Sistemas Distribuidos. Temario.

1. Introducción
2. Comunicación
3. Procesos
4. Nombrado y localización
5. Sincronización
6. Consistencia y replicación
7. Tolerancia a fallos

Tema 5.- Sincronización

1. Sincronización de relojes
2. Relojes lógicos
3. Estado global
4. Algoritmos de elección
5. Exclusión mutua
6. Transacciones distribuidas

Bibliografía: Capítulo 5 de Tanenbaum

1.- Sincronización de relojes

1. Introducción
2. Algoritmo de Cristian
3. Algoritmo de Berkeley
4. Otros algoritmos

1.1.- Introducción

- Sincronización: mecanismos que faciliten trabajo coordinado.
- Sincronización en sistemas distribuidos más compleja que en sistemas centralizados: p.ej:
  - Acceso a recurso compartido.
  - Ordenación de eventos
- Una forma de sincronizar habitual en sistemas centralizados se deriva de emplear un reloj único.
- En sistemas distribuidos, cada ordenador tiene un reloj.
- Pese a que los relojes son cada vez mejores, siempre hay divergencias en las velocidades de cada uno de ellos.
- Para tener un único reloj en sistemas distribuidos: sincronizar los relojes de los diferentes nodos para tener un reloj común.
  - P. ej: Utilidad *make* en sistemas operativos en red.

1.1.- Introducción

Introducción a los algoritmos de sincronización de relojes.

- Cada ordenador tiene un temporizador que genera una interrupción *H* veces por segundo.
- Un contador lleva la cuenta del número de *ticks* (interrupciones) ocurridas desde cierta base temporal prefijada. (ej.: 1 de Enero de 1970 a las 00.00.00)
- Si los relojes fueran perfectos, en todo instante t, todos los ordenadores tendrían relojes con el mismo valor del contador.
- Sin embargo, los chips de reloj actuales, tienen un error relativo de 10^{-5} :
  - Si un reloj genera 60 ticks por segundo, genera 216.000 ticks por hora: los relojes habituales correctos, podrán tener un valor entre 215.998 y 216.002
- Los algoritmos de sincronización de relojes tratarán de ajustar estos desfases..

1.2.- Algoritmo de Cristian

- Suponemos que existe un ordenador con un reloj muy preciso, posiblemente sincronizado con otros más precisos todavía.
- Los demás se sincronizarán con él.
- Llamaremos servidor al ordenador que tiene el reloj preciso y cliente al que se desea sincronizar.
- Problemas:
  - Los relojes no pueden retroceder.
  - El tránsito de mensajes por la red consume tiempo.

1.2.- Algoritmo de Cristian

Algoritmo:

- El cliente pide el valor del reloj al servidor en el instante T_0 (según reloj local del cliente).
- El servidor recibe la petición y contesta con el valor de su reloj: Cs
- La respuesta llega al servidor en T_1 (según reloj local del cliente).
- El cliente fija su reloj al valor de a $C = C_s + (T_1 – T_0)/2$.
- Si se conoce el tiempo de procesamiento del servidor (T_p), se fija el valor del reloj a $C = C_s + (T_1 – T_0 – T_p)/2$.
  - Si $C > C_c$, se fija el valor del reloj del cliente a C.
  - Si $C < C_c$, se almacena $LAG = C_c – C$ y se descartarán LAG ticks del reloj del cliente (se evita que retroceda el reloj: se hace que el reloj vaya más lento).

1.3.- Algoritmo de Berkeley

- No se supone que exista un ordenador con reloj preciso.
- Uno de los ordenadores actúa de servidor y los demás actúan de clientes.
- El servidor pregunta el valor de los relojes a los clientes de forma periódica.
- Dadas las respuestas, el servidor calcula la media y contesta a todos los clientes para que actualicen sus relojes.

1.4.- Otros algoritmos

- Los dos vistos son centralizados.
- Existen variaciones plenamente distribuidas: Algoritmos de media:
  - Cada nodo envía a todos los demás su valor
  - Cada nodo calcula la media de los valores recibidos y se ajusta.
- Algoritmo NTP (Network Time Protocol): Logra precisión a escala mundial con error entre 1 y 50 microsegundos.

