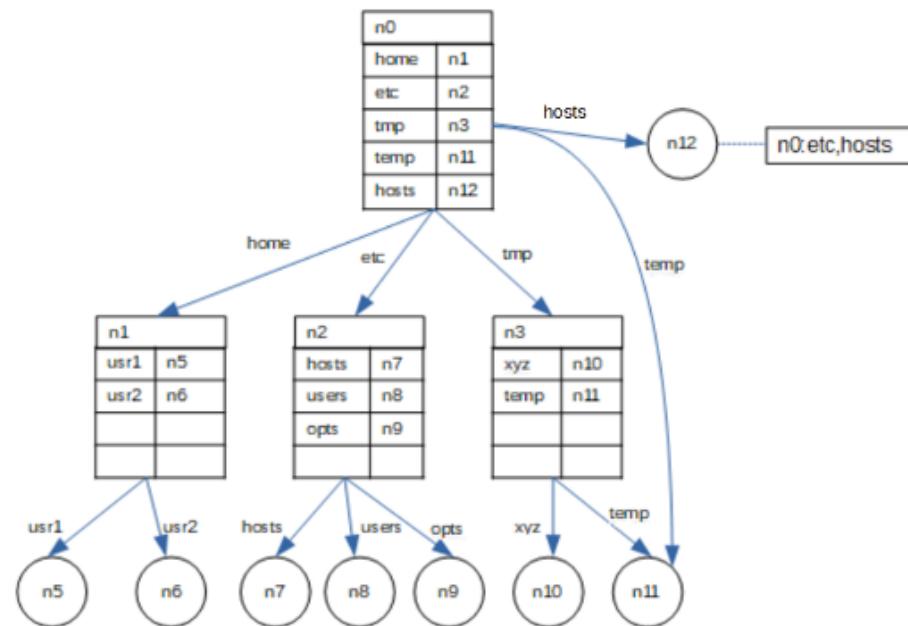
Existen dos formas de implementar un alias para una entidad: vínculo absoluto (*hard link*) y vínculo simbólico (*symbolic link*).

Un **vinculo absoluto** es simplemente un segundo nombre de ruta para una entidad en el espacio de nombres.

Un **vinculo simbólico** consiste en almacenar en un nodo hoja el nombre de ruta absoluto correspondiente a la entidad. Para encontrar la entidad se recorre el nombre de ruta hasta el nodo hoja que contiene el nombre de ruta absoluto, entonces se busca la entidad utilizando este nombre absoluto.

En el siguiente ejemplo podemos ver que el archivo "temp" puede ser resuelto mediante la ruta n0:tmp,temp y mediante la ruta n0:temp. En este caso se trata de un vinculo absoluto.

Por otra parte, el archivo "hosts" puede ser resuelto mediante la ruta n0:hosts y mediante la ruta n0:etc,hosts. En este caso se trata de un vínculo simbólico, ya que la resolución de la ruta n0:hosts lleva al nodo hoja n12 dónde se obtiene la ruta absoluta del nodo n7.



### Montar un espacios de nombres

La resolución de nombres que vimos anteriormente puede ser utilizada para enlazar diferentes espacios de nombres en forma transparente.

Montar un espacio de nombres B en un espacio de nombres A consiste en hacer que un nodo directorio del espacio de nombres A incluya el identificador de un nodo directorio del espacio de nombres B.

Al nodo directorio del espacio de direcciones A que contiene el identificador del nodo externo se le llama **punto de montaje**. Así mismo, se le llama punto de montaje al nodo directorio del espacio de direcciones B.

En general, el punto de montaje externo es un nodo raíz.

El concepto de montaje de espacios de nombre permite implementar sistemas de espacios de nombres distribuidos, dónde cada computadora podría administrar un espacio de nombres local.

Para montar un espacio de nombres externo se requiere al menos de lo siguiente:

1. El nombre del protocolo de comunicación.
2. El nombre de la computadora remota.
3. El nombre del punto de montaje en la computadora remota.

El nombre del protocolo de comunicación define cómo se va a comunicar la computadora local con la computadora remota.

El nombre de la computadora puede ser la dirección IP de la computadora o bien el nombre de dominio el cual puede ser resuelto mediante un servidor de nombres de dominio (DNS).

Finalmente, deberá existir un mecanismo de clausura en la computadora remota que resuelva el punto de montaje.

Recordemos que ya utilizamos un servidor de nombres llamado rmiregistry, donde se identifican los objetos remotos mediante una URL de la forma: rmi://ip-del-servidor/nombre-del-objeto.



- La clase de hoy veremos un ejemplo de espacio de nombres: el sistema de archivos distribuido NFS.

## El sistema de archivos distribuidos NFS

El sistema de archivos NFS (Network File System) permite que una computadora (cliente) tenga acceso de manera transparente a los archivos contenidos en un servidor remoto.

Supongamos que una computadora va a acceder mediante NFS los archivos que se encuentran en el directorio /home/usuario de un servidor remoto.

Si el dominio del servidor remoto es m4gm.com, para que el cliente tenga acceso al directorio remoto es necesario montar el espacio de nombres remoto en el espacio de nombres local.

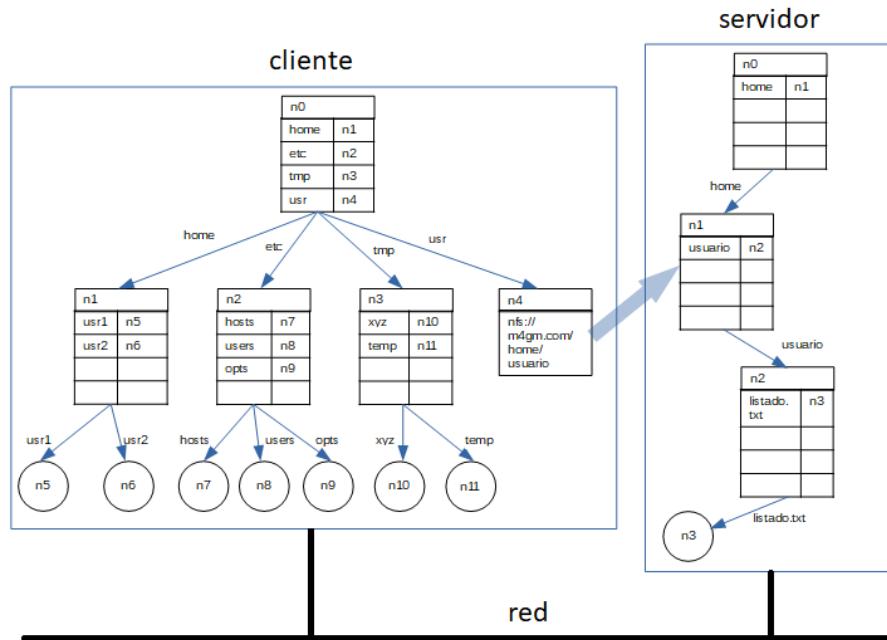
Para montar el espacio de nombres remoto se elige un punto de montaje en el cliente, por ejemplo se puede elegir el directorio /usr. Entonces el nodo directorio correspondiente a /usr va a contener una URL que define el protocolo, el dominio del servidor y el punto de montaje en el servidor, en este caso la URL sería:

nfs://m4gm.com/home/usuario

En general un cliente indica qué archivo va a acceder utilizando una URL de la forma:

nfs://dominio-o-ip-del-servidor/punto-de-montaje

En la figura se puede ver que el nodo directorio n4 contiene la URL que define el punto de montaje del espacio de nombres remoto.



