Transacciones Distribuidas

Ricardo Jiménez Peris, Marta Patiño Martínez

Lsd Distributed Systems Laboratory
Universidad Politécnica de Madrid (UPM)
http://lsd.ls.fi.upm.es/lsd/lsd.htm

Bibliografía

- Libros generales sobre transacciones:
  - Weikum, Vossen. Transactional Information Systems. Morgan-Kaufmann. 2002.
  - Lewis, Bernstein, Kifer. Databases and Transaction Processing. Addison-Wesley. 2002.
  - Garcia-Molina, Ullman, Widom. Database Systems: The Complete Book. Prentice Hall, 2002.
  - Gray, Reuter. Transaction Processing: Concepts and Techniques. Morgan-Kaufmann, 1993.
  - Bernstein, Newcomer. Principles of Transaction Processing. Morgan-Kaufmann, 1997.
  - Bernstein, Hadzilacos, Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
  - Lynch, Merrit, Weihl, Fekete. Atomic Transactions. Morgan-Kaufmann Publishers, 1994.

Motivación

- Transferencia entre dos cuentas bancarias.

  Read A ← Fallo inocuo
  Write A-100
  Read B ← Fallo perjudicial
  Write B+100

- ¿Qué ocurre si hay un fallo durante la transacción?
  - Los fallos pueden dejar la BD inconsistente.

Motivación

- ¿Qué ocurre si dos transferencias con alguna cuenta en común se ejecutan concurrentemente?

```
1  a=Read(A)              2  a=Read(A)
   if a >= 100 then          if a >= 100 then
     Write(A,a-100)            Write(A,a-100)
3    b=Read(B)            4    b=Read(B)
     Write(B,b+100)            Write(B,b+100)
   end if                    end if
```


Introducción

- Las transacciones se propusieron para simplificar la programación de las BDs.
- Una transacción consiste en una secuencia de operaciones que se ejecuta de forma atómica (indivisible).
- La atomicidad tiene dos aspectos:
  - Atomicidad de fallo: La transacción se hace en su totalidad (la transacción compromete) o no se hace (la transacción aborta).
  - Atomicidad de sincronización: Los estados intermedios de la transacción no se pueden observar.

Introducción

- El modelo de sistema tradicional para las transacciones se conoce como crash-recovery.
- En este modelo los procesos acceden a memoria persistente (no volátil) que sobrevive a los fallos.
- Los procesos fallan y se recuperan, pudiendo acceder a su memoria persistente para averiguar que hicieron en el pasado.

Introducción

- Las transacciones garantizan las propiedades ACID:
  - Atomicity: Efecto todo (compromete) o nada (aborta).
  - Consistency: Corrección del programa.
  - Isolation: Serialidad (serializability), efecto equivalente a una ejecución secuencial.
  - Durability: Los efectos de una transacción comprometida perduran (no se pierden).

Implementación de las transacciones

- El aislamiento se garantiza mediante protocolos de control de concurrencia.
- La atomicidad y durabilidad se garantizan mediante protocolos de recuperación.

Control de concurrencia

- La falta de aislamiento produce dos problemas: lecturas inconsistentes *(dirty reads)* y modificaciones perdidas *(lost updates)*.
- Las lecturas inconsistentes se producen por acceder a valores intermedios de una transacción.
- Las modificaciones perdidas se producen por dos lecturas concurrentes del mismo dato por parte de dos transacciones y éste es actualizado por las dos transacciones.

Control de concurrencia: Modificaciones Perdidas

a.Deposit(200)
- **bal= a.GetBalance()** (1)
- **a.SetBalance(bal+200)** (3)

a.Deposit(300)
- **bal= a.GetBalance()** (2)
- **a.SetBalance(bal+300)** (4)

bal	a	bal
	500 (0)	
500 (1)		
		500 (2)
	700 (3)	
	800 (4)	

Control de concurrencia: Lecturas inconsistentes

bank.Transfer(a, b, 100)
- **a.Withdraw(100)** (1)
- **b.Deposit(100)** (4)

branch.GetBalance()
- **bal= a.GetBalance()** (2)
- **bal = bal +b.GetBalance()** (3)

a	b	bal
500 (0)	300 (0)	
400 (1)		
		400 (2)
		700 (3)
	400 (4)	

Control de concurrencia

- Los protocolos se pueden clasificar en dos familias:
  - Pesimistas: Cuando dos transacciones acceden al mismo dato se retrasa a una de ellas hasta que la otra finaliza. P.ej. cerrojos (locks).
    - Puede haber interbloqueos (deadlocks).
  - Optimistas: Se deja acceder libremente a cualquier dato, cuando la transacción termina se efectúa un test para averiguar si hubo conflicto. En caso positivo se aborta, en caso afirmativo compromete. P.ej. marcas de tiempo (timestamps).
    - Pueden provocar un número elevado de abortos.

Control de concurrencia basado en cerrojos L/E

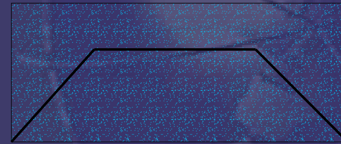
- Cuando una transacción accede a un dato solicita un cerrojo en el modo correspondiente (lectura/escritura).
- Si el cerrojo es compatible con los cerrojos activos (lectores con lectores) sobre el dato se concede, en caso contrario la transacción se bloquea.

T1:	**R(x)**	**R(y)**
T2: R(y)	**W(x)**	

Control de concurrencia basado en cerrojos L/E

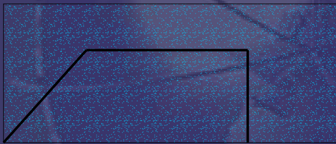
- Liberación de Cerrojos *(2 phase locking)*: Para garantizar el aislamiento se establecen dos fases: crecimiento (growing) y decrecimiento (shrinking).

Problema: ***Cascading aborts***



Control de concurrencia basado en cerrojos L/E

- Solución: *Strict two phase locking*, los cerrojos se liberan al final de la transacción.



