- 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

5. *Sistemas Distribuidos Principios y Paradigmas*, Tanenbaum, A. Van Steen, M., Ed. Pearson Educación, Segunda edición, 2008.
6. *Sistemas Distribuidos Conceptos y Diseño*, Coulouris, G. Dolimore, J. Kindberg, Ed. Pearson Educación, 2001.
7. *Mastering Cloud Computing: Foundations and Applications Programming*, Buyya, Rajkumar, Vecchiola, Christian, Ed. MK, 2013.
8. *Webservices, Theory and Practice*, Hrushikesha Mohanty, Prasant Kumar, Ed. Springer, 2018.

9. *Java Course*, http://youtube.com/watch?v=coK4jM5wvko&t=4s

Actividades individuales a realizar

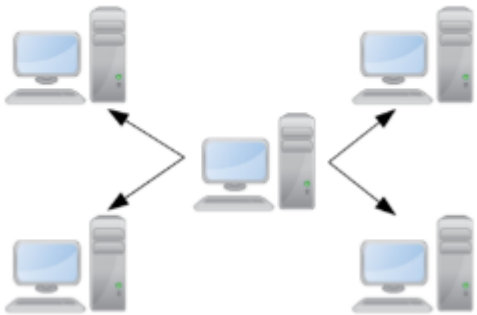
10. Obtener una cuenta de correo institucional del IPN.
11. Darse de alta en el programa Azure for Students.
12. Instalar en su computadora el JDK8 o superior.

- Tipos de comunicaciones

Los tipos de comunicaciones entre computadoras son los siguientes:

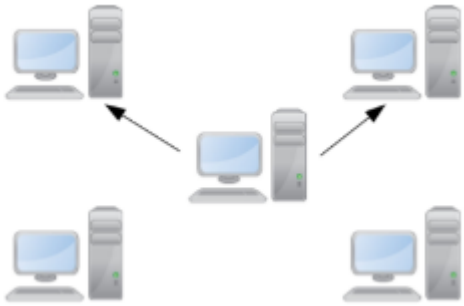
Unicast. El unicast es una comunicación punto a punto dónde una computadora envía mensajes a otra computadora.

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

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 **bi-direccional**, debido a que los datos pueden fluir en ambas direcciones.

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

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

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

Cliente.java

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

Primeramente vamos a crear un socket que se conectará al servidor. En este caso el servidor se llama "localhost"

(computadora local) y el puerto abierto en el servidor es el 50000. En general el nombre del servidor puede ser un nombre de dominio (como midominio.com o una dirección IP). El número de puerto es un número entero entre 0 y 65535.

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

En este caso declaramos una variable de tipo Socket llamada "conexión" la cual va a contener una instancia de la clase Socket. Es importante aclarar que antes de crear el socket, el programa servidor debe estar en ejecución y esperando una conexión, de otra manera la instrucción anterior produce una excepción, la cual desde luego debería controlarse dentro de un bloque **try**.

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

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

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

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

Ahora podemos enviar y recibir datos del servidor. Veamos algunos ejemplos.
Vamos a enviar un entero de 32 bits, en este caso el número 123, utilizando el método **writeInt**:

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

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

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

Vamos a enviar la cadena de caracteres "hola":

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

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

```
byte[] a = b.array();
```