Si el cliente requiere acceder al archivo remoto /home/usuario/listado.txt solo tiene que acceder al “archivo local” /usr/listado.txt

La localización del archivo listado.txt se realiza en tres pasos:

1. El espacio de nombres local resuelve el nombre /usr
2. El dominio m4gm.com se resuelve accediendo un DNS entonces se obtiene la dirección IP del servidor.
3. El servidor resuelve el nombre /home/usuario

La ventaja de utilizar archivos remotos mediante NFS es que el usuario accede a los archivos sin preocuparse por los detalles de la comunicación entre el cliente y el servidor.



- **Actividades individuales a realizar**

En esta actividad vamos a instalar NFS en dos máquinas virtuales en la nube.

Primeramente necesitamos crear dos máquinas virtuales (cliente y servidor) en Azure con Ubuntu 18 con las siguientes características:

- Tamaño de la memoria en la máquina virtual: 1 GB RAM
- Tipo de autenticación: Contraseña
- Tipo de disco del sistema operativo: HDD estándar

## Instalación en el servidor

1. Para instalar NFS en el servidor se ejecutan los siguientes comandos:

```
sudo apt update  
sudo apt install nfs-kernel-server
```

2. Crear el directorio compartido en el servidor;

```
sudo mkdir /var/nfs -p
```

3. El propietario del directorio /var/nfs es root debido a que este directorio se creó con sudo. Podemos ver el propietario del directorio /var/nfs ejecutando el comando:

```
ls -l /var
```

4. Debido a que NFS convierte el acceso del usuario root en el cliente en un acceso con el usuario "nobody:nogroup" en el servidor, es necesario cambiar el propietario y permisos del directorio creado anteriormente:

```
sudo chown nobody:nogroup /var/nfs  
sudo chmod 777 /var/nfs
```

5. Podemos verificar el nuevo propietario:

```
ls -l /var
```

6. Ahora se debe registrar el directorio creado en el archivo de configuración de NFS.

## 6.1 Editar el archivo /etc/exports:

```
sudo vi /etc/exports
```

6.2 Agregar la siguiente línea, guardar y salir del editor (en este caso 40.76.45.28 es la IP del cliente):

```
/var/nfs 40.76.45.28(rw,sync,no_subtree_check)
```

Para una descripción de los permisos se puede consultar [Understanding the /etc/exports File](#) y [exports linux page](#).

6.3 Actualizar la tabla de file systems exportados por NFS:

```
sudo exportfs -ra
```

6.4 Para ver los file systems exportados por NFS:

```
sudo exportfs
```

6.5 Para activar la nueva configuración, se requiere reiniciar el servidor NFS:

```
sudo systemctl restart nfs-kernel-server
```

7. Ahora debemos abrir el puerto 2049 en el portal de Azure:

Intervalos de puertos de destino: 2049

Protocolo: TCP

Nombre: puerto\_nfs

## Instalación en el cliente

8. Para instalar NFS en el cliente se ejecutan los siguientes comandos:

```
sudo apt update  
sudo apt install nfs-common
```

9. Crear el directorio de montaje en el cliente (punto de montaje):

```
sudo mkdir -p /nfs
```

10. Montar el directorio remoto (en este caso 40.87.94.140 es la IP del servidor):

```
sudo mount -v -t nfs 40.87.94.140:/var/nfs /nfs
```

11. Para desmontar el directorio remoto /nfs:

```
sudo umount /nfs
```

### Probar el acceso a archivos remotos

11. En el servidor crear un archivo, en este caso utilizamos el editor vi:

```
vi /var/nfs/texto.txt
```

12. Escribir un texto (como el siguiente) guardar y salir de la edición:

Esta es una prueba

13. En el cliente editar el archivo /nfs/texto.txt:

```
vi /nfs/texto.txt
```

14. Agregar la siguiente línea, guardar y salir de la edición:

Esta es otra prueba

15. En el servidor desplegar el contenido del archivo /var/nfs/texto.txt:

```
more /var/nfs/texto.txt
```

16. En el cliente desplegar el contenido del directorio /nfs:

```
ls -l /nfs
```

#### Instalación de NFS sobre un túnel SSH

Para encriptar el tráfico entre el cliente y el servidor NFS se puede utilizar un túnel SSH. Ver: [Mount NFS Folder via SSH Tunnel](#).

#### Instalación en el servidor

1. Para instalar NFS en el servidor se ejecutan los siguientes comandos:

```
sudo apt update
sudo apt install nfs-kernel-server
sudo apt install portmap
```

2. Crear el directorio compartido en el servidor;

```
sudo mkdir /var/nfs -p
```

3. El propietario del directorio /var/nfs es root debido a que este directorio se creó con sudo. Podemos ver el propietario del directorio /var/nfs ejecutando el comando:

```
ls -l /var
```

4. Debido a que NFS convierte el acceso del usuario root en el cliente en un acceso con el usuario "nobody:nogroup" en el

servidor, es necesario cambiar el propietario y permisos del directorio creado anteriormente:

```
sudo chown nobody:nogroup /var/nfs
sudo chmod 777 /var/nfs
```

5. Podemos verificar el nuevo propietario:

```
ls -l /var
```

6. Ahora se debe registrar el directorio creado en el archivo de configuración de NFS.

6.1 Editar el archivo /etc/exports:

```
sudo vi /etc/exports
```

6.2 Agregar la siguiente línea, guardar y salir del editor:

```
/var/nfs
localhost(insecure,rw,sync,no_subtree_check)
```

6.3 Actualizar la tabla de file systems exportados por NFS:

```
sudo exportfs -ra
```

6.4 Para ver los file systems exportados por NFS:

```
sudo exportfs
```

6.5 Para activar la nueva configuración, se requiere reiniciar el servidor NFS:

```
sudo systemctl restart nfs-kernel-server
```

7. Ahora debemos abrir el puerto 2049 en el portal de Azure:

Intervalos de puertos de destino: 2049

Protocolo: TCP

Nombre: puerto\_nfs

### Instalación en el cliente

8. Para instalar NFS en el cliente se ejecutan los siguientes comandos:

```
sudo apt update
sudo apt install nfs-common
sudo apt install portmap
```

9. Crear el directorio de montaje en el cliente (punto de montaje):

```
sudo mkdir -p /nfs
```

10. Para crear un túnel SSH entre cliente y el servidor se ejecuta el siguiente comando:

```
ssh -fNv -L 3049:localhost:2049
ubuntu@40.84.237.35
```

En este ejemplo el puerto local 3049 del cliente se conecta al puerto 2049 del servidor. En este caso "ubuntu" es un usuario en el servidor y "40.84.237.35" es la dirección IP del servidor.

11. Montar el directorio remoto:

```
sudo mount -t nfs -o port=3049 localhost:/var/nfs
/nfs
```

12. Para desmontar el directorio remoto /nfs:

```
sudo umount /nfs
```



- Domain Name System (DNS)

La clase de hoy veremos otro ejemplo de espacio de nombres: el sistema de nombres de dominio (DNS: *Domain Name System*).

Un DNS es un espacio de nombres distribuido a gran escala, organizado jerárquicamente en tres capas.

### Capa global

La capa global se compone de los nodos de más alto nivel, a saber, el nodo raíz y sus hijos. Todos los nodos en esta capa son **nodos directorio**.

