TRANSACCIONES DISTRIBUIDAS

Tema # V

Sistemas de operación II

Abril-Julio 2008

Yudith Cardinale

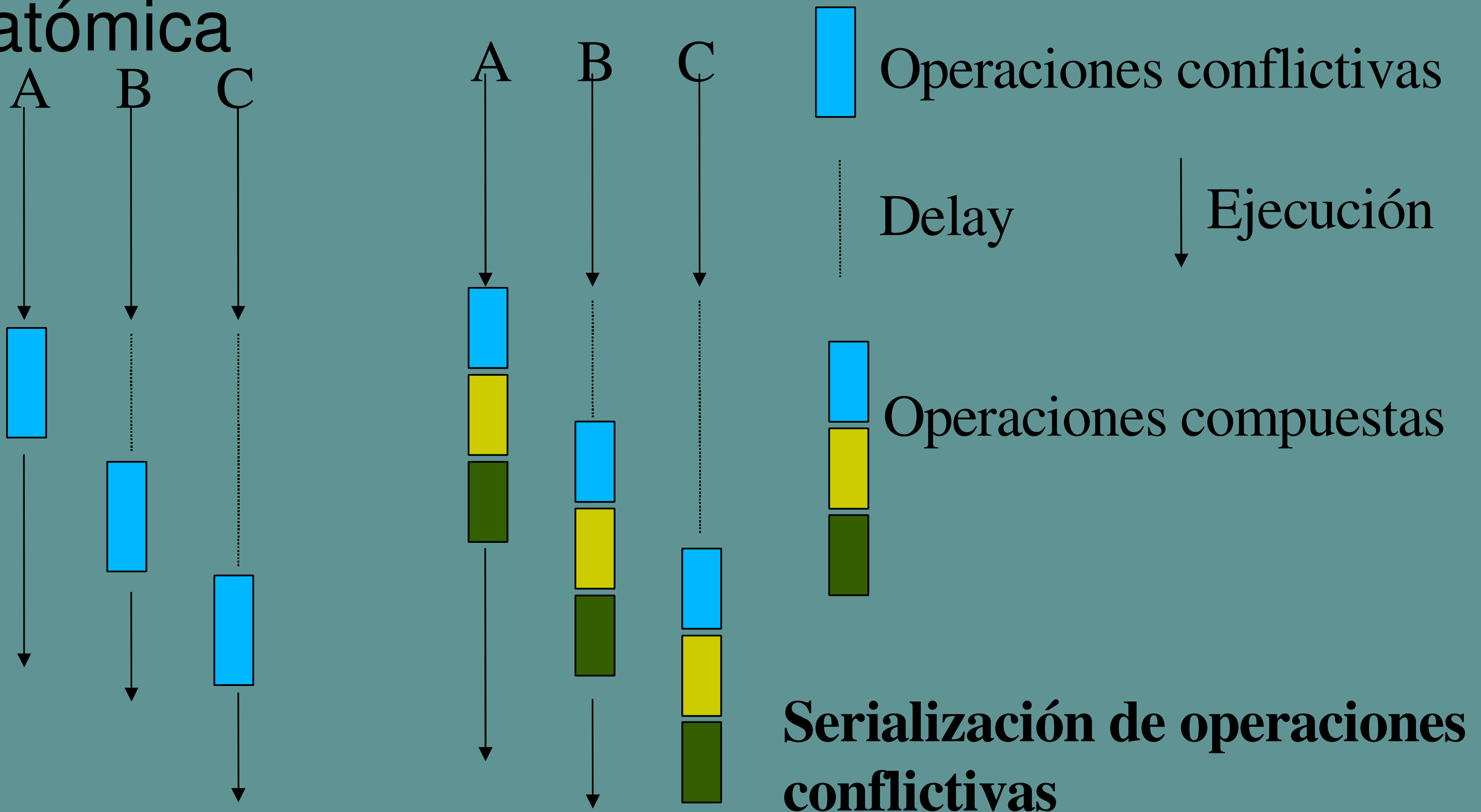
INDICE

- Introducción y definiciones
- Algoritmos de compromiso
  - Two Phase Commit
  - Three Phase Commit
- Algoritmos de control de concurrencia
  - Por bloqueo (*locking*)
  - Optimista
  - Por marcas de tiempo (timestamp)
- Tratamientos de interbloqueos

INTRODUCCIÓN Y DEFINICIONES

- Transacciones

Unidad de cálculo consistente, confiable y atómica



INTRODUCCIÓN Y DEFINICIONES

- Una transacción aplica a datos recuperables, puede estar formada por operaciones simples o compuestas y su intención es que sea atómica.
- Hay dos aspectos que se deben cumplir para lograr la atomicidad:

1. Todo-o-nada: si una transacción termina exitosamente, los efectos de todas sus operaciones son registrados en los ítems de datos.
   - Si falla no tiene ningún efecto.

INTRODUCCIÓN Y DEFINICIONES

- La propiedad *todo-o-nada* también considera:
  - Atomicidad ante fallas: los efectos son atómicos aún cuando el servidor falla.
  - Durabilidad: después que una transacción ha terminado exitosamente, todos sus efectos son salvados en almacenamiento permanente.

2. Aislamiento: cada transacción debe ser ejecutada sin interferencias de otras transacciones, es decir, los resultados intermedios de una transacción no deben ser visibles a otras transacciones.

- Estas propiedades tambien son conocidas como propiedades *ACID*

Propiedades ACID

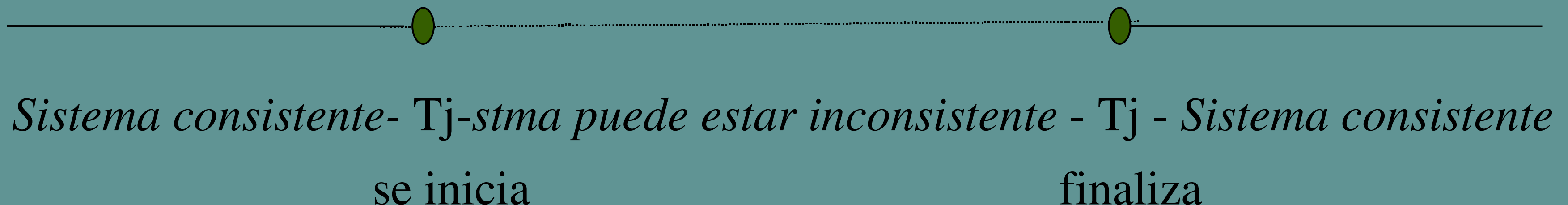
- *A*: Atomicidad (*atomicity*) - propiedad *todo-o-nada*.
  - Para soportar la *atomicidad ante fallas* y la *durabilidad*, los ítemes de datos deben ser recuperables.
  - Hay dos tipos de fallas:
    - Transacción falla por si misma por errores en los datos de entrada, deadlocks, etc. Recuperación de la transacción.
    - Fallas por caidas del sistema, de los medios de E/S, de los procesadores, de las líneas de comunicación, fallas de energía, etc. Recuperación de caidas.

Propiedades ACID

- *A*: Atomicidad (*atomicity*) - propiedad *todo-o-nada*.
  - En la recuperacion de caidas:
    - El sistema es quién tiene la responsabilidad de decidir qué hacer ante la recuperación de una falla:
    - Terminar de ejecutar el resto de las acciones.
    - Deshacer las acciones que se había realizado.
  - Para proveer la recuperación se usan técnicas de almacenamiento estable.

Propiedades ACID

- *C*: Consistencia (*consistency*) - una transacción toma el sistema en un estado consistente y lo deja en un estado consistente.



- *I*: Aislamiento (Isolation): implica *seriabilidad de las transacciones*. Se les permite a las transacciones ejecutarse concurrentemente si se obtiene el mismo efecto de una ejecución secuencial (*serialmente equivalentes*).
- *D*: Durabilidad (durability).

Primitivas sobre transacciones

- *begin_transaction* ⟶ *tid* : inicia una transacción y devuelve un identificador de la transacción
- *close_transaction(tid)* ⟶ *(Commit, Abort)*
  - *Commit:* la transacción termina exitosamente y sus efectos van a almacenamiento permanente.
  - *Abort*: no se reflejan los cambios. Los abortos pueden ser causados por la propia naturaleza de la transacción, por conflictos con otras transacciones o por fallas.
- *abort_transaction(tid)*: aborto intencional.
- *read y write*: típicamente una transacción se compone de una serie de lecturas y escrituras y algunos cálculos.

