SISTEMAS DISTRIBUIDOS

TEMA 6: TRANSACCIONES Y CONTROL DE CONCURRENCIA

vida laboral

Valentín Valero

Departamento de Sistemas Informáticos
Universidad de Castilla-La Mancha

valentin.valero@uclm.es
memilia.cambronero@uclm.es
vicente.sahori@uclm.es

Tercer Curso del Grado en Informática

CAP 16 : LIBRO

INDICE:

TRANSACCIONES

Definición (TRANSACCIÓN)

Una transacción es una serie de operaciones sobre uno o varios recursos para los cuales se garantiza la atomicidad en presencia de múltiples clientes y en presencia de fallos.

o se hace todo
o da fallo

Los efectos de una transacción no son visibles para los otros clientes de los recursos hasta que ha terminado y sus efectos han sido confirmados.

Soporte Java: ▸ Transaction API (JTA) [Glassfish]

TRANSACCIONES

PROPIEDADES DE LAS TRANSACCIONES (ACID):

- *Atomicidad*: Sus operaciones se realizan por completo o bien no se realizan (todo o nada).
- *Consistencia*: La ejecución entremezclada de las transacciones debe conducir el sistema a través de valores consistentes en los recursos u objetos afectados.
- *Independencia*: Su ejecución debe realizarse autónomamente del efecto parcial del resto de transacciones. Los efectos de una transacción sólo son visibles para las demás una vez confirmada.
- *Durabilidad*: Los efectos de la transacción deben ser permanentes, incluso ante fallos en el sistema.

TRANSACCIONES

¿Cómo garantizar estas propiedades?

- Mediante técnicas de control de concurrencia puede obtenerse la independencia y consistencia: *exclusión mutua*.
- Pero la exclusión mutua es muy restrictiva, más aun al afectar a una colección de recursos que deben ser retenidos por un tiempo indefinido.
- Además no garantiza la atomicidad: todo o nada en una serie de operaciones.
- Habrá que introducir mecanismos de regulación de la concurrencia más permisivos.

TRANSACCIONES

¿Cómo garantizar estas propiedades?

- Mediante técnicas de control de concurrencia puede obtenerse la independencia y consistencia: *exclusión mutua*.
- Pero la exclusión mutua es muy restrictiva, más aun al afectar a una colección de recursos que deben ser retenidos por un tiempo indefinido.
- Además no garantiza la atomicidad: todo o nada en una serie de operaciones.
- Habrá que introducir mecanismos de regulación de la concurrencia más permisivos.

TRANSACCIONES

¿Cómo garantizar estas propiedades?

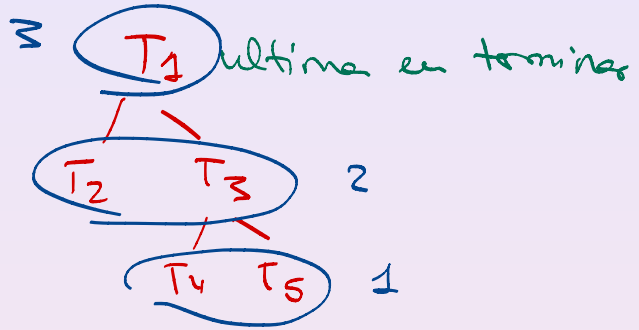
- Mediante técnicas de control de concurrencia puede obtenerse la independencia y consistencia: *exclusión mutua*.
- Pero la exclusión mutua es muy restrictiva, más aun al afectar a una colección de recursos que deben ser retenidos por un tiempo indefinido.
- Además no garantiza la atomicidad: todo o nada en una serie de operaciones.
- Habrá que introducir mecanismos de regulación de la concurrencia más permisivos.

TRANSACCIONES

¿Cómo garantizar estas propiedades?

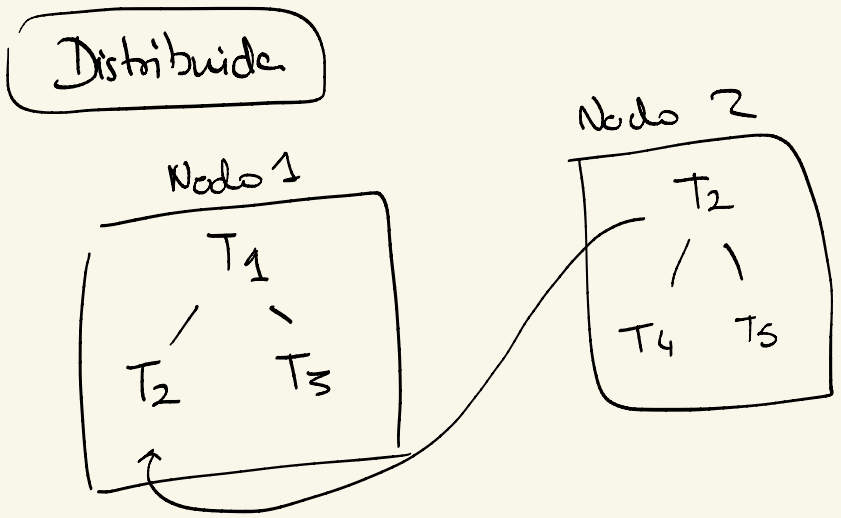
- Mediante técnicas de control de concurrencia puede obtenerse la independencia y consistencia: *exclusión mutua*.
- Pero la exclusión mutua es muy restrictiva, más aun al afectar a una colección de recursos que deben ser retenidos por un tiempo indefinido.
- Además no garantiza la atomicidad: todo o nada en una serie de operaciones.
- Habrá que introducir mecanismos de regulación de la concurrencia más permisivos.

TRANSACCIONES



Otras características a considerar:

- Las transacciones pueden *anidarse*, generando una estructura jerárquica (arborescente) de transacciones y subtransacciones.
  - aprovecha mejor recursos sys
- Las transacciones pueden ser *distribuidas*, afectando a recursos u objetos distribuidos: cada operación puede realizarse en un nodo diferente.


Distribuida
Nodo 1
T1
T2
T3
Nodo 2
T2
T4
T5

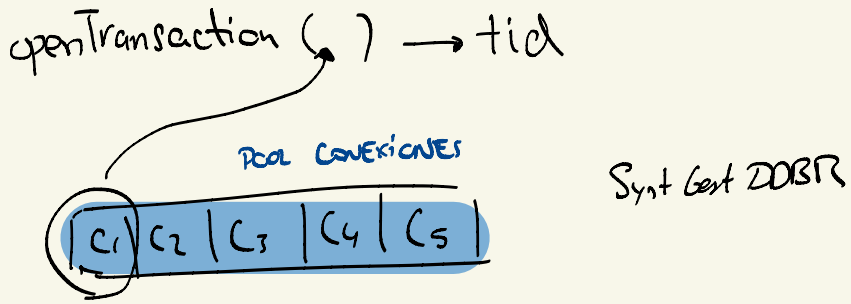
MODELO DE OPERACIONES

- Para poder cumplir con las propiedades (ACID), una transacción comenzará su ejecución con una operación de inicio de transacción: *openTransaction()* $\rightarrow$ *tid*.
  - Para bloquear los recursos que va a utilizar.
  - Para realizar copias (tentativas) de los recursos que va a modificar.
  - Para realizar una salvaguarda del estado del sistema por si es necesario un *roll-back*.
- Su ejecución terminará con una operación de cierre: *closeTransaction()* $\rightarrow \{commit, aborted\}$.
  - Comprobar si es posible la confirmación, y en su caso, actualizar los valores de los recursos. En otro caso realizar el *roll-back*.
  - Liberar los recursos bloqueados.

