CC5303 – Sistemas Distribuidos

6.- Tolerancia a Fallos

Parte 2

Sebastián Blasco V.

De la clase pasada...

- Conceptos Básicos
- Atenuación del proceso
  - Consenso frente a fallas parciales
- Multitransmisión confiable
  - Cómo hacer llegar con certeza un mensaje a todo un grupo.

Hoy

- Atomicidad (tx distribuidas)
- Cómo recuperarse de una falla

Atomicidad (Tx distribuidas)

La multitransmisión atómica analizada previamente es ejemplo de un problema todavía más general, conocido como **realización distribuida.**

Realización (ó *TX*) distribuida

- Lograr que una operación sea realizada por cada miembro de un grupo o por ninguno en absoluto. En el caso de multitransmisión confiable, la operación es la entrega de un mensaje.
- Conjuntos de operaciones englobadas dentro de un bloque cuya ejecución es completa.

Atomicidad (Tx distribuidas)

Cumplen las propiedades **ACID**

- **A**tomicity (Atomicidad)
  - La transacción se realiza completa o no se realiza nada.
- **C**onsistency (Consistencia)
  - Los estados anterior y posterior a la transacción son estados estables (consistentes).
- **I**solation (Aislamiento)
  - Los estados intermedios de la transacción son sólo visibles dentro de la propia transacción.
- **D**urability (Durabilidad)
  - Las modificaciones realizadas por una transacción completada se mantienen

Atomicidad (Tx distribuidas)

Ej. Viaje en avión

- SCL - TYO
  - Santiago (SCL) - Houston (IAH) ✔
  - Houston (IAH) - San Francisco (SFO) ✔
  - San Francisco (SFO) - Tokio (TYO) ✘

¡Todo o Nada!

No nos sirve quedar a mitad de camino! Debemos ser capaces de pedir la devolución en caso de que algún tramo no tenga disponibilidad.

Atomicidad (Tx distribuidas)

Ej. Update Perdido: **Serialización Incorrecta**

- A = $100, B = $200, C = $300

TransacciónA:	TransacciónB:
balance = b.getBalance()	balance = b.getBalance()
b.setBalance(balance*1.1)	b.setBalance(balance*1.1)
a.giro(balance/10)	c.giro(balance/10)

balance = b.getBalance()	//200		
		balance = b.getBalance()	//200
		b.setBalance(balance*1.1)	//220
b.setBalance(balance*1.1)	//220		
a.giro(balance/10)	//80		
		c.giro(balance/10)	//280

Atomicidad (Tx distribuidas)

Ej. Update Perdido: **Serialización Correcta**

- A = $100, B = $200, C = $300

TransacciónA:	TransacciónB:
balance = b.getBalance()	balance = b.getBalance()
b.setBalance(balance*1.1)	b.setBalance(balance*1.1)
a.giro(balance/10)	c.giro(balance/10)

balance = b.getBalance()	//200		
b.setBalance(balance*1.1)	//220		
		balance = b.getBalance()	//220
		b.setBalance(balance*1.1)	//242
a.giro(balance/10)	//80		
		c.giro(balance/10)	//278

Atomicidad (Tx distribuidas)

Ej. Lecturas inconsistentes: **Serialización Incorrecta**

- A = $100, B = $200

TransacciónA:	TransacciónB:
a.giro(100)	balanceTotal()
b.deposito(100)	

```
a.giro(100)                 //100

                                        total = a.getBalance()      //100

                                        total += b.getBalance()     //300

b.deposito(100)             //300
```

Atomicidad (Tx distribuidas)

Ej. Lecturas inconsistentes: **Serialización Correcta**

- A = $100, B = $200

TransacciónA:	TransacciónB:
a.giro(100)	balanceTotal()
b.deposito(100)	

```
a.giro(100)                 //100
b.deposito(100)             //300
                                        total = a.getBalance()      //100
                                        total += b.getBalance()     //400
```

Atomicidad (Tx distribuidas)

- Primitivas de transacción
  - BEGIN_TRANSACTION
  - END_TRANSACTION
  - ABORT_TRANSACTION

En medio de una transacción se podrán realizar diversas operaciones, según la aplicación. Las transacciones deberán ser todo o nada y además deben ejecutarse en exclusión mutua unas con otras.

¿Cómo lograr el efecto todo o nada (serializaciones)?

