Tema 4.
Sistemas distribuidos

Sincronización en sistemas distribuidos

Marisol García Valls

Departamento de Ingeniería Telemática
Universidad Carlos III de Madrid
mvalls@it.uc3m.es

Arquitectura de sistemas II

Grado en Ingeniería Telemática
Curso: 3º, cuatrimestre: 2º

Universidad Carlos III de Madrid

Índice

- Introducción a la sincronización en sistemas distribuidos.
- Sincronización mediante relojes.
  - Relojes lógicos y relojes físicos.
  - Algoritmos de sincronización de relojes lógicos
  - Algoritmos de sincronización de relojes físicos.
  - Aplicaciones de sincronización de relojes
- Exclusión mutua en sistemas distribuidos.
  - Algoritmos centralizados, distribuidos y en anillo.
- Algoritmos de elección.
- Transacciones atómicas.

Bibliografía

A.S. Tanenbaum. *Distributed Operating Systems*. Pearson. 2009

Capítulo 3.

A. S. Tanenbaum, M. Van Steen. Distributed systems: Principles and paradigms.
(Anteriormente editado por Pearson Education.)
Publicado pro M. Van Steen. 2016.

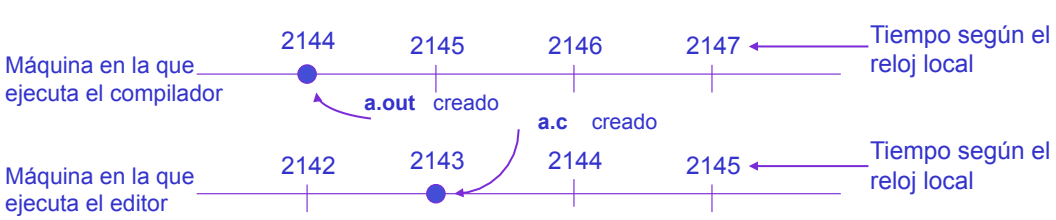
Capítulo 6.

Comunicación y sincronización distribuida

- La **comunicación** entre procesos puede realizarse mediante protocolos de capas, paso de mensajes (con protoc. petición-respuesta), RPC y comunicación de grupos.
- Los procesos no sólo comunican sino que deben **cooperar** y **sincronizarse**.
- ¿Cómo se implementan las **secciones críticas en un sistema distribuido**? ¿Cómo se asignan los recursos?
- En sistemas centralizados los problemas de sincronización como la exclusión mutua se resuelven con métodos como semáforos, monitores, etc., que **no sirven** en sistemas distribuidos porque se basan en el uso de **memoria compartida**.
- Dos procesos que usan un semáforo deben poder acceder a él. P.ej. si los procesos están en dos máquinas distintas este método no funciona.
- Determinar si un evento A ocurre antes que el evento B no es trivial.

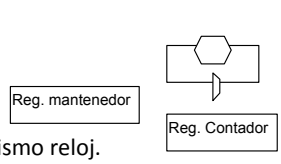
Sincronización mediante relojes

- La sincronización en sistemas distribuidos se basa en **algoritmos distribuidos**.
- Las características de un algoritmo distribuido son:
  - Tienen la **información relevante distribuida** entre múltiples máquinas.
  - Los procesos toman **decisiones** basándose únicamente en **información local**.
  - Deben evitarse los **puntos únicos de fallo** en el sistema.
  - **No existe un reloj común** o ningún tiempo global de precisión.
- En un **sistema centralizado** el **tiempo es no ambiguo**.
- En un **sistema distribuido** alcanzar un **acuerdo** sobre el tiempo **no es trivial**.



El reloj

- Los computadores tienen un **circuito** que permite **contabilizar el tiempo**: **el reloj** o **temporizador**.
- Es un cristal de cuarzo que oscila a una frecuencia bien definida cuando se le aplica una tensión.
- Cada cristal tiene **dos registros** asociados: un **contador** (*counter*) y un registro **mantenedor** (*holding register*).
- Cada **oscilación** del cristal **disminuye el contador** en una unidad.
- Cuando el contador llega **a 0** se genera una **interrupción** y el contador se **recarga** con el valor del registro mantenedor.
- Cada interrupción es conocida como un **tic de reloj** (*clock tick*).
- En un sistema con una sola CPU todos los procesos comparten el mismo reloj.



- Si hay **varias CPUs**, los **relojes software no estarán sincronizados**. Sus diferentes valores son conocidos como la **desviación del reloj** (*clock **skew***).
- ¿**Es posible sincronizar todos los relojes** para obtener un único y no ambiguo **estándar de tiempo**?