Las etiquetas de los arcos son los diferentes tipos de dominio (com, org, edu, net, mx, etc.). Las tablas de directorio en la capa global casi nunca se modifican.

### Capa de administración

La capa de administración se compone de **nodos directorio** que son administrados dentro de una misma organización.

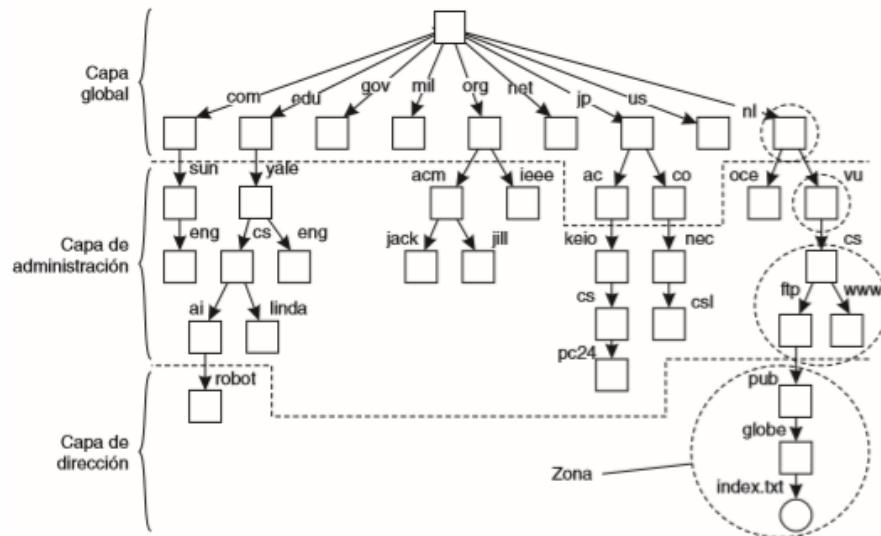
Por ejemplo, un nodo podría corresponder al subdominio llamado “sun” y este tener un subdominio llamado “eng”, en este caso el dominio sería eng.sun.com

Las tablas de directorio de la capa de administración se modifican poco, debido a que generalmente representan unidades administrativas dentro de una organización.

### Capa de dirección

La capa de dirección se compone de nodos que pueden modificarse con cierta frecuencia. Los nodos en esta capa representan servidores con el último subdominio

La siguiente figura muestra un ejemplo del espacio de nombres de un DNS. En esta figura se pueden ver partes del espacio de nombres llamadas zonas, las cuales se manejan mediante servidores de nombres por separado.



Fuente: Sistemas Distribuidos, Principios y Paradigmas, Andrew S. Tanenbaum, Pearson.

### Resolución de nombre en un DNS

Cuando un usuario escribe una URL en un navegador web, se inicia un proceso de resolución de nombres para la URL, en primer lugar, se debe resolver el dominio al cual se va a conectar el navegador.

La resolución del dominio la realiza un **solucionador de nombre** dentro del sistema operativo que ejecuta el navegador.

Supongamos que el usuario escribe la siguiente URL en su navegador web:

`ftp://ftp.cs.vu.nl/pub/globe/index.html`

El **nombre de ruta** correspondiente sería el siguiente:

`root:nl,vu,cs,ftp,pub,globe,index.html`

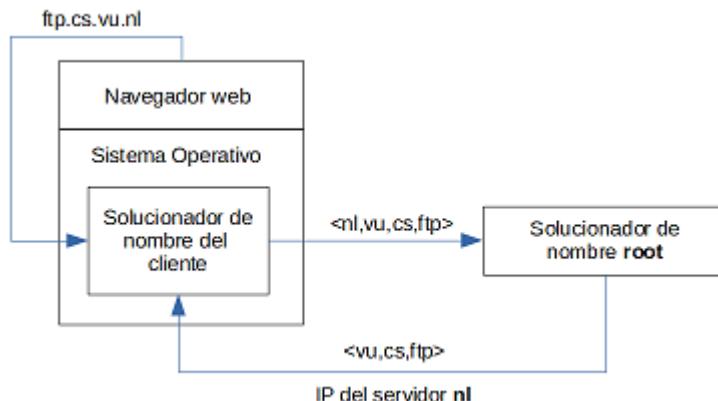
Esto significa que el usuario requiere acceder al archivo “index.html” el cual se encuentra en el directorio “/pub/globe” en el servidor cuyo dominio es “ftp.cs.vu.nl”.

Para resolver la URL existen dos técnicas, la resolución iterativa y la resolución recursiva.

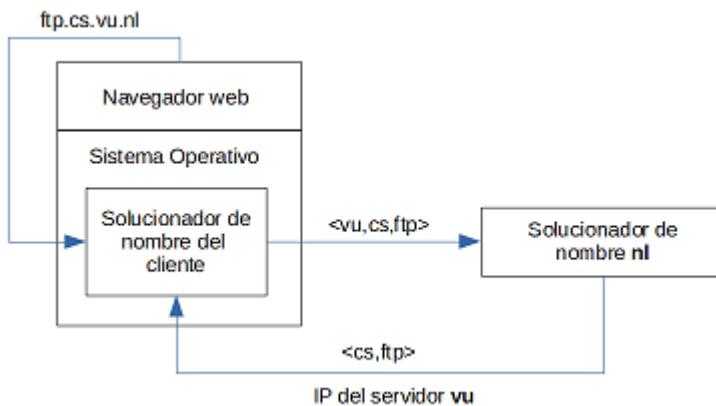
### Resolución iterativa

La resolución iterativa del nombre de ruta anterior se podría realizar en tres iteraciones:

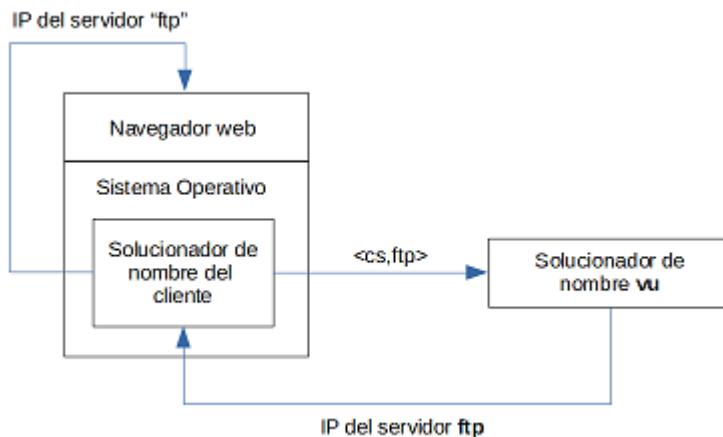
1) El solucionador de nombre del cliente se conecta a un solucionador de nombre **root** cuya dirección IP es conocida enviando la ruta <nl,vu,cs,ftp>. Este servidor resolverá el nombre hasta dónde le sea posible, en este caso solo puede resolver la etiqueta “nl”, entonces regresará al solucionador de nombre del cliente la dirección IP del solucionador de nombre **nl** y la ruta restante <vu,cs,ftp>.