Atomicidad (Tx distribuidas)

¿Cómo lograr el efecto todo o nada (serializaciones)?

Tres maneras de controlar concurrencia:

- Locks
- Control de concurrencia optimista
- Ordenamiento de timestamps

Atomicidad (Tx distribuidas)

¿Cómo lograr el efecto todo o nada (serializaciones)?

- Locks
  - Locks a nivel de cada objeto accedido
  - Mayor granularidad de locks permite más transacciones concurrentes
  - Mayor atención a posibilidad de deadlocks
  - Ventaja
    - Posibilidad de obtener mayor concurrencia de transacciones
  - Desventaja
    - Mayor overhead para evitar deadlocks en la medida que se aumenta la granularidad

Atomicidad (Tx distribuidas)

¿Cómo lograr el efecto todo o nada (serializaciones)?

- Control de concurrencia optimista
  - Supongamos que todo va a andar bien. No se hacen chequeos de conflicto durante la ejecución
  - Sólo al hacer commit se verifica la consistencia de los objetos

    Si se detectan conflictos, se abortan las transacciones involucradas.
  - Ventaja
    - Máximo paralelismo con mínimo overhead, cuando no hay conflictos
  - Desventaja
    - Transacciones son mucho más lentas cuando se producen conflictos.

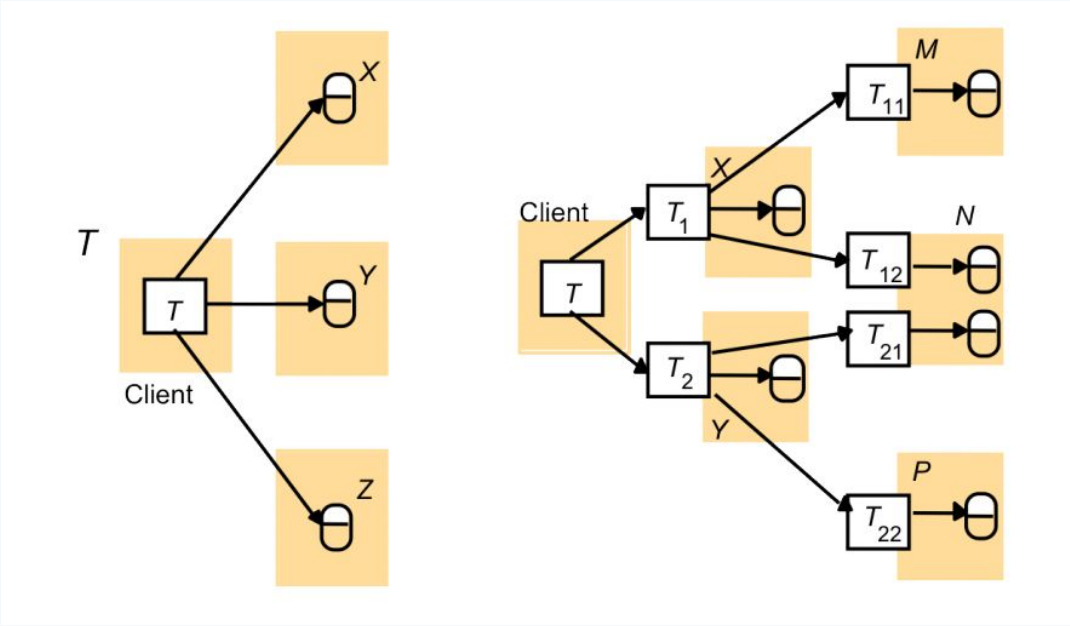
Atomicidad (Tx distribuidas)

¿Cómo lograr el efecto todo o nada (serializaciones)?

- Ordenamiento de timestamps
  - El servidor guarda el timestamp del *read* y *write* más recientes para cada operación sobre un objeto.
  - De acuerdo al timestamp, una nueva operación puede ser ejecutada:
    - inmediatamente
    - *delayed* (Operaciones delayed hacen que la transacción espere.) ó
    - *rejected* (Operaciones rejected hacen que la operación aborte.)
  - Ventajas
    - No produce deadlocks.
  - Desventajas
    - Overhead puede ser importante al tener que analizar cada operación.