Transacciones anidadas

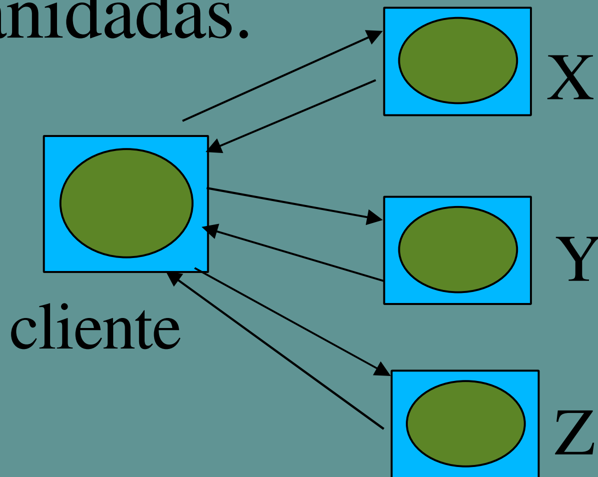
- Las transacciones pueden contener subtransacciones.
  - Problema: Si una transacción interna realiza *commit* y una más externa *abort*, se pierden las propiedades de atomicidad y aislamiento por cumplir con la durabilidad.
  - Generalmente la *durabilidad* sólo se considera para la transacción más externa.
  - En algunaos sistemas, la transacción superior puede decidir hacer commit aún cuando alguna subtransacción aborta.

Implementación de las transacciones

- Espacio de trabajo privado:
  - Se copian los datos en un espacio propio de cada transacción
  - Al finalizar exitosamente la transacción se actualizan en la base de datos
- Lista de intención (writeahead log)
  - Las actualizaciones son realizadas directamente en la base de datos
  - Se lleva un registro de los cambios realizados

Transacciones Distribuidas

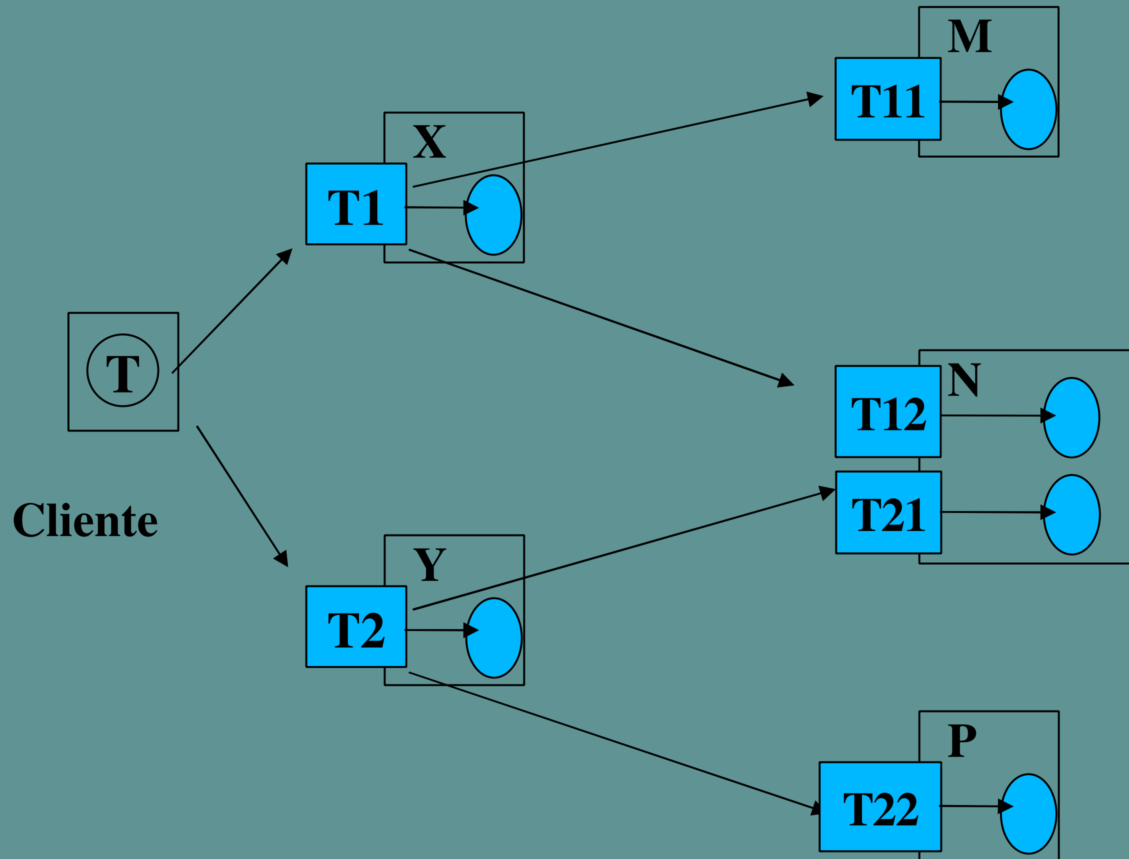
- Sus actividades envuelven múltiples servidores.
- Los ítemes de datos de un servidor pueden estar distribuidos entre varios servidores y, en general, una transacción de un cliente puede envolver múltiples servidores.
- Las transacciones distribuidas pueden ser simples o anidadas.



Cliente:

```
begin_transaction
    call X.x
    call Y.y
    call Z.z
end_transaction
```