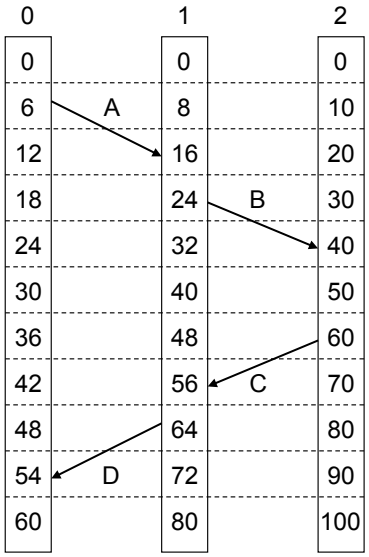
Relojes lógicos y relojes físicos

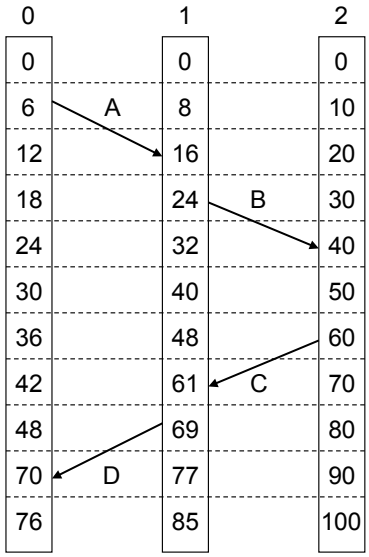
- Acordar el tiempo exacto no es lo más importante, sino **acordar el orden en que suceden los eventos**.
- En ciertos algoritmos basta con que las máquinas conozcan el **mismo tiempo**, es decir, que presenten una **consistencia interna** respecto a sus **relojes**.
- En estos tipos de algoritmos se suele hablar de **relojes lógicos**.
- Si además de acordar el mismo tiempo, los relojes no pueden desviarse del tiempo real más de una cierta cantidad se suele hablar de **relojes físicos**.

Sincronización de relojes lógicos. **Alg. de Lamport I**

- Se define la relación **sucede antes** como **a → b** : todos los procesos acuerdan que el evento *a sucede antes* que el evento *b*.
- **a → b** es cierto de forma directa en dos situaciones:
  - Si **a** y **b** son *eventos en el mismo proceso* y ***a** ocurre antes que **b***
  - Si **a** es el evento de un mensaje que es enviado por un proceso y **b** es el evento de un mensaje que es recibido por otro proceso
- **a → b** es transitiva.
- Si los *eventos **x** e **y** suceden en **distintos procesos*** que **no intercambian mensajes** entonces **x → y** no es cierta y tampoco lo es **y → x** .
- Se quiere asignar a los eventos un valor de tiempo **C** sobre el que todos los procesos estén de acuerdo.
- Los valores de tiempo cumplirán que si **a → b** entonces **C(a) < C(b)** .
- Además, se cumple que el tiempo de reloj **C** siempre se incrementa. Las **correcciones** se efectúan **sumando valores positivos**.

Sincronización de relojes lógicos. **Alg. de Lamport II**





Relojes físicos

- El algoritmo anterior ordena la sucesión de eventos pero no les asigna valores de tiempo real.
- En algunos sistemas (por ejemplo, en sistemas de tiempo real) **el valor real del reloj es importante**.
- Para estos sistemas se requieren varios **relojes físicos exteriores**.
- Esto presenta problemas de **sincronización de unos relojes respecto a otros** y **respecto al tiempo del mundo real**.
- El tiempo del mundo real se mide respecto al **Tiempo Universal Coordinado** (UTC).
- El UTC es difundido por el NIST (Instituto Nacional de Tiempo Estándar) que opera una estación de radio de onda corta WWV.
- WWV envía un pulso corto al comienzo de cada segundo UTC con una precisión de ±1 ms (debido a las condiciones atmosféricas ±10 ms).

Sincronización de relojes físicos

- Algoritmo de Berkeley
- Algoritmos de media
- Múltiples fuentes externas de hora
- Aplicaciones de relojes sincronizados

Sincronización de relojes físicos. **Alg. de Berkeley**

- El **servidor de tiempo** es activo; es un ***daemon*** *de tiempo*.
- Periódicamente **pregunta a cada máquina** su valor de tiempo.
- Según las respuestas **obtiene la media del valor de tiempo** e informa al resto de **máquinas que deben ajustar sus relojes**.
- Este sistema es adecuado cuando ninguna máquina tiene un receptor WWV.

Sincronización de relojes físicos. **Algoritmos de media**

- Al contrario que el anterior, éste es un algoritmo **descentralizado**.
- Se divide el tiempo en **intervalos de resincronización** y de duración fija.
- El *intervalo i-ésimo* es $[T_0 + iR, T_0 + (i+1)R]$ donde T_0 es un instante pasado y R es un parámetro de sistema.
- Al **comienzo de cada intervalo** cada máquina **difunde su tiempo actual**.
- Se arranca un temporizador para **recoger todas las difusiones de otros**.
- Cuando llegan, se ejecuta un **algoritmo que calcula el nuevo tiempo.**
- Variaciones de este algoritmo pueden ser:
  - **Media** de los valores.
  - **Descartar los valores más alejados** (más altos y más bajos) y obtener la media del resto (medida contra relojes defectuosos).
  - **Corrección** de cada mensaje con una **estimación del tiempo de propagación** del origen.