Tema 5.- Sincronización

1. Sincronización de relojes
2. Relojes lógicos
3. Estado global
4. Algoritmos de elección
5. Exclusión mutua
6. Transacciones distribuidas

Bibliografía: Capítulo 5 de Tanenbaum

2.- Relojes lógicos

1. Introducción
2. Marcas temporales de Lamport
3. Vectores de marcas temporales

2.1.- Introducción

- Para muchas aplicaciones es suficiente con que exista acuerdo sobre el valor de cierto reloj (más que en su correspondencia a los relojes reales).
- Se habla entonces de relojes lógicos.
- **Idea clave**: si dos nodos no interactúan (si no intercambian mensajes) no es necesario que tengan el mismo reloj.
- **Idea clave**: el general es importante el orden global en que ocurren los eventos y no el tiempo real en que suceden.

2.2.- Marcas temporales de Lamport

- Para sincronizar los relojes lógicos, se define la relación: **ocurre-antes**
- La expresión $a \rightarrow b$, se lee "*a* ocurre antes que *b*" y significa que todos los nodos están de acuerdo en que primero ven el evento *a* y después ven el evento *b*.
- Se puede observar la relación ocurre-antes de forma directa en dos situaciones:
  - Si *a* y *b* son eventos del mismo nodo y *a* ocurre antes que *b*, $a \rightarrow b$ es cierta.
  - Si *a* es el evento de envío de un mensaje por parte de un nodo, y *b* el evento de recepción de ese mensaje por parte de otro nodo, $a \rightarrow b$ es cierta.
- La relación ocurre-antes es transitiva, luego si $a \rightarrow b$ y $b \rightarrow c$, entonces $a \rightarrow c$.
- Si dos eventos *x*, *y* ocurren en nodos que no intercambian mensajes, $x \rightarrow y$ no es cierta y $y \rightarrow x$ tampoco. Entonces decimos que los eventos *x* e *y* son concurrentes.
- Dos eventos son concurrentes, si no se puede decir cuál de ellos ocurre antes (además no es relevante).
- Es necesaria una forma de medir esta relación: **reloj lógico de Lamport**.

2.2.- Marcas temporales de Lamport

- Los relojes lógicos deben marcar el instante en que ocurren los eventos, de forma que asocien un valor a cada evento.
- Llamamos C(a) al valor del reloj lógico del evento *a*.
- Los relojes lógicos deben satisfacer que:

  Si *a* → *b* , entonces C(a) < C(b).
- Si reescribimos las condiciones de la relación ocurre-antes:
  - Para dos eventos de un mismo nodo *a* y *b*, si *a* ocurre antes que *b*, entonces C(a) < C(b)
  - Para eventos correspondientes de envío y recepción *a* y *b*, C(a) < C(b).
- Adicionalmente el reloj nunca debe decrecer.

2.2.- Marcas temporales de Lamport

Algoritmo de Lamport

1. Todos los nodos empiezan con reloj a 0.
2. Cada ejecución de un evento en un nodo *p*, incrementa el valor de su contador *Cp* en 1.
3. El envío de mensajes incluye el valor del contador.
4. Cuando un nodo *p*, recibe un mensaje que incluye el tiempo *Cm*, calcula el valor que asocia al evento de recepción *r* como:

   Cr = max (Cp, Cm) +1

- **Con este algoritmo se pueden ordenar totalmente los eventos de un sistema distribuido.**
- **Para ello, sólo falta añadir el número de nodo a los tiempos para que no existan dos valores iguales.**

2.2.- Marcas temporales de Lamport

Aplicación: Multienvíos totalmente ordenados.

1. Un nodo que envía una difusión.
2. Todos los que reciben la difusión, envían reconocimiento a todos los demás.
3. Si se envían varias difusiones por diferentes nodos, todos los nodos estarán de acuerdo en el orden de las difusiones.

2.3.- Vectores de marcas temporales

- El reloj de Lamport garantiza que si ***a*** → ***b*** **, entonces C(a) < C(b).**
- Sin embargo, Si C(a) < C(b), no se desprende nada. Es decir, la simple observación de los relojes no implica nada acerca de su ordenamiento.
- Con relojes vectoriales se logra que si VT(a) < VT(b), entonces *a* → *b***.** Además, Si *a* → *b,* entonces VT(a) < VT(b).
- Los relojes vectoriales se construyen de forma que cada nodo *i* mantiene un vector de marcas temporales V_i. De forma que:
  - $V_i[i]$ es el número de eventos que han ocurrido en i.
  - *Si* $V_i[j]=k$, entonces P_i sabe que han ocurrido k eventos en P_j.
- Para actualizar el vector, cada nodo envía su vector en cada mensaje.
- Ante cada recepción, un nodo actualiza todas sus componentes, para reflejar el mayor valor de cada una de ellas, entre las locales y las que recibe en el mensaje.