Control de concurrencia basado en cerrojos L/E

- Las transacciones bloqueadas en un cerrojo de una transacción que ha finalizado intentan adquirir el cerrojo de nuevo.
- El problema que tienen los cerrojos es que se pueden producir interbloqueos:

T1:	**R(x)**	**W(y)**
T2: R(y)	**W(x)**	

Tratamiento de interbloqueos

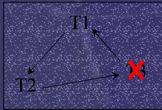
- Un interbloqueo se produce cuando dos o más transacciones se esperan mutuamente.
- Un interbloqueo se produce si y sólo si hay un ciclo en el grafo de esperas.
- Por lo tanto, es suficiente con abordar la formación de ciclos en el grafo de esperas, bien evitando su formación o eliminándolos cuando se producen.

Tratamiento de interbloqueos

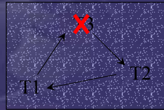
- Prevención (*prevention*): Garantizar que nunca se van a producir interbloqueos (p.ej. pedir todos los cerrojos al principio de forma atómica).

Tratamiento de interbloqueos

- Evitación (*avoidance*): Detectar interbloqueos potenciales abortando transacciones para romperlos y, así, evitar que se produzcan interbloqueos reales:
  - *wait & die*: Una transacción joven muere si espera por una vieja.
  - *wound & wait*: Una transacción vieja mata a la joven si le hace esperar.



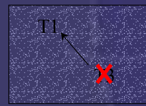
Wait & die



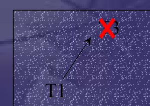
Wound & wait

Tratamiento de interbloqueos

- El principal inconveniente de la evitación es que aborta transacciones sin necesidad:



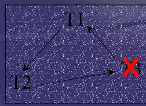
Wait & die



Wound & wait

Tratamiento de interbloqueos

- Detección/resolución (*detection/resolution*):
  - Se permiten los interbloqueos, pero hay un algoritmo encargado de reconocerlos y eliminarlos.
  - Se mantiene el grafo de esperas y cuando se detecta un ciclo se aborta una de las transacciones.



Detección y resolución

Cerrojos Semánticos: Conmutatividad

- Los cerrojos L/E se propusieron en el marco de transacciones en una BD.
- Sin embargo, hoy día existen aplicaciones que tienen operaciones más ricas: objetos con métodos para acceder a su estado, etc.
- En este contexto los cerrojos L/E ofrecen una concurrencia muy pobre.
- Para aumentar el grado de concurrencia se propuso un control de concurrencia que explotara la semántica de la aplicación: la conmutatividad [García-Molina TODS 83].

Cerrojos Semánticos: Conmutatividad

- Dos operaciones que conmuten da igual el orden en que se ejecuten, no tienen conflicto.
- Por ejemplo, dos incrementos conmutan por lo que podrían tener cerrojos compatibles.
- Un ejemplo, serían dos operaciones de depósito en una cuenta bancaria, que corresponderían a sendos incrementos.
- Al conmutar ambas operaciones de incremento, no imponen ningún orden de serialización, por lo que se pueden realizar en cualquier orden.

Cerrojos Semánticos: Conmutatividad

- Para ello, se definen por parte del usuario cerrojos específicos para cada operación.
- Estos cerrojos se encuentran parametrizados con los parámetros de interés (todos, parte o ninguno) de la operación correspondiente.
- El usuario proporciona una tabla de compatibilidad entre los distintos tipos de cerrojos.

Cerrojos Semánticos: Conmutatividad

Un ejemplo sería la clase Conjunto:

	Insert(y)	Remove(y)	IsIn(y)
Insert(x)	Yes	$x \neq y$	$x \neq y$
Remove(x)	$x \neq y$	Yes	$x \neq y$
IsIn(x)	$x \neq y$	$x \neq y$	Yes

¿Qué ocurre si aborta una transacción?

Cerrojos Semánticos: Conmutatividad

- Si se tiene en cuenta que las transacciones pueden abortar, el logging físico (almacena copias de los datos para deshacer los efectos de una transacción) es insuficiente ya que al abortar una transacción de desharían las actualizaciones que se han ejecutado concurrentemente sobre el dato.
- Por ello, es necesario definir la operación inversa de cada operación para que pueda deshacerse su efecto lógicamente (logging lógico).
- Para que dos operaciones no tengan conflicto, deben conmutar con sus inversas, y sus inversas entre ellas también.

Cerrojos Semánticos: Conmutatividad

La tabla de compatibilidad de la clase Conjunto quedaría:

	Insert(y)	Remove(y)	IsIn(y)
Insert(x)	**$x \neq y$**	$x \neq y$	$x \neq y$
Remove(x)	$x \neq y$	**$x \neq y$**	$x \neq y$
IsIn(x)	$x \neq y$	$x \neq y$	Yes

Una extensión de la conmutatividad es la recuperabilidad [Badrinath & Ramimrathan ICDE 87]

Control de Concurrencia Optimista

- Cuando hay muy pocos conflictos el control de concurrencia pesimista impone una sobrecarga innecesaria.
- Se ha propuesto como alternativa no emplear cerrojos y certificar la serialidad de las transacciones a posteriori, cuando van a comprometer.
- Si las transacciones violan la serialidad se abortan y se ejecutan de nuevo.
- Las transacciones trabajan sobre copias de los datos *(shadow copies)* y si la certificación tiene éxito las actualizaciones se propagan a los datos reales.

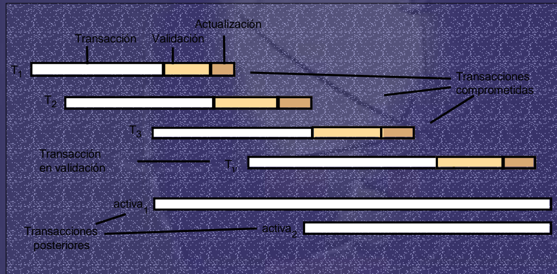
Control de Concurrencia Optimista

- Las transacciones se organizan en tres fases: fase de ejecución, validación y actualización.
- Las fases de validación y actualización se ejecutan de forma atómica.
- Las transacciones reciben una marca de tiempo (*timestamp)* cuando inician la validación.
- El orden de serialización corresponde al orden establecido por las marcas de tiempo.
- El control de concurrencia basado en cerrojos serializa las transacciones en función de sus conflictos, mientras que el optimista lo hace en función de su marca de tiempo.
- La certificación puede ser hacia atrás (backward validation) o hacia delante (forward validation).