Transacciones Distribuidas

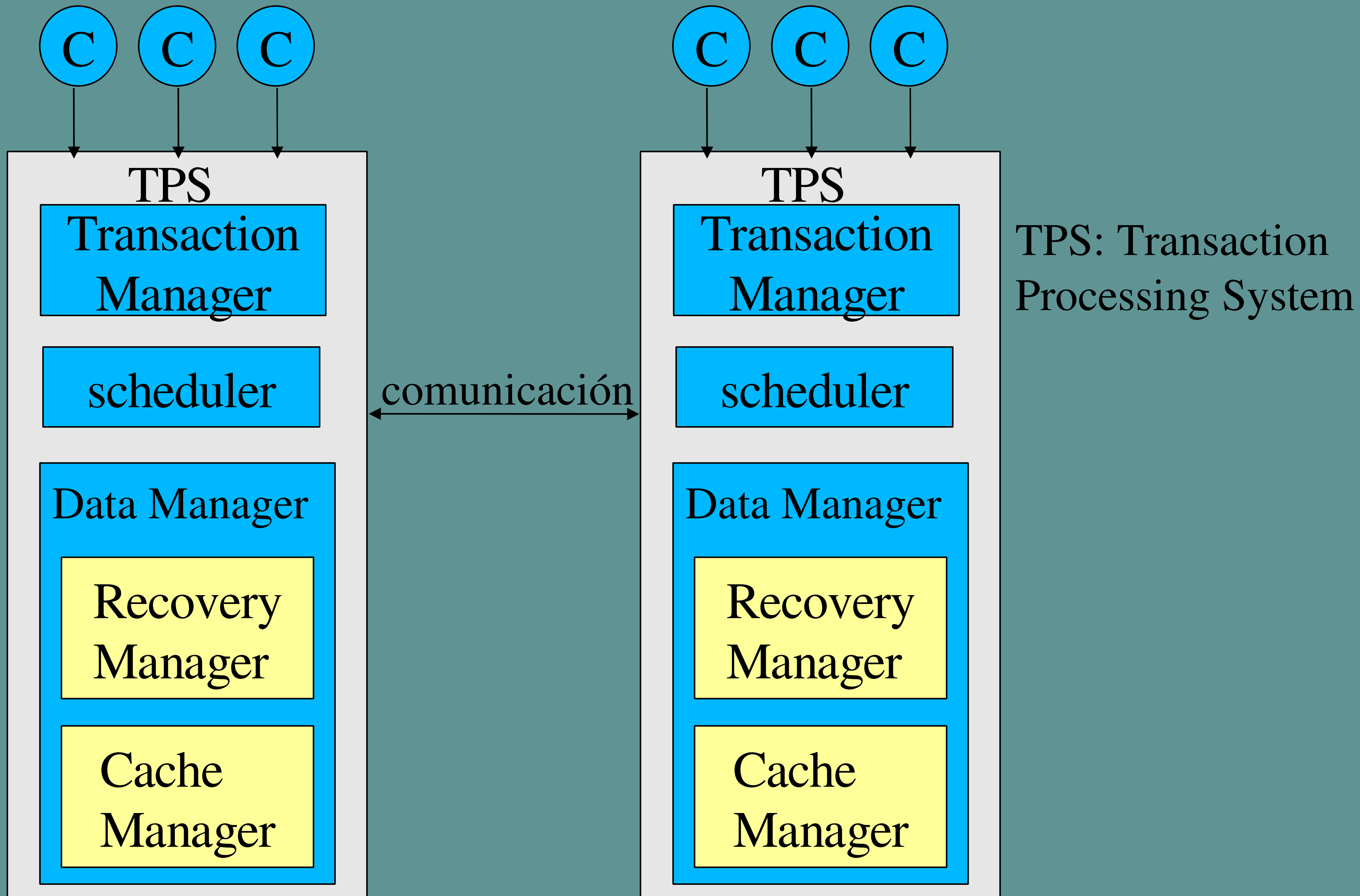


Transasccción distribuida anidada

Transacciones Distribuidas

- Cuando una transacción distribuida termina, la propiedad de atomicidad exige que todos los servidores acuerden lo mismo (*commit*) o todos aborten (*abort*). Existen **protocolos para llegar a compromisos** (Two-Phase-Commit y Three-Phase_Commit)
- Las transacciones distribuidas deben ser globalmente serializadas. Existen protocolos de **control de concurrencia distribuida**.

Procesamiento de transacciones distribuidas



Procesamiento de transacciones distribuidas

- Un cliente inicia una transacción (*begin_transaction*) sobre un TPS. El *Transaction Manager* identifica y localiza los objetos invocados por la transacción.
- Las invocaciones a los objetos locales son pasados al *Scheduler* local, las invocaciones a objetos remotos son pasados a TPS remotos correspondientes.
- Observaciones:
  - Un cliente inicia una transacción (*begin_transaction*) en un único nodo ==> Nodo coordinador.
  - Un objeto reside en único nodo (no hay replicación de objetos). La invocación de tal objeto toma lugar en ese nodo.

Procesamiento de transacciones distribuidas

- Observaciones (cont.):
  - Existen mecanismos para localizar un objeto, dado su identificador único.
  - Las instancias de TPS deben cooperar
- Coordinador de una transacción distribuida
  - Un cliente comienza una transacción enviando un *begin_transaction* a cualquier servidor TPS. Éste se convierte en el coordinador y los que se tengan que contactar a partir de aquí se convierten en trabajadores.

Procesamiento de transacciones distribuidas

- Coordinador de una transacción distribuida (cont.)
  - Para esto se requieren otras primitivas:

  AddServer (tid, server_id del coordinador):

  Es enviado por el coordinador a otro servidor informándole que está envuelto en la transacción *tid*.

  NewServer(tid, server_id del trabajador):

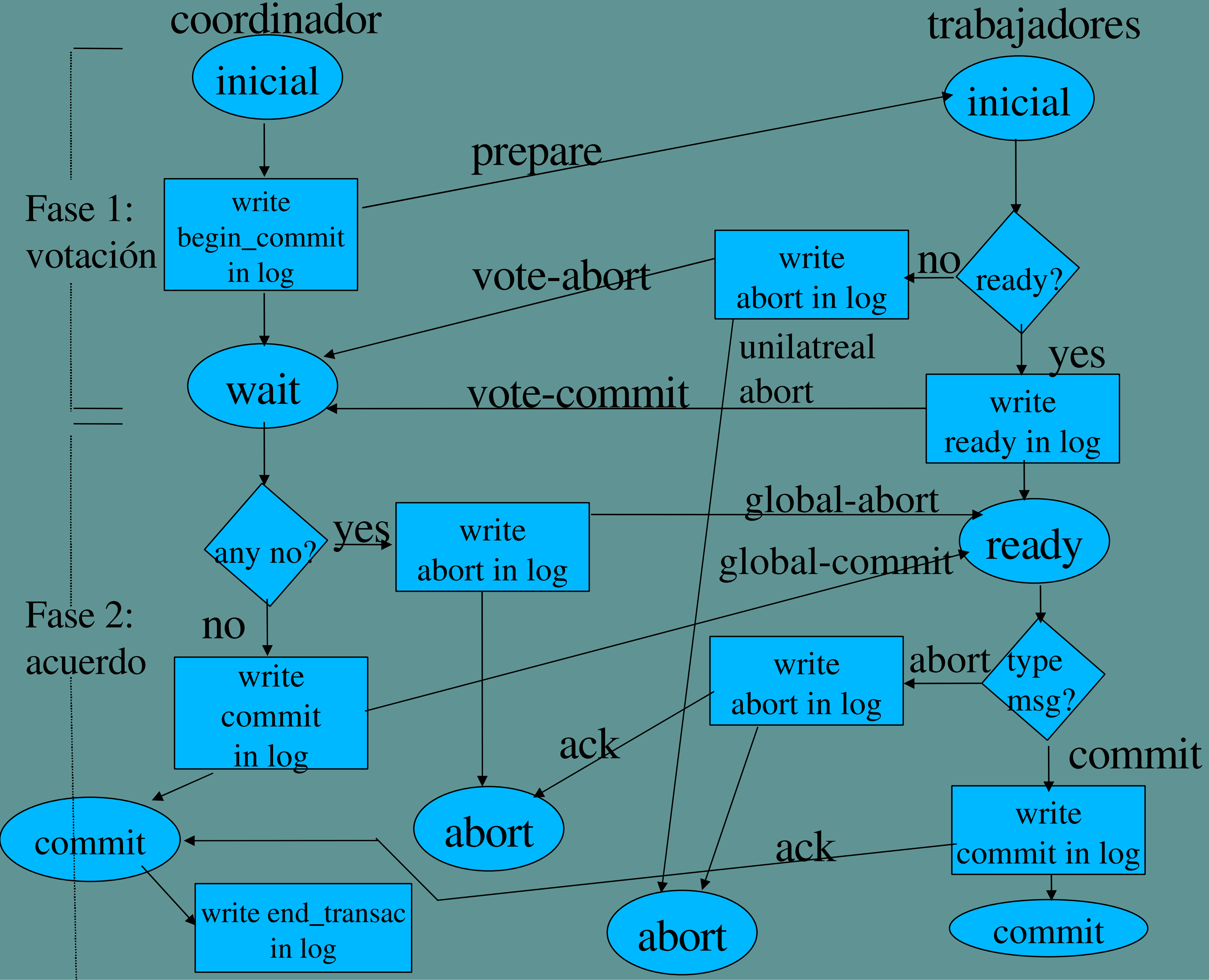
  Es la respuesta ante un *AddServer* de un trbajador al coordinador. El coordinador lo registra en su lista de trabajadores.

Algoritmos de compromiso

- Cuando el coordinador recibe un requerimiento *Commit* de una transación, tiene que asegurar:
  - *Atomicidad:* Todos los nodos se comprometen con los cambios o ninguno lo hace y cualquier otra transacción percibe los cambios en todos los nodos o en ninguno.
  - *Aislamiento:*Los efectos de la transacción no son visibles hasta que todos los nodos hayan tomado la decisión irrevocable *commit* o *abort*.

Algoritmos de compromiso

- El protocolo Two-Ohase Commit (TPC):
  - Durante el progreso de una transacción no hay comunicación entre el coordinador y los trabajadores, solo con *AddServer* y *NewServer*.
  - El requerimiento *commit* o *abort* del cliente, llega al coordinador.
  - Si es *abort*, el coordinador se lo informa inmediatamente a todos los trabajadores.
  - Si es *commit*, se aplica el protocolo TPC.


coordinador
trabajadores
inicial
inicial
prepare
write
begin_commit
in log
Fase 1:
votación
vote-abort
write
abort in log
no
ready?
yes
unilatreal
abort
wait
vote-commit
write
ready in log
global-abort
ready
any no?
yes
write
abort in log
global-commit
Fase 2:
acuerdo
no
write
commit
in log
write
abort in log
abort
type
msg?
commit
ack
commit
abort
write
commit in log
ack
write end_transac
in log
abort
commit

Algoritmos de compromiso

- ¿Cuáles serán las acciones del protocolo TPC ante timeouts?