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

```
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 BytetBuffer utilizando el método **wrap** de la clase ByteBuffer:

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

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

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

Finalmente, cerramos la conexión con el cliente:

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

- Actividades individuales a realizar

0. Compile los programas **Cliente.java** y **Servidor.java**
1. 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).
2. 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()
3. 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.
4. 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.
5. 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.
6. ¿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.

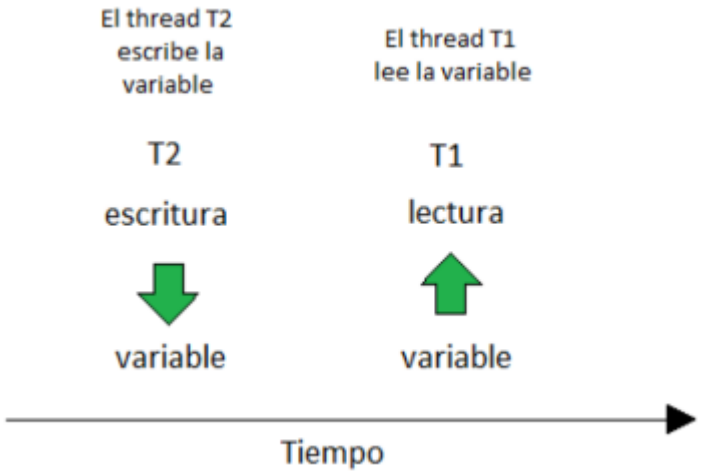
Por otra parte, si un programa crea un número de threads mayor al número de procesadores físicos (*cores*) disponibles en la computadora, entonces los threads ejecutarán en forma concurrente (por turnos).

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.

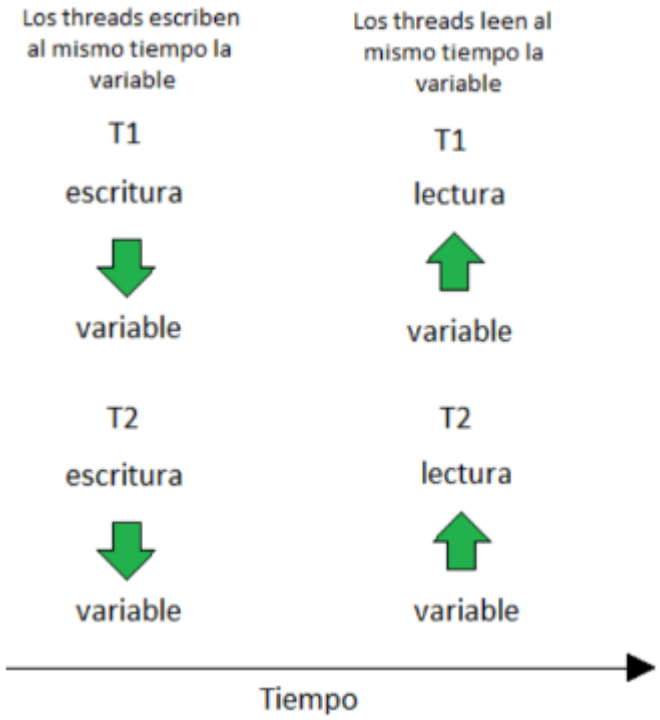
Supongamos que tenemos dos threads, el thread T1 y el thread T2. Si el thread T2 escribe a una variable y posteriormente el

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?

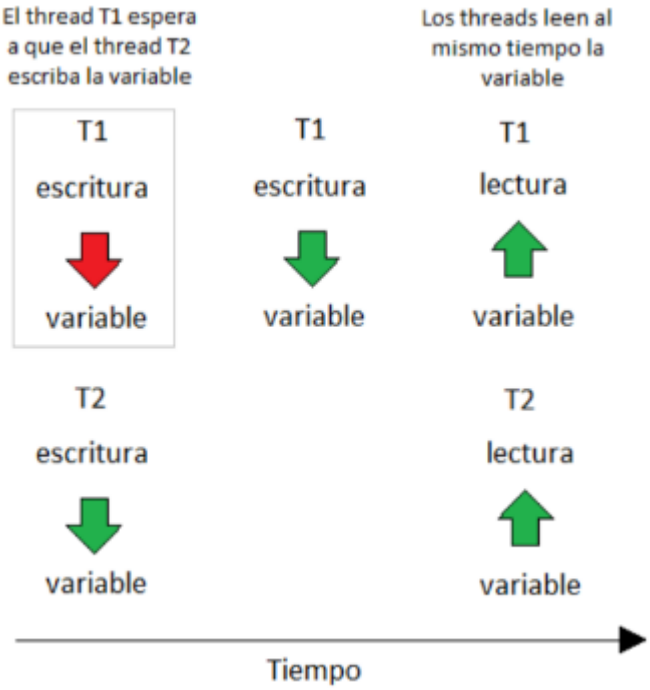


Evidentemente no es posible garantizar que el thread T1 haya leído el valor que escribió el thread T2 o que haya leído el valor que escribió el mismo thread T1. Igualmente, no es posible garantizar que el thread T2 haya leído el valor que escribió el

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.

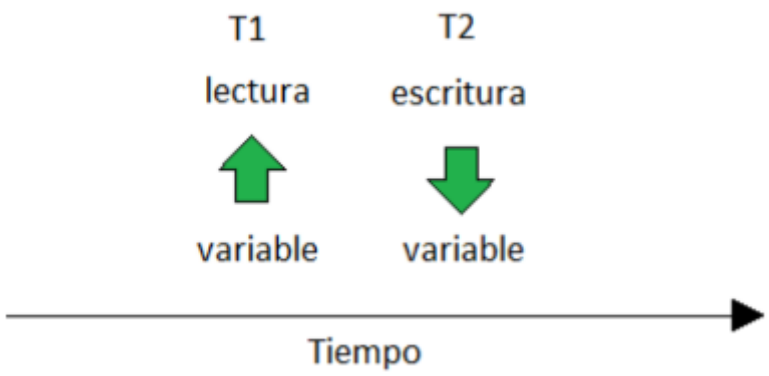


En este caso, el valor que leen los threads T1 Y T2 es el valor que escribió el thread T1.

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

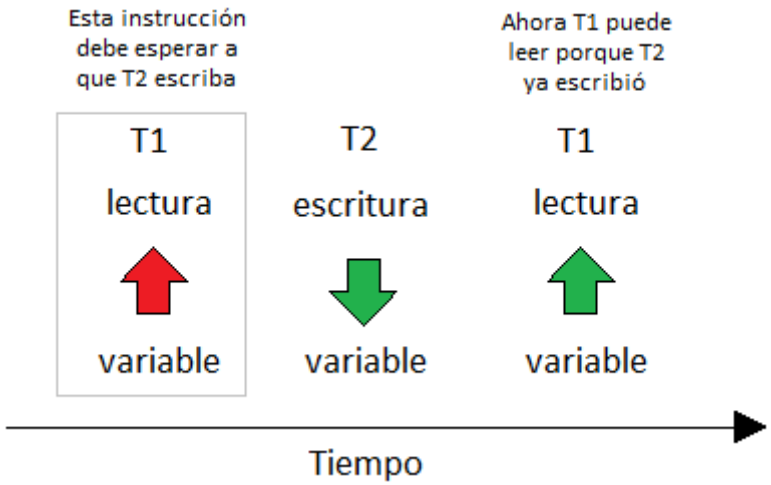
Orden escritura-lectura

Ahora supongamos que el therad 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.



Cuándo 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.

- Actividades individuales a realizar

0. Compile la clase A que no utiliza sincronización y la clase que utiliza sincronización
    - ¿Por qué el programa sin sincronización despliega un valor incorrecto?
    - ¿Por qué cada vez que se ejecuta el programa sin sincronización despliega un valor diferente?
    - ¿Por qué el programa con sincronización es más lento?
1. Compile los programas **Cliente2.java** y **Servidor2.java**

2. Ejecute el programa **Cliente2.java** en una ventana de comandos de Windows (o terminal de Linux) y 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.
3. Ejecute repetidamente el programa **Cliente2.java** en la ventana de comandos, como puede ver el servidor sigue en ejecución recibiendo las conexiones de los clientes y procesando los datos.

- **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*.

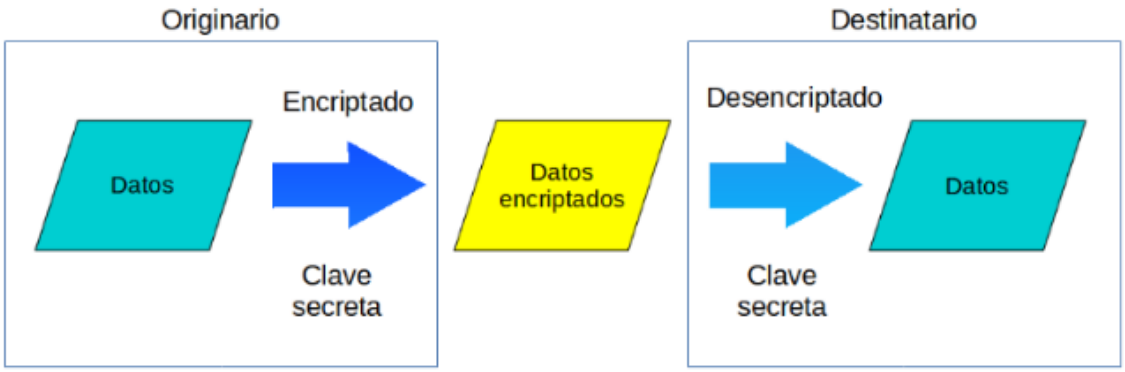
Ejemplos de algoritmos simétricos son el AES-128, AES-256, RC4, RC5, DES, 3DES, entre otros.

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.

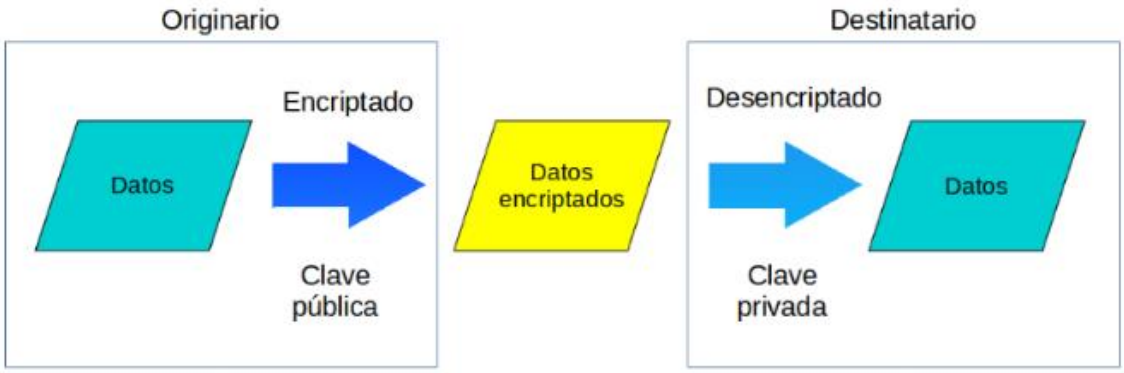


Criptografía asimétrica

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

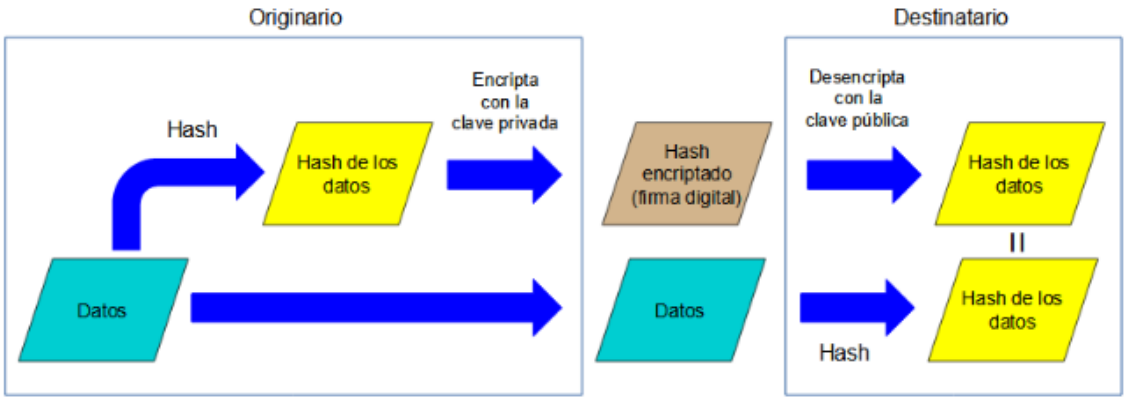
La clave privada es una clave que mantiene en secreto el destinatario de los datos, mientras que la clave pública es una clave que puede conocer cualquiera.

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 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 captura 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**).

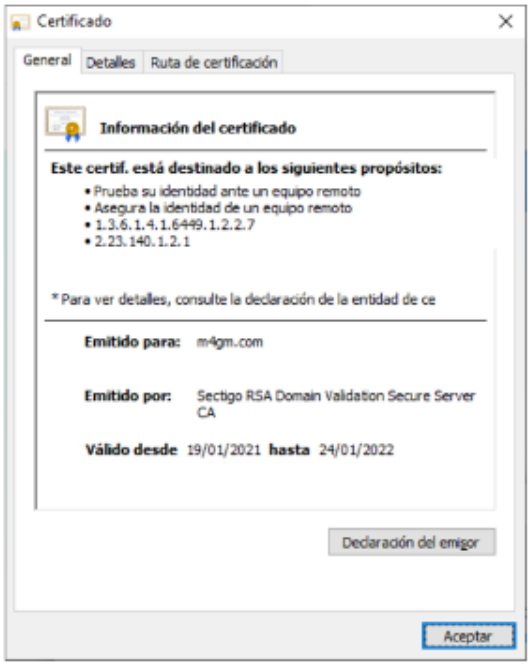
Existen dos tipos de certificados firmados por una CA, aquellos que verifican dominio y aquellos que adicionalmente verifican la empresa.

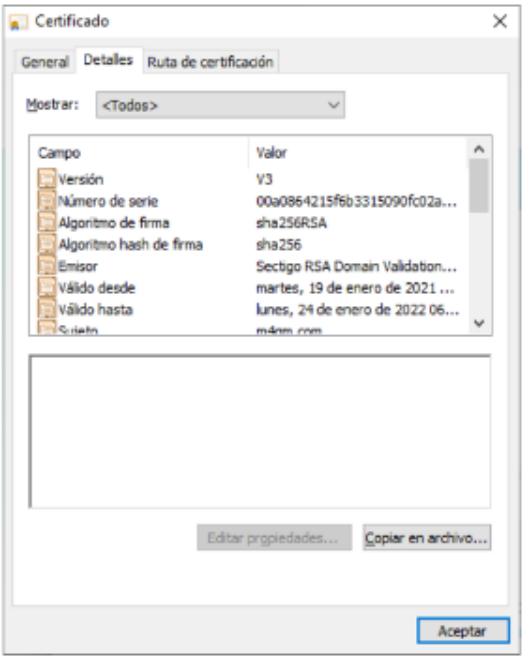
Para poder tener un certificado con **verificación de dominio**, es necesario ser propietario de un dominio.

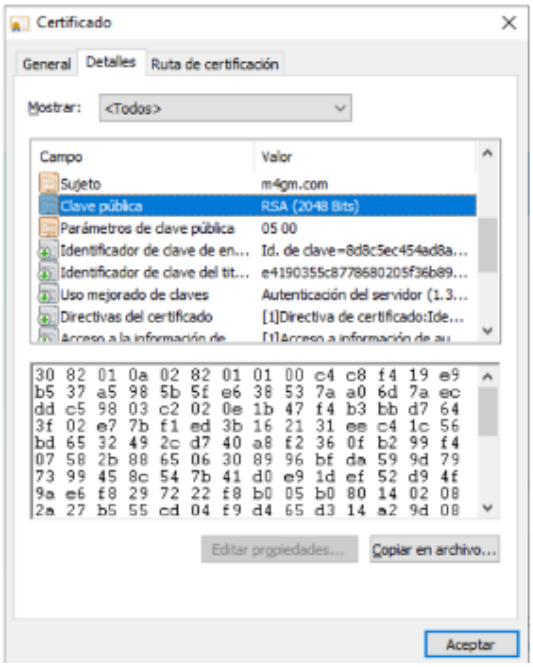
Por otra parte, para poder tener un certificado con **verificación de empres**a, es necesario tener una empresa y un dominio.

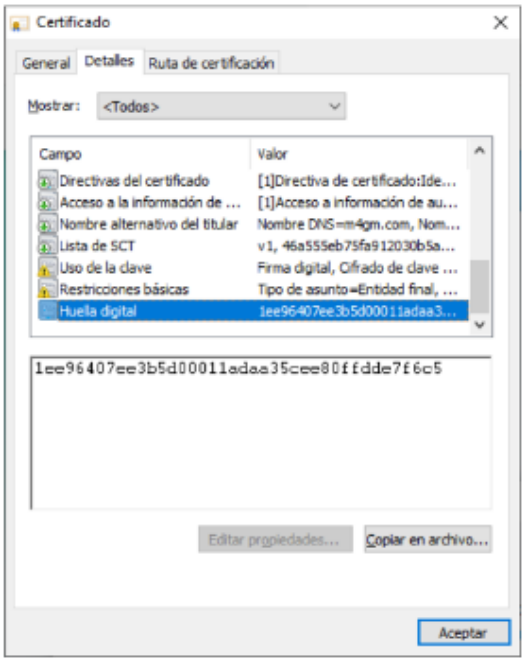
Cuando se adquiere un certificado firmado por una CA, además del certificado se obtiene un archivo conocido como *bundle*, el cual contiene los certificados de CA que forman una **ruta de certificación** desde el certificado emitido, hasta un certificado raíz pre-instalado en la computadora.

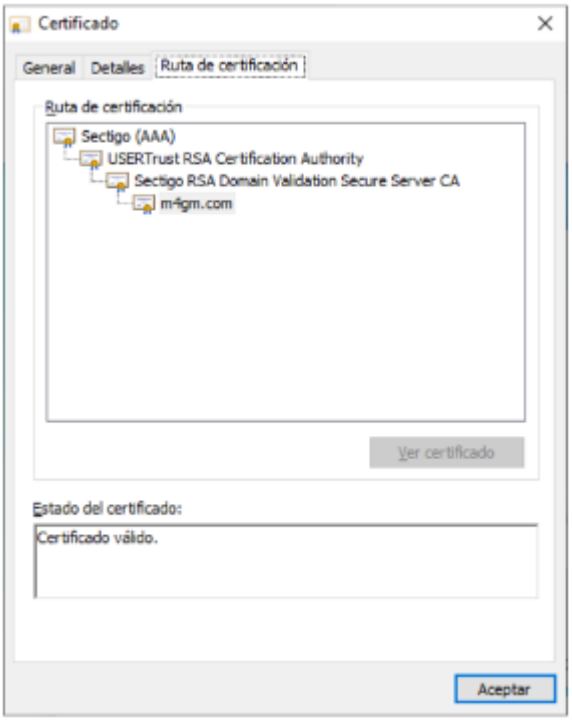
Veamos un ejemplo de certificado digital con verificación de dominio, en este caso el dominio es m4gm.com


Certificado
General
Detalles
Ruta de certificación
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
* Para ver detalles, consulte la declaración de la entidad de ce
Emitido para: m4gm.com
Emitido por: Sectigo RSA Domain Validation Secure Server CA
Válido desde 19/01/2021 hasta 24/01/2022
Declaración del emisor
Aceptar


Certificado
General
Detalles
Ruta de certificación
Mostrar: <Todos>
Campo
Valor
Versión
V3
Número de serie
00a0864215f6b3315090fc02a...
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...
Editar propiedades...
Copiar en archivo...
Aceptar


Certificado
General
Detalles
Ruta de certificación
Mostrar: <Todos>
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
[1]Directiva de certificado:Ide...
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 98 03 c2 02 0e 1b 47 f4 b3 bb d7 64
3f 02 e7 7b f1 ed 3b 16 21 31 ee c4 1c 56
bd 65 32 49 2c d7 40 a8 f2 36 0f b2 99 f4
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 08
Editar propiedades...
Copiar en archivo...
Aceptar


Certificado
General
Detalles
Ruta de certificación
Mostrar: <Todos>
Campo
Valor
Directivas del certificado
[1]Directiva de certificado:Ide...
Acceso a la información de ...
[1]Acceso a información de au...
Nombre alternativo del titular
Nombre DNS=m4gm.com, Nom...
Lista de SCT
v1, 46a555eb75fa912030b5a...
Uso de la clave
Firma digital, Cifrado de clave ...
Restricciones básicas
Tipo de asunto=Entidad final, ...
Huella digital
1ee96407ee3b5d00011adaa3...
1ee96407ee3b5d00011adaa35cee80ffdde7f6c5
Editar propiedades...
Copiar en archivo...
Aceptar



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
```