Validación hacia atrás

- En la validación hacia atrás se comprueba que la transacción que quiere comprometer t_v no tuvo conflicto con las transacciones con las que se ejecutó concurrentemente:
  - Las transacciones que no habían comprometido cuando se inició t_v.
  - Y que comprometieron antes que t_v iniciara la validación.
  - Dicho de otra forma: las transacciones que se han validado y comprometido durante la ejecución de t_v.
- En la validación hacia atrás hay que comprobar que t_v no leyó objetos escritos por transacciones concurrentes serializadas antes que ella.

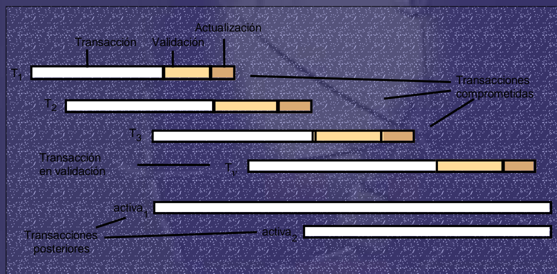
Validación hacia atrás



Validación hacia delante

- En la validación hacia delante t_v se valida con las transacciones activas en el momento que se inicia su validación.
- Se comprueba que sus actualizaciones no afectan a datos leídos por las transacciones activas (con cuidado porque es un blanco móvil).

Validación hacia delante



Recuperación

- Los métodos de recuperación pueden:
  - Rehacer (Redo) operaciones de transacciones comprometidas cuyo efecto pueda no estar reflejado en la BD.
  - Deshacer (Undo) operaciones de transacciones abortadas cuyo efecto pueda estar reflejado en la BD.
- Resulta necesario algún tipo de redundancia para garantizar la atomicidad.
  - Generalmente se hace uso de un log donde se registran operaciones realizadas y/o por realizar, así como el resultado final de las transacciones (compromiso o aborto).

Modelo de arquitectura

- La BD que vamos a considerar consta de:
  - Tablas en memoria persistente.
  - Log en memoria persistente.
  - Caché con las páginas/tuplas accedidas recientemente.
    - Los elementos de la caché pueden bloquearse (pin) para impedir su expulsión a disco y desbloquearse (unpin).

Tipos de recuperación

- Los tipos de recuperación se clasifican dependiendo del tipo de operaciones que haya que realizar durante la recuperación:
  - No Undo/Redo.
  - Undo/ No Redo.
  - Undo/Redo.
  - No Undo/No Redo.

Recuperación no undo/redo

- Si no hay registros *undo* y se escriben datos sucios a disco, en presencia de fallos se violaría la atomicidad de la transacción.
- Para garantizar la atomicidad sin escribir registros *undo* no hay que permitir la escritura de datos sucios (*dirty* writes, modificaciones no comprometidas) a disco.
- Es necesario escribir registros *redo* antes de comprometer para poder garantizar que las modificaciones de la transacción se aplicarán incluso si hay un fallo justo después del compromiso.

Recuperación no undo/redo

- Las páginas se bloquean antes de modificarse.
- Cuando la transacción compromete:
  - Se escribe en el log registros redo con las postimágenes de las páginas modificadas.
  - Se escribe en el log el compromiso de la transacción.
  - Se desbloquean las páginas modificadas.
- Durante la recuperación:
  - Para cada transacción comprometida en el log se aplican las actualizaciones correspondientes.
  - Las transacciones abortadas en el log se descartan.

Recuperación undo/no redo

- El inconveniente principal de no undo/redo consiste en que bloquea en la caché todos los datos modificados hasta que se realiza el compromiso (deja poca flexibilidad a la caché, rendimiento pobre).
- Para evitar este inconveniente se puede adoptar una política undo/no redo.
  - Si la caché se llena se pueden expulsar datos sucios a disco.
- Para garantizar la atomicidad es necesario garantizar que las escrituras sucias pueden deshacerse tanto si la transacción aborta, como si hay un fallo antes de que la transacción comprometa (durante la recuperación).

Recuperación undo/no redo

- Antes de modificar una página por primera vez se escribe en el log un registro undo con su preimagen.
- Las páginas se bloquean sólo durante cada modificación.

Recuperación undo/no redo

- Cuando la transacción compromete:
  - Se fuerzan todas las páginas modificadas a disco.
  - Se escribe un registro en el log con el compromiso de la transacción.
- Durante la recuperación:
  - Para cada transacción abortada en el log se deshacen sus actualizaciones.
  - Las transacciones comprometidas se descartan.

Recuperación undo/redo

- El principal inconveniente de la recuperación undo/no redo es que tiene que forzar los datos modificados a disco durante el compromiso (para evitar la generación de registros redo).
- ¿Por qué preocupa propagar las páginas de los datos y no preocupa tanto la propagación del log?
  - El log es una estructura de datos secuencia en la que la propagación se realiza con una sola escritura. Además, generalmente se dedica un disco al log con lo que el tiempo de acceso es el tiempo de rotación del disco.
  - Los datos, sin embargo, se encuentran en distintas partes del disco lo que requiere varios movimientos de la cabeza del disco, más las correspondientes rotaciones, por lo que resulta mucho más costoso.
- Por otro lado, la recuperación no undo/redo no deja libertad a la caché.
- Para evitar ambos inconvenientes se puede adoptar una recuperación undo/redo.

Recuperación undo/redo

- Las páginas sólo se bloquean durante su actualización.
- Las páginas no se fuerzan a disco.
- Antes de modificar una página por primera vez se escribe un registro undo con su preimagen al log.

Recuperación undo/redo

- Al comprometer:
  - Se escriben registros redo con las postimágenes de las páginas modificadas.
  - Escribir en el log un registro con el compromiso.
- Al recuperar:
  - Por cada transacción abortada en el log aplicar sus registros undo.
  - Por cada transacción comprometida en el log aplicar sus registros redo.
  - Descartar el resto de los registros.

Recuperación no undo/no redo

- Este esquema es conocido también como *shadowing*.
- Existe un directorio principal que apunta a la última versión comprometida de cada dato y uno secundario que apunta a la versión actual de cada dato.
- Cuando la transacción compromete:
  - Las páginas modificadas son apuntadas por un nuevo directorio.
  - Se conmuta atómicamente del antiguo al nuevo directorio.

Recuperación de desastres

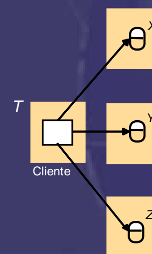
- Un desastre tiene lugar cuando se pierde parte de la memoria persistente.
- La única forma de combatirlos es el empleo de redundancias.
- *Mirroring*: Se duplica todo en dos discos.
- *Archiving*: Se saca un respaldo *(backup)* de la BD cada cierto tiempo. El log mantiene las modificaciones desde el último respaldo y se mantiene duplicado.