Sincronización de relojes físicos. **Múltiples fuentes externas de hora**

- **Sistemas que requieren mucha precisión** en la sincronización con el UTC pueden tener varios receptores WWV.
- El sistema operativo produce un rango o intervalo de tiempo en el que está el UTC que calcula.
- Distintas fuentes de UTC producirán distintos intervalos, por lo que deberán **alcanzar un acuerdo**.
- La base de este acuerdo está en que cada máquina con una fuente de UTC puede **difundir su intervalo periódicamente** (p.ej. al inicio de cada minuto UTC).
- Cada procesador recogerá los paquetes dentro de un tiempo determinado, según: la distancia entre ellos, el número de pasarelas, colisiones, etc.
- Diferentes sistemas operativos realizan este acuerdo de forma diferente.

Aplicaciones de sincronización de relojes

- **Entrega única de mensajes (*At-Most-Once Message Delivery*)**

- **Consistencia de caché basada en el reloj**

Entrega única de mensajes (*At-Most-Once Message Delivery*)

- Los mensajes llegan a través de una cierta conexión. Cada mensaje lleva un **identificador de conexión** y una **marca de tiempo** (*timestamp*).
- Para cada conexión el **servidor registra** en una tabla las **marcas más recientes**.
- Si un **mensaje** llega con una **marca de tiempo menor** que la almacenada para esa conexión, el mensaje es un **duplicado** y se rechaza.
- Para **descartar marcas de tiempo viejas**, cada servidor mantiene una variable:
  G = TiempoActual – MaxTiempoDeVida - MaxDesvíoReloj
- Cualquier marca de tiempo anterior a G puede ser borrada de forma segura.
- Los mensajes de conexiones desconocidas se aceptan si su marca es más reciente que G.
- Cada cierto tiempo ΔT, **el tiempo actual se escribe en el disco**.
- Después de una caída se recarga G del disco incrementado con el período de actualización ΔT.

Consistencia de caché basada en el reloj

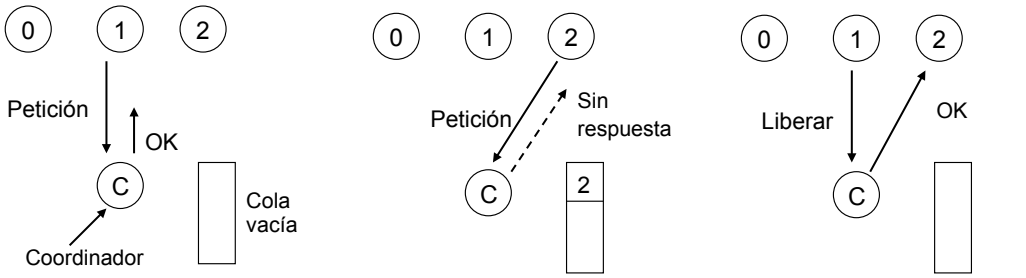
- **Problema:** Si un cliente tiene en caché un fichero para lectura y otro lo pide para escritura, el servidor debe pedir al primero que lo invalide.
- La utilización de **relojes sincronizados elimina esta sobrecarga**.
- Cuando un cliente pide un fichero, se le da un **alquiler** (*lease*) sobre el fichero que indica el tiempo durante el que la copia será válida.
- Cuando la copia está a punto de expirar el cliente puede pedir la **renovación**.
- Si expira, ya no podrá utilizarse y no hay que mandar un mensaje al servidor.
- Si expira y sigue en la caché, el cliente puede preguntar al servidor si la copia (**identificada por una marca de tiempo**) aún es la copia actual.
  - Si sigue siendo la actual, se genera un nuevo alquiler y la copia no necesita ser reenviada.
- Si **la copia está cacheada** en **varios clientes** y **uno pide escribir**, el servidor tiene que pedir al resto que terminen su alquiler prematuramente.
- Si un cliente cae, el servidor no puede saber si ha muerto o es demasiado lento.
- Utilizando temporizadores el servidor puede esperar y dejar que el alquiler expire.

Algoritmos de exclusión mutua en sistemas distribuidos

- Centralizado
- Distribuido
- Paso de testigo en anillo (*token ring*)

Algoritmo centralizado

- Se basa en simular el comportamiento en un monoprocesador.
- Se selecciona un proceso como el **coordinador**.
- Si un proceso P quiere entrar en la **sección crítica** envía **un *mensaje de petición*** al coordinador indicando qué sección crítica quiere obtener.
- Si actualmente **no hay otro proceso en la sección crítica**, el coordinador envía una ***respuesta de concesión***.
- Cuando P recibe la respuesta, entra en la sección crítica.



Algoritmo centralizado II

- Se garantiza la **exclusión mutua**.
- Es **justo**.
- No se produce **inanición**.
- Es un esquema fácil de implementar: sólo se requieren **tres mensajes** por utilización de la sección crítica (**petición**, **concesión**, **liberación**).
- Puede utilizarse para **gestión de recursos en general** (no sólo asignación de secciones críticas).
- Tener un coordinador centralizado también presenta problemas:
  - El coordinador constituye un **único punto de fallo**.
  - En un sistema grande, el coordinador puede presentar un **cuello de botella** en cuanto al rendimiento.