Timers en los estados del coordinador: *Wait, Commit o Abort*.

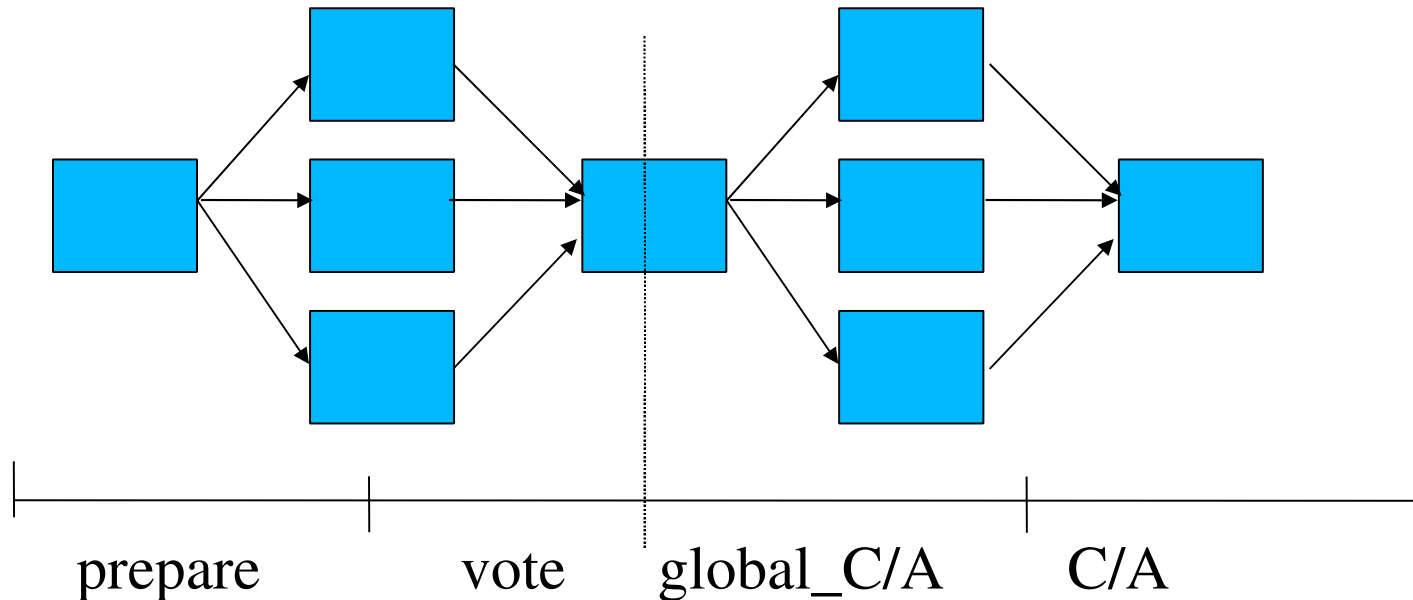
- Timeout en el estado *Wait*: Decide abortar la transacción, lo escribe en el log y envía el mensaje *global_abort* a los trabajadores.
- Timeout en los estados *commit* o *abort*: Envía el mensaje *global_commit* o *global_abort* respectivamente a los trabajadores que aún no han respondido y espera por sus ack.

Timers en los estados de los trabajadores: *Inicial o Ready*.

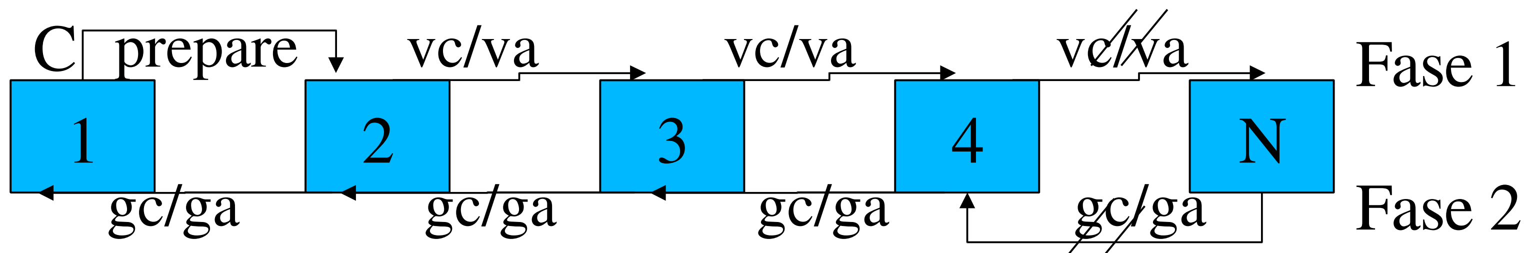
- Timeout en el estado *Inicial*: Decide abortar unilatealmente. Si el mensaje *prepare* llega después, el trabajador puede enviar un *vote_abort* o ignorarlo.
- Timeout en el estado *Ready*: Se debe quedar bloqueado esperando alguna noticia

Paradigmas de comunicación para el TPC

- TPC centralizado



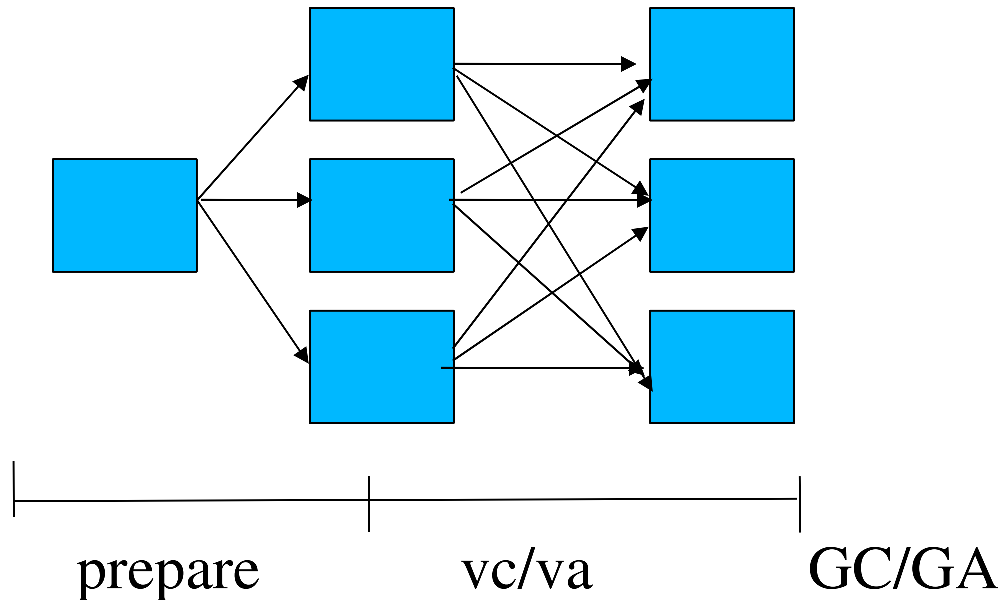
- TPC lineal



Menos mensajes, pero poco paralelismo

Paradigmas de comunicación para el TPC

- TPC distribuido



  - El GC/GA es una decisión de cada participante de acuerdo a los votos recibidos
  - Se elimina la fase 2
  - Lineal y distribuidos requieren conocer los Ids de todos los participantes.

Control de Concurrencia

- La idea es resolver las operaciones conflictivas

- *Operaciones conflictivas:* 2 operaciones son conflictivas cuando sus efectos combinados dependen del orden en el cual fueron ejecutadas.

- Para dos transacciones A y B, se consideran conflictivas las siguientes operaciones:

A	B	
read	read	no conflictivas
read	write	conflictivas
write	write	conflictivas

- Cuando dos o más transacciones son conflictivas es necesario su serialización para asegurar la consistencia de los datos después de su ejecución.