Transacciones distribuidas

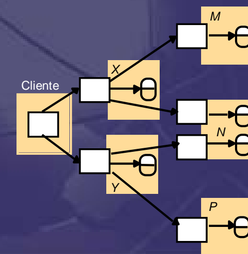
- Una transacción distribuida es aquella que accede a datos en distintos nodos de un sistema distribuido.
- Modelos de transacciones distribuidas:
  - Transacciones planas (flat): Una transacción se extiende a un número arbitrario de nodos.
  - Transacciones anidadas (nested): Una transacción se estructura en una jerarquía. Cada vez que se mueve a un nuevo nodo lo hace en un nuevo nivel de la jerarquía.

Modelos de Transacciones

(a) Transacción plana



(b) Transacciones anidadas



Transacciones distribuidas

- La distribución introduce nuevos problemas:
  - Es más difícil garantizar la atomicidad de fallo ya que puede haber fallos de los participantes.
  - El control de concurrencia afecta a varios nodos.
  - La detección de interbloqueos se complica ya que podemos observar grafos de dependencias irreales (estados globales inconsistentes, *phantom deadlocks*).
  - Cuando un nodo se cae, las transacciones iniciadas por él que estén ejecutándose en otros nodos quedan húerfanas (*orphan transactions*). Para ello se introducen algoritmos de *orphan killing*.

Transacciones anidadas: Atomicidad

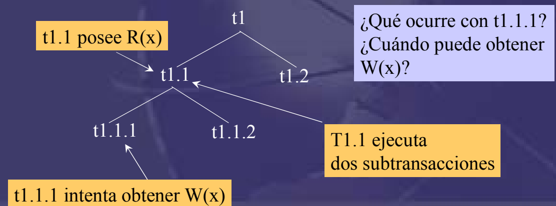
- Se pueden iniciar transacciones dentro de una transacción y se denominan subtransacciones.
- El aborto de una subtransacción no afecta a la transacción madre.
- Los efectos de una subtransacción comprometida están condicionados al compromiso de todos sus ancestros.

Transacciones anidadas: Aislamiento

- Es posible ejecutar subtransacciones concurrentemente.
- Subtransacciones concurrentes están aisladas entre sí.
- Una transacción no compromete hasta que todas sus subtransacciones hayan finalizado.

Transacciones anidadas: Control de Concurrencia

- El empleo de cerrojos en transacciones anidadas encierra algunas complicaciones.
- La transacción madre en el modelo tradicional queda bloqueada cuando ejecuta una subtransacción o conjunto de subtransacciones concurrentes.
- ¿Qué ocurre cuando una transacción hija (descendiente en general) compite con la transacción madre (ancestro en general) para obtener un cerrojo conflictivo?

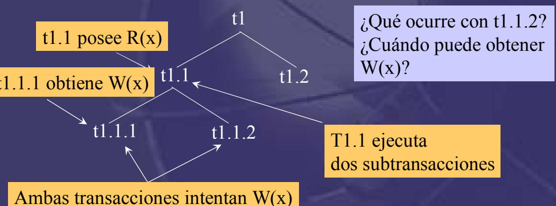


Transacciones anidadas: Control de Concurrencia

- Resulta natural que las transacciones hijas puedan acceder a los datos de las transacciones madre ya que realizan trabajo delegado por ésta.
- Para ello, resulta necesario un mecanismo para transferir los cerrojos de la transacción madre a las transacciones hijas.
- Generalmente, se emplea un mecanismo automático en el que un transacción hija puede obtener un cerrojo de la transacción madre.
- Una transacción puede obtener un cerrojo sobre un dato siempre que no exista ningún cerrojo conflictivo sobre el dato o bien los cerrojos conflictivos que hay sobre el dato corresponden a transacciones que son ancestros suyos.

Transacciones anidadas: Control de Concurrencia

- ¿Qué ocurre si dos transacciones hermanas intentan obtener un cerrojo conflictivo sobre el mismo dato y la transacción madre poseía este cerrojo?

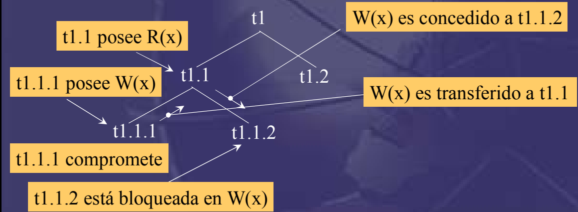


Transacciones anidadas: Control de Concurrencia

- Se necesita una regla adicional para que cuando las transacciones hijas comprometan se transfieran sus cerrojos a las transacción madre para que ésta puedan seguir trabajando.
- La nueva regla queda así:
  - Una subtransacción al comprometer transfiere sus cerrojos a la transacción madre.
  - Si la subtransacción aborta, sus cerrojos son liberados automáticamente.

Transacciones anidadas: Control de Concurrencia

- Con la nueva regla el ejemplo anterior se resuelve así:



Concurrencia entre transacciones madres e hija

- En el modelo tradicional siempre se considera que la transacción madre está bloqueada.
- Hoy día muchos de los entornos transaccionales tienen facilidades para realizar *multithreading*.
- En este caso un conjunto de *threads* puede actuar en el nombre de la misma transacción.
- Esto introduce problemas puesto que la transacción madre puede ejecutarse concurrentemente con sus hijas.

Concurrencia entre transacciones madres e hija

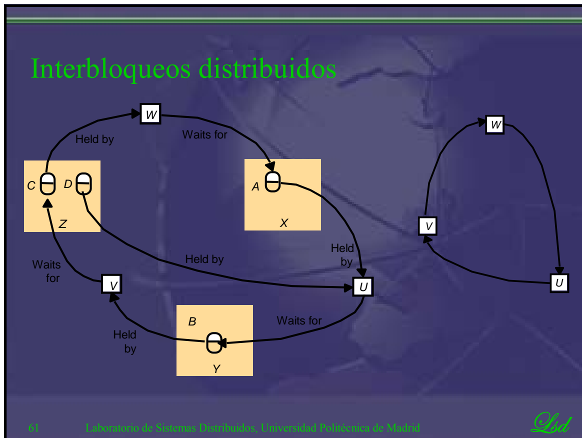
- Un problema es que una transacción hija acceda a un dato en mitad de una operación realizada por la madre:
  - Para garantizar la consistencia física de los datos se emplean *latches* (exclusión mutua).
- Otro problema es qué ocurre cuando un cerrojo conflictivo es obtenido por una transacción hija y la transacción madre intenta acceder al dato de nuevo:
  - Cuando la madre solicite el cerrojo se comprobará si existe un cerrojo conflictivo que no corresponda a un ancestro suyo y al haberlo (el de la transacción hija) quedará bloqueada hasta que la transacción hija finalice.