MODELO DE OPERACIONES

- Para poder cumplir con las propiedades (ACID), una transacción comenzará su ejecución con una operación de inicio de transacción: *openTransaction()* $\rightarrow$ *tid*.
  - Para bloquear los recursos que va a utilizar.
  - Para realizar copias (tentativas) de los recursos que va a modificar.
  - Para realizar una salvaguarda del estado del sistema por si es necesario un *roll-back*.
- Su ejecución terminará con una operación de cierre: *closeTransaction()* $\rightarrow \{commit, aborted\}$.
  - Comprobar si es posible la confirmación, y en su caso, actualizar los valores de los recursos. En otro caso realizar el *roll-back*.
  - Liberar los recursos bloqueados.

Inicio Transacción



Synt Gest DDBB

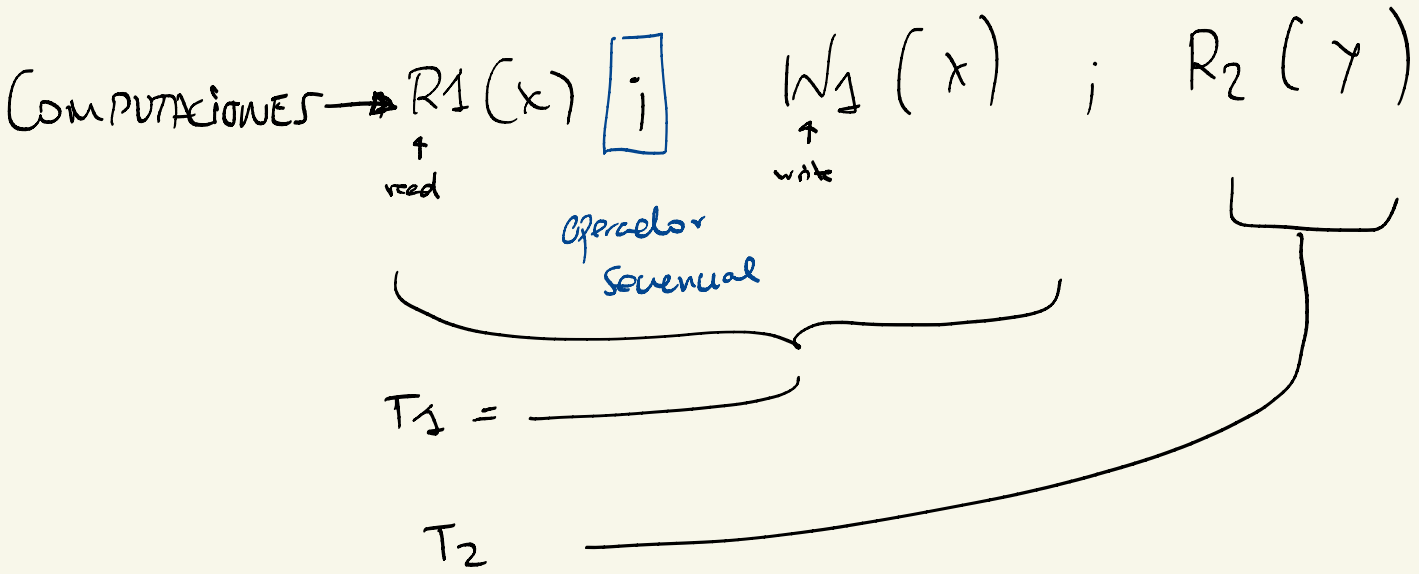
MODELO DE OPERACIONES

Cuerpo de una transacción:

- Ejecuta operaciones de **lectura** y de **escritura** sobre ciertos recursos.

  Modelo que seguimos:

  - $R_i(X)$: La transacción T_i realiza una lectura sobre el recurso u objeto X.
  - $W_i(X)$: La transacción T_i realiza una escritura sobre el recurso u objeto X.


COMPUTACIONES
R1(x)
i
W1(x)
;
R2(y)
read
write
Operador
secuencial
T1 =
T2

MODELO DE OPERACIONES

Otras operaciones:

- *abortTransaction(t)*: Aborta la transacción *t*. Dependiendo de la lógica de control de la aplicación puede ser necesario abortar la transacción actual u otra transacción, cuya ejecución no deba continuar.
- *commitTransaction()*: Podríamos separar esta operación de *closeTransaction*.

INDICE:

CONTROL DE CONCURRENCIA EN TRANSACCIONES

¿ Cómo garantizar la ejecución aislada e independiente de las transacciones ?

1. Ejecución serializada: una detrás de otra.
   Solución muy restrictiva.
2. Soluciones más permisivas, con mayor nivel de concurrencia: equivalentes a la serializada.

CONTROL DE CONCURRENCIA EN TRANSACCIONES

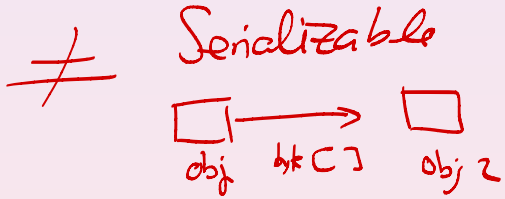
¿ Cómo garantizar la ejecución aislada e independiente de las transacciones ?

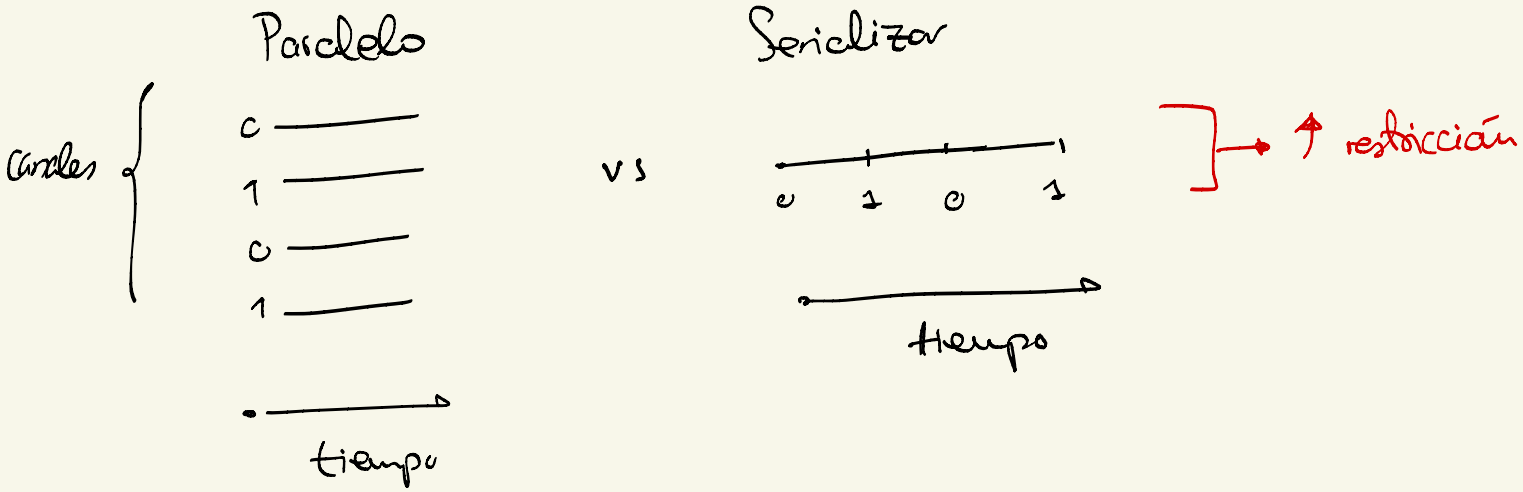
1. Ejecución serializada: una detrás de otra.
   Solución muy restrictiva.
2. Soluciones más permisivas, con mayor nivel de concurrencia: equivalentes a la serializada.