2.4.- Problemas de los relojes lógicos

- No capturan canales encubiertos
- Capturan causalidad potencial, aunque no exista causalidad : ineficiencias.

Tema 5.- Sincronización

1. Sincronización de relojes
2. Relojes lógicos
3. Estado global
4. Algoritmos de elección
5. Exclusión mutua
6. Transacciones distribuidas

Bibliografía: Capítulo 5 de Tanenbaum

3.- Estado global

1. Introducción
2. Algoritmo de Chandy-Lamport

3.1- Introducción

- Estado global: estado local de cada proceso + estado de cada canal.
- Estado de cada proceso: sólo las variables que nos interesen.
- Estado de cada canal: mensajes enviados y todavía no entregados.
- Usos del estado global:
  - Detección de terminación.
  - Depuración distribuida.

3.2.- Algoritmo de Chandy-Lamport

Conceptos previos

- Una **instantánea distribuida** refleja el estado que **pudo haberse alcanzado** en el sistema.
- Las instantáneas reflejan sólo estados consistentes:
  - Si P ha recibido un mensaje de Q en la instantánea, Q envió antes un mensaje a P.
- La noción de estado global se refleja por el concepto de **corte.**

3.2.- Algoritmo de Chandy-Lamport

- Cualquier proceso inicia el algoritmo.
- Sea P el proceso iniciador.
- P guarda su estado local, luego envía un mensaje de **marca** a todos los demás procesos.
- Cuando un proceso recibe el mensaje de marca:
  - Si no ha guardado su estado local: guarda su estado local y envía **marca** a todos los demás procesos. A partir de este momento almacenará lo que llegue por los canales entrantes.
  - Si ha guardado su estado local, guarda los mensajes recibidos por el canal (desde que el proceso guardó su estado local, hasta que recibió marca por este canal).
- Un proceso termina, cuando ha recibido marca por todos los canales.
- Cuando termina, envía su estado y el de sus canales al iniciador.

Tema 5.- Sincronización

1. Sincronización de relojes
2. Relojes lógicos
3. Estado global
4. Algoritmos de elección
5. Exclusión mutua
6. Transacciones distribuidas

Bibliografía: Capítulo 5 de Tanenbaum

4.- Algoritmos de elección

1. Introducción
2. Algoritmo Bully
3. Algoritmo para anillos

4.1- Introducción

- Muchas veces es necesario que un nodo coordine alguna acción en el sistema.
- Si todos los procesos son idénticos, **no hay forma de elegir a uno de ellos**. Luego asumiremos que tienen identificadores únicos.
- Supondremos que todos conocen la identidad de los demás miembros del grupo.

4.2.- Algoritmo Bully

1. Cuando un proceso quiere comenzar una elección (p.ej, porque el líder actual no contesta), envía ELECCION a todos los nodos con número mayor al suyo.
2. Si no responde nadie, él es el líder.
3. Si alguien responde, el nodo no hace nada más
4. Cuando un proceso recibe ELECCION, envía OK a quien se lo envío (para avisarle que participa y le ganará). Este nodo, procederá con el paso 1.
5. Será líder quien no obtenga respuesta de los nodos más altos, habiendo recibido ELECCION.

4.3.- Algoritmo para anillos

1. Suponiendo que tenemos a los nodos ordenados en una anillo lógico. (p.ej, en orden creciente de identificadores)
2. Cuando un nodo quiere elegir nuevo líder, envía ELECTION por el anillo (al siguiente del anillo que esté vivo: que responda con ACK).
3. Cada nodo que ve el mensaje incluye su propio ID en el mensaje.
4. Cuando el mensaje llega de vuelta al iniciador, contiene los IDs de todos los nodos participantes. En este momento, envía un mensaje LIDER por el anillo, avisando del nuevo líder (el nodo mayor).
5. Nótese que puede haber más de un mensaje simultáneo: pero los dos intentos de elección darán el mismo resultado.