Algoritmo distribuido

- Intenta evitar tener un único punto de fallo.
- Este algoritmo [Ricart y Agrawala 1981] asume:
  - la existencia de una **ordenación total de todos los eventos** en el sistema.
  - que el **envío de mensajes es fiable** (se reciben asentimientos).

- El algoritmo de Lamport se toma como base y se usa para obtener **marcas de tiempo** para la exclusión mutua distribuida.

- Para entrar en una sección crítica, P crea un **mensaje de petición** que contiene: su **número** de proceso, la **sección crítica** S que le interesa y el **tiempo actual**.

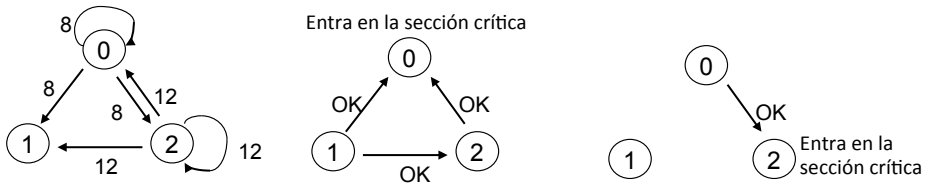
- P envía el mensaje **a todos los procesos** (él incluido).

- Cuando un proceso recibe un mensaje de petición:
  - Si **está en S**, **no contesta** y **encola la petición**.
  - Si el receptor **no está en S y no quiere entrar** en ella, envía a P un **mensaje OK**.
  - Si **no está en S pero quiere entrar**, **compara marcas de tiempo**. Gana la más baja y procede según sea el resultado.

- Al **enviar** la petición, P **espera** hasta recibir **todos los mensajes de concesión** para S.

- Cuando **salga de S,** **enviará mensajes OK** a todos los procesos de la cola y los borra.

Algoritmo distribuido II



- Se garantiza la **exclusión mutua** sin **interbloqueo ni inanición**.

- Se requiere **2(n-1) mensajes** por entrada en la sección crítica (más que antes).

- Ahora existen **n puntos de fallo**: ¿qué significa el silencio de un proceso?

- Puede **mejorarse** este algoritmo de forma que
  - al recibir una petición, el **receptor siempre envíe una respuesta** (sea de concesión o denegación), y
  - al recibir una denegación, deberá esperarse hasta recibir mensajes de concesión.

- Otras mejora:
  - utilizar un **temporizador** para tratar los **mensajes perdidos**.
  - **permitir entrada** en la sección crítica si recibe mensajes de **concesión de la mayoría** de procesos.

- Este algoritmo es más lento, complicado y menos robusto que el centralizado.

Algoritmo de paso de testigo en anillo (*Token Ring*)

- Se asume un **ordenamiento** software de los procesos de forma que se construye un **anillo lógico**: los procesos conocen a su sucesor en el anillo.
- En el comienzo, el **proceso 0** recibe un **testigo** que circulará por el anillo.
- **Para entrar** en la sección crítica un proceso deberá **poseer el testigo**.
- **No se permite** entrar en una **segunda sección crítica con el mismo testigo**.
- Se asegura la **exclusión mutua**: sólo uno tiene el testigo.
- **No** se da **inanición**.
  - Un proceso *esperará como máximo* que el resto entre una vez en su sección crítica.
- Es difícil detectar la **pérdida del testigo**.
- Es más fácil recuperarse de las **caídas de los procesos**. Esto se detectará cuando un proceso intente pasarle el testigo a su vecino y falle.

Comparación de los tres algoritmos

Algoritmo	Mensajes por entrada/salida	Retardo en la entrada (en tiempo de mensajes)	Problemas
Centralizado	3	2	Caída del coordinador
Distribuido	2(n-1)	2(n-1)	Caída de cualquier proceso
Anillo	1 a ∞	0 a n-1	Pérdida del testigo Caída de un proceso

Algoritmos de elección de coordinador

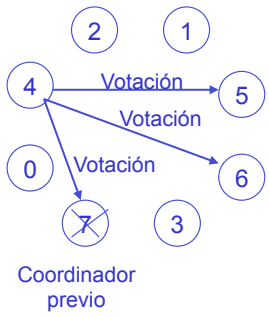
- En varios algoritmos hemos hablado de la existencia de coordinadores.
- … pero… ¿cómo se determina quién es el coordinador?
- Un proceso coordinador es aquél que se encarga de controlar los pasos a seguir en los procedimientos u operaciones típicas de un sistema distribuido.
- Veremos dos algoritmos básicos para la elección de procesos coordinadores:
  - Alg. Bully
  - Alg. en anillo

Algoritmo *bully*

- En el momento en que un proceso **detecta que el coordinador actual no responde**, inicia un proceso de **votación**.
- Un proceso P que realiza una votación procede de la siguiente forma:
  - P envía un **mensaje VOTACION** a todos los procesos con **número mayor** que él.
  - **Si no responde nadie**, P **gana** la votación y pasa a ser el nuevo coordinador.
  - Si un proceso de **mayor numeración responde**, éste **continua** el trabajo.
- Si un proceso **recibe un mensaje VOTACION de un proceso de menor numeración**, éste **responde** con un mensaje **OK**.
- El mensaje **OK** indica que el receptor **está vivo** y **seguirá** el proceso de **votación**.
- Al final todos los procesos **desisten menos** uno: el **nuevo coordinador**.
- Si un proceso que estaba **caído** arranca, **iniciará un proceso de votación**.

