



UNIVERSIDAD CATÓLICA  
de Colombia



UNIVERSIDAD CATÓLICA  
de Colombia

Vigilada Mineducación

# Transacciones de los Sistemas Distribuidos

Diego Alberto Rincón Yáñez  
darincon@ucatolica.edu.co



# Agenda

- Introducción
- Sincronización
- Transacciones
  - ACID
- Control de Concurrencia
- Bloqueos
- Ordenación por marcas de tiempo
- Two Phase Commit



# Introducción

- Primer acercamiento: Algoritmos “**Semi-Centralizados**”
- Objetivo de las **transacciones**:
  - Asegurar que todos los objetos gestionados por **un servidor** permanecen en un estado (c) cuando dichos objetos son **accedidos por múltiples transacciones** y en **presencia de fallos**



# Sincronización Sencilla

- Las operaciones realizadas en nombre de **diferentes clientes** pueden **interferir** a veces unas con otras.
- La **interferencia** puede producir valores **incorrectos** en los objetos.
- Frecuentemente se usan **hilos** para permitir concurrentemente las operaciones de varios clientes y aun accediendo, posiblemente, a los **mismos objetos**.





# Sincronización Sencilla

- Si los métodos no están **diseñados** para su utilización en un **programa multi-hilo**, es posible que las acciones de dos o más ejecuciones concurrentes del método puedan combinarse arbitrariamente y tener efectos extraños.
- Ejemplo en java:

```
public synchronized void deposita(int cantidad) throws RemoteException{  
  
}
```



# Sincronización Sencilla

- En varias ocasiones es **necesario** hacer que los **hilos se comuniquen**.
- Ejemplo en java:
  - Metodos: **wait** y **notify**.
  - Wait: Un hilo llama a wait en un objeto para suspenderse él mismo y permitir a otro hilo ejecutar un método en ese objeto.
  - Notify: Un hilo llama a notify en un objeto para informar a cualquier hilo que esta esperando en el objeto que ha cambiado alguno de sus datos.



# Modelos de fallos para transacciones

- En este modelo (**Lampson**) se intenta que los algoritmos trabajen correctamente en presencia de fallos predecibles, pero no se hacen consideraciones sobre su comportamiento cuando ocurre un desastre.
- El modelo establece:
  - Las **escrituras/lecturas** pueden fallar.





# Modelos de fallos para transacciones

- Los **servidores** pueden fallar **ocasionalmente**.
- Puede existir un **retardo arbitrario** antes que llegue un mensaje.



# Transacciones

- ¿Qué es?

- ACID

- Atomicidad

Todo o nada: Una transacción finaliza correctamente, y los efectos de todas sus operaciones son registrados en los objetos, o si fallan no tienen ningún efecto.



# Transacciones

- Isolation
  - Cada transacción debe realizarse sin interferencia de otras transacciones.
- Consistency
  - La transacción en el caso que se haga debe hacer pasar el sistema de un estado consistente a otro.
- Durability
  - Los cambios registrados en los recursos debido a transacciones que hagan COMMIT deben soportar fallos siguiendo el modelo de lampson.



# Transacciones

<i>Con éxito</i>	<i>Abortado por el cliente</i>	<i>Abortado por el servidor</i>
<i>AbreTransacción</i> <i>Operación</i> <i>Operación</i> • • <i>Operación</i>	<i>AbreTransacción</i> <i>Operación</i> <i>Operación</i> • • <i>Operación</i>	<i>AbreTransacción</i> <i>Operación</i> <i>Operación</i> • • <i>ERROR en la operación</i> <i>informado al cliente</i>
<i>CierraTransacción</i>	<i>AbortaTransacción</i>	

El servidor aborta  
la transacción →



# Transacciones

- Acciones de servicio relacionadas con la ruptura del proceso
  - Si un proceso **servidor** falla, se reemplaza en algún momento.
    - El nuevo proceso aborta todas las transacciones no finalizadas
    - Usa un proceso de **recuperación** para restablecer los valores de los objetos producidos por la transacción finalizada de forma correcta.



# Transacciones

- Acciones de un **cliente** relativas a la ruptura del proceso servidor
  - Si un servidor falla mientras una transacción esta en progreso, el cliente será consciente de ello cuando una de las operaciones devuelve una excepción.