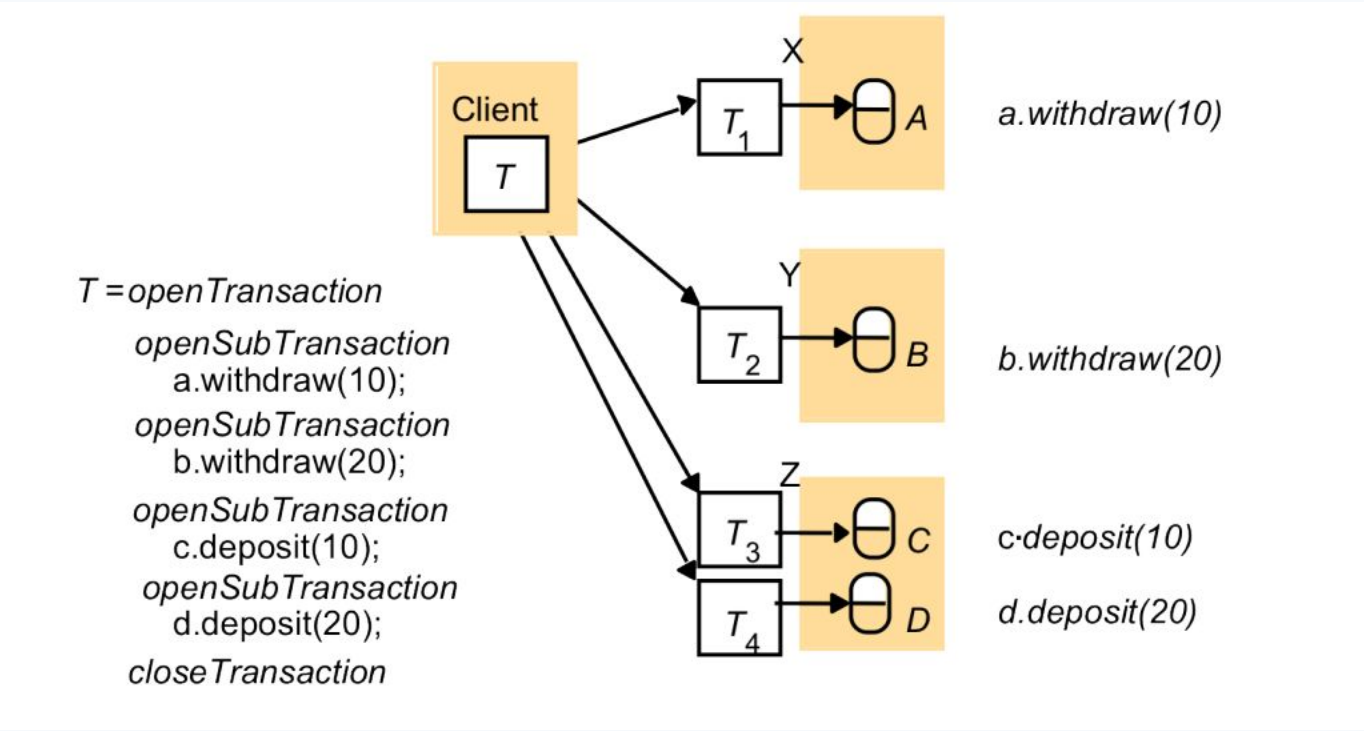
Atomicidad (Tx distribuidas)

Pero ojo, Tx distribuidas! => Los objetos y, por lo tanto, las operaciones, se ejecutan en distintos nodos.



Atomicidad (Tx distribuidas)

Ejemplo: transacción bancaria anidada y distribuida



Atomicidad (Tx distribuidas)

- Espacio de trabajo privado
  - Consiste en mantener una copia de los objetos o memoria que se quiera modificar.
  - Por ejemplo: si la transacción implica acceso a un directorio particular, mantenemos una copia. Intentamos llevar a cabo la transacción en la copia. Si nada falla al final modificamos el original. Si no, descartamos la copia.