CONTROL DE CONCURRENCIA EN TRANSACCIONES

¿ Cómo garantizar la ejecución aislada e independiente de las transacciones ?

1. Ejecución serializada: una detrás de otra. Solución muy restrictiva.
2. Soluciones más permisivas, con mayor nivel de concurrencia: equivalentes a la serializada.




Paralelo
Serializar
cores
0
1
0
1
vs
0
1
0
1
tiempo
tiempo
restricción

TRANSACCIONES CONCURRENTES

Ejemplo de transacciones concurrentes:

T1	T2
$X = V/10;$	$Y = V/10;$
$V = V + X;$	$V = V + Y;$

- Inicialmente $V = 200$.
- Ejecución serializada de $T1$ y $T2 \Rightarrow V = 242$.
- Ejecución concurrente de $T1$ y $T2 \Rightarrow V = 242$ ó $V = 240$.

① $X = 200/10 = 20$
② $Y = 200/10 = 20$
③ $V = 200 + 20 = 220$
④ $V = 220 + 20 = 240$

MODELO DE LECTURAS-ESCRITURAS

Es necesario un modelo de lecturas-escrituras para conocer las situaciones de conflicto entre operaciones de transacciones diferentes:

Op. de T1	Op. de T2	¿Conflicto?	Motivo
Read	Read	NO	No se altera el valor leído
Read	Write	SI	El efecto de la escritura afecta a la lectura
Write	Write	SI	Importa el orden de las escrituras

EQUIVALENCIA SERIAL

Definición (COMPUTACIONES)

Dadas dos transacciones cualesquiera T_i, T_j, una computación de las mismas es una ejecución entremezclada de sus operaciones, respetando el orden local.

Ejemplo: $T_1 = R_1(X); R_1(Y); W_1(Z), \quad T_2 = R_2(X); W_2(Z).$

Una computación posible es:

$C = R_1(X); R_2(X); W_2(Z); R_1(Y); W_1(Z)$

EQUIVALENCIA SERIAL

Definición (EJECUCIONES SERIALIZADAS)

Una computación o ejecución serializada de dos transacciones T_i y T_j consiste en una ejecución de las mismas en la cual todas las operaciones de una de ellas preceden a las de la otra.

Ejemplo:

$$R_i(X); W_i(X); R_i(Y); R_j(X); W_j(Y)$$

y también

$$R_j(X); W_j(Y); R_i(X); W_i(X); R_i(Y)$$

SIEMPRE MISMO ORDEN

COMPUTACIONES EQUIVALENTES

Definición (COMPUTACIONES EQUIVALENTES Y SERIALIZABLES)

Dos computaciones obtenidas de dos transacciones cualesquiera T_i, T_j se dice que son equivalentes si para todo par de operaciones en conflicto, el orden en que las realizan siempre es el mismo ($i;j$ ó $j;i$).

Una computación de dos transacciones arbitrarias T_i, T_j es serializable si es equivalente a una computación serializada de las mismas.

COMPUTACIONES SERIALIZABLES

Ejemplo:

$T_1 = R_1(X); W_1(X); W_1(Y), \quad T_2 = R_2(Y); W_2(Y); R_2(X).$

- $C = R_1(X); W_1(X); R_2(Y); W_2(Y); W_1(Y); R_2(X)$

  No es serializable: $W_1(X)$ está en conflicto con $R_2(X)$, luego la serialización tendría que ser $T_1; T_2$. Sin embargo, $R_2(Y)$ está en conflicto con $W_1(Y)$ y en la ordenación aparece en orden contrario al de $T_1; T_2$.

- $C = R_2(Y); R_1(X); W_2(Y); R_2(X); W_1(X); W_1(Y)$

  Es serializable, equivalente a $T_2; T_1$.

COMPUTACIONES SERIALIZABLES

Ejemplo:

$T_1 = R_1(X); W_1(X); W_1(Y), \quad T_2 = R_2(Y); W_2(Y); R_2(X).$

- $C = R_1(X); W_1(X); R_2(Y); W_2(Y); W_1(Y); R_2(X)$

  No es serializable: $W_1(X)$ está en conflicto con $R_2(X)$, luego la serialización tendría que ser $T_1; T_2$. Sin embargo, $R_2(Y)$ está en conflicto con $W_1(Y)$ y en la ordenación aparece en orden contrario al de $T_1; T_2$.

- $C = R_2(Y); R_1(X); W_2(Y); R_2(X); W_1(X); W_1(Y)$

  Es serializable, equivalente a $T_2; T_1$.

CONFIRMACIÓN DE LAS TRANSACCIONES

- **Lecturas sucias:** Una transacción T lee un objeto modificado previamente por U.

  La transacción U aborta, restaurando el estado original del objeto, lo que provoca que T también sea abortada, porque ha estado trabajando con un valor erróneo del estado del objeto.

  Consecuencia: podemos tener abortos en cascada.

  Solución estricta:

  Las transacciones sólo pueden leer aquellos objetos o recursos modificados por transacciones ya confirmadas.

CONFIRMACIÓN DE LAS TRANSACCIONES

- **Escrituras prematuras:** Una transacción *T* escribe sobre el objeto *O*, que ha sido modificado previamente por *U*.

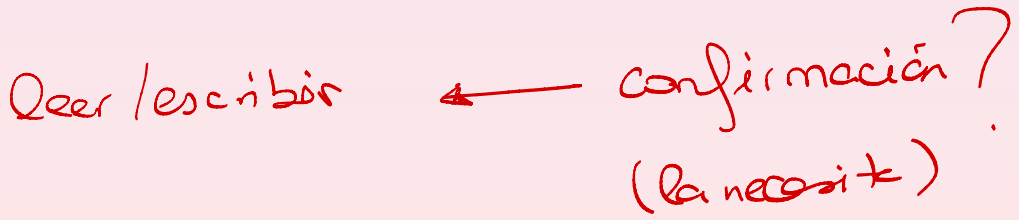
  U aborta y *O* recupera su estado original: se pierde la modificación hecha por *T* que debe ser abortada (estado inconsistente).

  Consecuencia: abortos en cascada.

  Solución estricta: Las transacciones sólo pueden escribir sobre objetos o recursos modificados por transacciones ya confirmadas.

SOLUCIONES ESTRICTAS

- Problema: las soluciones estrictas limitan el nivel de concurrencia.
- Una transacción que quiere leer/escribir de/sobre un objeto debe retrasar esta operación hasta que el estado del objeto sea confirmado (confirmación de la transacción que lo modificó).



VERSIONES TENTATIVAS

- *T* puede crear versiones tentativas de los recursos u objetos cuando intenta modificar su estado la primera vez.
- Después, todas las operaciones de *T* sobre este objeto se harán sobre la copia tentativa del mismo.
- Confirmación transacción: se actualiza el estado de los recursos usando las copias tentativas.
- Exclusión mutua mientras se realiza esta actualización del estado de los recursos originales.
- Ventaja: cuando se aborta una transacción, se eliminan sin más sus copias tentativas de los objetos que utiliza.

VERSIONES TENTATIVAS

- *T* puede crear versiones tentativas de los recursos u objetos cuando intenta modificar su estado la primera vez.
- Después, todas las operaciones de *T* sobre este objeto se harán sobre la copia tentativa del mismo.
- Confirmación transacción: se actualiza el estado de los recursos usando las copias tentativas.
- Exclusión mutua mientras se realiza esta actualización del estado de los recursos originales.
- Ventaja: cuando se aborta una transacción, se eliminan sin más sus copias tentativas de los objetos que utiliza.