Algoritmos de elección. Algoritmo bully

- El proceso 7 era el coordinador y cae.
- El proceso 4 es el primero que se da cuenta e inicia el proceso de votación enviando mensajes de VOTACION a los procesos más altos.
- Al final, el proceso 6 será el nuevo coordinador.
- Si se arranca de nuevo el proceso 7, éste enviará un mensaje COORDINADOR al resto y ganará al resto.

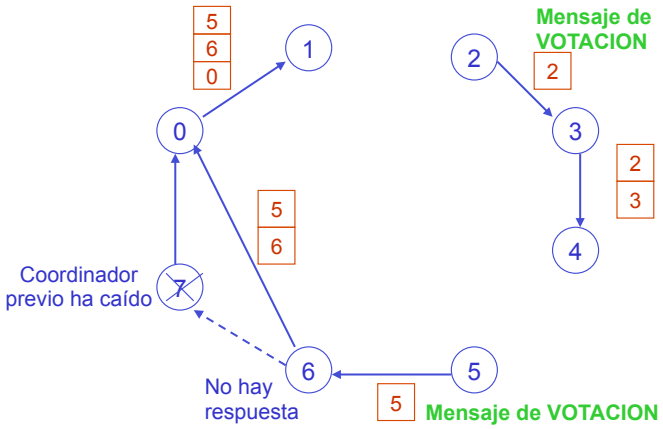


Algoritmo en anillo (*Ring*)

- Todos los **procesos** están **físicamente o lógicamente ordenados**.
- Cada proceso sabe cuál es su sucesor; forman un **anillo** pero sin pasar un testigo.
- Cuando un proceso nota que el **coordinador no funciona**, construye un **mensaje de VOTACION** con su número de proceso.
- Envía el mensaje VOTACION a su sucesor.
- Si el sucesor está caído el emisor lo salta y lo pasa al siguiente proceso en el anillo hasta encontrar un proceso no caído.
- En cada paso, **el emisor añade su número de proceso** en la lista del mensaje.
- **Cuando el mensaje vuelve al proceso origen**, reconoce su número en el mensaje.
- **El tipo de mensaje se cambia a COORDINADOR** y circula otra vez para informar a todos quién es el coordinador (el miembro de la lista con mayor numeración).
- Cuando ha circulado el mensaje COORDINADOR es retirado y se sigue de forma normal.

Algoritmo en anillo (*Ring*)

- Los procesos 2 y 5 descubren de forma simultánea que el coordinador previo, el proceso 7, ha caído.
- Los dos comienzan un proceso de votación enviando un mensaje VOTACION.



Transacciones atómicas. Introducción

- Las técnicas de sincronización vistas hasta ahora son de muy bajo nivel.
- Existen mecanismos de **mayor nivel de abstracción** que **ocultan todos los detalles técnicos**.
- Esto permite concentrarse en los algoritmos en sí que se quieren programar y en cómo los procesos trabajan juntos en paralelo.
- Estos mecanismos de alto nivel se llaman **transacciones atómicas** (también se conocen como **acciones atómicas** o, simplemente, **transacciones)**.

Transacciones atómicas. Introducción (cont.)

- El modelo original de las transacciones atómicas está **inspirado** en los contratos que tienen lugar **en el mundo de los negocios**.
- Un proceso P anuncia que quiere comenzar una transacción con otros procesos.
- Entonces comienzan una serie de negociaciones (creación y destrucción de objetos, etc.).
- P anunciará que quiere que los otros procesos se **comprometan** respecto a **las operaciones** que han realizado hasta entonces.
- Si todos están de **acuerdo**, los resultados se hacen **definitivos** (permanentes).
- Si alguno **no está de acuerdo**, **el trabajo se deshace** y se vuelve al mismo punto en el que se estaba antes de la transacción.
- Por lo tanto, las transacciones tienen la propiedad ***todo o nada***.