Interbloqueos distribuidos

- Las transacciones centralizadas pueden dar lugar a interbloqueos.
  - El empleo de timeouts es complejo:
    - Si se ponen cortos: detección rápida, pero las transacciones largas no pueden terminar.
    - Si se ponen largos: desaparece el problema de las transacciones largas, pero la detección se vuelve lenta (contención).
  - Para la detección de interbloqueos locales es preferible la detección-resolución basada en grafos de espera *(wait-for graphs)*.
- Las transacciones distribuidas dan lugar a interbloqueos distribuidos:
  - Un ciclo en el grafo de espera que no aparece en los grafos de espera locales.
- Dos alternativas:
  - Servidor centralizado: Construir los grafos de espera globales a partir de los locales en un servidor.
  - Algoritmo distribuido: Intercambiar mensajes para detectar los interbloqueos sin reconstruir el grafo global de esperas.

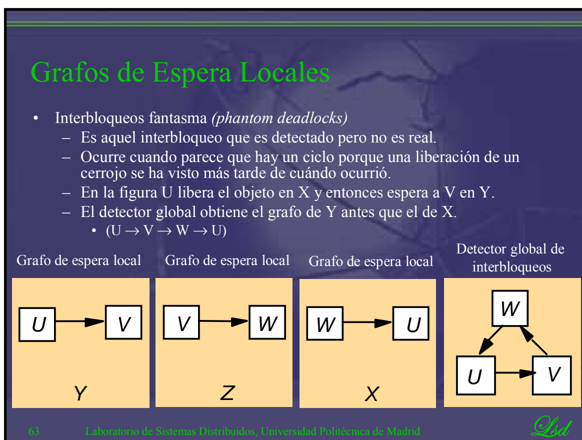
Interbloqueos distribuidos

U		*V*		*W*	
d.deposit(10)	lock *d*				
		b.deposit(10)	lock *b* at *Y*		
a.deposit(20)	lock *a* at *X*				
U → *V* at *Y*				*c.deposit(30)*	lock *c* at *Z*
b.withdraw(30)	wait at *Y*	*V* → *W* at *Z*			
		c.withdraw(20)	wait at *Z*	W → *U* at *X*	
				a.withdraw(20)	wait at *X*



Grafos de Espera Locales

- Se pueden construir grafos de espera locales
  - nodo *Y*: *U* → *V* añadido cuando *U* ejecuta *b.withdraw*(30)
  - nodo *Z*: *V* → *W* añadido cuando *V* ejecuta *c.withdraw*(20)
  - nodo *X*: *W* → *U* añadido cuando *W* ejecuta *a.withdraw*(20)
- Para encontrar un ciclo global es necesaria la cooperación entre nodos.
- Detección centralizada
  - Un nodo toma el papel de detector de interbloqueos globales.
  - Los demás nodos le envían periódicamente sus grafos locales.
  - Cuando observa un ciclo, ha detectado un interbloqueo.
  - Entonces decide qué transacción abortar e informa al resto de los nodos.
  - Este enfoque sufre los problemas típicos de soluciones centralizadas: falta de tolerancia a fallos y sobrecarga del nodo que realiza la detección.

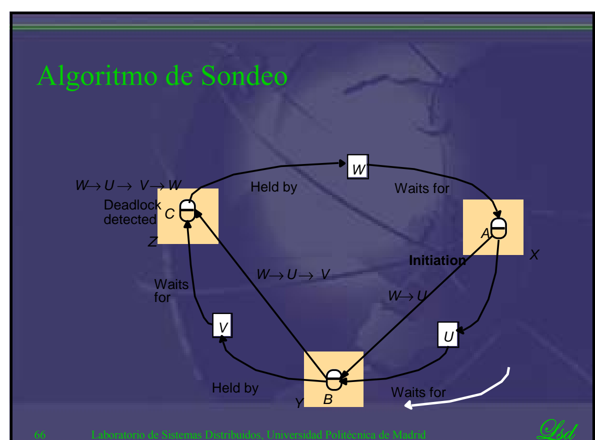


Algoritmo Distribuido de Sondeo *(edge chasing)*

- El problema del que adolece la solución centralizada para la detección de interbloqueos es un problema general conocido como estado global consistente.
- No se construye un grafo local, cada nodo conoce su grafo local e intercambia mensajes con los demás nodos.
  - Los nodos tratan de encontrar ciclos mediante el envío de sondas que siguen los arcos del grafo a través del sistema distribuido.
  - ¿Cuándo tiene un nodo que enviar una sonda?
  - Enviar una sonda cuando se crea un arco *T1* → *T2* cuando *T2* se bloquea.
  - Cada nodo registra si sus transacciones están activas o bloqueadas.
    - Los gestores de cerrojos locales comunican al detector de interbloqueos si las transacciones se bloquean o desbloquean.
    - Cuando una transacción se aborta para eliminar un interbloqueo, se informa a los participantes de los cerrojos liberados y los arcos eliminados de los grafos de espera.

Algoritmo de Sondeo

- Fases:
  - Iniciación:
    - Cuando un nodo observa que T empieza a esperar por U, cuando U está esperando en otro servidor, inicia la detección mediante el envío de una sonda conteniendo el arco < T → U > al nodo donde U está bloqueada.
    - Si U está compartiendo un cerrojo (p.ej., es de lectura), las sondas se envían a todos los poseedores del cerrojo.
  - Detección:
    - La detección consiste en la recepción de sondas y decidir cuándo ha ocurrido un interbloqueo, así como cuándo propagar las sondas recibidas.
      - P.ej. Cuando un nodo recibe la sonda < T → U > comprueba si U está esperando, p.ej. U → V, y si es así, propaga < T → U → V > al nodo donde V espera.
      - cuando un servidor añade un arco nuevo, comprueba si hay un ciclo.
  - Resolución:
    - Cuando se detecta un ciclo, una transacción involucrada en el ciclo es abortada para eliminar el interbloqueo.