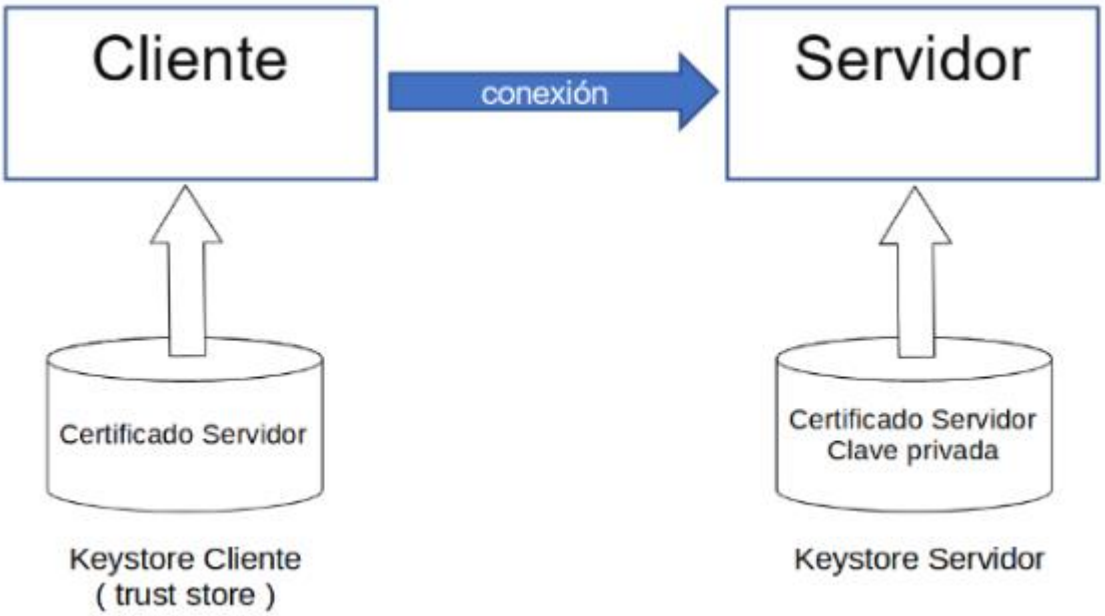
La opción **exportcert** lee del keystore el certificado identificado por el alias y genera un archivo texto que contiene el certificado, en este caso se genera el archivo certificado_servidor.pem