VERSIONES TENTATIVAS

- *T* puede crear versiones tentativas de los recursos u objetos cuando intenta modificar su estado la primera vez.
- Después, todas las operaciones de *T* sobre este objeto se harán sobre la copia tentativa del mismo.
- Confirmación transacción: se actualiza el estado de los recursos usando las copias tentativas.
- Exclusión mutua mientras se realiza esta actualización del estado de los recursos originales.
- Ventaja: cuando se aborta una transacción, se eliminan sin más sus copias tentativas de los objetos que utiliza.

VERSIONES TENTATIVAS

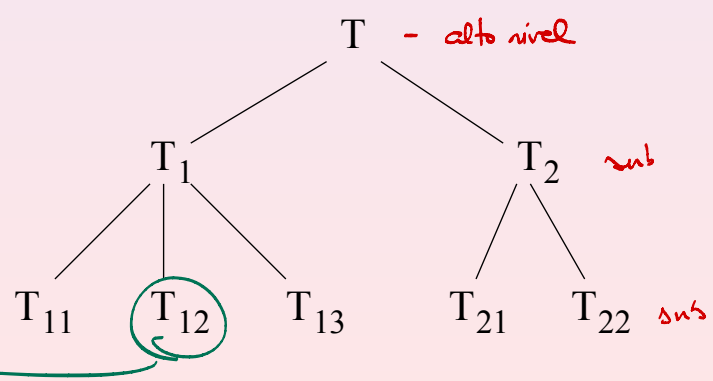
- *T* puede crear versiones tentativas de los recursos u objetos cuando intenta modificar su estado la primera vez.
- Después, todas las operaciones de *T* sobre este objeto se harán sobre la copia tentativa del mismo.
- Confirmación transacción: se actualiza el estado de los recursos usando las copias tentativas.
- Exclusión mutua mientras se realiza esta actualización del estado de los recursos originales.
- Ventaja: cuando se aborta una transacción, se eliminan sin más sus copias tentativas de los objetos que utiliza.

VERSIONES TENTATIVAS

- *T* puede crear versiones tentativas de los recursos u objetos cuando intenta modificar su estado la primera vez.
- Después, todas las operaciones de *T* sobre este objeto se harán sobre la copia tentativa del mismo.
- Confirmación transacción: se actualiza el estado de los recursos usando las copias tentativas.
- Exclusión mutua mientras se realiza esta actualización del estado de los recursos originales.
- Ventaja: cuando se aborta una transacción, se eliminan sin más sus copias tentativas de los objetos que utiliza.

TRANSACCIONES ANIDADAS

- Una transacción puede englobar otras transacciones: pueden iniciar (y terminar) otras transacciones.
- Transacciones planas: no tienen subtransacciones.
- Se llama transacción de alto nivel a la que ocupe la raíz del árbol correspondiente de transacciones.
- Las transacciones creadas por una transacción se dice que son subtransacciones de ella.



TRANSACCIONES ANIDADAS

- Cada transacción es atómica para su progenitora.
- La progenitora queda bloqueada mientras se ejecutan sus subtransacciones.
- Ejecución concurrente de las subtransacciones, a menos que haya interferencias entre ellas (accesos a objetos comunes).
- Fallo de transacción progenitora $\Rightarrow$ se abortan todas sus subtransacciones.
- Fallo de subtransacción $\not\Rightarrow$ Abortar la progenitora.

TRANSACCIONES ANIDADAS

- Cada transacción es atómica para su progenitora.
- La progenitora queda bloqueada mientras se ejecutan sus subtransacciones.
- Ejecución concurrente de las subtransacciones, a menos que haya interferencias entre ellas (accesos a objetos comunes).
- Fallo de transacción progenitora $\Rightarrow$ se abortan todas sus subtransacciones.
- Fallo de subtransacción $\not\Rightarrow$ Abortar la progenitora.

TRANSACCIONES ANIDADAS

- Cada transacción es atómica para su progenitora.
- La progenitora queda bloqueada mientras se ejecutan sus subtransacciones.
- Ejecución concurrente de las subtransacciones, a menos que haya interferencias entre ellas (accesos a objetos comunes).
- Fallo de transacción progenitora $\Rightarrow$ se abortan todas sus subtransacciones.
- Fallo de subtransacción $\not\Rightarrow$ Abortar la progenitora.

TRANSACCIONES ANIDADAS

- Cada transacción es atómica para su progenitora.
- La progenitora queda bloqueada mientras se ejecutan sus subtransacciones.
- Ejecución concurrente de las subtransacciones, a menos que haya interferencias entre ellas (accesos a objetos comunes).
- Fallo de transacción progenitora $\Rightarrow$ se abortan todas sus subtransacciones.
- Fallo de subtransacción $\not\Rightarrow$ Abortar la progenitora.

TRANSACCIONES ANIDADAS

- Cada transacción es atómica para su progenitora.
- La progenitora queda bloqueada mientras se ejecutan sus subtransacciones.
- Ejecución concurrente de las subtransacciones, a menos que haya interferencias entre ellas (accesos a objetos comunes).
- Fallo de transacción progenitora $\Rightarrow$ se abortan todas sus subtransacciones.
- Fallo de subtransacción $\not\Rightarrow$ Abortar la progenitora.

TRANSACCIONES ANIDADAS

¿ Para qué podemos usar las transacciones anidadas?

- Subtransacciones: Permiten realizar acciones concurrentes dentro de una misma transacción, que de otra forma serían secuenciales.
- Mayor fiabilidad: el fallo de una subtransacción no invalida los efectos de sus compañeras o su progenitora.

  El fallo de una subtransacción no causa que toda la transacción sea abortada o reiniciada, se pueden considerar diferentes acciones cuando falla una subtransacción.

TRANSACCIONES ANIDADAS

¿ Para qué podemos usar las transacciones anidadas?

- Subtransacciones: Permiten realizar acciones concurrentes dentro de una misma transacción, que de otra forma serían secuenciales.
- Mayor fiabilidad: el fallo de una subtransacción no invalida los efectos de sus compañeras o su progenitora.

  El fallo de una subtransacción no causa que toda la transacción sea abortada o reiniciada, se pueden considerar diferentes acciones cuando falla una subtransacción.

TRANSACCIONES ANIDADAS

¿ Para qué podemos usar las transacciones anidadas?

- Subtransacciones: Permiten realizar acciones concurrentes dentro de una misma transacción, que de otra forma serían secuenciales.
- Mayor fiabilidad: el fallo de una subtransacción no invalida los efectos de sus compañeras o su progenitora.

  El fallo de una subtransacción no causa que toda la transacción sea abortada o reiniciada, se pueden considerar diferentes acciones cuando falla una subtransacción.

TRANSACCIONES ANIDADAS

Reglas de confirmación para transacciones anidadas:

- Una (sub)transacción sólo puede confirmarse (provisionalmente) cuando lo han hecho todas sus hijas (las que no fueron abortadas).
- Las subtransacciones se confirman provisionalmente, hasta la confirmación definitiva de la raíz.
- Transacción abortada: se abortan todas sus subtransacciones, aunque estén confirmadas provisionalmente.
- Transacción de alto nivel confirmada:
  - Todas las subtransacciones confirmadas provisionalmente (no abortadas) en el árbol se confirman definitivamente.

TRANSACCIONES ANIDADAS

Reglas de confirmación para transacciones anidadas:

- Una (sub)transacción sólo puede confirmarse (provisionalmente) cuando lo han hecho todas sus hijas (las que no fueron abortadas).
- Las subtransacciones se confirman provisionalmente, hasta la confirmación definitiva de la raíz.
- Transacción abortada: se abortan todas sus subtransacciones, aunque estén confirmadas provisionalmente.
- Transacción de alto nivel confirmada:
  - Todas las subtransacciones confirmadas provisionalmente (no abortadas) en el árbol se confirman definitivamente.