# Control de concurrencia

- Para describir los problemas de transacciones concurrentes se usa el contexto de un ejemplo bancario.
  - Tres UP (objetos), el estado solo consiste en una variable entera.  $A=100$ ,  $B=200$ ,  $C=300$
- Los problemas teniendo en cuenta el **solapamiento de las operaciones** son:
  - Actualizaciones perdidas
  - Recuperaciones inconsistentes





# Control de concurrencia (Actualizaciones perdidas)

Transacción <i>T</i> :	Transacción <i>U</i> :
<i>balance = b.obtenBalance( );</i> <i>b.deposita(balance/10);</i> <i>a.extrae(balance/10)</i>	<i>balance = b.obtenBalance( );</i> <i>b.deposita(balance/10);</i> <i>c.extrae(balance/10)</i>
<i>balance = b.obtenBalance( );</i> 200 \$	<i>balance = b.obtenBalance( );</i> 200 \$
<i>b.deposita(balance/10);</i> 220 \$	<i>b.deposita(balance/10);</i> 220 \$
<i>a.extrae(balance/10)</i> 80 \$	<i>c.extrae(balance/10)</i> 280 \$



# Control de concurrencia (Recuperaciones inconsistentes)

Transacción V :		Transacción W :	
<i>a.extrae(100);</i> <i>b.deposita(100)</i>		<i>unasucursal.totalSucursal( )</i>	
<i>a.extrae(100);</i>	0 \$	<i>total = a.obtenBalance( )</i>	0 \$
		<i>total = total + b.obtenBalance( );</i>	200 \$
		<i>total = total + c.obtenBalance( )</i>	500 \$
<i>b.deposita(100)</i>	300 \$	• •	



# Control de concurrencia

- Para describir los problemas de transacciones concurrentes se usa el contexto de un ejemplo bancario.
  - Tres UP (objetos), el estado solo consiste en una variable entera.  $A=100$ ,  $B=200$ ,  $C=300$
- Los problemas teniendo en cuenta el **estado final de las transacciones** son:
  - Lecturas sucias
  - Escrituras prematuras



# Control de concurrencia

## (Recuperabilidad de transacciones abortadas)

### Lecturas sucias

Transacción <i>T</i> :		Transacción <i>U</i> :	
<i>a.obtenBalance( )</i>		<i>a.obtenBalance( )</i>	
<i>a.ponBalance(balance + 10)</i>		<i>a.ponBalance(balance + 20)</i>	
<i>balance = a.obtenBalance( )</i>	100 \$	<i>balance = a.obtenBalance( )</i>	110 \$
<i>a.ponBalance(balance + 10)</i>	110 \$	<i>a.ponBalance(balance + 20)</i>	130 \$
		<i>consumar transacción</i>	
<i>abortar transacción</i>			



# Control de concurrencia

## (Recuperabilidad de transacciones abortadas)

### Escrituras prematuras

Transacción <i>T</i> :		Transacción <i>U</i> :	
<i>a.deposita(5)</i>		<i>a.deposita(10)</i>	
	100 \$		
<i>a.deposita(5)</i>	105 \$		105 \$
		<i>a.deposita(10)</i>	115 \$
<i>CT</i>	<i>AT</i>	<i>AT</i>	<i>CT</i>
<i>AT</i>	<i>AT</i>	<i>AT</i>	<i>AT</i>



# Control de concurrencia (Equivalencia secuencial)

Transacción <i>T</i> :		Transacción <i>U</i> :	
<i>balance = b.obtenBalance( );</i>		<i>balance = b.obtenBalance( );</i>	
<i>b.deposita(balance/10);</i>		<i>b.deposita(balance/10);</i>	
<i>a.extrae(balance/10);</i>		<i>c.extrae(balance/10);</i>	
<i>balance = b.obtenBalance( );</i>	200 \$	<i>balance = b.obtenBalance( );</i>	220 \$
<i>b.deposita(balance/10);</i>	220 \$	<i>b.deposita(balance/10);</i>	242 \$
<i>a.extrae(balance/10);</i>	80 \$	<i>c.extrae(balance/10)</i>	278 \$



# Control de concurrencia (Equivalencia secuencial)

Transacción V :		Transacción W :	
<i>a.extrae(100);</i> <i>b.deposita(100)</i>		<i>unasucursal.totalSucursal( );</i>	
<i>a.extrae(100);</i>	0 \$		
<i>b.deposita(100)</i>	300 \$		
		<i>total = a.obténBalance( )</i>	0 \$
		<i>total = total + b.obténBalance( );</i>	300 \$
		<i>total = total + c.obténBalance( );</i>	600 \$
		...	





# Control de concurrencia (Operaciones conflictivas)

Operaciones de diferentes transacciones		Conflicto	Causa
<i>Lee</i>	<i>Lee</i>	No	Porque el efecto de un par de operaciones de <i>lectura</i> no depende del orden en el que son ejecutadas.
<i>Lee</i>	<i>Escribe</i>	Sí	Porque el efecto de una operación de <i>lectura</i> y una de <i>escritura</i> dependen del orden de su operación.
<i>Escribe</i>	<i>Escribe</i>	Sí	Porque el efecto de un par de operaciones de <i>escritura</i> depende del orden de su ejecución.