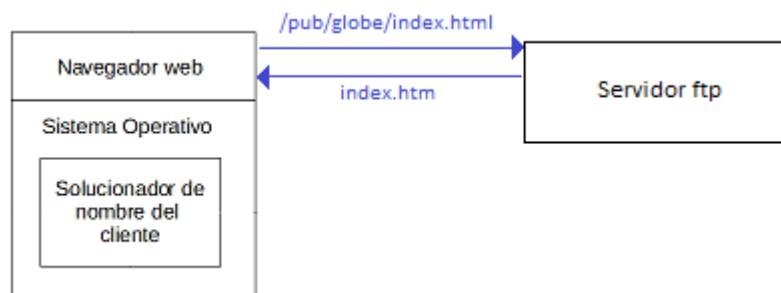
2) El solucionador de nombre del cliente se conecta al solucionador de nombre **nl** enviando la ruta <vu,cs,ftp>. Este servidor solo puede resolver la etiqueta “vu”, entonces regresará al solucionador de nombre del cliente la dirección IP del solucionador de nombre **vu** y la ruta restante <cs,ftp>.



- 3) El solucionador de nombre del cliente se conecta al solucionador de nombre **vu** enviando la ruta **<cs,ftp>**. Este servidor puede resolver las etiquetas “cs” y “ftp”, entonces regresará al solucionador de nombre del cliente la dirección IP del servidor **ftp**.



Finalmente el navegador web se conecta al servidor **ftp** enviando la ruta **/pub/globe/index.html**. Finalmente el servidor FTP regresa el archivo solicitado.



### Resolución recursiva

Debido a que el solucionador de nombre del cliente suele estar lejos de los solucionadores de nombres, la resolución iterativa

puede ser más costosa en términos de latencia de la comunicación.

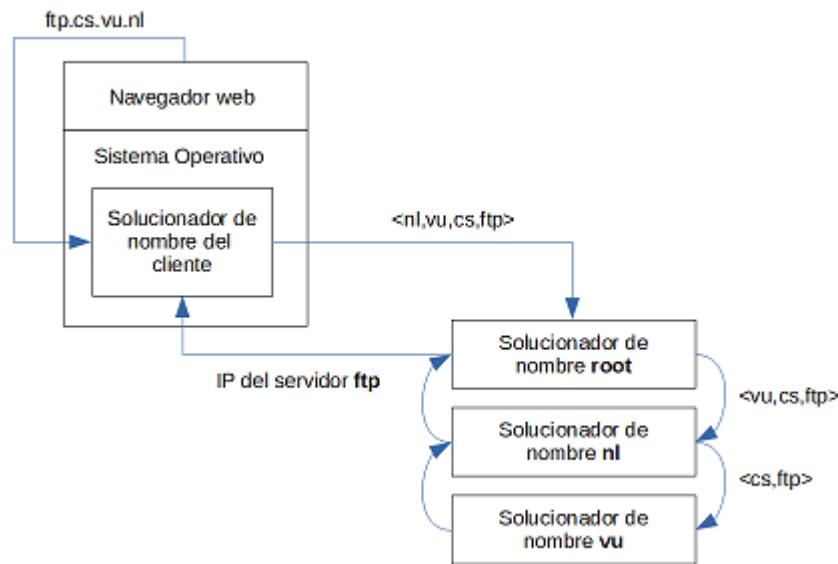
Una alternativa a la resolución iterativa de nombres es el uso de la técnica de resolución recursiva.

1) En la resolución recursiva, el solucionador de nombre del cliente se comunica con el solucionador de nombre **root** enviando la ruta **<nl,vu,cs,ftp>**, este servidor solo puede resolver la etiqueta “nl” por tanto se comunica con el solucionador de nombre **nl** enviando el resto de la ruta **<vu,cs,ftp>**.

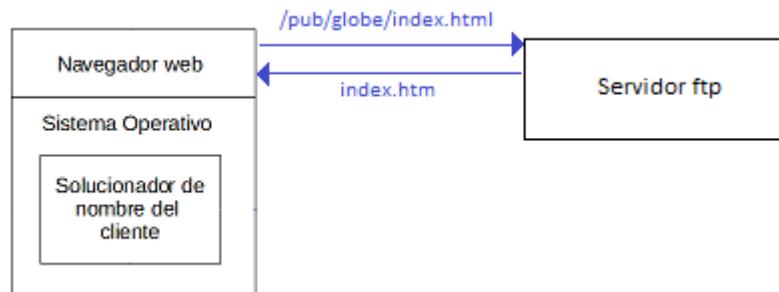
2) El solucionador de nombre **nl** sólo puede resolver la etiqueta “vu” por tanto se comunica con el solucionador de nombre **vu** enviando el resto de la ruta **<cs,ftp>**.

3) Finalmente, el solucionador de nombre **vu** resuelve las etiquetas “cs” y “ftp”, entonces regresará al solucionador de nombre **nl** la dirección IP del servidor **ftp**. El solucionador de nombre **nl** le envía la dirección IP al solucionador de nombre **root**, y este le enví la IP al solucionador de nombre del cliente.

4) El solucionador de nombre que resuelve la última etiqueta regresa la IP al servidor que se comunicó con él, y así sucesivamente hasta que el solucionador de nombre **root** regresa la IP al solucionador de nombre del cliente



Finalmente el navegador web se conecta al servidor ftp enviando la ruta /pub/globe/index.html. Finalmente el servidor FTP regresa el archivo solicitado.



La desventaja de la resolución recursiva es que representa una mayor carga en cada servidor de nombre, ya que debe mantener abierta una conexión al siguiente solucionador mientras el proceso de resolución esté en curso.

Por esta razón, los servidores de nombre de la capa global (los cuales son los que más peticiones reciben) soportan solamente la resolución iterativa.



- Replicación de datos

Los datos son el principal activo de las empresas e instituciones.

Si bien es cierto que los sistemas informáticos son de gran importancia para automatizar los procesos en las empresas, los

datos representan los objetos de negocio que tienen mayor persistencia en el tiempo, ya que ellos corresponden a lo que se sabe de los clientes, los productos, los insumos, los activos, los pasivos, las ventas, los procesos, los empleados, y muchos objetos de negocio más.

La replicación de los datos es una estrategia utilizada para mantener copias consistentes de los datos en diferentes ubicaciones, con el objetivo de tener la capacidad de recuperar la operación en caso de desastre.

### ¿Por qué replicar los datos?

Los datos se replican para satisfacer dos requerimientos no funcionales de los sistemas distribuidos: la **confiabilidad** y el **rendimiento**.

Replicar los datos en diferentes sistemas de archivos, aumenta la **confiabilidad** del sistema, ya que si una copia de los datos falla es posible seguir trabajando con otra copia de los datos.

Por ejemplo, en los DBMS se acostumbra configurar copias "espejo" de las tablas, de tal manera que cuando se inserta, modifica o borra datos en una tabla, se realicen las mismas operaciones sobre una tabla "espejo", si la lectura de la tabla principal falla, entonces el DBMS automáticamente realiza la lectura sobre la copia "espejo", sin mayor intervención del sistema que accede a la base de datos.

La replicación de datos también mejora el **rendimiento** de un sistema distribuido que requiere escalar en tamaño y geografía.

La replicación de los datos permite acercar los datos al sistema, lo cual disminuye la latencia en el acceso. Por ejemplo, la replicación de datos que se modifican poco como es el caso de los catálogos de un sistema (clientes, productos, cuentas, etc.) permite acceder a estos datos más rápido.

No obstante las ventajas que tiene la replicación de los datos, la principal dificultad es mantener la consistencia entre las copias. La **consistencia** de los datos significa que todas las copias deben tener los mismo datos.