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

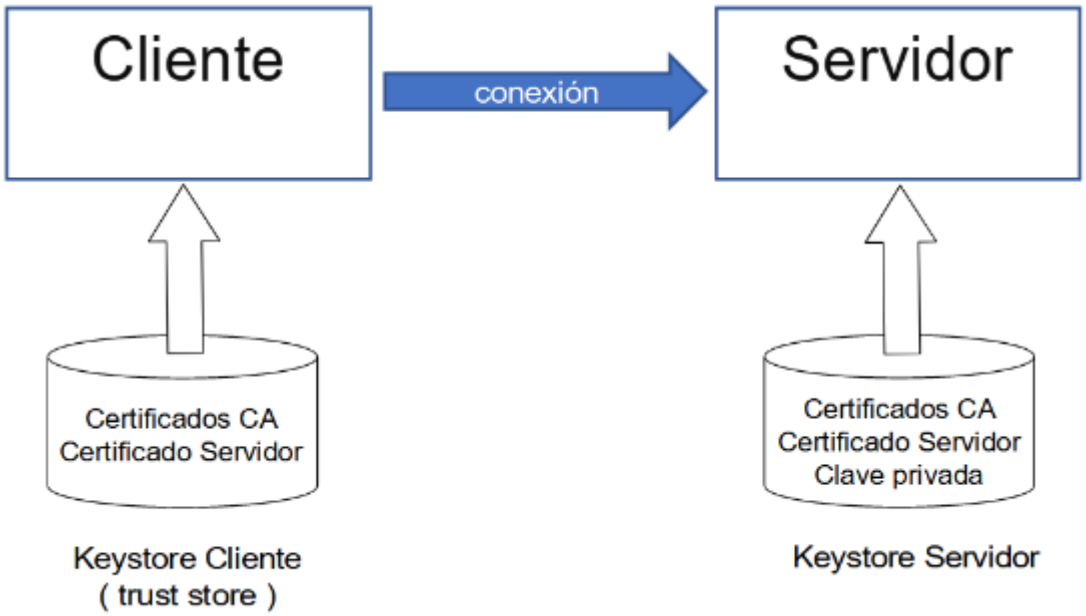
```
keytool -import -alias certificado_servidor -file
certificado_servidor.pem -keystore keystore_cliente.jks -
storepass 123456
```

La opción **import** lee el archivo certificado_servidor.pem e inserta el certificado en el keytore **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 SSLServerSocketFactory:

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

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

```
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
```

- keystore_cliente.jksArchivo
- keystore_servidor.jksArchivo
- Actividades individuales a realizar