# Bloqueos

- Si los datos son compartidos por varias transacciones debe ser secuencialmente equivalentes.
- La equivalencia secuencial requiere que todos los accesos de una transacción a un objeto particular sean secuenciados con respecto a los accesos por otras transacciones
- Mecanismo sencillo
  - **Bloqueo exclusivo**



# Bloqueos Exclusivos

Transacción <i>T</i> :		Transacción <i>U</i> :	
<i>bal = b.obtenBalance( )</i> <i>b.deposita(bal /10)</i> <i>a.extrae(bal /10)</i>		<i>bal = b.obtenBalance( )</i> <i>b.deposita(bal /10)</i> <i>c.extrae(bal /10)</i>	
Operaciones	Bloqueos	Operaciones	Bloqueos
<i>abreTransacción</i>		<i>abreTransacción</i>	
<i>bal = b.obtenBalance( )</i>	bloquea <i>B</i>	<i>bal = b.obtenBalance( )</i>	espera por el fin del bloqueo en <i>B</i> de <i>T</i>
<i>b.deposita(bal/10)</i>		...	
<i>a.extrae(bal/10)</i>	bloquea <i>A</i>		
<i>cierraTransacción</i>	desbloquea <i>A, B</i>	<i>b.deposita(bal/10)</i>	bloquea <i>B</i>
		<i>c.extrae(bal/10)</i>	bloquea <i>C</i>
		<i>cierraTransacción</i>	desbloquea <i>A, B</i>



# Bloqueos

- Para asegurar la equivalencia secuencial se necesita ( **bloqueo de dos fases** )
  - No está permitido a una transacción ningún nuevo bloqueo después que ha liberado uno.
  - 2 Fases:
    - Primera fase -> Fase de **crecimiento** -> Solicitan todos los bloqueos
    - Segunda fase -> Fase de **acortamiento** -> Liberan todos los bloqueos usados por la transacción.



# Bloqueos

- **Bloque de dos fases estricto:** Cuando se consuma una transacción, para asegurar la recuperabilidad, los bloqueos se mantienen hasta que todos los objetos que “ella” actualizo han sido escrito en memoria permanente.



# Bloqueos

- Utilizar siempre un **bloqueo exclusivo** tanto para las operaciones de lectura como de escritura **reduce la concurrencia** más de lo necesario.
- Maximizar la concurrencia: **Método de muchos lectores/un escritor**, tipos de bloqueo:
  - Bloqueos de lectura
  - Bloqueos de escritura.



# Bloqueos

- El método de muchos lectores/un escritor tiene en cuenta las **reglas de conflicto**:
  - Dependiendo de las operaciones de lectura y/o escritura.
- El método debe cumplir las siguientes dos reglas:
  - Si una transacción T ha realizado ya una operación de **lectura** en un objeto particular, entonces una transacción concurrente U **no** debe **escribir** ese objeto hasta la finalización de T(C/A)
  - Si una transacción T ha realizado ya una operación de **escritura** en un objeto en particular, entonces una transacción concurrente U **no** debe **leer** o **escribir** ese objeto hasta la finalización de T(C/A)





# Bloqueos

<i>Para un objeto</i>		<i>Bloqueo solicitado</i>	
		<i>Lectura</i>	<i>Escritura</i>
<i>Bloqueo ya activado</i>	<i>Ninguno</i>	Bien	Bien
	<i>Lectura</i>	Bien	Espera
	<i>Escritura</i>	Espera	Espera



# Bloqueos

- Reglas de uso de una técnica de bloqueos de dos fases estricto.
  1. Cuando se accede a un objeto en una transacción:
    - (a) Si el objeto no estaba bloqueado, se bloquea y comienza la operación.
    - (b) Si el objeto tiene activado un bloqueo, y es conflictivo, la transacción debe esperar hasta que esté desbloqueado.
    - (c) Si el objeto tiene activado un bloqueo, y no es conflictivo, se comparte el bloqueo y comienza la operación.
    - (d) Si el objeto ya ha sido bloqueado en la misma transacción, y no existe conflicto en la promoción, el bloqueo será promovido y comienza la operación. (Si la promoción es conflictiva, se utiliza la regla (b)).
  2. Cuando una transacción se consuma o aborta, el servidor desbloquea todos los objetos bloqueados por la transacción.



# Bloqueos

- Implementación de bloqueos: La concesión de bloqueos será implementado por un objeto separado -> gestor de bloqueo.
- El gestor de bloqueo debe mantener un conjunto de bloqueos.
- Cada bloqueo es una instancia de la clase Bloqueo que mantiene la siguiente información:
  - El identificador del objeto bloqueado.
  - Los identificadores de las transacciones que mantienen actualmente el bloqueo
  - Un tipo de bloqueo.