Transacciones atómicas. Ejemplo

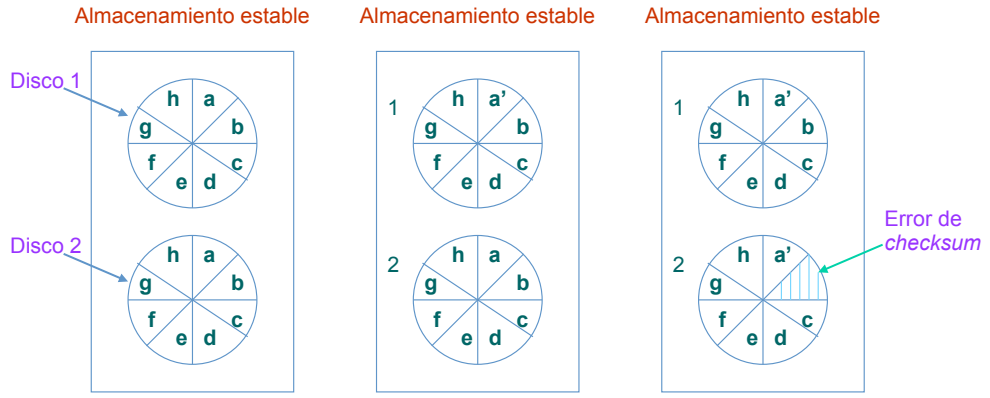
- Supongamos el sistema informático de un banco.
- A través de este sistema se pueden realizar operaciones de transferencias entre dos cuentas distintas de un cliente.
- Estas transferencias actualizan una base de datos que mantiene la información relevante de la situación de las cuentas.
- Un usuario se conecta al sistema del banco para sacar dinero de una cuenta A y traspasarlo a otra cuenta B.
- La operación se efectúa en dos pasos:

```
Reintegrar(cantidad, cuentaA)
Depositar(cantidad, cuentaB)
```

- Supongamos que la conexión se corta después del reintegro: el dinero se esfuma.
- La solución sería **agrupar** las dos operaciones en **una** transacción atómica.
  De esta forma, o **se completan las dos o ninguna** de ellas.
- Las transacciones **deberán poder deshacerse** (***roll back***), es decir, volver al estado inicial en caso de ser abortadas antes de completarse.

El modelo de Transacción. **Almacenamiento estable**

- Existen tres categorías de almacenamiento: **RAM**, **disco** y **almacenamiento estable**.
- El almacenamiento estable está diseñado para aguantar cualquier tipo de fallos.
- Puede implementarse con **dos discos**.
- Cuando suceden actualizaciones, los bloques de uno de ellos son copiados en el otro disco.



El modelo de Transacción. **Primitivas de transacción**

- La programación utilizando transacciones requiere la utilización de ciertas primitivas.
- Estas son ofrecidas por el **sistema operativo** o por el **entorno de ejecución del lenguaje** utilizado.

```
BEGIN_TRANSACTION
END_TRANSACTION
ABORT_TRANSACTION
READ
WRITE
```

- Las primitivas dependen del tipo de objetos o información utilizada en la transacción.
- Las **operaciones entre las dos primitivas** `BEGIN_TRANSACTION` y `END_TRANSACTION` son el **cuerpo de la transacción**.
- Veamos un ejemplo de reserva de billetes utilizando transacciones:

```
BEGIN_TRANSACTION
    reservar MAD-JFK;
    reservar JFK-SLO;
    reservar SLO-ABQ;
END_TRANSACTION
```

```
BEGIN_TRANSACTION
    reservar MAD-JFK;
    reservar JFK-SLO;
    SLO-ABQ lleno → ABORT_TRANSACTION;
END_TRANSACTION
```

El modelo de Transacción. **Propiedades de las transacciones**

- Una transacción debe presentar las siguientes propiedades (**ACID**).

- **ATOMICIDAD**: la transacción sucede de forma **indivisible**.
  - O sucede completamente o no suceden ninguna de sus operaciones.
  - Mientras está en curso ningún proceso puede ver sus estados intermedios.

- **CONSISTENCIA**: **se mantienen todas las invariantes** del sistema.
  - En el entorno bancario, una invariante es la conservación del dinero.

- **AISLAMIENTO**: las transacciones concurrentes **no interfieren** entre sí.

```
BEGIN_TRANSACTION          BEGIN_TRANSACTION          BEGIN_TRANSACTION
    x = 0;  x = x + 1;         x = 0;  x = x + 2;         x = 0;  x = x + 3;
END_TRANSACTION            END_TRANSACTION            END_TRANSACTION
```

Plan 1	x = 0;	x = x + 1;	x = 0;	x = x + 2;	x = 0;	x = x + 3;	Bien
Plan 2	x = 0;	x = 0;	x = x + 1;	x = x + 2;	x = 0;	x = x + 3;	Bien
Plan 3	x = 0;	x = 0;	x = x + 1;	x = 0;	x = x + 2;	x = x + 3;	Mal

- **DURACION**: una vez la transacción **se compromete** (*commit*) los **cambios** ocasionados por una transacción son **permanentes**.