Atomicidad (Tx distribuidas)

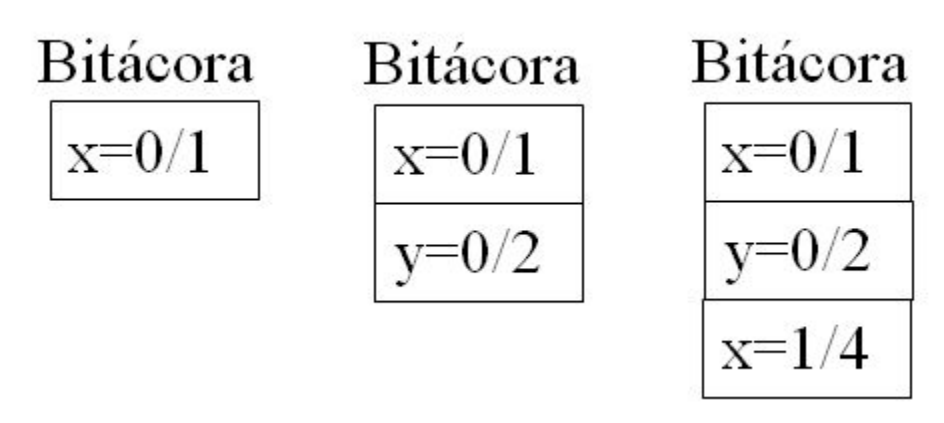
- Espacio de trabajo privado
  - Bitácora (Logging)
    - La bitácora es una lista de los cambios que se van realizando sobre cada objeto implicado en la transacción.
    - En la lista se incluye el estado anterior y posterior del objeto
    - Podemos hacer los cambios en los objetos reales, pues con la bitácora tenemos información para deshacer: partimos del final hacia atrás.

x=0; y=0;

BEGIN_TRANSACTION

x=x+1; y=y+2; x=y*y;

END_TRANSACTION



Atomicidad (Tx distribuidas)

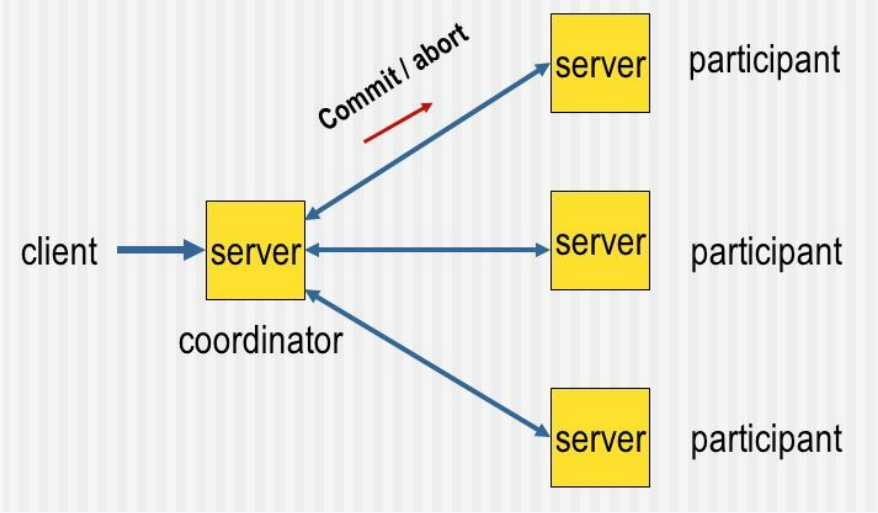
Para garantizar atomicidad de las Tx distribuidas la coordinación de los nodos corre por cuenta de los **protocolos de commit**.

Objetivo: garantizar atomicidad al hacer commit en una transacción distribuida (hacer o no una operación)

- one-phase commit
- two-phase commit
- three-phase commit

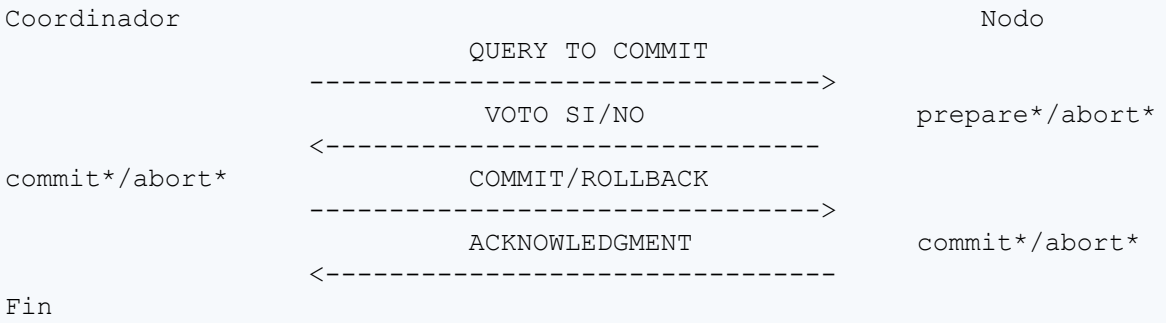
Atomicidad (Tx distribuidas)