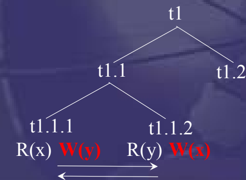


Algoritmo de Sondeo

- El algoritmo de sondeo estudiado detecta interbloqueo en las siguientes condiciones:
  - Las transacciones involucradas en el ciclo no abortan.
- Mejoras que pueden hacerse:
  - Evitar que todas las transacciones del ciclo inicien la detección.

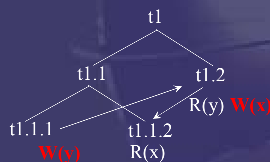
Detección de Interbloqueos en Transacciones Anidadas

- Con transacciones anidadas podemos tener interbloqueos entre transacciones de alto nivel, pero también entre subtransacciones de una transacción de alto nivel.



Detección de Interbloqueos en Transacciones Anidadas

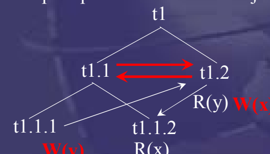
- Las transacciones anidadas introducen nuevos tipos de dependencia ya que una transacción madre no puede terminar hasta que terminen sus transacciones hijas. Esto genera interbloqueos sin ciclos explícitos.



¿Cómo se caracteriza un interbloqueo en transacciones anidadas?

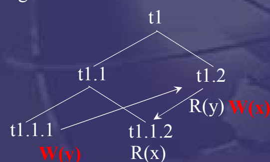
Detección de Interbloqueos en Transacciones Anidadas

- Los interbloqueos reales siempre se producen entre transacciones hermanas.
- El interbloqueo en este caso se produce porque t1.2 espera por un cerrojo que no tendrá hasta que termine t.1.1
- y t1.1 espera a que termine t1.1.1, que no terminará hasta que termine t1.2 para poder obtener el cerrojo W(y).



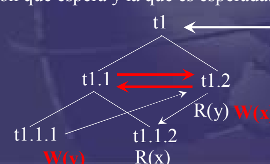
Detección de Interbloqueos en Transacciones Anidadas

- Una posibilidad consiste en enviar sondas al ancestro de la transacción esperada que sea descendiente directo del mínimo ancestro común.
- Estas sondas se propagarían a todos los descendientes.
- Este es el algoritmo de Moss.



Detección de Interbloqueos en Transacciones Anidadas

- Otra posibilidad más interesante es la propuesta por M. Rukoz.
- Como los interbloqueos ocurren en un nivel del árbol, su ancestro más inmediato puede encargarse de su detección.
- Cada vez que se produce una espera se envía el arco correspondiente al mínimo ancestro común de entre la transacción que espera y la que es esperada.



t1 centraliza la detección de interbloqueos en el siguiente nivel.

Otros Tipos de Interbloqueo

- Modelo AND: Un proceso pide cada vez un conjunto de recursos y hasta que no los obtiene todos permanece bloqueado.
- Modelo OR: Un proceso pide en cada ocasión un conjunto de recursos y se desbloquea en cuanto obtiene uno o más.
- Modelo AND-OR: Se pueden pedir un conjunto de conjuntos de recursos en los que hay que obtener al menos uno de cada uno de los subconjuntos.

Compromiso atómico distribuido (atomic commitment)

- Objetivo: Todos los participantes comprometen la transacción o ésta aborta.
- Una transacción no distribuida escribe en el log para asegurar la atomicidad ante fallos. Si la transacción es distribuida hace falta un protocolo de compromiso.
- Debido a los fallos no basta con enviar un mensaje con el resultado.

Compromiso atómico distribuido

- Un nodo puede caerse antes de procesar el mensaje de commit y de que el resto de los nodos hayan procesado el mensaje. La transacción no es atómica.
- El primer protocolo propuesto fue el compromiso en dos fases (2PC, 2 phase commit).

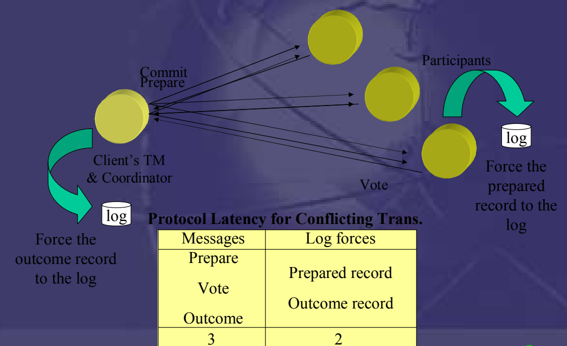
Compromiso en dos fases (2PC)

- Un nodo (*coordinador*) inicia el protocolo para una transacción. El resto de los nodos en la transacción se denominan *participantes*.
- Una transacción no compromete en ningún nodo hasta que todos los participantes están *preparados*.
- Un participante está preparado cuando ha escrito en el log las postimágenes de los datos que la transacción ha accedido .

Compromiso en dos fases (2PC)

- El coordinador registra la transacción en el log y los participantes.
- El coordinador envía un mensaje *prepare* a los participantes.
- Cada participante añade su voto al log (sí o no) y fuerza su log a disco.
- A continuación cada participante responde al coordinador con su voto.

Two Phase Commit



Protocol Latency for Conflicting Trans.

Messages	Log forces
Prepare Vote Outcome	Prepared record Outcome record
3	2

Compromiso en dos fases (2PC)

- Si el coordinador recibe *un voto negativo* aborta la transacción. Registra el aborto en el log y se lo comunica a todos los participantes.
- Si el coordinador recibe *todos los votos* y son *afirmativos*. Registra el compromiso en el log, fuerza el log a disco y envía la decisión a los participantes.
- Los participantes registran la decisión en su log y la aplican. Envían un mensaje *done* al coordinador.
- El coordinador escribe un *done record* cuando recibe el *done* de todos los participantes.

Compromiso en dos fases (2PC)

- Fallos en el 2PC:
- Desde el punto de vista del coordinador:
  - Al coordinador no le llegan todas las respuestas al *prepare*. Aborta unilateralmente la transacción.
  - No le llegan los mensajes *done*. Los sigue pidiendo.
- Desde el punto de vista de un participante:
  - No recibe el prepare. Aborta la transacción. Si después recibe este mensaje contesta no o lo ignora (es equivalente).
  - Si no recibe el resultado, se bloquea.