En esta actividad vamos a desarrollar un servidor multithread que recibará 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;
}
```

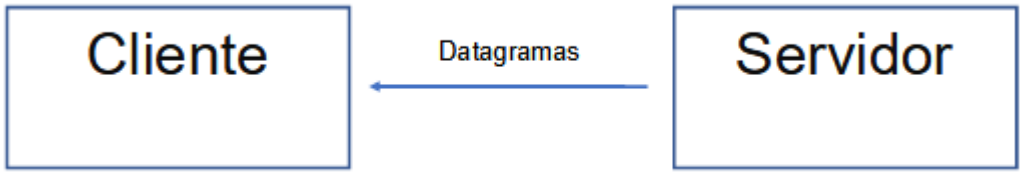
4. El cliente deberá enviar al servidor el nombre del archivo utilizando el método **writeUTF**() de la clase DataOutputStream, deberá enviar la longitud del archivo utilizando el

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

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

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

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

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

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

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

0. Compile los programas ServidorMulticast.java y ClienteMulticast.java.

1. 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 dónde cada nivel representa una capa de hardware que almacena datos.

---

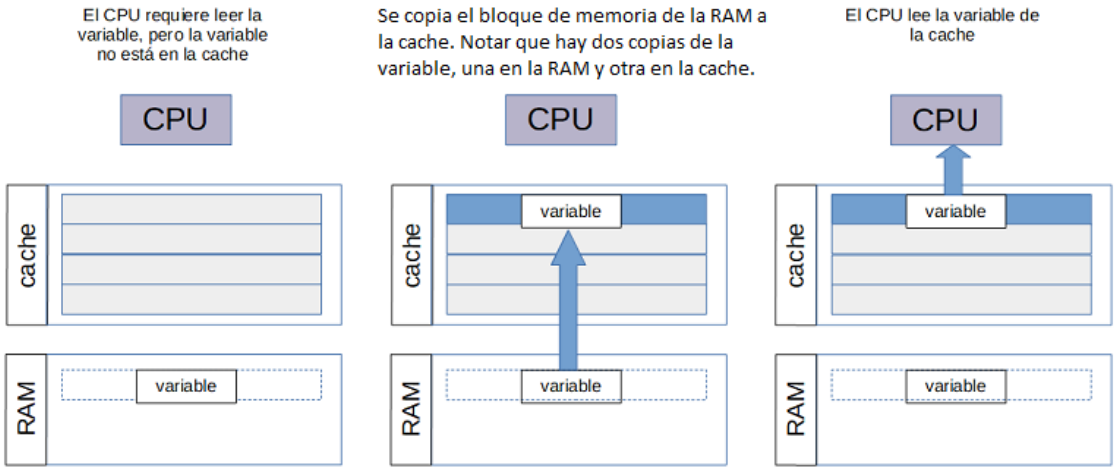
0. El CPU utiliza los **registros** para realizar operaciones aritméticas, lógicas y de control.
1. La memoria **cache** consiste en una memoria asociativa. Las memorias asociativas son muy rápidas, pero como son costosas, suelen ser de poca capacidad. La memoria cache puede estar dividida en varios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

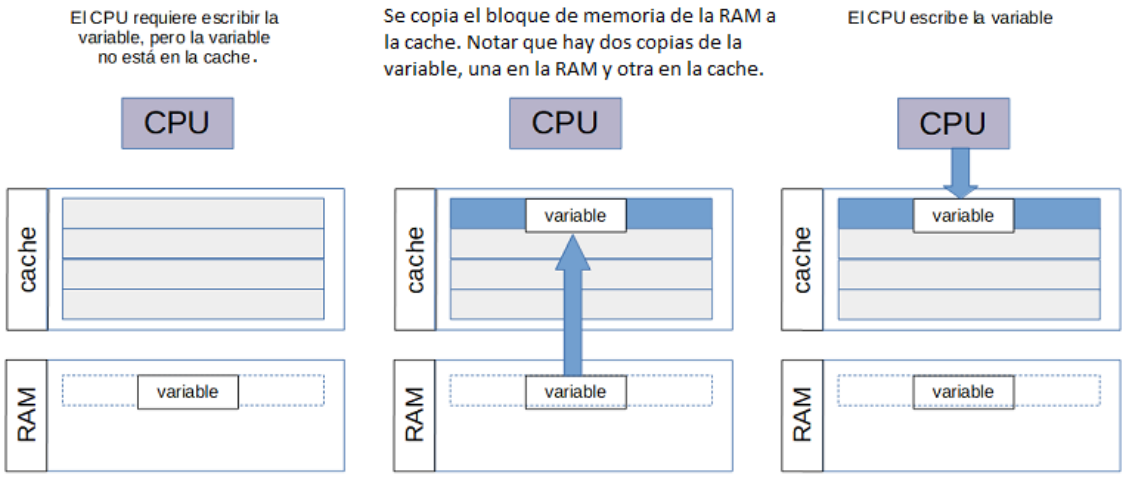
en la cache, copia el bloque de datos (que contiene a la variable) a la cache, entonces el CPU lee la variable de la cache.



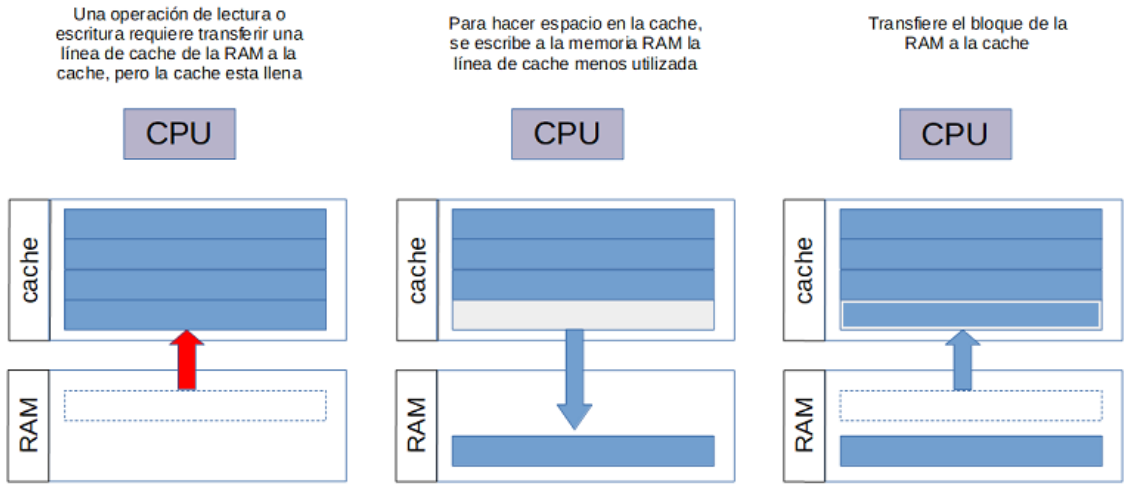
Al bloque de memoria que se transfiere de la RAM a la cache se le llama **línea de cache**. El tamaño de una línea de cache típicamente es de cientos de Kilobytes o Megabytes.

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:

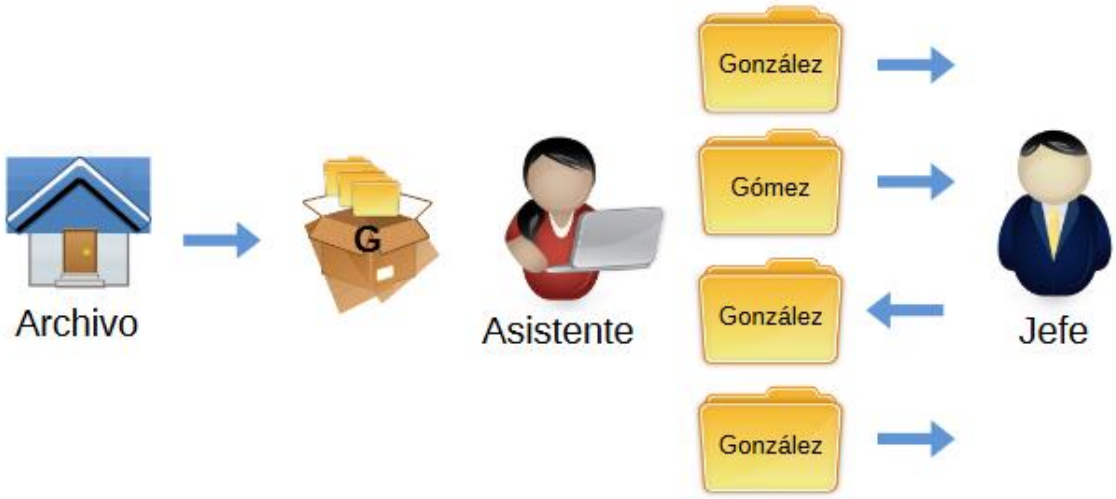
Caso 1. La asistente obtiene expedientes individuales del archivo.

4. El jefe le pide a su asistente el expediente del Sr. González.
5. La asistente va al archivo a traer el expediente del Sr. González.
6. La asistente le da a su jefe el expediente del Sr. González
7. El jefe le pide a su asistente el expediente del Sr. Gómez.
8. La asistente va al archivo a traer el expediente del Sr. Gómez.
9. La asistente le da a su jefe el expediente del Sr. Gómez.
10. El jefe regresa a su asistente el expediente del Sr. González.
11. La asistente va al archivo a dejar el expediente del Sr. González
12. El jefe le pide a su asistente el expediente del Sr. González.
13. La asistente va al archivo a traer el expediente del Sr. González.
14. 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.

15. El jefe le pide a su asistente el expediente del Sr. González.
16. La asistente va al archivo a traer la caja correspondiente a la letra "G".
17. La asistente le da a su jefe el expediente del Sr. González.
18. El jefe le pide a su asistente el expediente del Sr. Gómez.
19. La asistente le da a su jefe el expediente del Sr. Gómez.
20. El jefe regresa el expediente del Sr. González.
21. El jefe le pide a su asistente el expediente del Sr. González.
22. La asistente le da a su jefe el expediente del Sr. González.

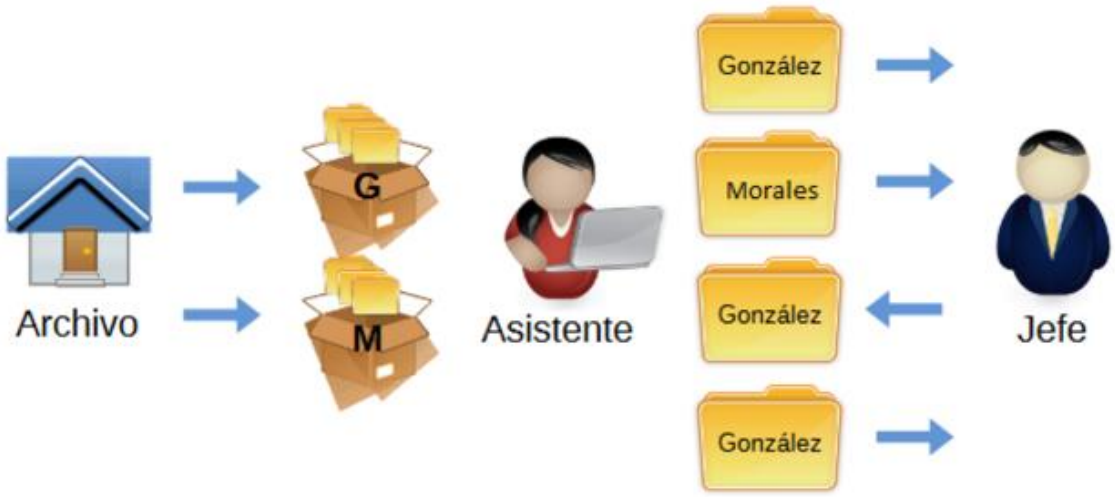


Entonces la asistente sólo va una vez al archivo. Los expedientes solicitados por el jefe se encuentran en la misma caja. El jefe pide más de una vez el expediente del Sr. González el mismo día.

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

---

23. El jefe le pide a su asistente el expediente del Sr. González.
24. La asistente va al archivo a traer la caja correspondiente a la letra “G”.
25. La asistente le da a su jefe el expediente del Sr. González.
26. El jefe le pide a su asistente el expediente del Sr. Morales.
27. La asistente va al archivo a traer la caja correspondiente a la letra “M”.
28. La asistente le da a su jefe el expediente del Sr. Morales.
29. El jefe regresa el expediente del Sr. González.
30. El jefe le pide a su asistente el expediente del Sr. González.
31. 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++)
      {
        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.

- MultiplicaMatriz.javaArchivo

- MultiplicaMatriz_2.javaArchivo

- Actividades individuales a realizar

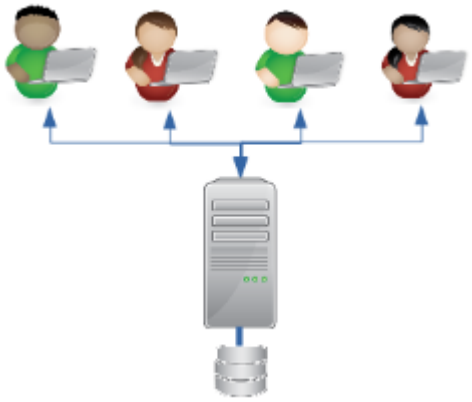
0. Compilar y ejecutar los programas MultiplicaMatriz.java y MultiplicaMatriz_2.java que vimos en clase, para los siguientes valores de N: 100, 200, 300, 500, 1000.
1. Utilizando Excel o LibreOffice Calc hacer una gráfica de dispersión (con líneas, sin marcadores) dónde se muestre el tiempo de ejecución de ambos programas con respecto a N (N en el eje X y el tiempo en el eje Y).
2. ¿Por qué el segundo programa es más rápido que el primero?
3. ¿Podría plantear otro programa dónde el aumento de la localidad espacial y/o temporal hace más eficiente la ejecución?

- **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 dónde el código y los datos residen en una sola computadora.



Un sistema centralizado tiene las siguientes ventajas:

- **Facilidad de programación**. Los sistemas centralizados son fáciles de programar, ya que no existe el problema de comunicar diferentes procesos en diferentes computadoras, tampoco es un problema la consistencia de los datos debido a que todos los procesos ejecutan en una misma computadora con una sola memoria.
- **Facilidad de instalación**. Es fácil instalar un sistema central. Basta con instalar un solo *site* el cual va a requerir una acometida de energía eléctrica, un sistema de enfriamiento (generalmente por agua), conexión a la red de datos y comunicación por voz. Más adelante en el curso veremos cómo el cómputo en la nube está cambiando la idea de instalación física en pos de sistemas virtuales en la nube.