Implementación de las transacciones

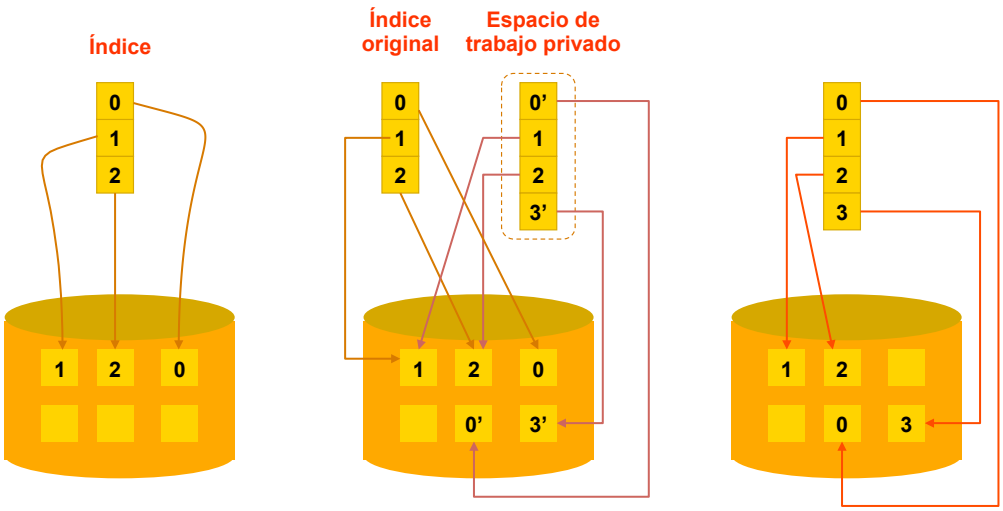
- Espacio privado de trabajo
- Lista de intenciones (*Writeahead Log*)
- Protocolo de compromiso en dos fases

Espacio de trabajo privado

- El **espacio de trabajo de un proceso** comprende el conjunto de todos los **ficheros y objetos que utiliza** durante su ejecución.
- Cuando un proceso comienza una transacción, se le da un espacio de trabajo privado.
- Hasta que la transacción se compromete o se aborta, todos los **accesos se hacen sobre los objetos del espacio privado**.
- El problema es el **coste prohibitivo de copiar todo** a un espacio privado.
- Existen varias formas de solucionar este problema.
- Si un objeto se abre sólo **para lectura**, **no se hace una copia** al espacio privado.
- De esta forma, el espacio privado sólo contiene una referencia al espacio padre.
- Si un objeto se abre **para escritura**, se localiza a través de la referencia pero antes de ser modificado **se copia al espacio privado**.

Espacio de trabajo privado

- Una mejora consiste en no copiar todo el objeto o fichero. Sólo **se copiará el índice del fichero** al espacio privado.



Lista de intenciones

- Los ficheros son modificados directamente (no hay copias locales).
- Las modificaciones se registran en una **lista de intenciones** en **almacenamiento estable**.
- Para **cada modificación** se escribe una **entrada o registro** en la lista que contiene:
  - La transacción que la realiza.
  - Fichero y bloque que son modificados.
  - Antiguo y nuevo valor.
- El cambio se hará efectivo después de que la lista de intenciones haya sido actualizada de forma exitosa.
- Durante una transacción, **antes de** ejecutar una operación que implique una **modificación**, **se escribe un registro con los valores antiguo y nuevo.**
- Si la transacción **se compromete**, se escribe un **registro de compromiso** en la lista.
- Si la transacción **se aborta**, la lista se utiliza para **volver al estado original**.
  - Empezando por el final, cada registro es leído y se deshace el cambio que éste describe.
- Después de una caída, utilizando la lista de intenciones es posible terminar una transacción o deshacerla.

Lista de intenciones (cont.)

- Tenemos una transacción simple que utiliza dos variables (u objetos) compartidas: ***x*** e ***y***.

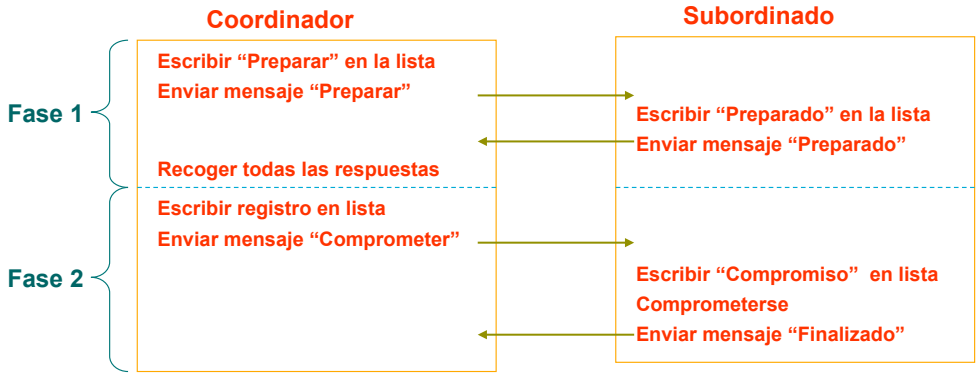
```
x = 0;
y = 0;

BEGIN_TRANSACTION
      x = x + 1;
      y = y + 2;
      x = y * y;
END_TRANSACTION
```

Lista de intenciones	Lista de intenciones	Lista de intenciones
x = 0 / 1	x = 0 / 1	x = 0 / 1
	y = 0 / 2	y = 0 / 2
		x = 1 / 4

Protocolo de compromiso en dos fases

- Cuando **finalizan todas las operaciones de una transacción**, ésta debe **comprometerse** (*commit*), es decir, indicar que ha finalizado de forma exitosa.
- En un entorno distribuido el compromiso de una transacción puede requerir la **cooperación de múltiples procesos** en máquinas diferentes.
- El **compromiso debe hacerse** de forma atómica (de **forma instantánea e indivisible**).
- Se basa en la existencia de un **proceso coordinador**.
- Normalmente, el coordinador es el proceso que ejecuta la transacción.