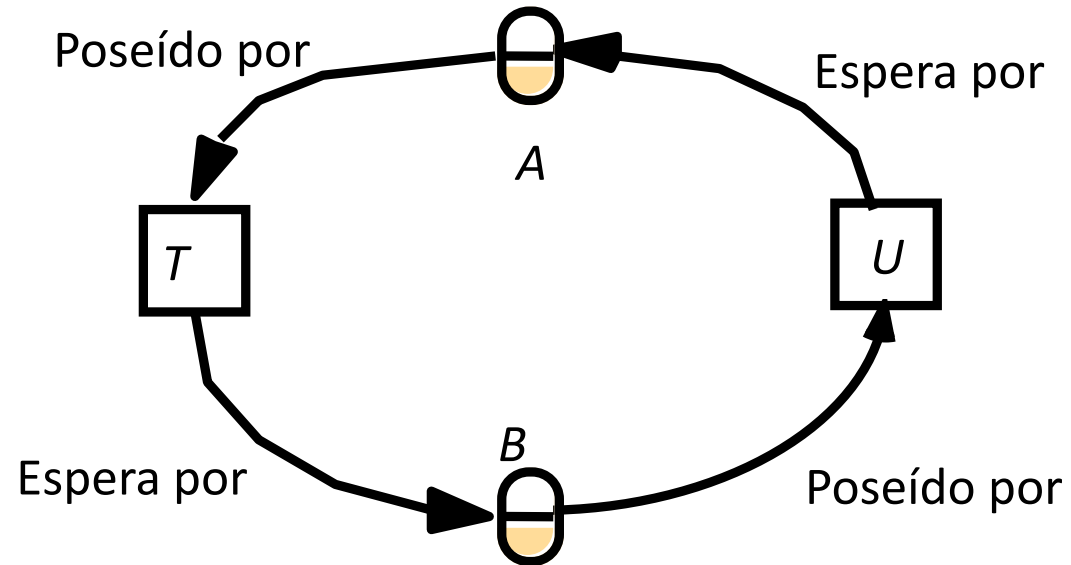
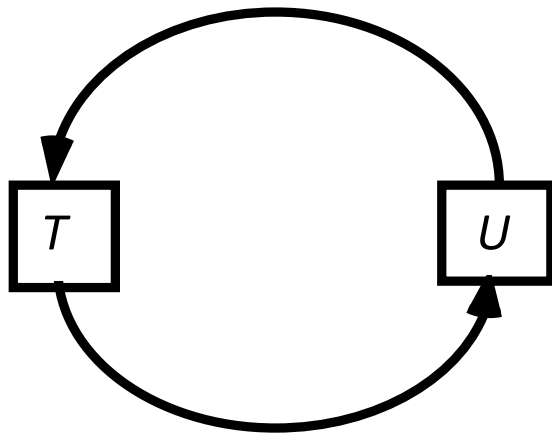


# Bloqueos indefinidos

Transacción <i>T</i>		Transacción <i>U</i>	
Operaciones	Bloqueos	Operaciones	Bloqueos
<i>a.deposita(100);</i>	Escribe bloquea A		
<i>b.extrae(100);</i>		<i>b.deposita(200);</i>	Escribe bloquea <i>B</i>
• • •	Espera por <i>U</i>	<i>a.extrae(200);</i>	Espera por <i>T</i>
• • •	Bloqueo en <i>B</i>	• • •	
• • •		• • •	Bloqueo en <i>A</i>
• • •		• • •	