- **Facilidad de operación**. Es fácil operar un sistema central, ya que la administración la realiza un solo equipo de operadores, incluyendo las tareas de respaldos, mantenimiento preventivo y correctivo, actualización de versiones, entre otras.
- **Seguridad**. Es fácil garantizar la seguridad física y lógica de un sistema centralizado. La seguridad física se implementa mediante sistemas CCTV, controles de cerraduras electrónicas, biométricos, etc. La seguridad lógica se implementa mediante un esquema de permisos a los diferentes recursos como son el sistema operativo, los archivos, las bases de datos.
- **Bajo costo**. Dados los factores anteriores, instalar un sistema centralizado resulta más barato que un sistema distribuido ya que solo se pagan licencias para un servidor,

sólo se instala un *site*, se tiene un solo equipo de operadores.

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

- **El procesamiento es limitado**. El sistema centralizado cuenta con un número limitado de procesadores, por tanto a medida que incrementamos el número de procesos en ejecución, cada proceso ejecutará más lentamente. Por ejemplo, en Windows podemos ejecutar el Administrador de Tareas para ver el porcentaje de CPU que utiliza cada proceso en ejecución, si la computadora ha llegado a su límite, entonces veremos que el porcentaje de uso del CPU es 100%.
- **El almacenamiento es limitado**. Un sistema centralizado cuenta con un número limitado de unidades de almacenamiento (discos duros). Cuando un sistema llega al límite del almacenamiento se detiene, ya que no es posible agregar datos a los archivos ni realizar *swap*.
- **El ancho de banda es limitado**. Un sistema centralizado puede llegar al límite en el ancho de banda de entrada y/o de salida, en estas condiciones la comunicación con los usuarios se va a alentar.
- **El número de usuarios es limitado**. Un sistema centralizado tiene un máximo de usuarios que se pueden conectar o que pueden consumir los servicios. Por ejemplo, por razones de licenciamiento los manejadores de bases de datos tienen un máximo de usuarios que pueden conectarse, así mismo, el sistema operativo tiene un límite en el número de *descriptores de archivos* que puede crear. Recordemos que cada vez que se abre un archivo y cada vez que se crea un socket se ocupa un descriptor de archivo.
- **Baja tolerancia a fallas**. En un sistema centralizada una falla suele ser catastrófica, ya que sólo se tiene una computadora y una memoria. Cualquier falla suele producir la inhabilitación del sistema completo.

Ejemplos de sistemas centralizados

Un servidor Web centralizado

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

Un DBMS centralizado

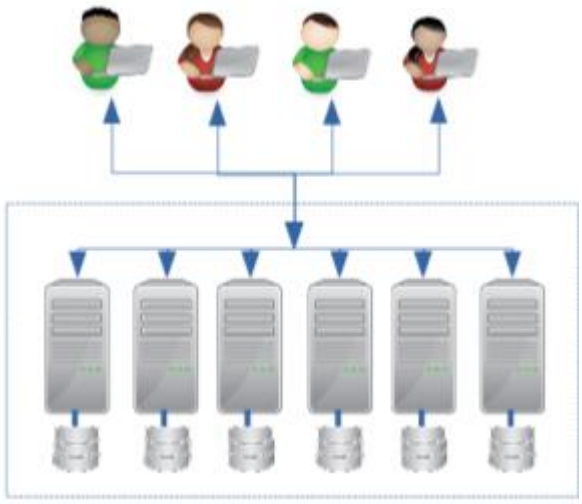
Generalmente los sistemas manejadores de bases de datos (DBMS) son centralizados debido a que resulta más fácil programar sistemas que accedan los datos que se encuentran en una base de datos central. Sin embargo, las plataformas de alcance mundial como Facebook, Twitter, Uber, etc. requieren distribuir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

Una computadora stand-alone

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

Sistema distribuido

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



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

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

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

Las ventajas de un sistema distribuido son, entre otras:

- **El procesamiento es (casi) ilimitado**. Un sistema distribuido puede tener un número casi ilimitado de CPUs ya que siempre será posible agregar más servidores, por tanto a medida que incrementamos el número de CPUs podemos esperar que los procesos ejecuten más rápido debido a que los procesos ejecutarán en paralelo en diferentes CPUs. El límite del paralelismo queda definido por la **ley de Amdahl**.
- **El almacenamiento es (casi) ilimitado**. Un sistema distribuido cuenta con un número casi ilimitado de

unidades de almacenamiento (discos duros). Siempre es posible conectar más servidores de almacenamiento.
- **El ancho de banda es (casi) ilimitado**. En un sistema distribuido cada computadora aporta su ancho de banda, esto es, en la medida que agregamos servidores podemos enviar y recibir una mayor cantidad de datos por unidad de tiempo (es decir, aumentamos el ancho de banda).
- **El número de usuarios es (casi) ilimitado**. El número de usuarios que pueden conectarse a un sistema distribuido aumenta en la medida que agregamos servidores. Si bien es cierto que cada servidor tiene un límite en el número de *descriptores de archivos*, y con ello un límite al número de conexiones que puede abrir, cada servidor en el sistema distribuido agrega descriptores (conexiones).
- **Alta tolerancia a fallas**. En un sistema distribuido la falla de un servidor no es catastrófica, ya que el sistema está diseñado para retomar el trabajo que realizaba el servidor que falla. Más adelante en el curso veremos las estrategias que se utilizan en los sistemas distribuidos para la replicación de datos y la replicación del sistema completo.

Las desventajas de los sistemas distribuidos son:

- **Dificultad de programación**. La definición que hace Tanenbaum de los sistemas distribuidos implica que los usuarios del sistema tienen la impresión de utilizar un sistema único, esto incluye a los programadores. Sin embargo, en la realidad actual los sistemas distribuidos son difíciles de programar ya que el programador es quien el que tiene que implementar la comunicación entre los diferentes componentes del sistema.
- **Dificultad de instalación**. Es complicado instalar un sistema distribuido. Es necesario interconectar múltiples computadoras, lo cual implica la necesidad de una red de alta velocidad.