Compromiso en dos fases (2PC)

- La principal desventaja del 2PC es que es bloqueante cuando el coordinador ha recibido todos los votos, éstos son afirmativos, y el coordinador se cae antes de enviar el resultado.
- Si todos los participantes votaron afirmativamente aunque conversen entre sí, son incapaces de determinar si el coordinador decidió comprometer (escribió con éxito en el log la decisión de comprometer), o si falló antes de decidirlo (en cuyo caso la transacción deberá abortar).

Compromiso en dos fases (2PC): Recuperación

- El estado del coordinador puede ser:
  - No tiene el registro de inicio del 2PC en el log. Ningún participante ha recibido el mensaje prepare y han abortado la transacción.
  - Tiene el registro de inicio, pero no la decisión. Aborta la transacción y envía un mensaje a los participantes. Si ya han abortado la transacción ignoran ese mensaje.
  - Tiene la decisión, pero no el registro *done*. Los participantes pueden estar esperando el resultado, se envía.

Compromiso en dos fases (2PC): Recuperación

  - Tiene el reg. *done*. No hace nada.
- El estado de un participante puede ser:
  - No tiene el reg. *prepare*. La transacción no pudo comprometer. Aborta unilateralmente.
  - Tiene el reg. *prepare*, pero no la decisión. No sabe qué hacer. Ejecuta un *protocolo de terminación*.
  - Tiene la decisión. Puede mandar un *done* al coordinador o esperar a que éste se lo pida.

2PC: Recuperación. Protocolo de terminación

- Protocolos de terminación:
  - Esperar a que el coordinador se recupere. El coordinador envía el resultado.
  - Intervención manual. Cuando se recupere el coordinador habrá que detectar posibles inconsistencias. Otras transacciones pueden haber visto resultados intermedios.
  - Protocolo de terminación cooperativa. El coordinador incluye en el mensaje *prepare* la lista de participantes.

2PC: Recuperación. Protocolo de terminación

- Cuando un participante se recupera pregunta al resto por el resultado.
- Cuando un participante recibe ese mensaje, si no tiene el registro *prepare* la transacción ha abortado. Si tiene ese registro, pero no el resultado no puede ayudar. Si tiene el resultado lo envía. El participante que se está recuperando se lo envía al resto.

- Los participantes no eliminan el resultado de la transacción de memoria inmediatamente. Lo hacen pasado un tiempo o cuando el coordinador envía otro mensaje después de recibir todos los *done*.

Compromiso en dos fases (2PC): Eficiencia.

- La eficiencia está determinada por el número de mensajes y escrituras en el log.
- Las escrituras en el log pueden ser *eager* (hay que esperar a que finalicen antes de continuar con el siguiente paso del protocolo) o *lazy* (se pueden realizar a posteriori y no tienen impacto en la duración).
- En el coordinador:
  - El registro de la transacción es *lazy*. No conoce a los participantes de la transacción.
  - El resultado es eager. Si lo envía antes de que se escriba y se cae, cuando se recupere abortará la transacción (inconsistencias).

Compromiso en dos fases (2PC): Eficiencia.

- Después de recibir todos los *done*. Esta escritura es *lazy*.

- En un participante:
  - *Eager* cuando recibe el mensaje *prepare*. Si contesta antes de prepararse, cuando se recupere abortará. Inconsistencias.
  - *Lazy* cuando escribe la decisión antes de responder con *done*. Si se cae y no ha escrito la decisión, el participante tendrá que preguntar por el resultado de la transacción cuando se recupere.

Compromiso en dos fases (2PC): Optimizaciones.

- *Presumed abort*.
  - El coordinador no escribe el registro de inicio. Cuando se recupera, si no hay un registro de commit, la transacción ha abortado.
  - Un participante escribe la decisión (si es abortar) en el log de manera perezosa y no envía el mensaje *done*.
  - Si la decisión es abortar, el coordinador no escribe en el log.
- Las bases de datos usan esta optimización.

Compromiso en dos fases (2PC): Optimizaciones

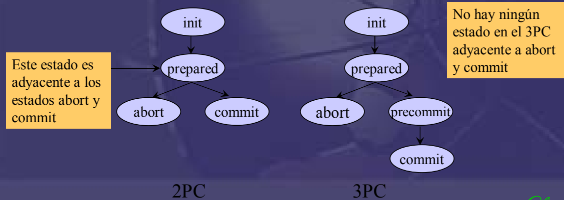
- Transacciones de sólo lectura. Si un participante sólo ha leído datos, indica en la respuesta al prepare que es una transacción de sólo lectura y libera los cerrojos.
- El coordinador ya no le envía el resto de los mensajes.
- Las bases de datos no las usan porque otro participante puede adquirir cerrojos (lanza un *trigger* al recibir el *prepare*) en el nodo que ya ha enviado su voto y violar el aislamiento.

Compromiso en 3 fases (3PC)

- Propiedad no bloqueante: Si un proceso vivo no conoce el resultado, ningún proceso (vivo o muerto) ha podido comprometer.
- Se introduce una fase adicional, preparado-para-comprometer *(precommit)*, para evitar la situación de bloqueo.
- Esta fase se introduce entre la preparación y el compromiso.
- El estado de un participante puede ser: init, prepared (cualquiera de estos dos es incierto), abort, precommit, commit.
- El coordinador sólo envía el mensaje commit cuando ha recibido acks de precommit de todos los participantes.

Compromiso en 3 fases (3PC)

- Si representamos con un autómata los estados del 2PC y 3PC podemos observar por qué uno es bloqueante y el otro no lo es.



Compromiso en 3 fases (3PC)

- Timeouts
  - A un participante no le llega el prepare o al coordinador los votos -> abortar (igual que en 2PC)
  - No llegan los acks del precommit. Como todos han votado sí, el coordinador compromete.
  - No llega el precommit o el abort.
  - No llega el commit.
- El los dos últimos casos se hace un protocolo de terminación.