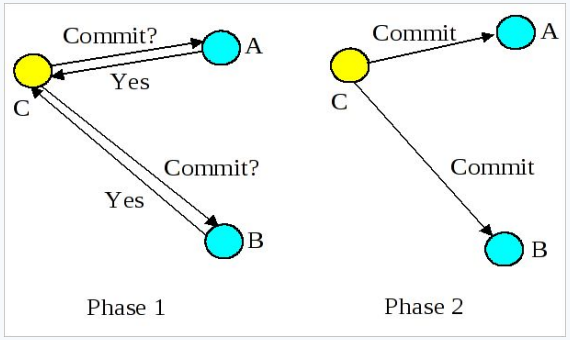
- one-phase commit
  - Coordinador indica *commit* o *abort* a todos hasta que digan ACK.
  - Problema
    - Algunos procesos pueden haber hecho *commit* y otros *abort*
    - Si alguien dice *abort*, la transacción completa debería hacer *abort* Pero algún proceso ya podría haber hecho *commit*



Atomicidad (Tx distribuidas)

- two-phase commit
  - Uno de los procesos actúa como **coordinador**
  - El proceso se separa en dos etapas
    - Etapa 1: Los procesos votan por commit o abort
      - No pueden arrepentirse de votar commit
      - Pero pueden ser obligados a hacer abort
      - Procesos guardan estados antes de la modificación.
    - Etapa 2: La decisión final es transmitida
      - Si un proceso vota abort, se hace abort
      - Los procesos ejecutan el commit o el abort



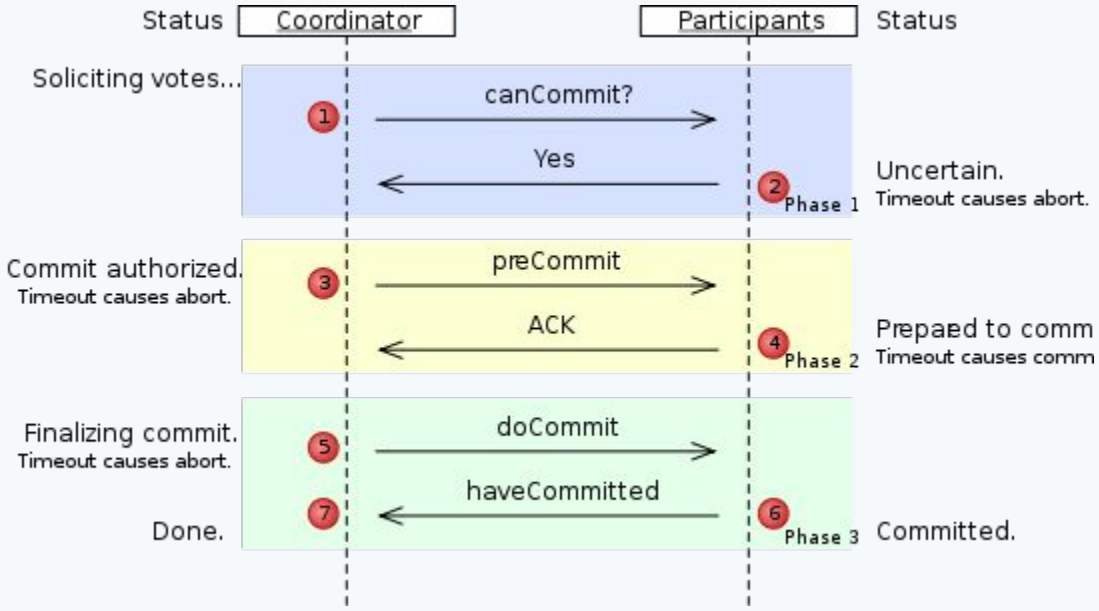


Atomicidad (Tx distribuidas)

- three-phase commit
  - Un problema con el protocolo de realización bifásica es que cuando el coordinador se congela, es posible que los participantes no puedan llegar a una decisión final.
  - Al igual que el 2PC, el 3PC también está formulado en función de un coordinador y varios participantes.
  - Con este método, como cada nodo tiene la decisión almacenada independientemente, si el coordinador o cualquier otro nodo cae la ejecución de la transacción no es afectada.