- **Dificultad de operación**. Es complicado operar un sistema distribuido, ya que se requiere un equipo de administración por cada *site*. Los equipos deberán coordinarse para realizar las tareas de respaldos, mantenimiento preventivo y correctivo, actualización de versiones, entre otras.
- **Seguridad**. Es complicado garantizar la seguridad física y lógica de un sistema distribuido. Tanto la seguridad física

como la seguridad lógica requieren la coordinación de múltiples equipos dedicados a la seguridad del sistema. La interconexión remota de los diferentes servidores implica el riesgo de ataques al sistema a través de los puertos de comunicación.
- **Alto costo**. Instalar un sistema distribuido resulta más costoso que un sistema centralizado ya que será necesario pagar licencias para cada servidor, para cada *site* se requiere un equipo de operadores, así mismo, cada *site* requiere su propia acometida de energía, un sistema de seguridad física, infraestructura de refrigeración, etc.

Tipos de distribución

Distribución del procesamiento

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

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

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

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

Distribución de los datos

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

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

Ejemplos de sistemas distribuidos

World Wide Web

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

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

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

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

Cómputo en la nube

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

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

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

- Platform as a Service (**PaaS**): DBMS, plataformas de desarrollo y pruebas como servicio.
- Software as a Service (**SaaS**): aplicaciones de software como servicio

SETI

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

TOP500

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 direción 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:

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

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

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

1. Dar click al icono de "hamburguesa" (las tres líneas horizontales) localizado en la parte superior izquierda de la pantalla.
2. Seleccionar "Todos los recursos".
3. Seleccionar cada recursos (dar click en cada checkbox)
4. Seleccionar "Eliminar".
5. Verificar la lista de recursos a eliminar.
6. Escribir la palabra: sí (con acento en la i).
7. Dar click en el botón "Eliminar".

Ver los videos:

- **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:

0. Facilidad en el acceso a los recursos.
1. Transparencia.
2. Apertura.
3. 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 utilizara el protocolo HTTPS para acceder el 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, mas 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 cuándo 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

Posiblemente la técnica más interesante para escalar un sistema es la elasticidad en la nube. La **elasticidad** es la adaptación a los cambios en la carga mediante aprovisionamiento y des-aprovisionamiento de recursos en forma automática. 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 sub-utilizados los días entre semana, lo cual resulta en pérdidas económicas.

Entonces la solución es utilizar la posibilidad que les ofrece la nube para crecer y decrecer los recursos aprovisionados en forma automática. Más adelante en el curso veremos cómo utilizar la elasticidad en la nube.

- Actividades individuales a realizar

En esta actividad 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:

- **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 particiona físicamente el sistema, la organización del sistema en sub-sistemas

de diseño, la especificación del entorno tecnológico, los requisitos de operación, administración, seguridad, control de acceso, así como los requisitos de calidad, esto es, las características que el sistema debe cumplir.

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

Calidad de Servicio (QoS)

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

Balance de carga

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

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

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

Tolerancia a fallas

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

Los requisitos de tolerancia a fallas de un sistema distribuido definen las estrategias que el sistema implementará para soportar la falla en determinados componentes, algunas estrategias

empleadas para la tolerancia a fallas son la replicación de datos y la replicación de código.

Seguridad

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

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

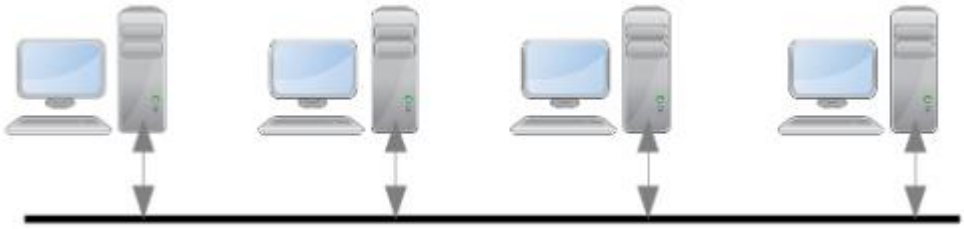
Por tanto, es muy importante definir los **requisitos de seguridad** para el sistema, entre otros: seguridad física del sistema, comunicación encriptada (SSL, TLS, HTTPS), utilización de usuarios no administrativos, configuración detallada de los permisos, programar para la prevención de ataques (p.e.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

Tipos de sistemas distribuidos

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

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

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

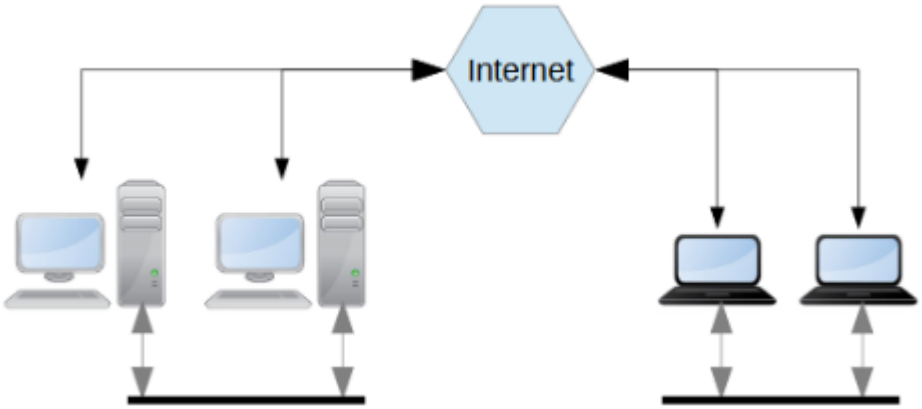


Los clusters se utilizan para el cómputo de alto rendimiento, dónde los programas se distribuyen entre los

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