Mantener la consistencia entre copias puede impactar el rendimiento del sistema, debido a que la actualización de las copias representa un costo en tiempo y recursos (CPU, red, almacenamiento, etc). Entonces será necesario evaluar el costo de mantener consistentes las copias y el beneficio del aumento del rendimiento que trae la replicación de los datos.

### Modelos de consistencia

Un **modelo de consistencia** es un acuerdo entre los procesos que acceden un almacén de datos y el almacén de datos.

El acuerdo establece las reglas que deben obedecer los procesos cuando acceden el almacén de datos de manera que los procesos puedan tener una imagen consistente de los datos.

El **almacén de datos** puede ser una base de datos distribuida, un sistema de archivos distribuido o una combinación de ambos.

El principio en que se basa un modelo de consistencia es que si un proceso realiza una operación de lectura sobre elemento de datos, se espera leer el resultado de la última escritura sobre el mismo elemento de datos, independientemente de qué proceso realizó la escritura.

### Consistencia secuencial

El modelo de consistencia secuencial fue propuesto por Lamport (1979), y dice que un almacén de datos es secuencialmente consistente si:

*"El resultado de cualquier ejecución es el mismo que si las operaciones (de lectura y escritura) de todos los procesos efectuados sobre el almacén de datos se ejecutaran en algún orden secuencial y las operaciones de cada proceso individual aparecieran en esa secuencia en el orden especificado por su programa".*

Esto significa que el almacén de datos debería "ver" las operaciones de lectura y escritura que realizar todos los procesos como si tratara de una secuencia de operaciones de lectura y escritura realizadas por un solo proceso.

La consistencia secuencial corresponde a operaciones de lectura y escritura ordenadas mediante la relación happen-before. Si A es un dato en el almacén de datos, entonces para cada par de operaciones write(A), read(A) se deberá cumplir write(A) happen-before read(A), es decir, la escritura de un dato debe preceder a la lectura del dato.

### Consistencia de entrada

El modelo de consistencia secuencial propuesto por Lamport es un modelo de granularidad fina pensado originalmente para ser implementado en hardware para acceder localidades de memoria compartida en sistemas multiprocesadores, sin embargo, este modelo de consistencia resulta muy costoso para los sistemas distribuidos dónde los datos tienen granularidad gruesa, como son los registros, tablas, archivos, etc.

B.N. Bershad et al. ([The Midway Distributed Shared Memory System](#), 1993) propuso un modelo de consistencia basado en la relación entre objetos de sincronización (locks) que protegen secciones críticas y los datos compartidos dentro de las secciones críticas.

El modelo de **consistencia de entrada** utiliza objetos de sincronización exclusiva y no-exclusiva (compartida) para

garantizar el orden en que se ejecutan las operaciones de lectura y escritura sobre un mismo elemento de datos.

Una sección crítica comienza con una operación de adquisición del lock y termina con la liberación de lock. A estas operaciones se les llama generalmente "lock" y "unlock".

Las reglas que se debe cumplir en este modelo de consistencia son:

1. Cuando un proceso ejecuta la operación "lock" ésta debe esperar a que se realicen todas las operaciones de escritura de los datos compartidos por el proceso.
2. Un proceso no puede adquirir un lock si algún otro proceso lo adquirió ya sea en forma exclusiva o compartida.
3. Si un proceso adquirió un lock en forma exclusiva, ningún otro proceso puede adquirir el lock en forma compartida.

Estas reglas garantizan que las lecturas de datos compartidos (que se realizan dentro de una sección crítica) obtendrán los datos actualizados por la última escritura, la cual también se debió realizar dentro de una sección crítica.

Como puede observarse, no importa el orden en que se realizan las lecturas y escrituras a los elementos de datos, lo que importa es el orden en que se realizan las operaciones "lock" y "unlock".

Para lograr la consistencia de los elementos de datos compartidos, los objetos de sincronización (locks) deberán implementarse en forma global, tal como se explicó en el tema de "Sincronización y coordinación".

Supongamos que tenemos dos threads  $t_1$  y  $t_2$ , y cada thread ejecuta un ciclo donde se incrementa la variable global  $n$ .

```
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);
    }
}
```

Al ejecutar varias veces el programa podemos ver que el valor final de **n** no es el mismo ¿por qué?

La instrucción **n++** se compone de tres operaciones:

1. Copiar el valor de la variable **n** a un registro del procesador.
2. Incrementar el valor del registro.
3. Escribir el valor del registro a la variable **n**.

Debido a que los threads **t1** y **t2** ejecutan en paralelo, en una computadora con dos o más núcleos el thread **t1** leerá y escribirá la variable **n** al mismo tiempo que el thread **t2**:

T1

T2

<b>r=Read(n)</b>	<b>r=Read(n)</b>
<b>r++</b>	<b>r++</b>
<b>Write(n,r)</b>	<b>Write(n,r)</b>

Dado que la lectura que realiza **t1** no está ordenada con respecto a la escritura que hace **t2** y que la lectura que realiza **t2** no está ordenada con respecto a la escritura que hace **t1**, es posible que no se escriba algún incremento en la variable **n**, lo que produce un valor final menor a 200000.

Para resolver el problema se requiere que el programador identifique las secciones críticas y agregue las operaciones "lock" y "unlock" necesarias.

En este caso, la sección crítica es la instrucción **n++**, que es dónde se lee y escribe la variable **n** compartida por los dos threads.

Por lo tanto, es necesario ejecutar "lock" antes de **n++** y ejecutar "unlock" después.

En java se utiliza la instrucción **synchronized(objeto){ bloque-de-instrucciones }** para definir un bloque de instrucciones como sección crítica controlada por el lock que contiene el **objeto** (recordemos que en Java todos los objetos incluyen un lock).

Entonces el código del programa queda de la siguiente manera:

```
class A extends Thread
{
    static long n;
    static Object obj = new Object();
    public void run()
    {
        for (int i = 0; i < 100000; i++)
    }
```

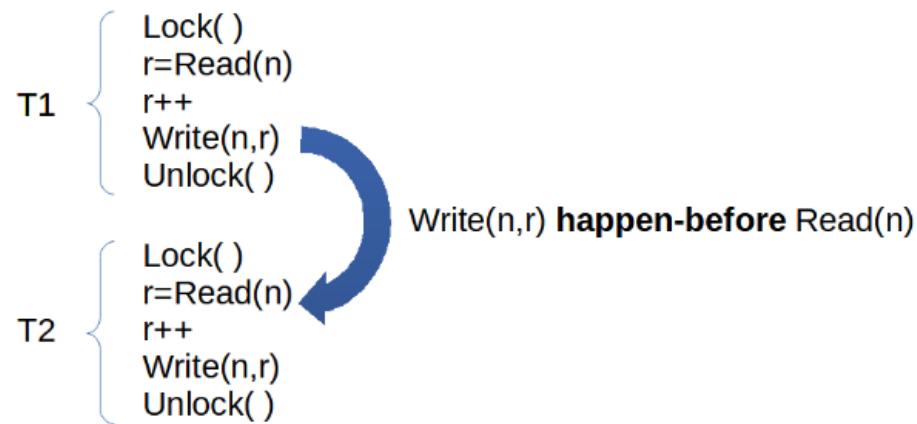
```

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);
}
}