TRANSACCIONES ANIDADAS

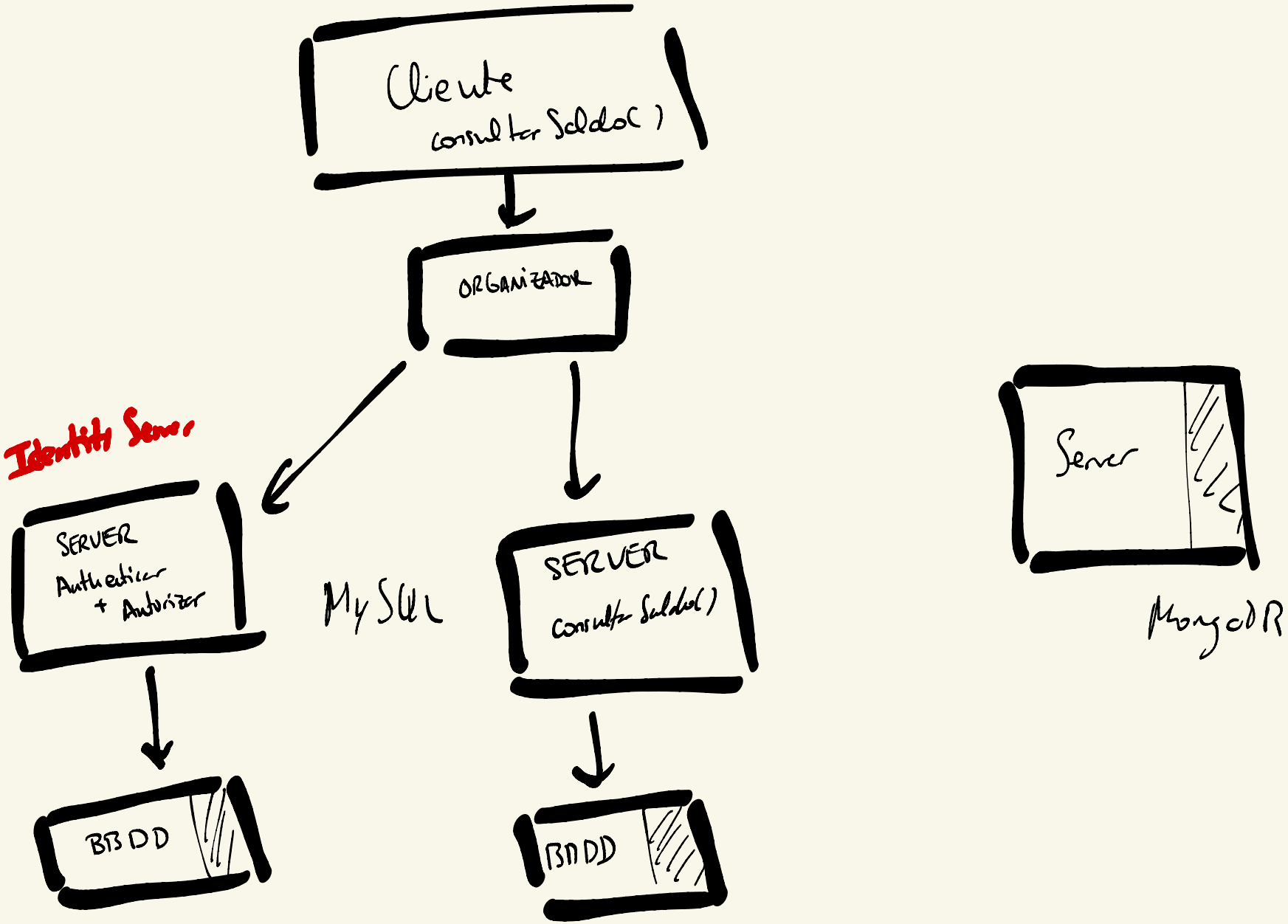
Reglas de confirmación para transacciones anidadas:

- Una (sub)transacción sólo puede confirmarse (provisionalmente) cuando lo han hecho todas sus hijas (las que no fueron abortadas).
- Las subtransacciones se confirman provisionalmente, hasta la confirmación definitiva de la raíz.
- Transacción abortada: se abortan todas sus subtransacciones, aunque estén confirmadas provisionalmente.
- Transacción de alto nivel confirmada:
  - Todas las subtransacciones confirmadas provisionalmente (no abortadas) en el árbol se confirman definitivamente.

TRANSACCIONES ANIDADAS

Reglas de confirmación para transacciones anidadas:

- Una (sub)transacción sólo puede confirmarse (provisionalmente) cuando lo han hecho todas sus hijas (las que no fueron abortadas).
- Las subtransacciones se confirman provisionalmente, hasta la confirmación definitiva de la raíz.
- Transacción abortada: se abortan todas sus subtransacciones, aunque estén confirmadas provisionalmente.
- Transacción de alto nivel confirmada:
  - Todas las subtransacciones confirmadas provisionalmente (no abortadas) en el árbol se confirman definitivamente.


Cliente
consultarSaldo()
ORGANIZADOR
Identity Server
SERVER
Authenticar
+ Autorizar
MySQL
SERVER
consultarSaldo()
BBDD
BDDD
Server
MongoDB

PROTOCOLO DE CONFIRMACIÓN DISTRIBUIDO ATÓMICO

- Atómico: todos los participantes deciden si la transacción puede confirmarse o debe ser abortada.

Protocolo

- 1 fase → inicia (commit) coordinador
  - luego done ← problema = aborta
  - cualquier cliente
- 2 fases → inicia elección consensuada (joined)

PROTOCOLO DE CONFIRMACIÓN DISTRIBUIDO ATÓMICO

IMPORTANTE EXAMEN

Solución simple: **protocolo de confirmación de 1 fase**.

- Los participantes envían un mensaje *Done* al coordinador cuando en su nodo hayan finalizado las acciones locales de la transacción, y esperan el mensaje *Commit* del coordinador.
- Cuando el coordinador ha recibido todos los mensajes *Done*, envía a todos el mensaje *Commit* (abort en caso de no haber recibido el mensaje *Done* de alguno).