Atomicidad (Tx distribuidas)

- three-phase commit



Recuperación frente a fallas

Permitir a un sistema continuar funcionando, o retomar su funcionamiento ante eventos externos

- Caídas (bloqueos)
- Falta de conectividad

4 Enfoques

- El algoritmo del avestruz (ignorar el problema)
- Detección (permitir que ocurran bloqueos, detectarlos e intentar recuperarse)
- Prevención (imponer restricciones para que podamos asegurar que no se van a dar bloqueos)
- Evitarlos (que los procesos hagan una cuidadosa asignación de recursos para que no se den bloqueos)

¿Solución?

- Reiniciar todo el sistema, o recuperar la ejecución desde un estado establecido.
  - ¿Desde qué estado?

Recuperación frente a fallas

Detección distribuida de un bloqueo

- Cada máquina mantiene su gráfica de recursos y procesos
- Un coordinador recibe (mediante mensajes) esa información. Con la visión global, toma las decisiones
- Cuando el coordinado detecta un ciclo, elimina los procesos para romper el bloqueo

Recuperación frente a fallas

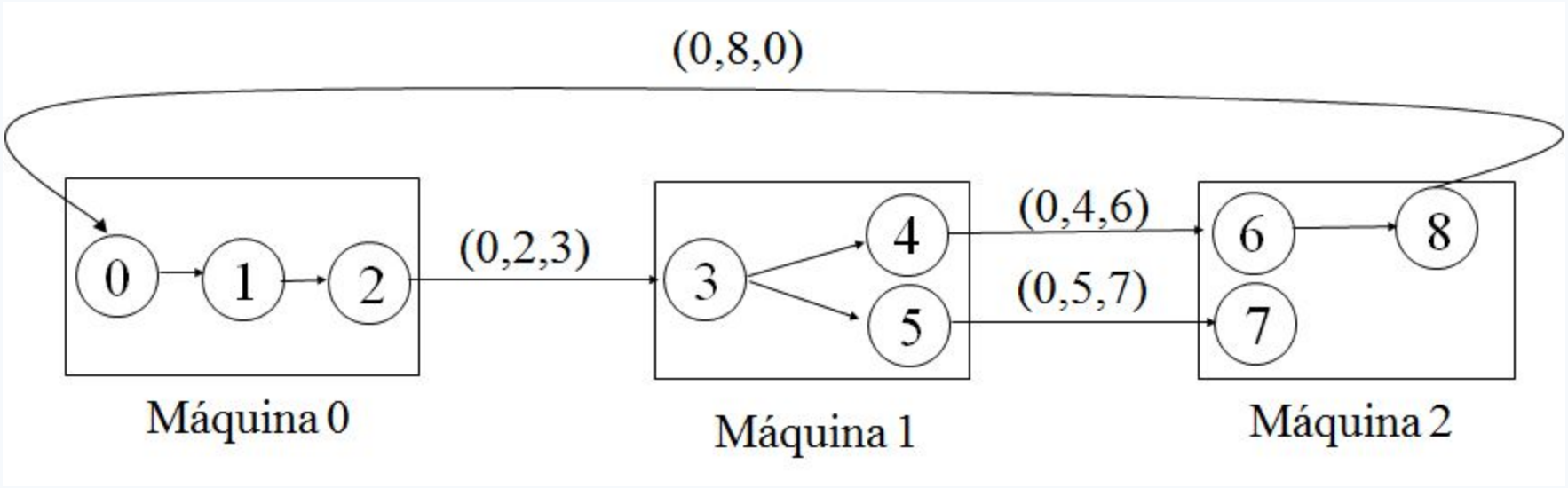
Detección distribuida de un bloqueo

- Chandy-Misra-Haas
  - Cuando un proceso debe esperar por un recurso, construye un mensaje especial de exploración, que envía al proceso o procesos que retienen el recurso.
  - El mensaje consta de tres números: el proceso que espera, el proceso que envía el mensaje y el proceso al cual se envía.
    - [A, A, B] (enviado de A a B)
  - Al llegar el mensaje, el receptor comprueba si él también espera a algunos procesos. En ese caso el mensaje se actualiza, conservando el primer campo pero reemplazando el segundo por su propio número de proceso y el tercero por el nº del proceso al cual espera.
    - [A, B, C] (Enviado de B a C)
  - El mensaje se reenvía entonces al proceso por el cual espera.
  - Si un mensaje regresa al emisor original (el proceso enumerado en el primer campo) es que hay un ciclo y el sistema está bloqueado