```

Al ejecutar varias veces el programa anterior podemos ver que el valor final de la variable n siempre es 200000, debido a que ahora las operaciones de lectura y escritura se ejecutan en el orden correcto:



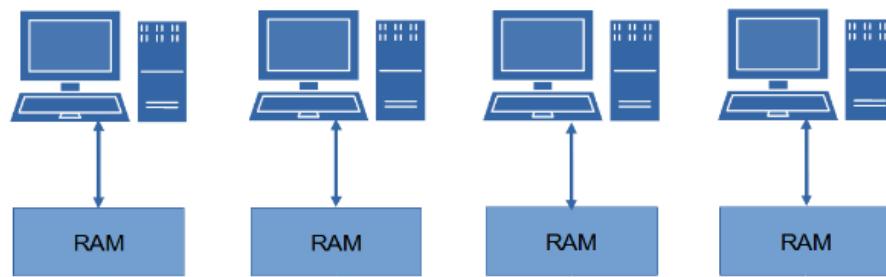
### Memoria compartida distribuida

Un sistema de memoria compartida distribuida (DSM: *Distributed Shared Memory*) es una capa de software o hardware que implementa un área de memoria compartida a la

que cada computador en un cluster tiene acceso además de su memoria local.

En una red de computadoras, cada computadora tiene acceso a su memoria local (RAM). Cada computadora realiza dos operaciones sobre la memoria local:

1. La operación de escritura a memoria Write(dirección,valor) la cual escribe un valor a una dirección de memoria.
2. La operación de lectura a memoria Read(dirección) la cual lee el valor que se encuentra en una dirección de memoria.

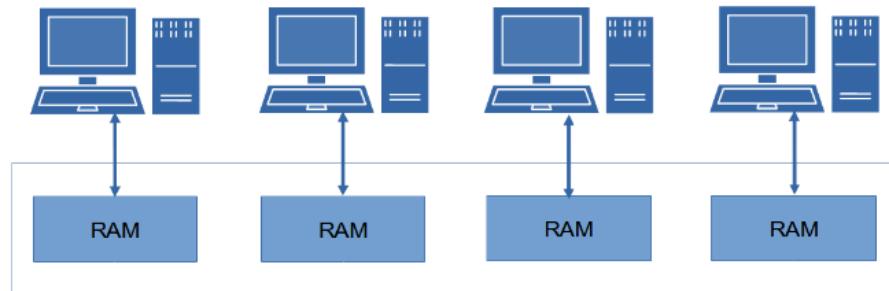


- Write(dirección, valor)
- valor = Read(dirección)

Un DSM requiere garantizar la consistencia de los datos, esto es, cada nodo deberá ver los mismos datos en la memoria compartida. Por tanto, la implementación de un DSM implica la utilización de un modelo de consistencia.

Para implementar un DSM podemos utilizar el modelo de consistencia de entrada, debido a que en este modelo de consistencia solo se debe ordenar las operaciones de bloqueo y desbloqueo, lo cual es más eficiente que ordenar todas las operaciones de escritura y lectura, como sería el caso del modelo de consistencia secuencial.

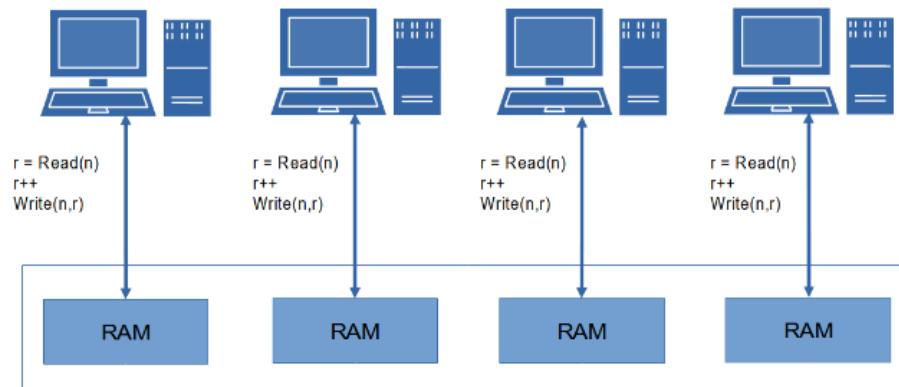
Entonces, el DSM deberá implementar cuatro operaciones: 1) escritura a memoria compartida, 2) lectura a memoria compartida, 3) bloqueo distribuido y 4) desbloqueo distribuido.



- Write(dirección, valor)
- valor = Read(dirección)
- Lock( )
- Unlock ( )

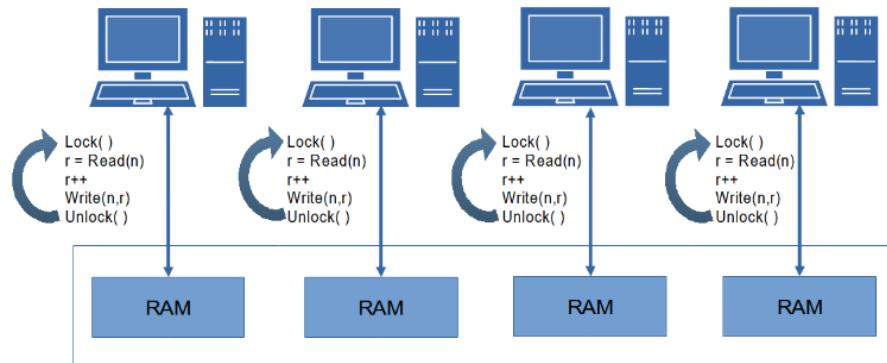
### Ejemplo de ejecución distribuida utilizando DSM

Supongamos que se quiere incrementar una variable  $n$  utilizando cuatro nodos. Cada nodo ejecutará un ciclo de 100 iteraciones donde se lee una variable compartida  $n$  a un registro  $r$ , se incrementa el registro  $r$  y se escribe el registro  $r$  a la variable  $n$ . Al final, el nodo 0 desplegará el valor de la variable  $n$ .



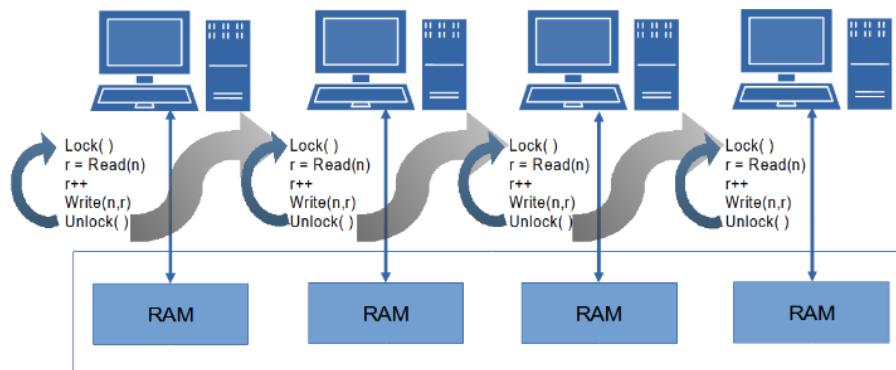
Debido a que cada nodo lee y escribe una copia local de la variable  $n$ , el valor final de la variable  $n$  (en cada nodo) es 100, no obstante el resultado debería ser 400.