PROTOCOLO DE CONFIRMACIÓN DISTRIBUIDO ATÓMICO

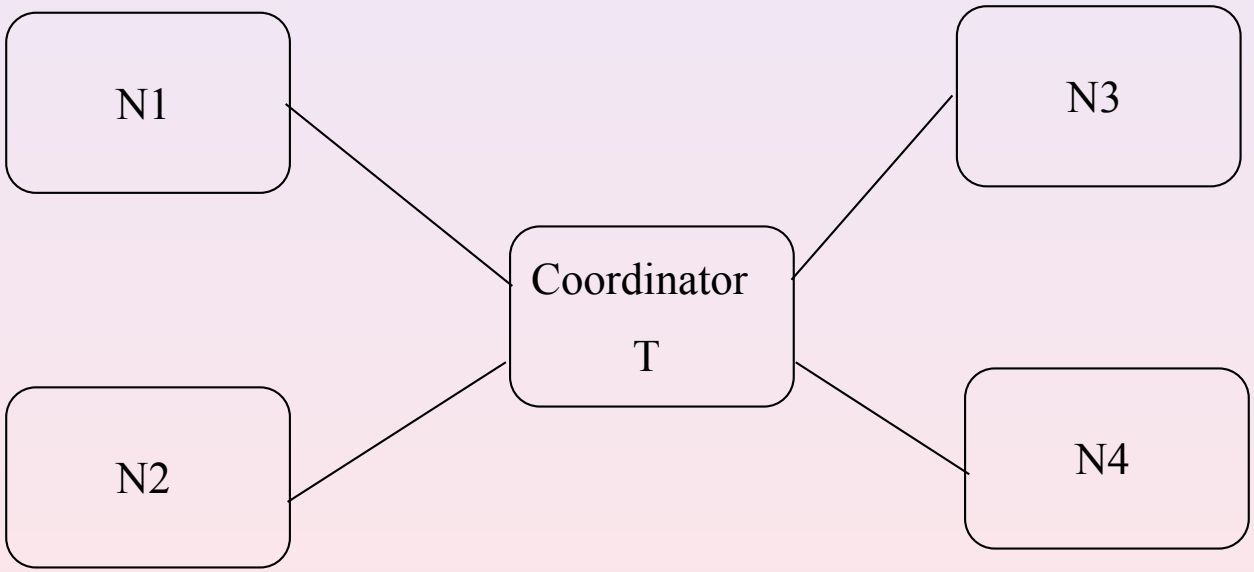
- Los nodos envían un mensaje *ACK* al coordinador, para indicar que se han confirmado localmente los efectos de la transacción.
- Si no se recibe el mensaje ACK de alguno de los nodos, se envía a todos el mensaje *roll-back*.

PROTOCOLO DE CONFIRMACIÓN DE 1 FASE

- Un participante puede enviar *abort* al coordinador, si localmente no puede validar la confirmación o no puede completar la transacción.
- Si las acciones de una transacción quedan bloqueadas en un nodo, o son abortadas por cualquier motivo, este participante no podrá informar al coordinador.
- El coordinador no podrá tomar la decisión acerca de confirmar o no la transacción, al haber algún nodo que no responde.
- Se puede fijar un *time-out*, y enviar *abort* a todos los participantes cuando algún nodo no ha respondido o no se reciben sus mensajes del tipo *SIGO_VIVO*.

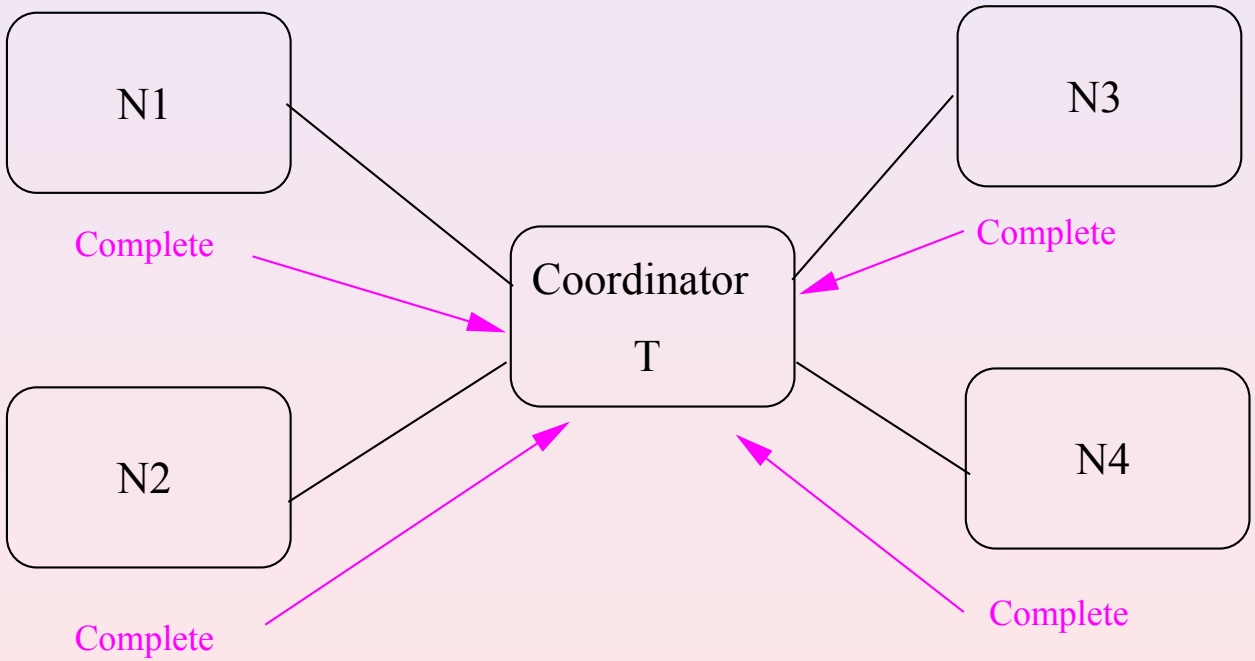
PROTOCOLO DE CONFIRMACIÓN DE 2-FASES

Todos los participantes votan para decidir la confirmación.



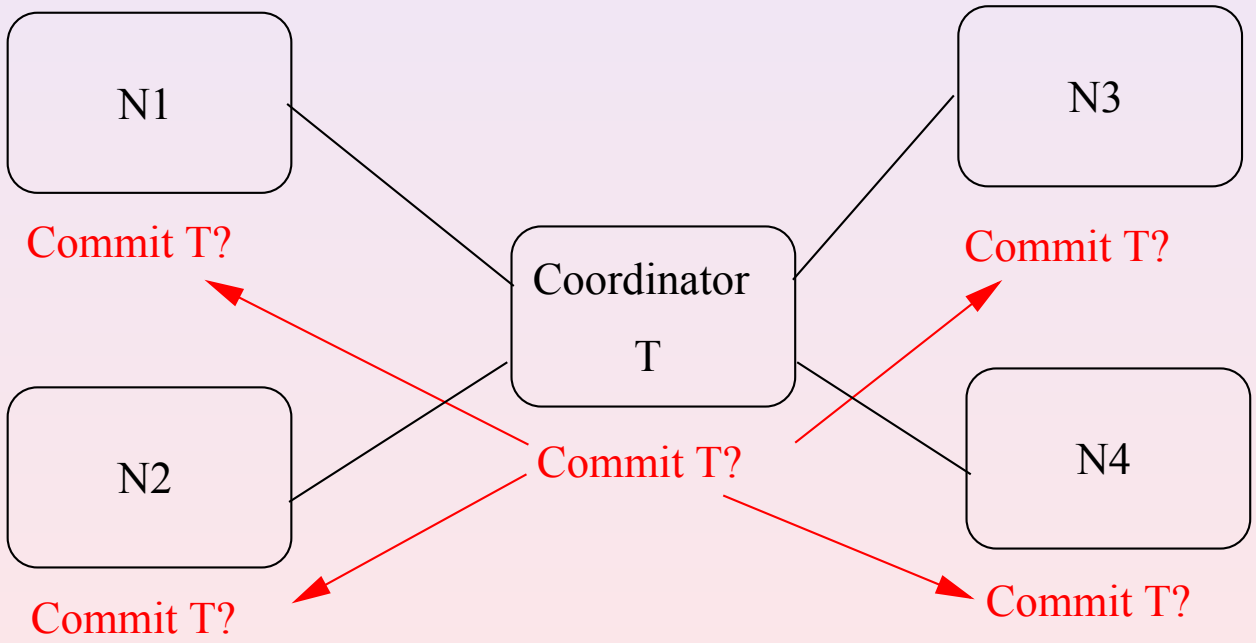
PROTOCOLO DE CONFIRMACIÓN DE 2-FASES

Todos los participantes votan para decidir la confirmación.