Tema 5.- Sincronización

1. Sincronización de relojes
2. Relojes lógicos
3. Estado global
4. Algoritmos de elección
5. Exclusión mutua
6. Transacciones distribuidas

Bibliografía: Capítulo 5 de Tanenbaum

5.- Exclusión mutua

1. Un algoritmo centralizado
2. Un algoritmo distribuido
3. Un algoritmo para anillos

5.1.- Un algoritmo centralizado

1. Se elige un nodo como coordinador (líder).
2. Cuando un nodo quiere entrar en la sección crítica, envía un mensaje al líder, pidiendo permiso.
3. Si ningún otro proceso está en la sección crítica, el líder responde CONCECER, respondiendo DENEGAR en caso contrario.
4. Cuando un proceso sale de su sección crítica, avisa al líder. Si el líder sabe de otro proceso que intentó entrar después, le envía CONCEDER.

5.2.- Un algoritmo distribuido

- Suponemos que todos los eventos están ordenados (p.ej, usando relojes de Lamport junto al número de nodo).

1. Cuando un proceso quiere entrar en la sección crítica, envía un mensaje TRY a todos.
2. Cuando un proceso recibe un mensaje TRY:
    1. Si no está intentando entrar en su sección crítica, envía OK.
    2. Si está en su sección crítica, no contesta y encola en mensaje.
    3. Si no está en su sección crítica, pero quiere entrar, compara el número de evento del mensaje entrante con el que él mismo envió al resto. Vence el número más bajo:
        1. Si el mensaje entrante es más bajo, responde OK.
        2. Si el más alto, no responde y encola el mensaje
3. Un proceso entra en la sección crítica, cuando recibe OK de todos.
4. Cuando sale, envía OK a todos los mensajes que retuvo en su cola.

5.3.- Un algoritmo para anillos

1. Un token circula por un anillo lógico.
2. Un proceso que quiere entrar en su sección crítica, espera a tener el token.
3. Sólo puede entrar en la sección crítica un proceso: el que tiene el token.
4. Si un nodo cae, hay que reconstruir el token.

Tema 5.- Sincronización

1. Sincronización de relojes
2. Relojes lógicos
3. Estado global
4. Algoritmos de elección
5. Exclusión mutua
6. Transacciones distribuidas

Bibliografía: Capítulo 5 de Tanenbaum

6.- Transacciones distribuidas

1. Introducción
2. Transacciones planas
3. Transacciones anidadas
4. Transacciones distribuidas
5. Implementación de transacciones
6. Control de concurrencia

6.1.- Introducción

- Similar a exclusión mutua en que permiten el acceso a datos en exclusión mutua.
- En lugar de un solo dato, a múltiples datos de forma atómica: Se modifican todos los datos o ninguno.
- Un proceso podrá modificar varios datos mediante una transacción y finalmente decidir si hace commit o rollback.
  - Commit: todos los datos se efectuarán de forma definitiva y serán visibles al resto de procesos.
  - Rollback: los datos volverán a su valor previo al comienzo de la transacción.

6.1.- Introducción

Modelo para transacciones

- Se utiliza BEGIN_TRANSACTION y END_TRANSACTION para delimitar el ámbito de la transacción.
  - END_TRANSACTION significa COMMIT.
  - Para abortar una transacción se emplea ABORT_TRANSACTION.
- Como operaciones, se permite READ(x) y WRITE(x), es decir, se distingue entre operaciones de lectura y escritura.
- Las transacciones cumplen 4 propiedades (**ACID**):
  - Atomicidad (Atomicity) : las transacciones se ejecutan de forma indivisible.
  - Consistencia (Consistency): La transacción no viola invariantes del sistema. (operaciones individuales si)
  - Aislamiento (Isolation): Transacciones concurrentes no interfieren entre ellas. También llamada serializabilidad.
  - Durabilidad (Durability): Una vez finaliza una transacción, sus cambios permanecen.

6.- Transacciones distribuidas

1. Introducción
2. Transacciones planas
3. Transacciones anidadas
4. Transacciones distribuidas
5. Implementación de transacciones
6. Control de concurrencia

6.2.- Transacciones planas

- Transacciones efectuadas por un mismo proceso.
- Pueden ser de duración arbitrariamente larga.
  - Ejemplo: actualizar los links que apuntan a cierta página web.
- Dos deficiencias de las transacciones planas:
  - No permiten el acceso a datos físicamente distribuidos.
  - No permiten el “commit” o “rollback” de modificaciones parciales sobre los datos.

6.- Transacciones distribuidas

1. Introducción
2. Transacciones planas
3. Transacciones anidadas
4. Transacciones distribuidas
5. Implementación de transacciones
6. Control de concurrencia

6.3.- Transacciones anidadas

Tipo de transacción para permitir actualizaciones parciales sobre los datos.