# Bloqueos indefinidos

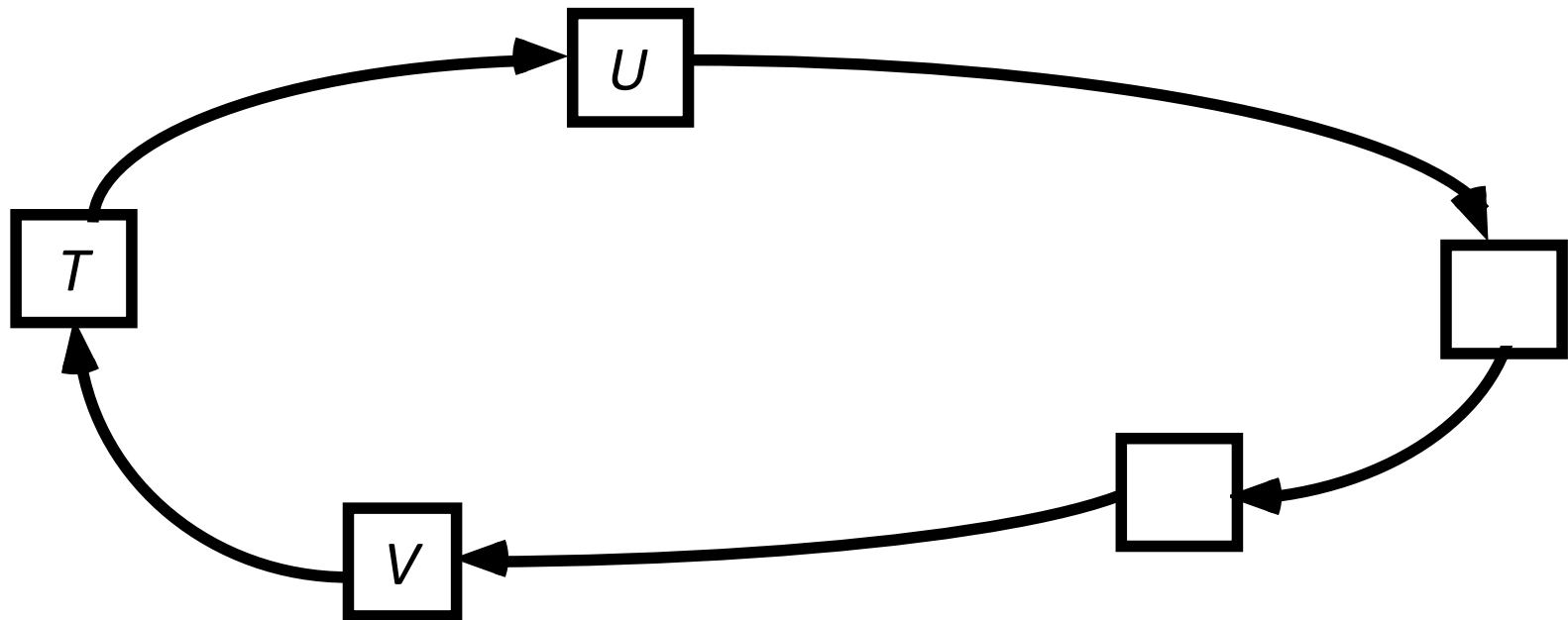


# Bloqueos indefinidos (Definición)

- Estado en el que un grupo de transacciones( $m_2$ ) se presentan las siguientes situaciones:
  - Una  $T(T)$  posee un bloqueo que necesita otra transacción del grupo ( $U$ ).
  - Una  $T(T)$  necesita un recurso que esta siendo bloqueado por otro miembro del grupo ( $U$ ).



# Bloqueos indefinidos (Definición)

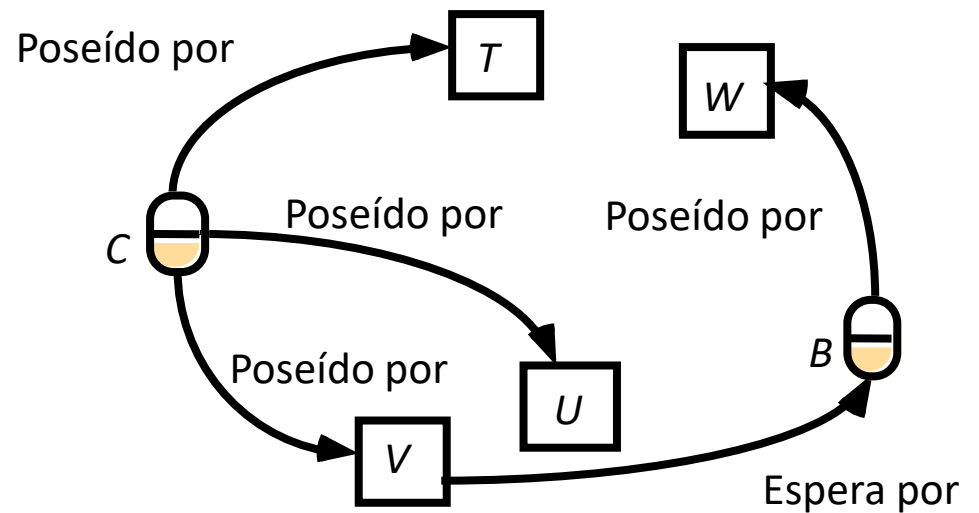
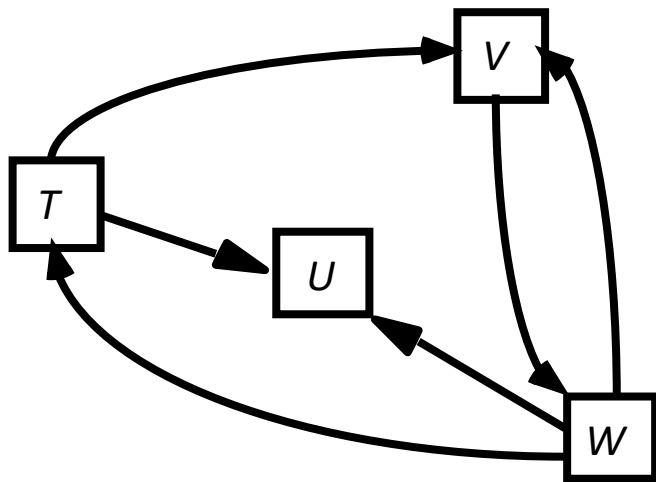


Un ciclo en el grafo *espera por*.