Control de Concurrencia

- Las *Operaciones conflictivas* derivan en dos problemas:
  - Actualizaciones perdidas

Transacción T	Transacción U
balance=b.getBalance()	balance=b.getBalance()
b.setBalance(balance*1.1)	b.setBalance(balance*1.1)

Ejecución

T	U
balance=200$	
	balance=200$
	balance=220$
balance=220$	

(balance debió ser 200+20+ 22)

  - Recuperaciones inconsistentes

Transacción V	Transacción W
a.retirar(100)	unasucursal.totalSucursal(a,b)
b.deposita(100))	

Ejecución; a=200$, b=200$

V	W
a=200-100	
	lee a=100$
	lee b=200$
b=200$+100$	
	total=300$

Según la ejecución el total es 300$ y debió ser 400$

Ambos problemas se resuelven definiendo ***Equivalencia Secuencial ==> Control de concurrencia***

Control de Concurrencia

- Las transacciones pueden abortar, ante esta situación surgen otros problemas: lecturas sucias y escrituras prematuras

  - Lecturas Sucias

Transacción T	Transacción U
a.getBalance() (100$)	
a.depositar(10) (110$)	
	a.getBalance() (110$)
	a.deposita(20) (130$)
	commit
aborta	

U tomó el valor 110$ que ahora no es válido.

- La estrategia para la recuperación es retrasar la acción de commit de U hasta que T finalice
- Esto conlleva a la posibilidad de **Abortos en Cascada** (si T aborta, U debe abortar también)

Control de Concurrencia

- Escrituras prematuras (perdida de actualizaciones)

  - La estrategia para la recuperación es retrasar los *writes* hasta el momento del commit

- Para evitar ambos problemas se debe proveer ejecución estricta de las transacciones (propiedad de aislamiento)

Algoritmo de *locking* o bloqueo

- Nivel de granularidad del bloqueo: tiene que ver con el tamaño del objeto o dato que se está bloqueando
- A mayor granularidad (mayor fineza del grano), más pequeño es el tamaño del objeto.
- El nivel del bloqueo es directamente proporcional al grado de paralelismo y concurrencia, pero también es directamente proporcional al grado de complejidad de los sistemas
- Mientras mayor sea la fineza del grano, mejor será el grado de paralelismo/concurrencia, pero mayor será la complejidad del sistema.
- El bloqueo puede ser a nivel de item, página, archivo, base de datos (donde item representa el grano más fino y base de datos corresponde al grano más grueso)

Algoritmo de *locking* o bloqueo

- Consiste en que cada vez que un proceso necesita leer o escribir en un objeto como parte de una transacción, el objeto se bloquea hasta que la transacción culmine exitosamente (commit) y cualquier otra transacción que desee hacer alguna operación sobre dicho objeto tendrá que esperar hasta que él sea desbloqueado.
- Los *locks* son adquiridos y liberados por el administrador de transacciones, esto implica que todo lo concerniente al control de concurrencia es transparente para el programador.
- El administrador de *locks* puede ser centralizado o local para cada máquina

Algoritmo de *locking* o bloqueo

lock otorgado	lock solicitado
Ninguno	read OK - write OK
read	read OK - write Espera
write	read Espera - write Espera

- Una mejora: utilizar *locks* de escritura y *locks* de lectura para ofrecer un mejor paralelismo al permitir que se realicen concurrentemente transacciones que hagan operaciones no conflictivas.
- Otra mejora: promoción de *locks*, si varias transacciones necesitan un objeto para lectura y luego para escritura, se les puede otorgar un *lock* de lectura hasta que alguna necesite escribir en el objeto. Se le otorgará el *lock* de escritura después de que todas las demás transacciones que tengan *locks* de lectura sobre el mismo objeto, lo liberen. La ventaja de esta mejora es que provee un mayor grado de paralelismo.

Algoritmo de *locking* o bloqueo

- Resuelve “recuperaciones inconsistentes”

  No hay posibilidad de que dos operaciones conflictivas se ejecuten concurrentemente

- Resuelve “pérdida de actualizaciones”

  Si dos transacciones leen el mismo dato y luego lo modifican, la 2da. espera (ya sea por promoción o por no otorgamiento)

- El problema del algoritmo de *locking* es que puede ocasionar *deadlocks* y abortos en cascada, por lo que se han propuesto algunas variaciones para evitar tales problema.

Algoritmo de *locking* o bloqueo

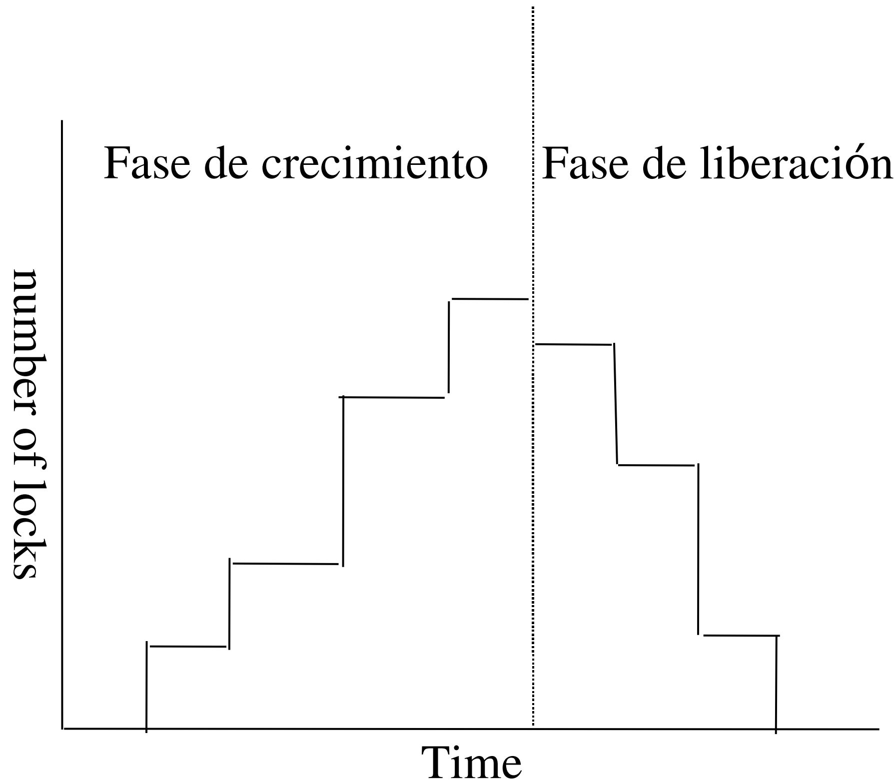
- Two Phase Locking: “obtención” y “liberación”
  - Durante la fase de “obtención”, la transacción trata de obtener todos los *locks* que necesite. Si no es posible obtener alguno, entonces espera.
  - La segunda fase comienza cuando la transacción libera alguno de los *locks*, a partir de ese momento no podrá solicitar ningún otro *lock* (si lo hace, será abortada).
  - Desventaja: si una transacción en la fase de liberación había desbloqueado algunos objetos y los mismos habían sido accedidos por otras transacciones antes de que la primera hiciera commit, entonces las demás transacciones deberían abortar (esto es abortos en cascada).