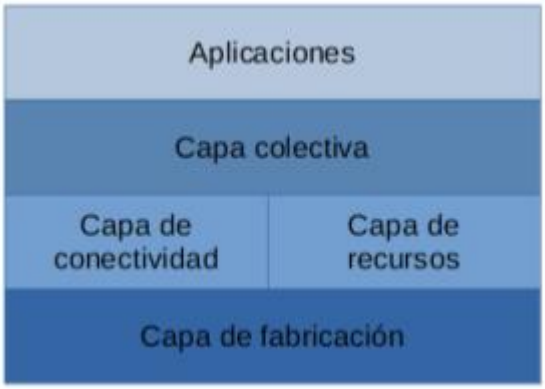
En el TOP500 474 sistemas son clusters, mientras que sólo 40 sistemas son MPP (*Massively Parallel Processing*). Un ejemplo de sistema MPP es la malla (grid).

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



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

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



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

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

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

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

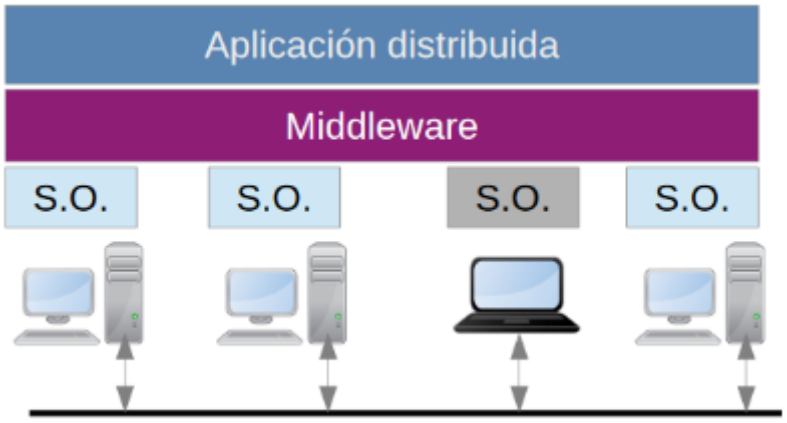
La **capa de aplicaciones** está compuesta por las aplicaciones que ejecutan dentro de la organización virtual.

Middleware

Un middleware (software 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 deferentes 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:

Captura de una imagen administrada de una máquina virtual generalizada en Azure

Información y uso del agente de Linux de Azure

- 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".

- 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.

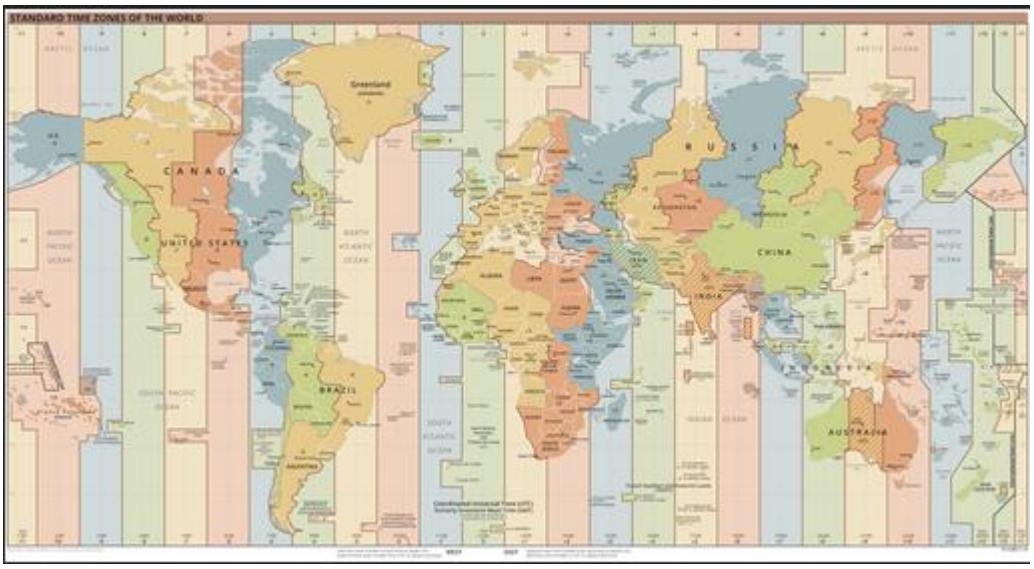
Supongamos una plataforma de comercio electrónico que funciona a nivel global, en cada país se tiene un servidor con una base de datos dónde se registran las compras, incluyendo la fecha y hora en la que se realiza cada compra.

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:

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

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

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



Mapa de los husos horarios oficiales vigentes (dominio público)

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

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

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

Sincronización de relojes

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

grupos de servidores se conectan entonces ambos grupos deberán acordar una hora.

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

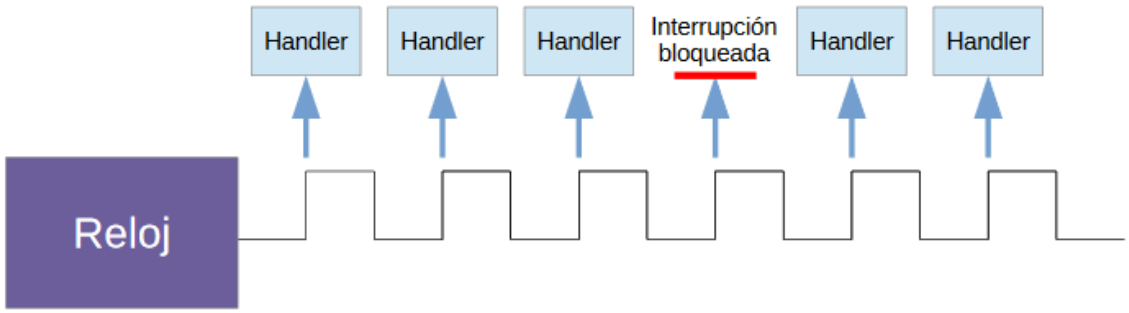
Más adelante veremos que éste orden puede establecerse utilizando relojes físicos (mecanismos que marcan el tiempo real) o bien relojes lógicos (contadores).

Relojes físicos

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

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

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



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

---

2. Los relojes físicos se construyen utilizando cristales de cuarzo con la finalidad de tener un periodo de oscilación constante, sin embargo los cambios en la temperatura

modifican el periodo del pulso, lo que ocasiona que el reloj se adelante o se atrase.
3. Cuando se produce la interrupción al CPU, el sistema podría estar ejecutando una rutina de mayor prioridad, por tanto la rutina que incrementa los "ticks" se bloquea lo que provoca que algunos pulsos de reloj no incrementen la cuenta de "ticks".

Segundos solares

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

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

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

Segundos atómicos

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

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

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

La frecuencia natural de resonancia del Cesio 133 es de 9,192,631,770 ciclos/segundo, es decir, el átomo de Cesio 133

muestra un máximo de absorción de energía cuando se le aplica microondas con una frecuencia de 9,192,631,770 Hertzios.

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

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

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

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

Tiempo atómico internacional TAI

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

Tiempo universal coordinado UTC

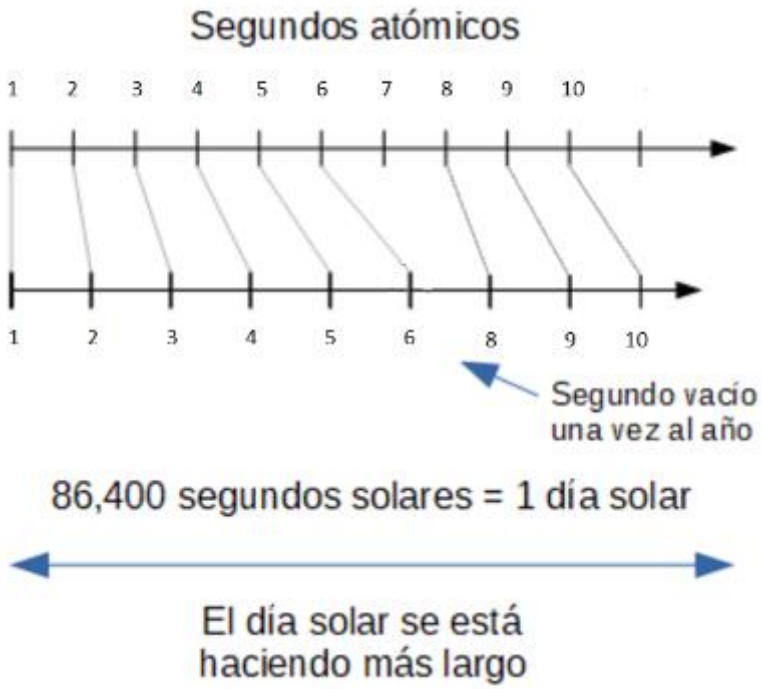
El tiempo universal coordinado UTC (*Coordinated Universal Time*, CUT) es el estándar de tiempo que regula actualmente el tiempo de los relojes a nivel internacional.

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

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

Debido a que nuestro planeta disminuye su velocidad angular lentamente, el segundo solar dura más que el segundo atómico. Para sincronizar los segundos UTC con los segundos TAI, el tiempo UTC se debe "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 esta 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 sicronizado con una computadora que tenga un reloj atómico.

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

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

Dado que la computadora A conoce sus tiempos locales T1 y T4, si el servidor B envía sus tiempos locales T3 y T2 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:

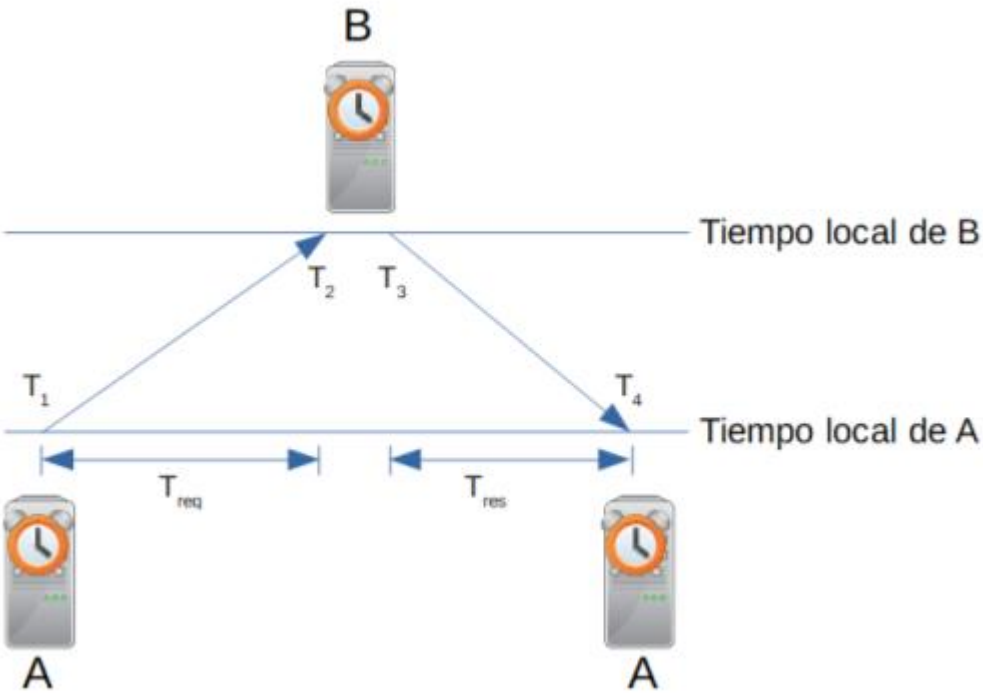
$T4-T1 = T_{req}+T_{res}+(T3-T2) = 2T_{res}+(T3-T2)$

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

$T_{res}= ((T4-T1)-(T3-T2))/2$

Por tanto, cuando la computadora A recibe el mensaje de la computadora B, la computadora A cambiará su tiempo local a $\mathbf{T3+T_{res}}$.

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.

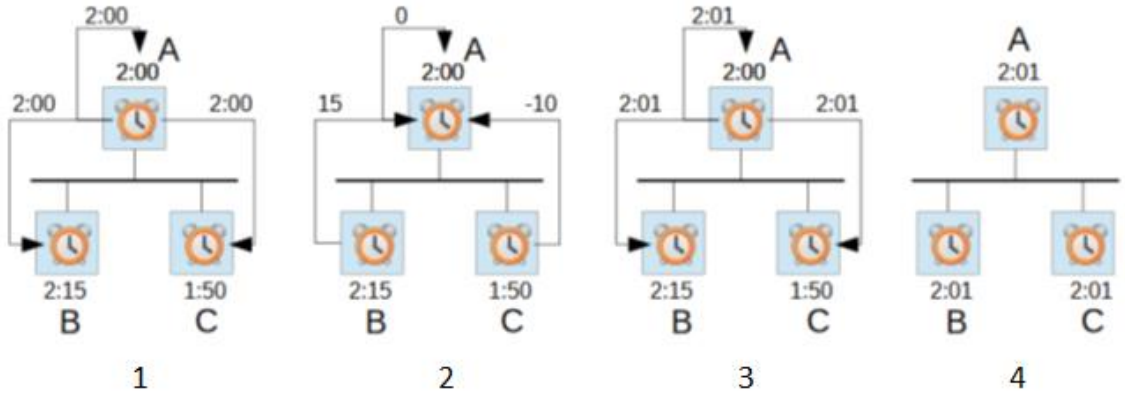
El algoritmo de Berkeley es el siguiente:

---

0. El nodo A (servidor) le envía a los nodos A, B y C su tiempo.
1. Los nodos A, B y C les envían al nodo A las diferencias entre sus tiempos y el tiempo en el nodo A.

2. El nodo A calcula el promedio de las diferencias. El nodo A envía a los nodos A, B y C la corrección de tiempo.
3. 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.