# Bloqueos indefinidos (Definición)



Otro grafo *espera por*.



# Bloqueos indefinidos (Soluciones)

- Prevenirla: Bloquear todos los objetos utilizados por una transacción cuando comienza.
- Detección de bloqueos indefinidos: Detectar ciclo en grafos de espera por.
- Timeouts



# Ordenación por marcas de tiempo

- La regla de ordenación básica por marca de tiempo está basada en los conflictos de operación y es muy sencilla.

“Una solicitud de una transacción para escribir un objeto es válida sólo si ese objeto fue leído y escrito por última vez por transacciones anteriores”.

“Una petición de lectura de un objeto por una transacción es válida sólo si ese objeto fue escrito por última vez por una transacción anterior”



# Ordenación por marcas de tiempo

- Como es normal, las operaciones de escritura son registradas en versiones tentativas de los objetos y son invisible por las demás transacciones hasta que se realiza una solicitud de cierre de transacción.
- Cada objeto tiene una marca de tiempo de escritura y un conjunto de marcas de tiempo de lectura.



# Ordenación por marcas de tiempo

- Regla de escritura por ordenación de marca de tiempo

if( $T_c \geq$  la máxima marca de tiempo de lectura en D &&  $T_c >$  la marca de tiempo de escritura en la versión consumada de D)

realiza la operación de escritura en la versión tentativa de D con marca de tiempo  $T_c$

else

aborta la transacción  $T_c$

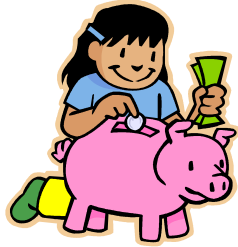


# Ordenación por marcas de tiempo

Regla	$T_c$	$T_i$	
1.	Escritura	Lectura	$T_c$ no debe <i>escribir</i> un objeto que haya sido <i>leído</i> por cualquier $T_i$ , donde $T_i > T_c$ esto requiere que $T_c \geq$ la mayor marca de tiempo de lectura del objeto.
2.	Escritura	Escritura	$T_c$ no debe <i>escribir</i> un objeto que haya sido <i>escrito</i> por cualquier $T_i$ , donde $T_i > T_c$ esto requiere que $T_c >$ la marca de tiempo de escritura del objeto consumado.
3.	Lectura	Escritura	$T_c$ no debe <i>leer</i> un objeto que haya sido <i>escrito</i> por cualquier $T_i$ , donde $T_i > T_c$ esto requiere que $T_c >$ la marca de tiempo de escritura del objeto consumado.



# Ejemplo



cliente

Transferir \$1000  
Desde A:\$3000  
A B:\$2000

Banco A

Banco B

- Clientes quieren transacciones todo o nada
  - La transferencia pasa o no pasa.