- Una transacción anidada está formada por una transacción padre y varias sub-transacciones.
- Cada sub-transacción puede a su vez tener sus propias transacciones anidadas.
- Cuando una sub-transacción hace “commit”, los cambios serán visibles dentro de su transacción padre.
- Cuando la transacción padre haga “commit”, se harán visibles los cambios que hayan sido efectuados en ella y en sus sub-transacciones que hayan efectuado “commit”.
- Si una sub-transacción hace “commit” y la transacción padre hace “rollback”, hay que deshacer también los cambios efectuados por la sub-transacción, pese a que hizo “commit”.

6.- Transacciones distribuidas

1. Introducción
2. Transacciones planas
3. Transacciones anidadas
4. Transacciones distribuidas
5. Implementación de transacciones
6. Control de concurrencia

6.4.- Transacciones distribuidas

Tipo de transacción para permitir que los datos estén físicamente distribuidos.

- Una transacción que afecte a datos ubicados en distintos ordenadores, iniciará una sub-transacción en cada uno de estos ordenadores.
- Dificultad: utilizar un protocolo distribuido que permita coordinar el “commit” o “rollback” de todas las sub-transacciones.
- Diferencia con transacciones anidadas:
  - Las anidadas están formadas por sub-transacciones a modo de jerarquía para permitir cambios parciales.
  - Las distribuidas están formadas por sub-transacciones para poder alcanzar datos físicamente distribuidas, pero a nivel lógico son transacciones planas.

6.- Transacciones distribuidas

1. Introducción
2. Transacciones planas
3. Transacciones anidadas
4. Transacciones distribuidas
5. Implementación de transacciones
6. Control de concurrencia

6.5.- Implementación

- Dos alternativas para la implementación de transacciones: Espacio de trabajo privado y Registro previo de actualizaciones (write-ahead log).
- Espacio de trabajo privado: a cada transacción se le proporciona una copia de todos los datos al empezar la transacción.
  - La transacción opera sobre su copia privada.
  - Si hace “commit”, se copia el valor de sus datos en los datos globales.
  - Si hace “rollback” se descarta la copia privada.
  - Nótese que la copia es a nivel lógico, pues son posibles muchas optimizaciones.
- Registro previo de actualizaciones: Antes de efectuar una modificación, se apunta en el registro, el valor previo del dato y el valor que tomará el dato. Después se realiza la modificación.
  - Si se hace “commit”, se anota “commit” en el registro y no es necesario nada más.
  - Si se hace “rollback”, se recorre el log hacia detrás, para restaurar el valor original de todo dato modificado,

6.- Transacciones distribuidas

1. Introducción
2. Transacciones planas
3. Transacciones anidadas
4. Transacciones distribuidas
5. Implementación de transacciones
6. Control de concurrencia

6.6.- Control de concurrencia

- Atomicidad y durabilidad se logran con alguna de las alternativas de implementación vistas.
- Aislamiento y consistencia requieren de control de concurrencia.
- Los mecanismos de control de concurrencia garantizan que las operaciones que realizan concurrentemente distintas transacciones procedan en exclusión mutua.
- Interesa centrarse en las operaciones conflictivas: las escrituras.
- Las escritura están en conflicto con las lecturas y con otras escrituras.
  - Conflicto read-write: Conflicto de una escritura con múltiples lecturas.
  - Conflicto write-write: Más de una escritura.

6.6.- Control de concurrencia

Serializabilidad

Dos transacciones A y B son serializables, si el resultado de su ejecución sobre un sistema transaccional, es el mismo que habría ocurrido si se ejecuta primero A y luego B, o primero B y luego A.

- Para lograr serializabilidad hay varios métodos:
  - Bloqueo en dos fases (Two-phase locking)
  - Métodos basados en marcas de tiempo.

6.6.- Control de concurrencia

Bloqueo en dos fases

- Las transacciones proceden de forma ordenada:
  - Primero piden los bloqueos
  - Segundo: trabajan sobre los datos.
  - Tercero: liberan los bloqueos.
- Si se pide el bloqueo de un dato después de haber liberado algún bloqueo, se considera un error de programación y se aborta la transacción.
- Ventajas:
  - Los datos que se leen siempre son válidos. Se hizo commit en la transacción que los modificó.
  - Los bloqueos pueden hacerse transparentes al programador: el sistema pide el bloqueo ante la primera modificación de un dato y se liberan todos los bloqueos al finalizar la transacción.
- Deventajas:
  - Interbloqueos.