Protocolo de compromiso en dos fases (cont.)

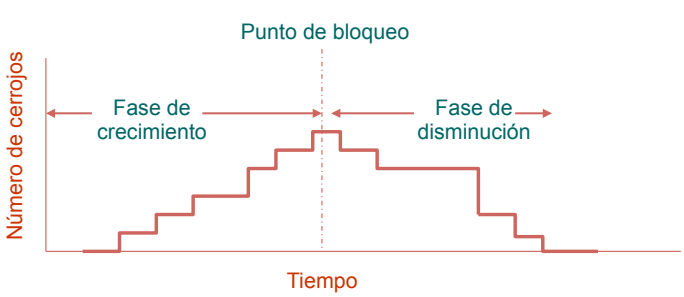
- Cuando el coordinador escribe un registro de compromiso (en la fase 2), es cuando realmente se compromete la transacción.
- En ese momento la transacción se realizará sin importar lo que suceda.
- Este protocolo es muy flexible ante las caídas de las máquinas.
- Si el **coordinador cae después de escribir el registro inicial** ("Preparar"), cuando se recupere continuará donde lo dejó.
- Si **cae después de escribir el resultado recibido de los subordinados**, cuando se recupere reinformará a los subordinados del resultado.
- Si un **subordinado cae antes de haber contestado al primer mensaje**, el coordinador continuará mandándole mensajes por un tiempo.
- Si **el subordinado cae después de contestar el primer mensaje**, cuando se recupere leerá la lista y verá cómo proceder.

Control de la concurrencia. Cerrojos (*Locking*)

- Cuando un proceso va a **leer o escribir un objeto** como **parte de una transacción** debe **bloquear** (echarle el cerrojo) el fichero.
- El bloqueo puede hacerse a través de un único **gestor de bloqueo centralizado** o con un **gestor local** de bloqueos en cada máquina para gestionar ficheros locales.
- El gestor mantiene una lista de objetos bloqueados y rechaza peticiones de bloquear estos objetos.
- Los cerrojos sobre estos objetos normalmente son **adquiridos y liberados por el sistema de transacciones** y no por los programadores.
- Este mecanismo puede ser mejorado distinguiendo **cerrojos de lectura** de **cerrojos de escritura**.
- Los cerrojos de lectura son compartidos; los de escritura son exclusivos.
- La unidad de bloqueo suele ser el objeto entero, aunque puede ser más amplio o menor. El tamaño del objeto a bloquear se llama **granularidad del bloqueo**.
- Con una granularidad pequeña se tienen **bloqueos más precisos** y **mayor paralelismo**, pero se requieren **más cerrojos**, es **más compleja** y **mayor riesgo de interbloqueos**.

Control de la concurrencia. Cerrojos (*Locking*) (cont.)

- La mayoría de transacciones implementadas con cerrojos (o bloqueo de objetos) usan el **bloqueo de dos fases**.
- En la fase de crecimiento, el proceso primero adquiere todos los cerrojos que necesita.
- En la fase de disminución, el proceso los libera.
- Existe una variante de bloqueo estricto en dos fases donde la fase de disminución no se comienza hasta que la transacción no ha terminado.
- Así la transacción siempre leerá el valor escrito por la transacción que se ha comprometido.



Control de la concurrencia. Control de concurrencia optimista

- Este protocolo se basa en que los **procesos se ejecutan sin preocuparse de lo que está haciendo el resto**.
- Si existe algún problema, éste se tratará cuando aparezca.
- Funciona bastante bien porque en la práctica los conflictos se producen en raras ocasiones.
- El protocolo anota los objetos que han sido leídos o escritos.
- **Cuando la transacción A va a comprometerse, comprueba las otras** transacciones para ver si alguno de sus ficheros ha sido modificado desde que A comenzó.
- Si es así, A es abortada. Si no A se compromete.
- Este protocolo se suele implementar con la técnica de los **espacios de trabajo privados**.
- Sus ventajas son que **no se producen interbloqueos** y permite **máximo paralelismo**.
- El problema es que si una transacción falla, deberá ejecutarse de nuevo. Además en casos de alta carga, este protocolo baja su rendimiento de forma notable.

Control de concurrencia. Marcas de tiempo (*timestamps*)

- A cada **transacción** se le asigna una **marca de tiempo**.
- Cada objeto tiene una **marca de tiempo** para **lectura** y otra para **escritura**.
- El procesamiento de las transacciones se considerará correcto siempre que cuando desde una transacción A se acceda a un fichero u objeto F, se encuentre que **la marca de tiempo de F es menor que la de A**.
- Si la marca de F es mayor que la de A significará que una transacción que empezó más tarde que A ha modificado el objeto y se ha comprometido. A será abortada.
- La transacción con una marca menor siempre deberá ejecutarse primero.