# Solución de Strawman



cliente

Transferir \$1000  
Desde A:\$3000  
A B:\$2000

Coordinador  
de transacciones

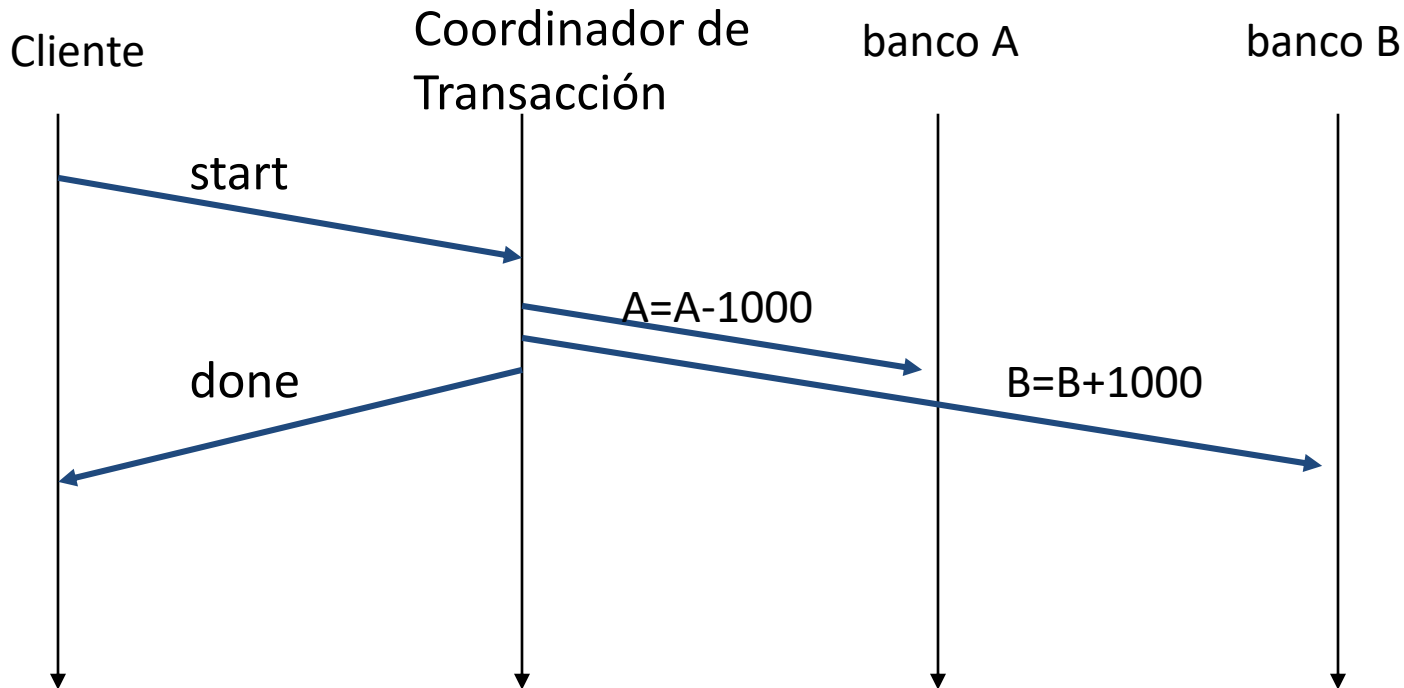
Banco A

Banco B





# Solución de Strawman



- Que puede pasar mal?
  - A no tiene suficiente dinero
  - La Cuenta de B no existe
  - B ha colapsado
  - El coordinador Falla