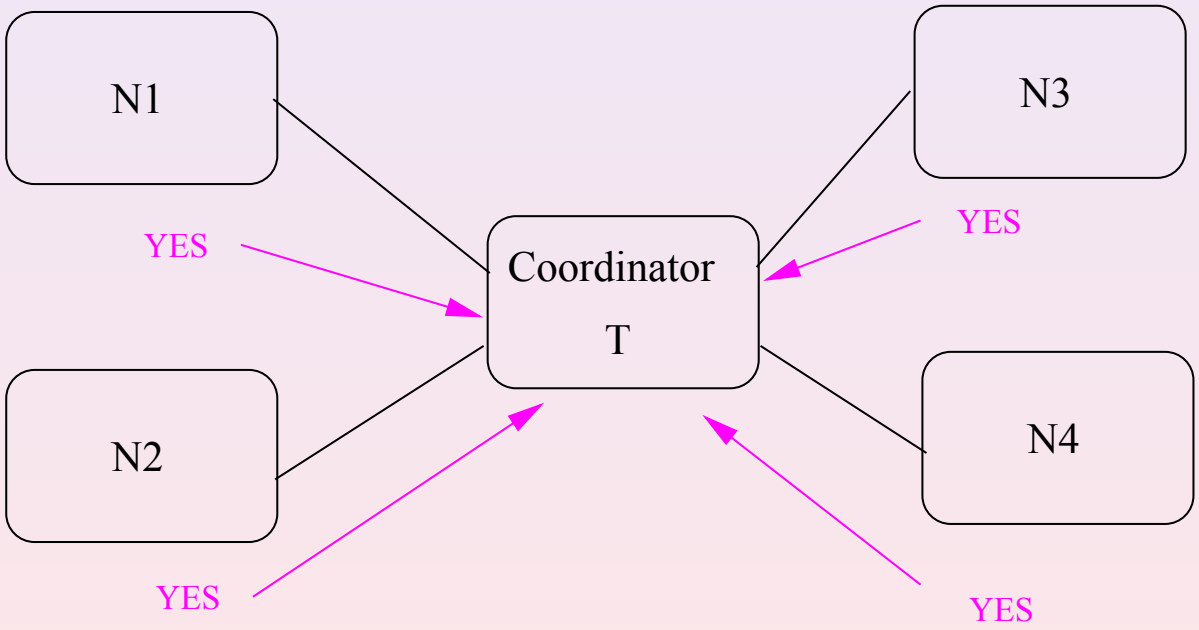
PROTOCOLO DE CONFIRMACIÓN DE 2-FASES

Todos los participantes votan para decidir la confirmación.



PROTOCOLO DE CONFIRMACIÓN DE 2-FASES

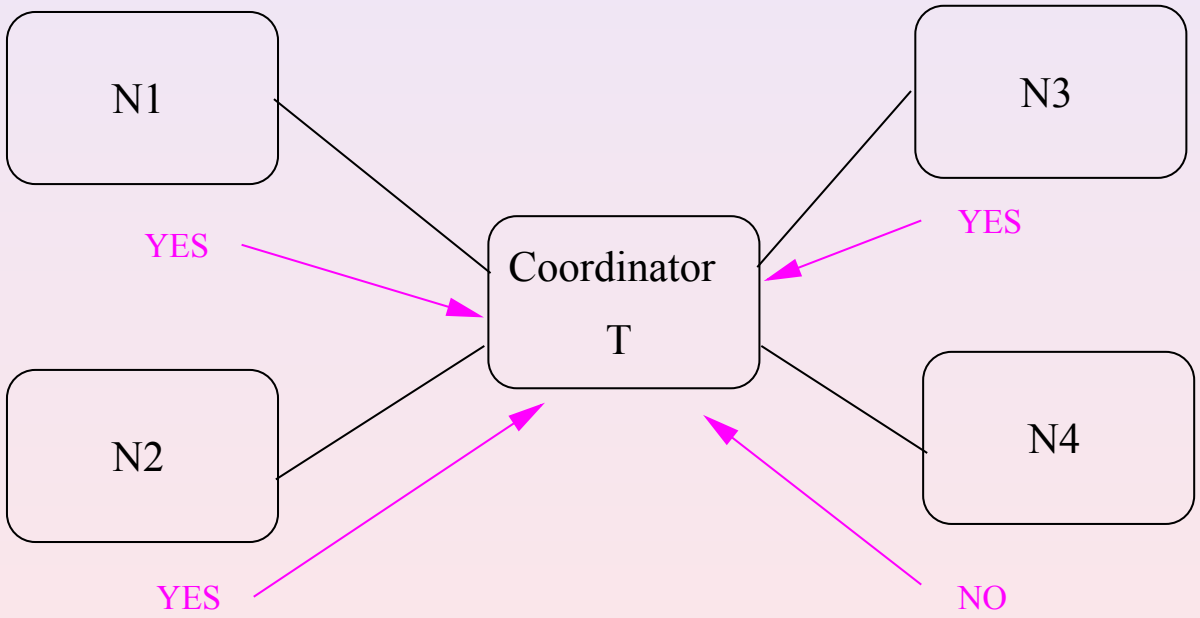
Todos los participantes votan para decidir la confirmación.



Coordinator sends commitment message to all participants

PROTOCOLO DE CONFIRMACIÓN DE 2-FASES

Todos los participantes votan para decidir la confirmación.



Coordinator sends abort to all participants

PROTOCOLO DE CONFIRMACIÓN DE 2-FASES

[Gray, 1978]

Fase 1:

- Votación: el coordinador les pide que voten, una vez recibidos todos los mensajes de finalización.
- Los participantes envían sus votos (Yes/No).
- Los que han votado *Yes* salvan en memoria permanente el estado de los objetos que hayan modificado. Si hay un fallo después de votar, se puede recuperar el estado de los objetos y realizar la confirmación.
- Decisión coordinada: un solo *No* de un participante implica abortar la transacción.

PROTOCOLO DE CONFIRMACIÓN DE 2-FASES

Fase 2:

- Una vez recibido el mensaje con la decisión del coordinador, todos los nodos confirman o abortan.
- Si alguno falla después de votar, su estado se recupera desde memoria permanente antes de realizar la acción enviada desde el coordinador (confirmar o abortar).

PROTOCOLO DE CONFIRMACIÓN DE 2-FASES

Posibles fallos:

- Un participante no responde a la petición de votar:

  Después de un tiempo razonable (*time-out*), el coordinador envía *abort* a todos.

- Si falla el coordinador, y no es restaurado en un cierto tiempo, los participantes abortan sus transacciones huérfanas.

SERIALIZACIÓN

Queremos estudiar diferentes técnicas que nos garanticen la serialización, es decir, las computaciones resultantes son siempre equivalentes a una ejecución ordenada (serializada) de las transacciones (y subtransacciones).

ORDENACIÓN POR MARCAS DE TIEMPO

- Mecanismo sencillo de serialización: asociar marcas de tiempo a las transacciones al inicio de su fase de trabajo (openTransaction).

$$TS(T)$$

- Uso de la marca de tiempo (timestamp): orden de ejecución serializada de la computación equivalente.

$$TS(T_1) < TS(T_2) < \ldots < TS(T_n) \rightarrow \text{equivalente a } T_1; T_2; \ldots; T_n$$

- Cada operación debe ser validada antes de su ejecución.
- Fallo en validación de una operación $\Rightarrow$ abortar transacción y reiniciarla posiblemente tras un rato.