Algoritmo de *locking* o bloqueo

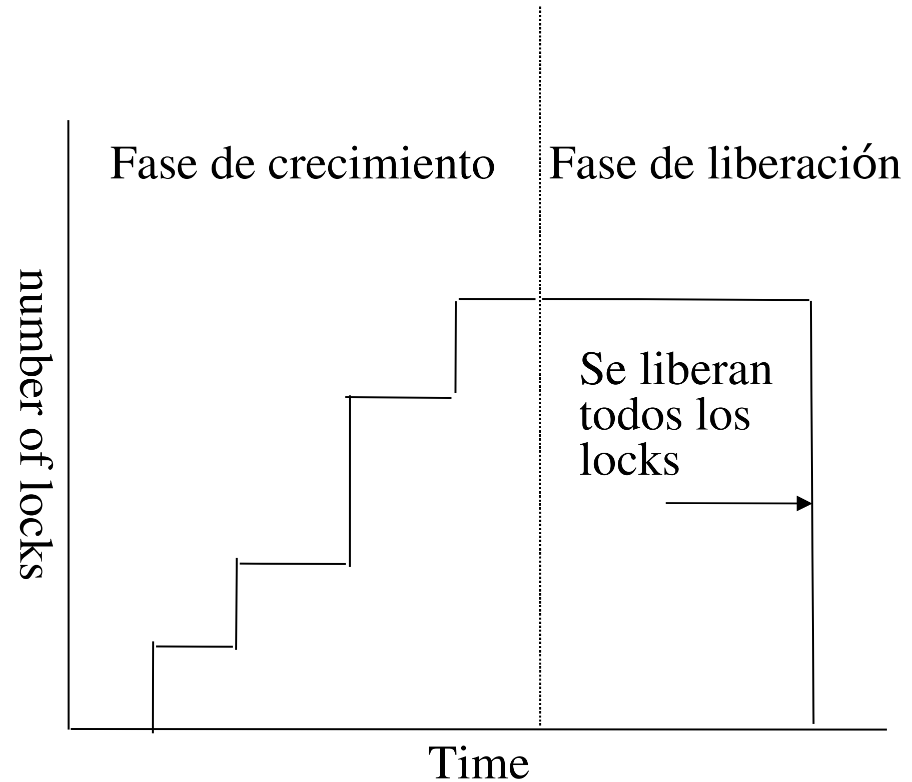
- **Strict Two Phase Locking**:
  - La fase de "liberación" se realiza sólo cuando la transacción hace commit
  - La mejora: evita los abortos en cascada
  - Desventajas:
    - El nivel de paralelismo se degrada
    - Permanece la posibilidad de *deadlock*
    - Aún representa un alto costo de mantenimiento

Algoritmo de *locking* o bloqueo

Two Phase Locking



Strict Two Phase Locking



Algoritmo Optimista

- Se basa en las siguientes premisas:
  - "Los conflictos suceden poco"
  - "Como vaya viniendo vamos viendo"
  - "¡Adelante!, haz lo que quieras sin atender lo que los otros hacen, no te preocupes por los problemas ahora, preocúpate más tarde"

- Las modificaciones/accesos se hacen sobre **espacios privados** y se lleva registro de los datos que han sido modificados/accedidos. Al momento del commit, se chequea que los espacios privados sean válidos, de no serlos, se aborta la transacción.

- A toda transacción se le asigna un identificador (orden secuencial ascendente) para llevar una sucesión de transacciones en el tiempo.

Algoritmo Optimista

- Cada transacción cumple tres fases:
  - Trabajo:Todos los *reads* se ejecutan inmediatamente sobre la última versión "comprometida" del dato. Los *writes* crean versiones tentativas. Se mantiene un conjunto de lectura (datos leídos) y un conjunto de escritura (versiones tentativas de los datos).

    No hay posibilidad de "lecturas sucias", ¿por qué?
  - Validación: Ante la solicitud de un commit, se valida si la transacción realizó operaciones conflictivas con otras transacciones.
  - Escritura: Si la transacción es validada, todos los cambios hechos sobre los **espacios privados** son actualizados en las versiones originales.

Algoritmo Optimista

- **Fase de validación**:
  - Ante el *close_transaction,* a cada transacción se le asigna un número (secuencial ascendente, i) que define su posición en el tiempo.
  - La validación se basa en las siguientes reglas (i < j):

Ti	Tj	Regla
read	write	Ti no debe leer datos escritos por Tj
write	read	Tj no debe leer datos escritos por Ti
write	write	Ti no debe escribir datos escritos por Tj y Tj no debe escribir datos escritos por Ti

  - Simplificación: fases de validación y escritura son secciones críticas, entonces se satisface la regla 3. Sólo hay que validar las reglas 1 y 2

Algoritmo Optimista

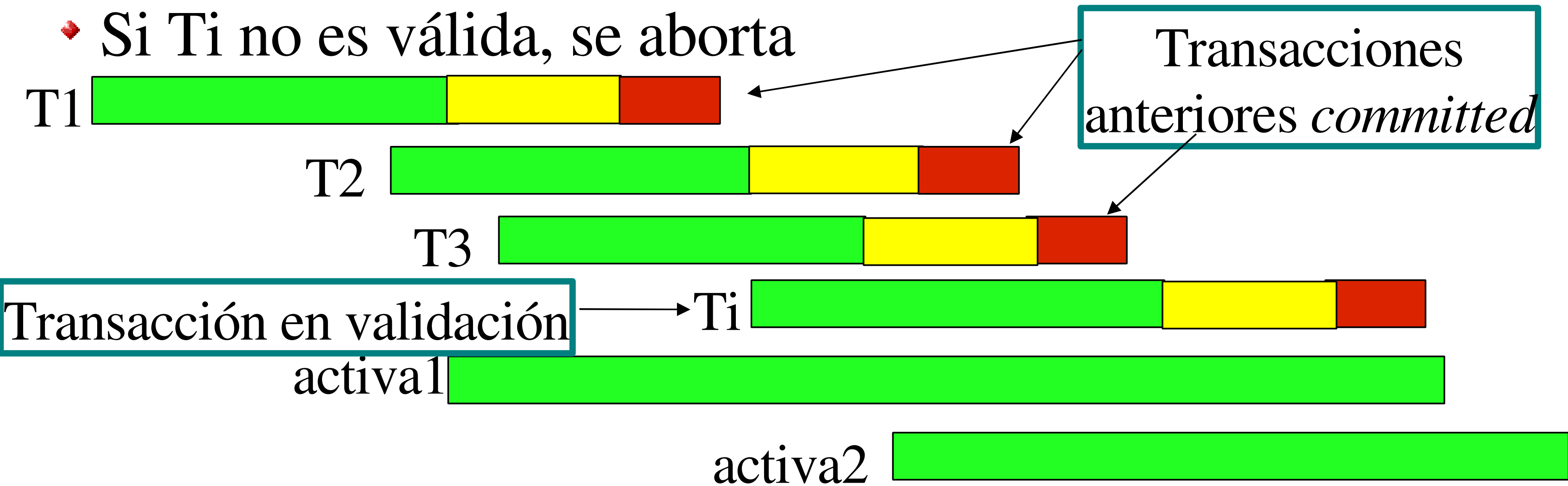
- **Validación hacia atrás**:
  - Los *reads* de las Tj se realizaron antes que la validación de Ti, entonces se cumple la regla 1.
  - Sólo se valida la regla 2 para cada Tj:

```
valid= true;
for (Tj=startTn+1;Tj<=finishTn,Tj++) {
  if ("read_set" of Ti intersects "write_set" Tj)
    valid=false;
}
```

  - startTn: Tj más grande asignado a una transacción *committed* al momento que Ti entra a su fase de trabajo
  - finishTn: Tj más grande asignado al momento que Ti entra a su fase de validación

Algoritmo Optimista

- **Validación hacia atrás**:
  - Sólo es necesario validar los conjuntos de lectura. Las transacciones que sólo hacen escritura no se validan.
  - Si Ti no es válida, se aborta