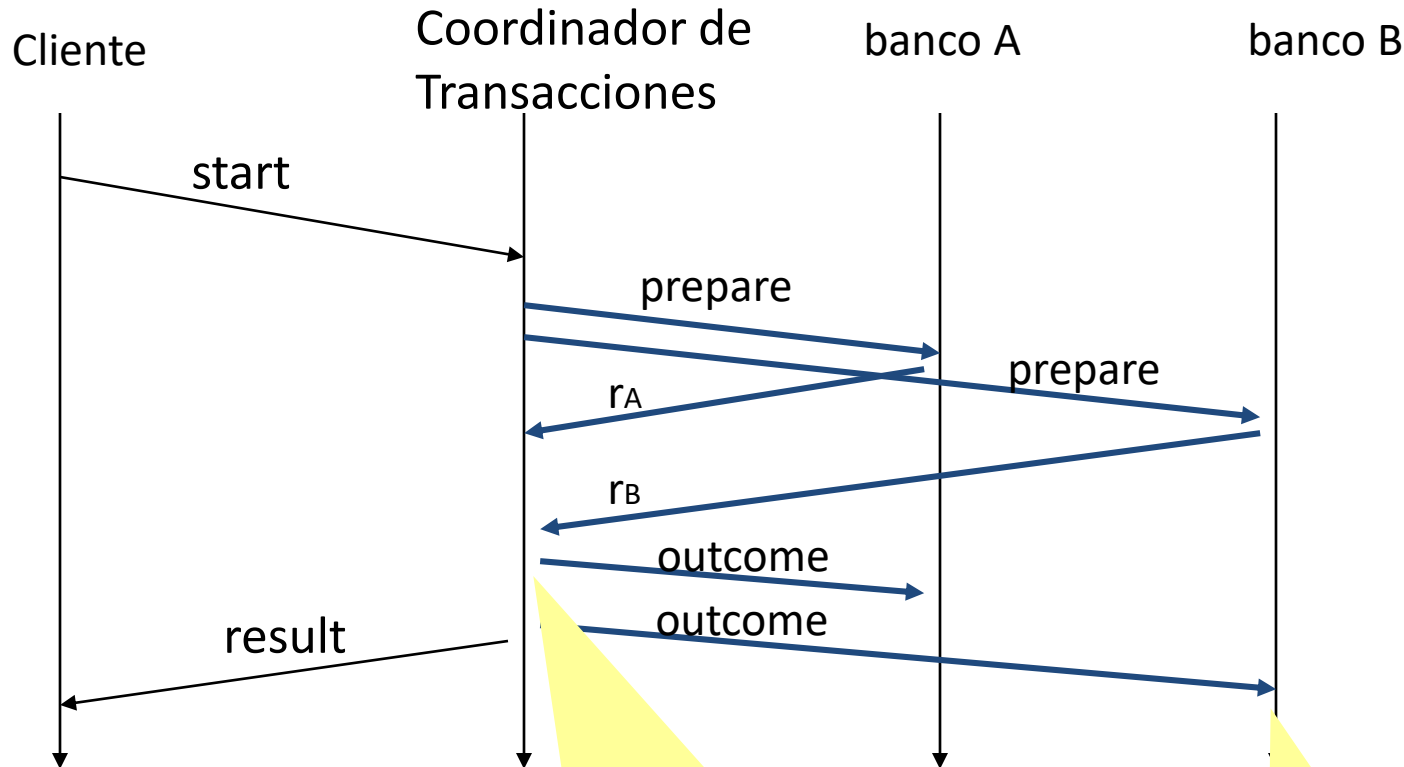


# Razonando.....

- CT, A, B cada uno tiene una noción del Commit
- Correcto:
  - Si uno hace commit, nadie aborta
  - Si uno aborta, nadie hace commit
- Desempeño:
  - Si no hay fallas , a y B pueden hacer commit, entonces hacen commit
  - Si hay una falla, encontrar la razón de la falla



# Correctness first



If  $r_A == \text{yes} \ \&\& \ r_B == \text{yes}$   
outcome = "commit"  
else  
outcome = "abort"

B hace commit una vez  
recibe la señal "commit"

# Problemas de Desempeño

- Que hay acerca de los timeouts?
  - CT falla esperando la respuesta de A
  - A falla esperando la respuesta del TC
- Que hay acerca de los reinicios?
  - Como un participante hace cleanup?



# Manejando los Tiempos de a y B

- El CT hace time out esperando A (o B)'s “yes/no” como respuesta
- Puede el CT decidir unilateralmente commit?
- Puede el CT decidir unilateralmente abort?



# Manejando el Timeout con el CT

- Si B responde con no
  - Puede unilateralmente abortar?
- Si B responde con si
  - Puede unilateralmente abortar?
  - Puede unilateralmente hacer commit?



# Posible terminación

- Protocolo ejecuta la terminación si B tiene un time out con el CT y ha votado “yes”
- B envia “status” mensaje a A
  - Si A recibe “commit”/”abort” de CT ...
  - Si A no ha respondido a CT, ...
  - Si A ha respondido con “no”, ...
  - Si A ha respondido con “yes”, ...

Resuelve la mayoría de los casos cuando falla el CT



# Manejando caída y reinicio

- Nodos no pueden retractarse si el commit esta decidido.
- Si el CT falla después de decidir “commit”
  - No puede olvidar la decisión después de reiniciar
- A/B falla después de enviar el “yes”
  - No puede olvidar la decisión después de reiniciar





# Manejando caída y reinicio

- Todos los nodos deben registrar el avance del protocolo
- Cuando y donde el CT registra en disco?
- Cuando y donde el A/B registra en disco?



# Recuperación tras Reinicio

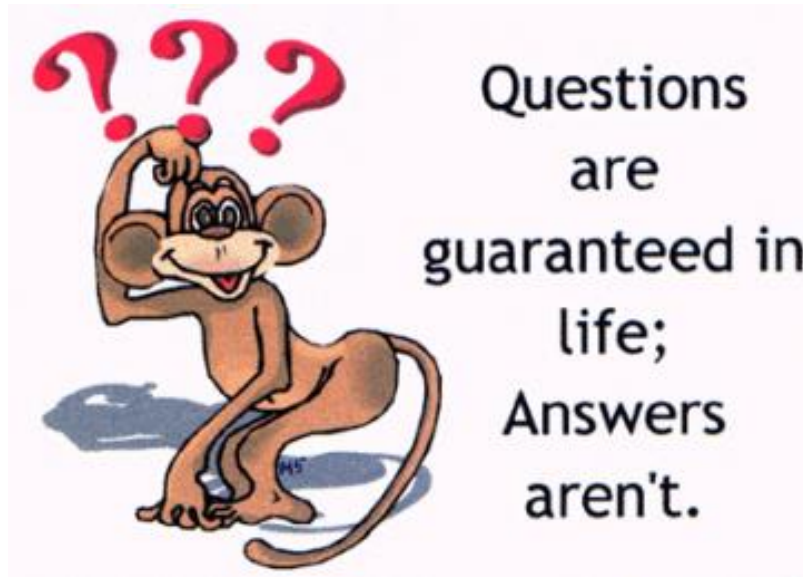
- SI CT no encuentra “commit” en disco , abort
- SI CT encuentra “commit”, commit
- SI A/B no encuentra “yes” ” en disco , abort
- Si A/B encuentra “yes”, corre el protocolo de terminación para decidir.



# Resumen: two-phase commit

1. Todos los nodos deben llegar a la misma decisión.
2. Nadie hace commit hasta que todos digan “yes”
3. No hay fallos si todos dijeron “yes” y luego hacen commit
4. Si hay fallas, reparar, esperar lo suficiente para recuperar y tomar una decisión.





# ¿Preguntas?

Diego Alberto Rincón Yáñez MCSc.  
Twitter: @d1egoprog.