Compromiso en 3 fases (3PC): Terminación

- Se elige un nuevo coordinador mediante un protocolo de elección de líder.
- El nuevo coordinador pide el estado a los participantes:
  - Si algún participante tiene el resultado, lo envía al resto.
  - Si algún participante está en precommit, pero ninguno en commit. Envía precommit a los que están inciertos y cuando recibe los acks envía el commit.
  - Si todos están inciertos, aborta la transacción.

Compromiso en 3 fases (3PC): recuperación

- Si un participante falló antes de enviar el voto, aborta.
- Si tiene la decisión, conoce el resultado de la transacción.
- Ha votado afirmativamente, pero no tiene el resultado. Pregunta por el resultado.
- Si tiene el precommit no puede comprometer unilateralmente. La transacción ha podido abortar por lo que tiene que preguntar.

Compromiso en 3 fases (3PC)

- El mensaje de precommit no ayuda en la recuperación, no se graba en el log.
- Se hacen las mismas escrituras en el log que en 2PC.
- Introduce una ronda más de mensajes que en el 2PC.

Replicación de datos

- Objetivo: mejorar la disponibilidad y la eficiencia.
  - Las lecturas se pueden hacer en cualquier copia.
  - Acceder a datos locales.
- Corrección: *one-copy serializable.* La ejecución de las transacciones en una base de datos replicada debe ser equivalente a ejecutarlas en una base de datos no replicada.
- Problema: Hay que actualizar todas las copias.

Replicación de datos

- Las escrituras se pueden propagar dentro de la misma transacción-> *Eager replication* (síncrona).
  - No hay inconsistencias.
  - Aumenta la latencia.
- En otra transacción-> *Lazy replication* (asíncrona).
- Las escrituras se pueden realizar en cualquier réplica -> *Update everywhere.*
- Las escrituras sólo se pueden hacer en una réplica -> *primary copy*. Las *queries* se pueden hacer en todas las réplicas. Es la que usan las bases de datos.

	Dónde se pueden realizar las actualizaciones	
Cuándo se propagan las actualizaciones	En el primario	En cualquier réplica
Dentro de la transacción	Eager primary copy	Eage update everywhere
Fuera de la transacción	Lazy primary copy	Lazy update everywhere

Replicación de datos

- Una alternativa a propagar las escrituras es replicar la peticiones (*replicación activa*). Hay que asegurar que todas las réplicas serializan las transacciones en el mismo orden.
- En el *reliable transaction router* de Digital una petición replicada se ejecuta como una transacción distribuida. Emplea 2PC en la terminación. Si falla una réplica, el protocolo continúa y la transaccción compromete.

Replicación de datos

- Cuando la replicación es *eager* las escrituras se pueden hacer en todas las réplicas simultáneamente o se puede hacer la escritura local y cuando la transacción compromete se escribe en el resto de las copias (en diferido).
  - El número de mensajes es menor en diferido. Las escrituras se pueden enviar en un mensaje (prepare).
  - El coste de los abortos es menor (sólo se deshace en un nodo).
  - Los conflictos se detectan más tarde (si es *update everywhere*).

Replicación de datos

- Si se puede escribir en todos los nodos disponibles, cuando hay particiones se producen inconsistencias.
- *Quórum consensus o majority quorum:* sólo una componente puede procesar transacciones.
  - Cada nodo tiene un peso.
  - Todos los nodos conocen el peso de los demás nodos en el sistema.

Replicación de datos

- Un quórum es un conjunto de nodos con más de la mitad del peso total. Sólo la componente con un quórum puede progresar cuando hay una partición.
- Un quórum de lectura (RT) (escritura WT) es un conjunto de copias 2.WT y WT+RT > total suma de los pesos.
- Una lectura y una escritura, o dos escrituras de un dato siempre se solapan, al menos en una copia.
- Cada copia de un dato tiene un número de versión
- Una lectura lee de un quórum la versión más reciente.

Replicación de datos

- Las lecturas no son locales.
- Una escritura se realiza en un quórum de escritura. Tiene que tomar la última versión en el quórum, incrementarla y etiquetar todas las copias del quórum con ese número de versión.
- Hace falta un número alto de copias para tolerar un número dado de fallos. Un fallo, tres copias. Dos fallos cinco copias...

Quorums de Lectura-Escritura

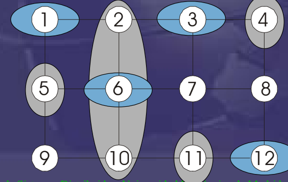
- Los sistemas de quorums se han adaptado para permitir la realización de lecturas en paralelo.
- Los quorums de exclusión mutua se emplean como quorums de escritura y además se definen los quorums de lectura.
- Un quórum de lectura cumple la propiedad de que tiene intersección no nula con cualquier quórum de escritura.

Quorums de Lectura-Escritura

- **Rejilla rectangular *(rectangular grid)*.**
  - Los elementos del universal se organizan en una rejilla rectangular de *f* filas y *c* columnas.
  - Un quórum de escritura está formado por una columna completa y un representante de cada una de las columnas restantes.
  - Un quórum de lectura está formado por un representante de cada columna.

Write Quorum

Read Quorum

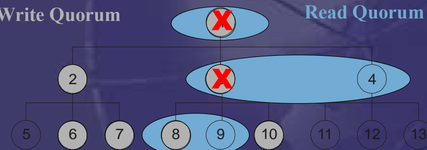


Quorums de Lectura-Escritura

- **Arborescente.**
  - Los elementos del universal se organizan en un árbol de grado impar (>= 3).
  - Los quorums de escritura se definen recursivamente del siguiente modo:
    - La raíz forma parte del quorum.
    - Una mayoría de sus descendientes pertenece al quórum y así sucesivamente.
  - El quórum de lectura lo compone la raíz.
  - Si la raíz ha fallado, el quórum de lectura lo compone una mayoría de sus hijos.
  - Cualquiera de sus hijos puede ser sustituido por una mayoría de sus nietos (si no hay suficientes vivos para conseguir una mayoría) y así sucesivamente.

Quórum Arborescente

Write Quorum



Quorums de Lectura-Escritura

- **Arborescente Logarítmico.**
  - Los elementos del universal se organizan en un árbol de grado impar (>= 3).
  - Los quorums de escritura son un camino de la raíz a las hojas.
  - El quórum de lectura lo compone la raíz.
  - Si la raíz ha fallado, el quórum de lectura lo componen todos sus hijos.
  - Cualquiera de sus hijos puede ser sustituido por todos sus nietos (si está caído).