Entonces es necesario implementar un modelo de consistencia para garantizar que todos los nodos vean "la misma" variable  $n$ , en este caso, implementamos el modelo de consistencia de entrada:



Ahora se ha agregado las operaciones Lock y Unlock. De acuerdo al modelo de consistencia de entrada, cada vez que se ejecute la operación Lock, el nodo deberá recibir los cambios a las variables compartidas realizados por el nodo que ejecutó Unlock, en este caso la variable compartida es n.

Debido a que la sección crítica no puede ser ejecutada por más de una computadora, cada vez que un nodo ejecuta la operación Lock, solo la computadora que ejecutó la operación Unlock deberá enviará las escrituras realizadas a las variables compartidas, en este caso la variable n.



- Actividades individuales a realizar

En esta actividad se implementará el modelo de consistencia de entrada en lenguaje Java.

### Requerimientos funcionales

1. Se deberá implementar las operaciones lock y unlock utilizando exclusión mutua distribuida, utilizar el algoritmo de

Ricart o el algoritmo de token-ring, los cuales implementamos en actividades anteriores.

2. Cada nodo creará un arreglo M de 10 enteros de 64 bits, el cual representará la memoria compartida.

3. Cada nodo creará un arreglo B de 10 booleanos los cuales indicarán qué elemento del arreglo de enteros fue modificado dentro de un bloque lock-unlock.

4. Cada nodo implementará las operaciones Read, Write, Lock y Unlock. La operación Read(n) leerá del arreglo M el elemento n, la operación Write(n,valor) escribirá el valor en el elemento n del arreglo M. Cuando se ejecute la operación Write(n,valor) se deberá asignar true al elemento n del arreglo B.

5. En la operación lock se deberá asignar false a todos los elementos del arreglo B.

6. Cuando se ejecute unlock, antes de desbloquear, se deberá enviar los cambios realizados en el arreglo M al resto de nodos.

7. Ejecutar el programa en cuatro nodos, en cada nodo se ejecutará:

7.1 Una barrera que espere que todos los nodos se encuentran en ejecución.

7.2 Ejecutar las siguientes instrucciones en un ciclo de 100 iteraciones:

```
Lock()  
r=Read(0)  
r++  
Write(0,r)  
Unlock()
```

7.3 Al final de las iteraciones el nodo 0 ejecutará los siguientes:

```
Lock()
```

```
System.out.println(Read(0))
```

```
Unlock();
```



- **Respaldos incrementales y respaldos continuos**

En la clase anterior vimos que la replicación de los datos es una estrategia utilizada para aumentar la confiabilidad y rendimiento de los sistemas.

Replicar los datos permite mantener copias de los datos en diferentes lugares, si una copia falla entonces se puede acceder a otra copia.

La replicación de los datos tiene también un efecto positivo en el rendimiento de un sistema, debido a que es más rápido acceder a datos cercanos.

Para mantener consistentes las diferentes copias de los datos, es necesario implementar un modelo de consistencia. Mantener la consistencia de los datos suele ser complicado y costoso en tiempo de comunicación.

### **Respaldos incrementales**

La replicación de los datos mediante respaldos es una solución medianamente buena para garantizar la continuidad de un sistema.

Si un sistema falla, entonces se puede restaurar el **último respaldo** con el fin de recuperar el estado del sistema hasta un cierto punto del tiempo.

Los **respaldos incrementales** son aquellos que sólo guardan los cambios realizados desde el último respaldo, entonces se pueden realizar respaldos anuales, mensuales, semanales, diarios, cada hora, etc.

En estas condiciones, el respaldo más frecuente solo deberá guardar los cambios realizados desde el último respaldo y no

todo el almacén de datos.

No obstante, aún realizando respaldos cada minuto, si el sistema falla se perderán los cambios efectuados los últimos segundos.

Para mitigar los riesgos debidos a errores humanos o el secuestro de datos (*ransomware*), los proveedores de cómputo en la nube ofrecen servicios escalables de respaldo de datos, los cuales permiten respaldar y restaurar máquinas virtuales completas o bien archivos, carpetas o bases de datos.

Mas adelante veremos cómo realizar respaldos en Azure.

### Respaldos continuos

Los sistemas manejadores de bases de datos (DBMS) implementan una estrategia de **respaldos continuos** de las transacciones.

A partir del inicio de una transacción (*begin work*), se va guardando copias de los registros que son actualizados dentro de la transacción, si el DBMS falla o la computadora se apaga, es posible recuperar la base de datos hasta un estado consistente a partir de los registros contenidos en el respaldo continuo.

El respaldo continuo de los registros modificados en una transacción también se utiliza para mantener un estado consistente de la base de datos. Si la transacción termina con *rollback*, entonces se restablecen todos los registros modificados al estado anterior a la transacción.

Por otra parte, si la transacción termina con *commit*, entonces los cambios quedan firmes y se desecha el respaldo continuo de los registros modificados dentro de la transacción.

A la funcionalidad del DMBS que garantiza la consistencia de la base de datos se le llama **ACID** (*Atomicity, Consistency, Isolation, & Durability*).

En el caso de bases de datos distribuidas, el control de las transacciones distribuidas se realiza mediante un protocolo llamado *two-phase commit* (2PC). En este caso el respaldo continuo también se realiza en forma distribuida.

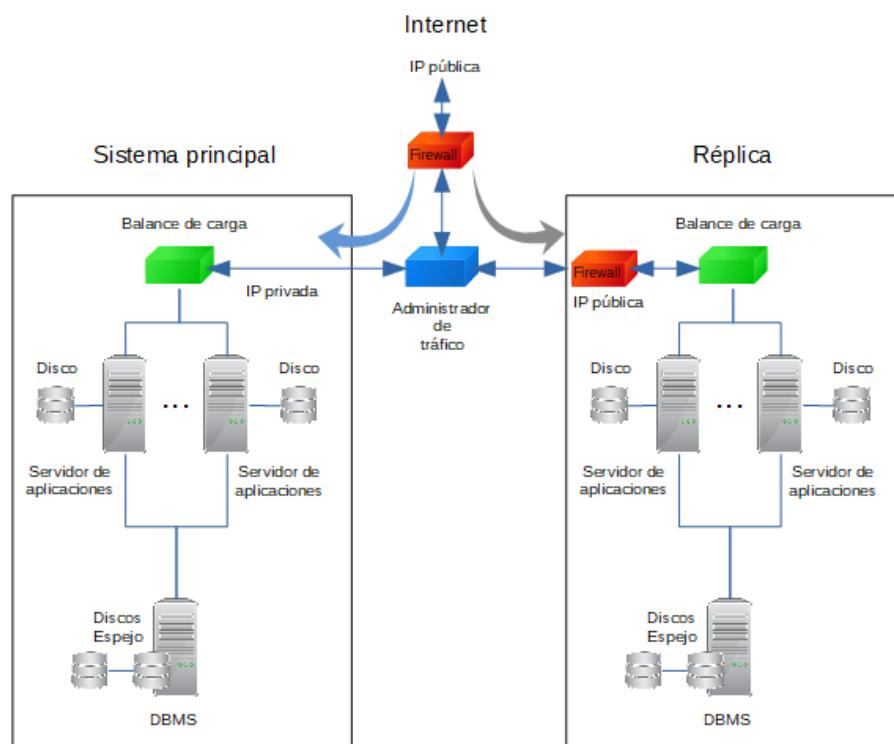
## Replicación de un sistema completo

En la actualidad el cómputo en la nube nos permite realizar el aprovisionamiento dinámico de recursos de forma fácil (automática), rápida y a bajo costo.