Algoritmo Optimista

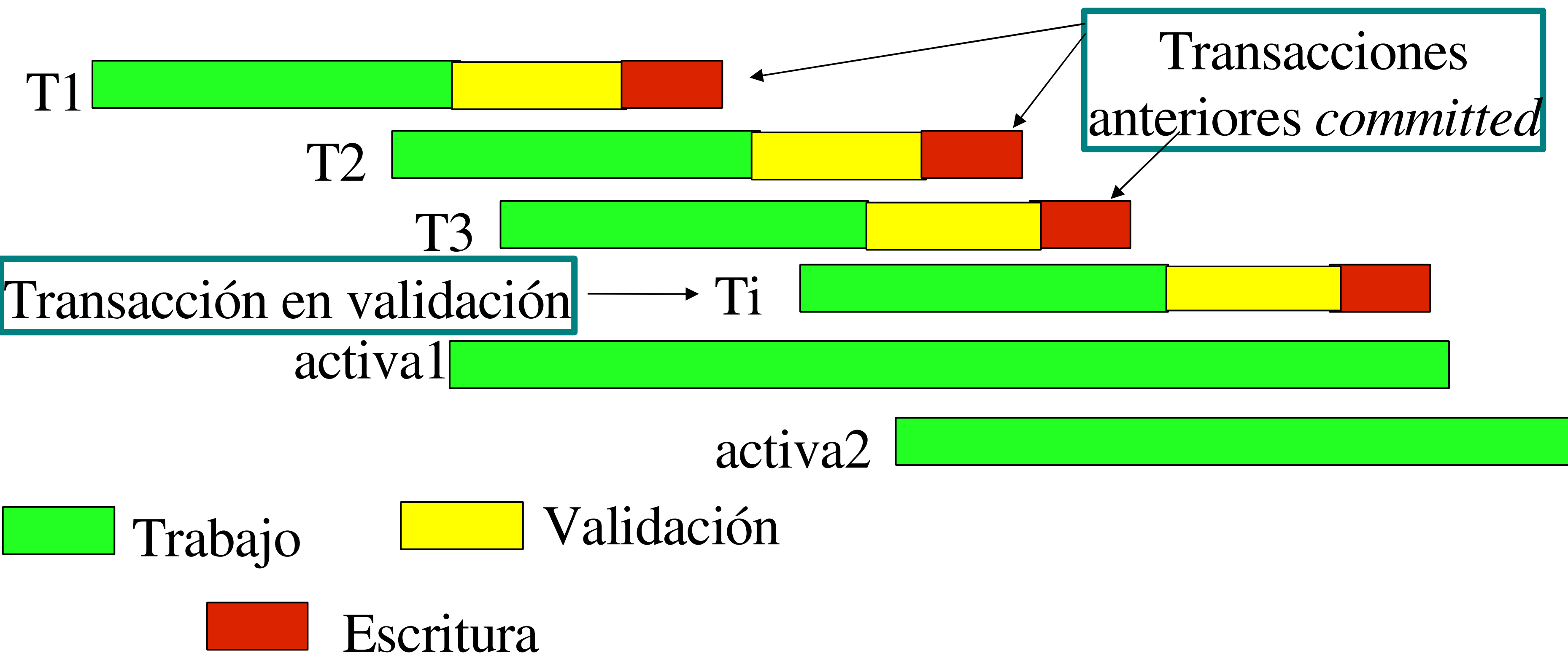
- **Validación hacia delante**:
  - Se satisface la regla 2 porque las transacciones activas no escriben mientras que Ti no se ha completado.
  - Sólo se valida la regla 1 para cada Tid:

```
valid= true;
for (Tid=activa1;Tid<=activaN,Tid++) {
  if (“write_set” of Ti intersects “read_set” Tid)
    valid=false;
}
```

  - activaX: Representan transacciones que aún no han entrado a la fase de validación
  - Las transacciones que sólo hacen lecturas no requieren ser validadas

Algoritmo Optimista

- **Validación hacia adelante**:
  - Si Ti no es válida:
    - Aplazar la validación (¿le irá mejor en el futuro?)
    - Abortar las activas y consumar Ti
    - Abortar Ti (¿qué pasa si alguna de las futuras Tj es abortada?



Algoritmo Optimista

- **Desventajas:**
  - Hay posibilidad se inanición: una transacción puede abortar indefinidas veces y no se contempla mecanismo para evitar esto.
  - También es importante saber que este algoritmo no serviría para nada en sistema con carga alta.
  - Otra desventaja es que este algoritmo produce mucha sobrecarga porque hay que mantener los conjuntos de escritura de transacciones que ya terminaron (hacia atrás)

Algoritmo por Marcas de Tiempo

- Las operaciones se validan al momento de ser ejecutadas
- Cuando una transacción comienza, se le asigna un *timestamp*
- Se trabaja con versiones tentativas
- Cada item de dato tiene asociado:

  Un *timestamp* de escritura (Twrite_commit), un *timestamp* de lectura (Tread) y un conjunto de versiones tentativas con su propio *timestamp*

- Un *write* aceptado genera una versión tentativa
- Un *read* se dirige a la versión con el máximo *timestamp* menor que el *timestamp* de la transacción

Algoritmo por Marcas de Tiempo

- La validación se basa en las siguientes reglas:

Regla	Tj	Ti	Condición
1	write	read	Tj no debe escribir un dato leído por Ti>Tj (requiere que Tj >= max(Tread) del dato)
2	write	write	Tj no debe escribir un dato escrito por Ti>Tj (requiere que Tj > max(Twrite_commit) del dato)
3	read	write	Tj no debe leer un dato escrito por Ti>Tj (requiere que Tj > Twrite_commit)

Algoritmo por Marcas de Tiempo

- Para saber cuando un *write* es válido, se aplica el siguiente algoritmo (validación de las reglas 1 y 2- regla de escritura):

Sea Tj una transacción que desea hacer un *write* sobre el objeto D.

If ((Tj >= Max (Tread en D)) &&

(Tj > write_commit en D))

Proceder con el *write* sobre una versión tentativa nueva;

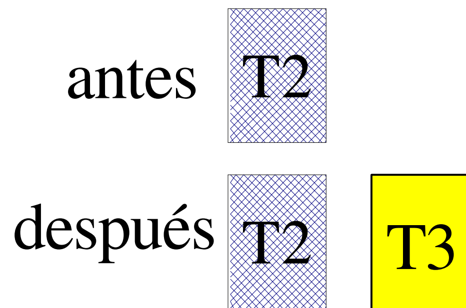
else // *write is too late*

Abortar Tj;

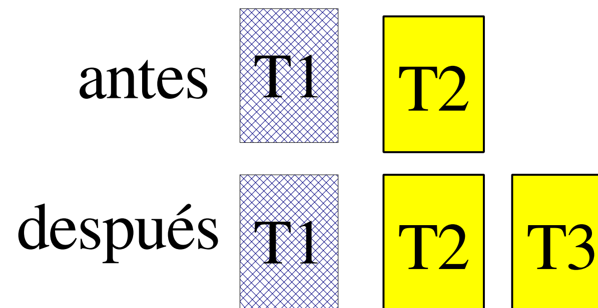
Algoritmo por Marcas de Tiempo

- Regla de escritura

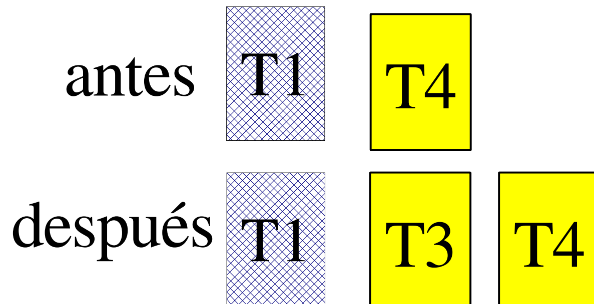
a) T3->*write*



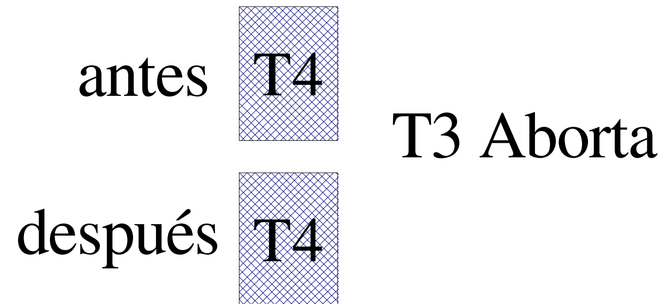
b) T3-> *write*

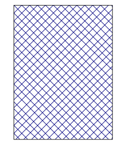


c) T3->*write*

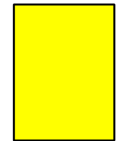


d) T3-> *write*





Versión committ



Versión tentativa

Algoritmo por Marcas de Tiempo

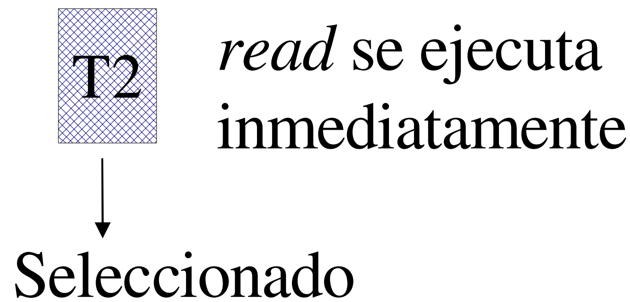
- **V**alidación de la regla 3 (regla de lectura): Tj hace *read*(D)

```
if (Tj > Twrite_commit en D) {
     Dselected with Max (Twrite <=Tj en D);
 if (esCommit (Dselected)) {
     Procede el read sobre Dselected;
  } else {
         Esperar hasta que la Ti que hizo la versión
        tentativa de Dselected haga commit o abort;
        volver a comenzar;
 } else {
   Abortar Tj;
}
```