Recuperación frente a fallas

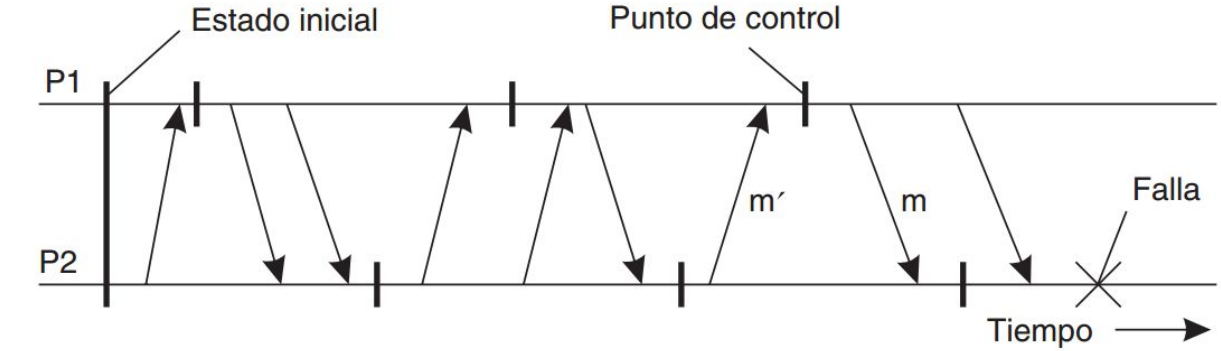
Detección distribuida de un bloqueo

- Chandy-Misra-Haas



Recuperación frente a fallas

- La recuperación se logra marcando con **puntos de control** el estado del sistema en forma regular.
  - La marcación de puntos de control es completamente distribuida.
  - La marcación de puntos de control debe hacerse a intervalos regulares.
  - A este proceso se le llama también *Checkpointing*
- Problema: La toma de un punto de control es una operación cara.



En un escenario de recuperación, cada proceso se reinicia desde su último punto de control