Entonces, ¿por qué no replicar el sistema completo?

### Arquitectura de un sistema replicado

La siguiente figura muestra la arquitectura general de un sistema replicado:



En este caso se supone que el sistema consta de dos o más servidores de aplicaciones conectados a un servidor de bases de datos, el cual tiene implementada la replicación de datos mediante discos espejo.

El administrador de tráfico funciona como un proxy transparente subrogado (*reverse transparent proxy*) el cual recibe las conexiones y peticiones de los clientes y las envía al sistema principal con copia a la réplica.

El sistema principal procesa las peticiones del cliente y envía las respuestas al administrador de tráfico, el cual las re-envía al cliente. Notar que el administrador de tráfico deberá ignorar las respuestas de la réplica.

Para garantizar la recuperación en caso de desastre catastrófico en el *site* dónde ejecuta el sistema principal, la réplica deberá ejecutar en una locación diferente, por lo tanto el administrador de tráfico y la réplica deberán tener cada uno una IP pública.

Mientras el sistema principal puede estar ejecutando en la nube o en una instalación propia de la empresa (*on-premise*), la réplica estaría funcionando en la nube, en alguna locación geográfica diferente a la locación dónde ejecuta el sistema principal.

El administrador de tráfico puede ser un programa o un *appliance* localizado en mismo *site* dónde ejecuta el sistema principal, de manera que el administrador de tráfico se pueda conectar al sistema principal mediante una red privada.

Como puede observarse, tanto el sistema principal como la réplica realizarán las mismas transacciones sobre la base de datos (y/o el sistema de archivos), por lo que la consistencia de los datos está garantizada.

Si se produce un error en la comunicación entre el administrador de tráfico y la réplica, el cliente deberá recibir un de error de comunicación.

En caso de falla del sistema principal y/o falla del administrador de tráfico, solo habrá que re-definir el dominio del sistema a la IP pública de la réplica, entonces los clientes estarán conectados directamente al sistema de respaldo.

Desde luego, posteriormente será necesario: 1) sacar de producción la réplica, 2) realizar una copia espejo de la réplica, y 3) restaurar el sistema principal y el administrador de tráfico.



- **Actividades individuales a realizar**

En esta actividad vamos a realizar un ejercicio de replicación de un sistema completo, en este caso la replicación de una plataforma de servicios web con Tomcat y MySQL.

Como vimos en clase, para replicar un sistema, podemos crear una máquina virtual en la nube (réplica) que procese todas las peticiones que realizan los clientes, en paralelo al proceso de las mismas peticiones que realiza el sistema principal.

Vamos a utilizar el programa [SimpleProxyServer.java](#) el cual es un proxy escrito en Java, modificado por el profesor para que funcione como un administrador de tráfico.

Se deberá realizar lo siguiente:

1. Crear dos máquinas virtuales en la nube de Azure con Ubuntu 18, 1 GB de RAM y disco HDD estándar a partir de la imagen creada en la tarea 6.
2. Abrir el puerto 80 protocolo TCP en la máquina virtual 1.
3. Abrir el puerto 8080 protocolo TCP en la máquina virtual 2, ingresar en el campo "Origen" ("Source" si la pantalla está en inglés) la IP de la máquina virtual 1 (por seguridad, la máquina virtual 1 es la única computadora que podrá acceder la máquina virtual 2).
4. Conectar a la máquina virtual 1 (sistema principal) utilizando el programa ssh.
5. Utilizando el programa sftp enviar a la máquina virtual 1 el archivo: [SimpleProxyServer.java](#)
6. Compilar en la máquina virtual 1 el programa [SimpleProxyServer.java](#)
7. Iniciar Tomcat en las máquinas virtuales 1 y 2.
8. Ejecutar el máquina virtual 1 el proxy:

```
sudo java SimpleProxyServer ip-maquina-virtual-2
8080 80 8080 &
```

Donde *IP-máquina-virtual-2* es la IP de la réplica, 8080 es el puerto abierto en la réplica (servidor Tomcat remoto), 80 es el puerto abierto en el sistema principal (proxy local) y 8080 es el puerto en la máquina virtual 1 donde Tomcat recibe las peticiones (puerto de Tomcat local). Notar que no es necesario abrir el puerto 8080 en la máquina virtual 1, ya que el proxy y Tomcat se comunican localmente mediante *loopback*.

En este caso ejecutamos el proxy con "sudo" para que este pueda abrir el puerto 80 en la máquina virtual 1.

### Probar el servicio web utilizando HTML-Javascript

9. En la computadora local (Windows, Linux o MacOS):

9.1 Ingresar la siguiente URL en un navegador, notar que no es necesario ingresar el nombre del puerto, ya que se utiliza el puerto default 80:

<http://ip-máquina-virtual-1/prueba.html>

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

9.3 Mostrar los registros insertados en la base de datos en la máquina virtual principal y la réplica (no desplegar el contenido del campo foto).

9.4 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”.

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

9.6 Mostrar los registros modificados en la base de datos en la máquina virtual principal y la réplica.

9.7 Consultar el usuario modificado, para verificar que la modificación se realizó.

9.8 Dar clic en el botón “Borra usuario” para borrar el usuario.

9.9 Mostrar los registros insertados en la base de datos en la maquina virtual principal y la réplica.

9.10 Capturar el email del usuario borrado y dar clic en el botón “Consulta”.



- </>

- **5. Cómputo en la nube**

- 

En el pasado el aprovisionamiento de recursos informáticos *On-premise* (en las instalaciones de la empresa) representaba el concurso de diferentes proveedores de bienes y servicios, como eran los representantes de ventas, ingenieros de pre-venta, fabricantes de los equipos, fabricante del sistema operativo, fabricante de la base de datos, agentes aduanales, transportistas, instaladores del *site*, proveedor de energía, proveedor de comunicaciones, instaladores del hardware, instaladores del software, entre otros.

Entonces, el aprovisionamiento de recursos informáticos era un proceso complejo y tardado, el cual culminaba con el sistema en producción.

Después había que re-aprovisionar cuándo crecían las necesidades de la empresa.

### 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", así, se ofrece infraestructura virtual y física como servicio (IaaS: Infrastructure as a Service), DBMS, plataformas de desarrollo y pruebas como servicio (PaaS: Platform as a Service), aplicaciones de software como servicio (SaaS: Software as a Service) y otros servicios con la terminación "as a Service", como Data as a Service (DaaS), Disaster Recovery as a Service (DRaaS), entre otros.

### La elasticidad en la nube

Debido a que el cómputo en la nube está basado fundamentalmente en la virtualización de los recursos informáticos, este modelo de arquitectura ofrece una ventaja única, la posibilidad de hacer crecer y decrecer los recursos aprovisionados.

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 del 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.

Sin lugar a dudas, el éxito que han alcanzado las empresas proveedoras de streaming bajo demanda, se debe a que su modelo de negocio está basado en la posibilidad que les ofrece la nube para crecer y decrecer los recursos aprovisionados, a esta característica de la nube se le llama *elasticidad*.

El cómputo elástico es la habilidad de hacer crecer y decrecer rápidamente la capacidad de cómputo (CPUs), la memoria y el almacenamiento para adaptarse a la demanda.

Para implementar el cómputo elástico se utilizan herramientas de monitoreo, las cuales aprovisionan y des-aprovisionan recursos conforme son necesarios, sin detener la operación.