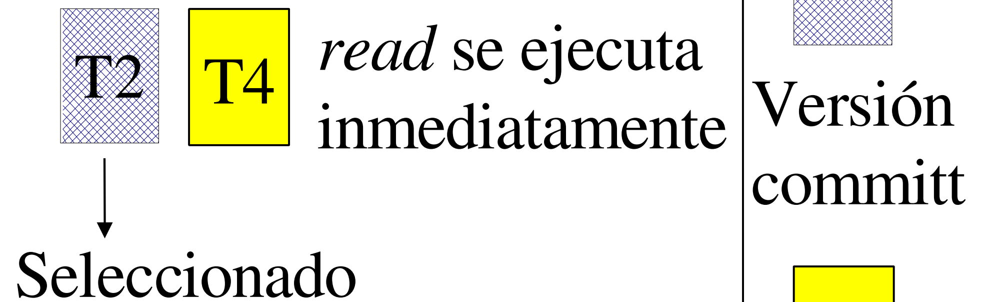
Algoritmo por Marcas de Tiempo

- Regla de lectura

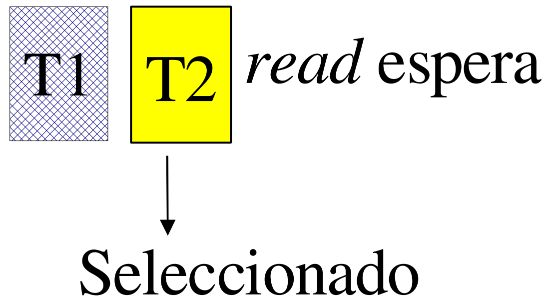
a) T3->*read*



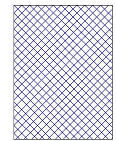
b) T3-> *read*



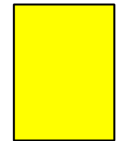
c) T3->*read*



d) T3-> *read*



Versión committ



Versión tentativa

Tratamiento de Interbloqueos

- Condiciones para un bloqueo:

1.- Condición de exclusión mutua. Cada recurso está asignado a un único proceso o está disponible.

2.- Condición de posesión y espera.Los procesos que tienen, en un momento dado, recursos asignados con anterioridad, pueden solicitar nuevos recursos.

3.- Condición de no apropiación. Los recursos otorgados con anterioridad no pueden ser forzados a dejar un proceso. El proceso que los posee debe liberarlos en forma explícita.

4.- Condición de espera circular.Debe existir una cadena circular de dos o más procesos , cada uno de los cuales espera un recurso poseído por el siguiente miembro de la cadena.

Tratamiento de Interbloqueos

- Políticas frente a los bloqueos:

  1.- Ignorar:el tratamiento del deadlock es responsabilidad del programador y/o de las acciones.

  2.- Detectar:dejar que suceda y luego recuperarse.

  3.- Prevenir:Evitar que estructuralmente sea posible el deadlock, es decir, asegurar que al menos una de las cuatro condiciones no se cumpla.

  4.- Algoritmo del Banquero:Se necesita conocer los requerimientos de recursos del proceso. (No es aplicable en sistemas distribuidos por su complejidad de conocer los requerimientos de recursos de los procesos con anterioridad).

Tratamiento de Interbloqueos

- Se distinguen los siguientes tipos de Deadlocks:

1.- Deadlock de Comunicación:

A	B	C
send(B)	send(C)	send(A)
recv(C)	recv(A)	recv(B)

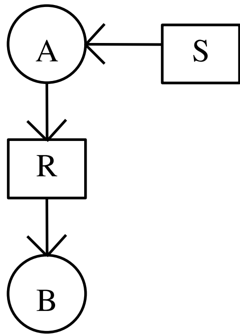
- Si el send es bloqueante entonces estamos en presencia de deadlock.
- Este tipo de deadlock es más frecuente en Sistemas Paralelos, pues la filosofía de comunicación es de todos con todos, mientras que en el caso de sistemas distribuidos con la filosofía de cliente-servidor, es menos probable, aun cuando es posible.

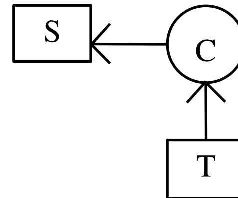
2.- Deadlock de Recursos: Es más frecuente en sistemas distribuidos.

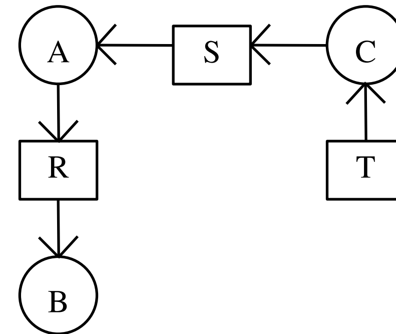
Algoritmos de Detección

1.- Centralizado: basado en grafos de espera

Máquina 0 **Máquina 1** **Coordinador**







Cómo se mantiene el grafo en el coordinador ?

- Cada vez que ocurra una variación en su grafo notifica al coordinador .
- Periodicamente cada máquina notifica sus últimos cambios .
- Periodicamente el coordinador solicita la información.

Problema: Los 3 casos pueden conducir a un Deadlock falso.

Ejemplo: Si se pierden mensajes

Si B solicita a T y C libera a T y llega primero el mensaje de B al coordinador, entonces el cree que hay deadlock.

Algoritmos de Detección

2.- Distribuido: basado en grafo de recursos y procesos.

- Asuma comunicación confiable.
- A los procesos (o transacciones) se les permite pedir varios recursos a la vez.
  - Cuando un proceso tiene que esperar un recurso ocupado:

1.- Envía un mensaje "Prueba" al que tiene el recurso

El mensaje consiste en:

.- Número del proceso que tiene que esperar.

.- Número del proceso que envía el mensaje.

.- Número del proceso que recibe el mensaje.

2.- Cuando un proceso recibe un mensaje de Prueba

Si él espera por otro recurso

Si recibe (0,X,0) entonces hay Deadlock.

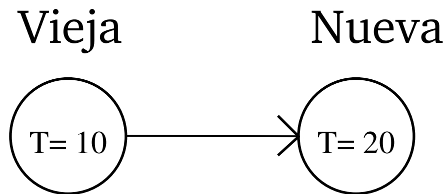
- Recuperación: Eliminar una transacción, el proceso decide terminarse (Suicidio: puede inducir a suicidios colectivos innecesarios), seleccionar victima.
- ¿Puede suceder deadlock falso?

Algoritmos de Prevención

- Se basan en asignar a cada transacción un timestamp: Si una transacción requiere un recurso que otra transacción tiene se chequean los timestamp, y se toma una acción dependiendo de la política seleccionada.

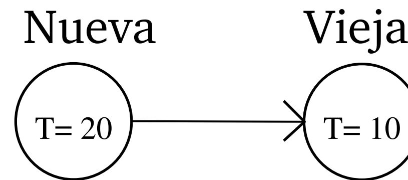
1.- Wait-Die:

Vieja Nueva



Espera

Nueva Vieja

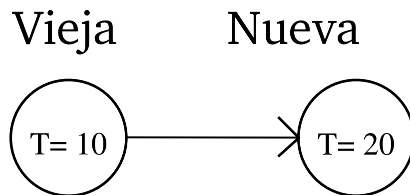


Muere

Problema: Puede ocurrir inanición

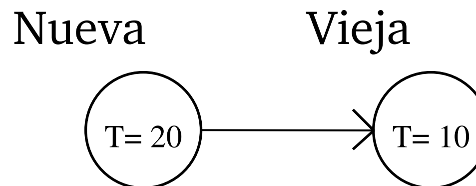
2.- Wound-Wait:

Vieja Nueva



Muere

Nueva Vieja



Espera

¿Qué sucedería si las transacciones nuevas tienen la prioridad?
No es justo.