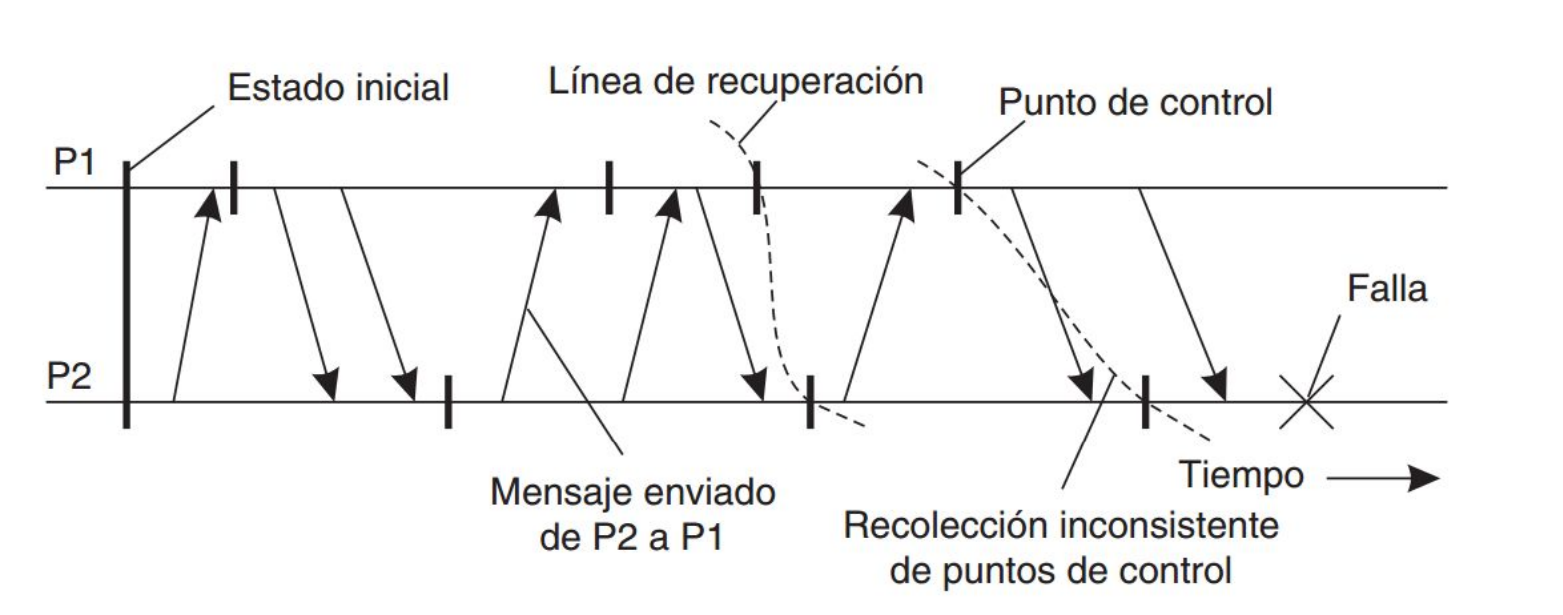
Recuperación frente a fallas

- En un escenario de recuperación, cada proceso se reinicia desde su último punto de control
  - PERO!
    - Esto no garantiza que el estado global de recuperación sea consistente
  - Sol? Los procesos que hacen el estado inconsistente se reinician desde su checkpoint anterior
    - Peor caso: se regresa al estado inicial (efecto dominó)

En la práctica en SSDD es un proceso coordinador el responsable de iniciar el algoritmo de **snapshots coordinado** y que se asegura que los estados almacenados sean consistentes.

Recuperación frente a fallas

En la práctica en SSDD es un proceso coordinador el responsable de iniciar el algoritmo de **snapshots coordinado** y que se asegura que los estados almacenados sean consistentes.



Recuperación frente a fallas

Para recuperarse a un estado previo, es necesario que la información requerida para habilitar la recuperación sea guardada con seguridad. Podemos tener tres tipos de almacenamiento:

- Memoria RAM
  - Se borra al fallar la energía o en un fallo de la máquina
- Disco
  - Sobrevive a fallos anteriores pero podría fallar la cabeza lectora del disco
- Almacenamiento estable
  - Diseñado para sobrevivir a todo.
  - Es lo que más anhelamos!

Recuperación frente a fallas

El almacenamiento estable se puede lograr con un par de discos:

- Cuando se quiere escribir se escribe primero en el disco 1 y luego en el disco 2
- Si el sistema falla tras escribir en la unidad 1, tras recuperar podemos comprobar que ambos discos son inconsistentes. Hacemos entonces que el 2 copie lo distinto en el 1, pues el 1 es siempre el que se modifica primero
- Cuando se detecte error (p.ej. por CRC) en un sector de uno de los discos, se repara con la información del otro

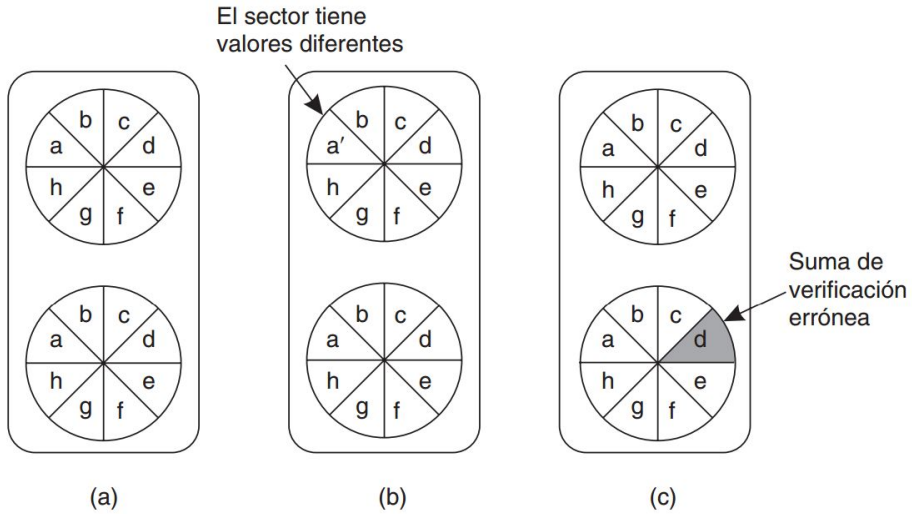


Figura 8-23. (a) Almacenamiento estable. (b) Congelación después de que se actualiza el disco 1. (c) Punto defectuoso.