ORDENACIÓN POR MARCAS DE TIEMPO

- Mecanismo sencillo de serialización: asociar marcas de tiempo a las transacciones al inicio de su fase de trabajo (openTransaction).

$$TS(T)$$

- Uso de la marca de tiempo (timestamp): orden de ejecución serializada de la computación equivalente.

$$TS(T_1) < TS(T_2) < \ldots < TS(T_n) \rightarrow \text{equivalente a } T_1; T_2; \ldots; T_n$$

- Cada operación debe ser validada antes de su ejecución.
- Fallo en validación de una operación $\Rightarrow$ abortar transacción y reiniciarla posiblemente tras un rato.

ORDENACIÓN POR MARCAS DE TIEMPO

- Mecanismo sencillo de serialización: asociar marcas de tiempo a las transacciones al inicio de su fase de trabajo (openTransaction).

$$TS(T)$$

- Uso de la marca de tiempo (timestamp): orden de ejecución serializada de la computación equivalente.

$$TS(T_1) < TS(T_2) < \ldots < TS(T_n) \rightarrow \text{equivalente a } T_1; T_2; \ldots; T_n$$

- Cada operación debe ser validada antes de su ejecución.
- Fallo en validación de una operación $\Rightarrow$ abortar transacción y reiniciarla posiblemente tras un rato.

ORDENACIÓN POR MARCAS DE TIEMPO

- Mecanismo sencillo de serialización: asociar marcas de tiempo a las transacciones al inicio de su fase de trabajo (openTransaction).

$$TS(T)$$

- Uso de la marca de tiempo (timestamp): orden de ejecución serializada de la computación equivalente.

$$TS(T_1) < TS(T_2) < \ldots < TS(T_n) \rightarrow \text{equivalente a } T_1; T_2; \ldots; T_n$$

- Cada operación debe ser validada antes de su ejecución.
- Fallo en validación de una operación $\Rightarrow$ abortar transacción y reiniciarla posiblemente tras un rato.

ORDENACIÓN POR MARCAS DE TIEMPO

Marca de tiempo de objetos/recursos:

- **Read timestamp**: TS de la última transacción que realizó una lectura: *ReadTS(X)*.
- **Write timestamp**: TS de la última transacción que realizó una escritura: *WriteTS(X)*.

Inicialmente: *ReadTS(X)=WriteTS(X)=(0,0) (orden extendido)*.

ORDENACIÓN POR MARCAS DE TIEMPO

REGLAS:

- Una petición de escritura de una transacción sobre un objeto sólo es válida si las últimas lecturas y escrituras sobre el objeto fueron realizadas por transacciones anteriores (si las hay):

```
if (ReadTS(X)>TS(T) or WriteTS(X)>TS(T)) then
   Abort T;
   Rollback T;
else {
   Execute write operation on X;
   WriteTS(X)=TS(T);
}
```

ORDENACIÓN POR MARCAS DE TIEMPO

REGLAS:

- Una petición de lectura de un objeto sólo es válida si la última escritura sobre el objeto fue realizada por una transacción anterior (si la hay):

```
if (WriteTS(X)>TS(T)) then
   Abort T;
   Rollback T;
else {
   Execute read operation on X;
   ReadTS(X)=Max{ReadTS(X), TS(T)};
}
```

ORDENACIÓN POR MARCAS DE TIEMPO

PROBLEMA:

- Puede haber numerosos abortos, y también abortos en cascada.
- Ejemplo: sean las transacciones T_1, T_2, T_3, con marcas de tiempo 1, 2, 3 (suponemos enteros por simplicidad):

$$W1(X); W2(X); R3(X); R1(X); R2(X)$$

T_1 debe ser abortada al intentar ejecutar $R_1(X)$, y entonces T_3 también debe ser abortada ($W_1(X)$ antes que $R_3(X)$: lectura sucia). También T_2 es abortada (escritura prematura: $W_1(X)$).

ORDENACIÓN POR MARCAS DE TIEMPO

Escritura prematura:

- Consideremos:

$$W1(X); W2(X); R2(X); W3(X); R2(X)$$

T_2 aborta al intentar ejecutar la última operación ($R_2(X)$), y entonces T_3 también es abortada ($W_2(X)$ antes que $W_3(X)$).

ORDENACIÓN POR MARCAS DE TIEMPO

- Pueden producirse numerosos abortos de transacciones.
- Se puede plantear una versión mejorada empleando versiones tentativas de los objetos/recursos, con marcas de tiempo asociadas a dichas copias tentativas (omitido).

INDICE:

SERIALIZACIÓN: BLOQUEOS

Solución más simple:

Bloqueo de objetos/recursos al inicio:

- Al inicio de una transacción: bloqueo de todos los recursos u objetos que necesite durante su ejecución.
- Solución muy restrictiva: objetos o recursos bloqueados mucho tiempo. Bajo nivel de concurrencia.
- Varios intentos hasta lograr obtener todos los recursos.
- *Starvation:* puede producirse cuando las transacciones usan muchos recursos durante mucho tiempo.

SERIALIZACIÓN: BLOQUEOS

Solución más simple:

Bloqueo de objetos/recursos al inicio:

- Al inicio de una transacción: bloqueo de todos los recursos u objetos que necesite durante su ejecución.
- Solución muy restrictiva: objetos o recursos bloqueados mucho tiempo. Bajo nivel de concurrencia.
- Varios intentos hasta lograr obtener todos los recursos.
- *Starvation:* puede producirse cuando las transacciones usan muchos recursos durante mucho tiempo.

SERIALIZACIÓN: BLOQUEOS

Solución más simple:

Bloqueo de objetos/recursos al inicio:

- Al inicio de una transacción: bloqueo de todos los recursos u objetos que necesite durante su ejecución.
- Solución muy restrictiva: objetos o recursos bloqueados mucho tiempo. Bajo nivel de concurrencia.
- Varios intentos hasta lograr obtener todos los recursos.
- *Starvation:* puede producirse cuando las transacciones usan muchos recursos durante mucho tiempo.

SERIALIZACIÓN: BLOQUEOS

Solución más simple:

Bloqueo de objetos/recursos al inicio:

- Al inicio de una transacción: bloqueo de todos los recursos u objetos que necesite durante su ejecución.
- Solución muy restrictiva: objetos o recursos bloqueados mucho tiempo. Bajo nivel de concurrencia.
- Varios intentos hasta lograr obtener todos los recursos.
- *Starvation:* puede producirse cuando las transacciones usan muchos recursos durante mucho tiempo.

PROTOCOLO DE BLOQUEO DE 2-FASES

Mejor solución:

- Adquisición de bloqueos de recursos conforme se necesiten, asumiendo el riesgo de *deadlocks*.
- **Criterio:** No se pueden adquirir nuevos bloqueos una vez liberado algún bloqueo.
- Ejecución en 2 fases:
  1. Adquisición de bloqueos.
  2. Liberación de bloqueos.
- Puede haber DEADLOCKS.

PROTOCOLO DE BLOQUEO DE 2-FASES

Mejor solución:

- Adquisición de bloqueos de recursos conforme se necesiten, asumiendo el riesgo de *deadlocks*.
- **Criterio:** No se pueden adquirir nuevos bloqueos una vez liberado algún bloqueo.
- Ejecución en 2 fases:
  1. Adquisición de bloqueos.
  2. Liberación de bloqueos.
- Puede haber DEADLOCKS.

PROTOCOLO DE BLOQUEO DE 2-FASES

Mejor solución:

- Adquisición de bloqueos de recursos conforme se necesiten, asumiendo el riesgo de *deadlocks*.
- **Criterio:** No se pueden adquirir nuevos bloqueos una vez liberado algún bloqueo.
- Ejecución en 2 fases:
  1. Adquisición de bloqueos.
  2. Liberación de bloqueos.
- Puede haber DEADLOCKS.

PROTOCOLO DE BLOQUEO DE 2-FASES

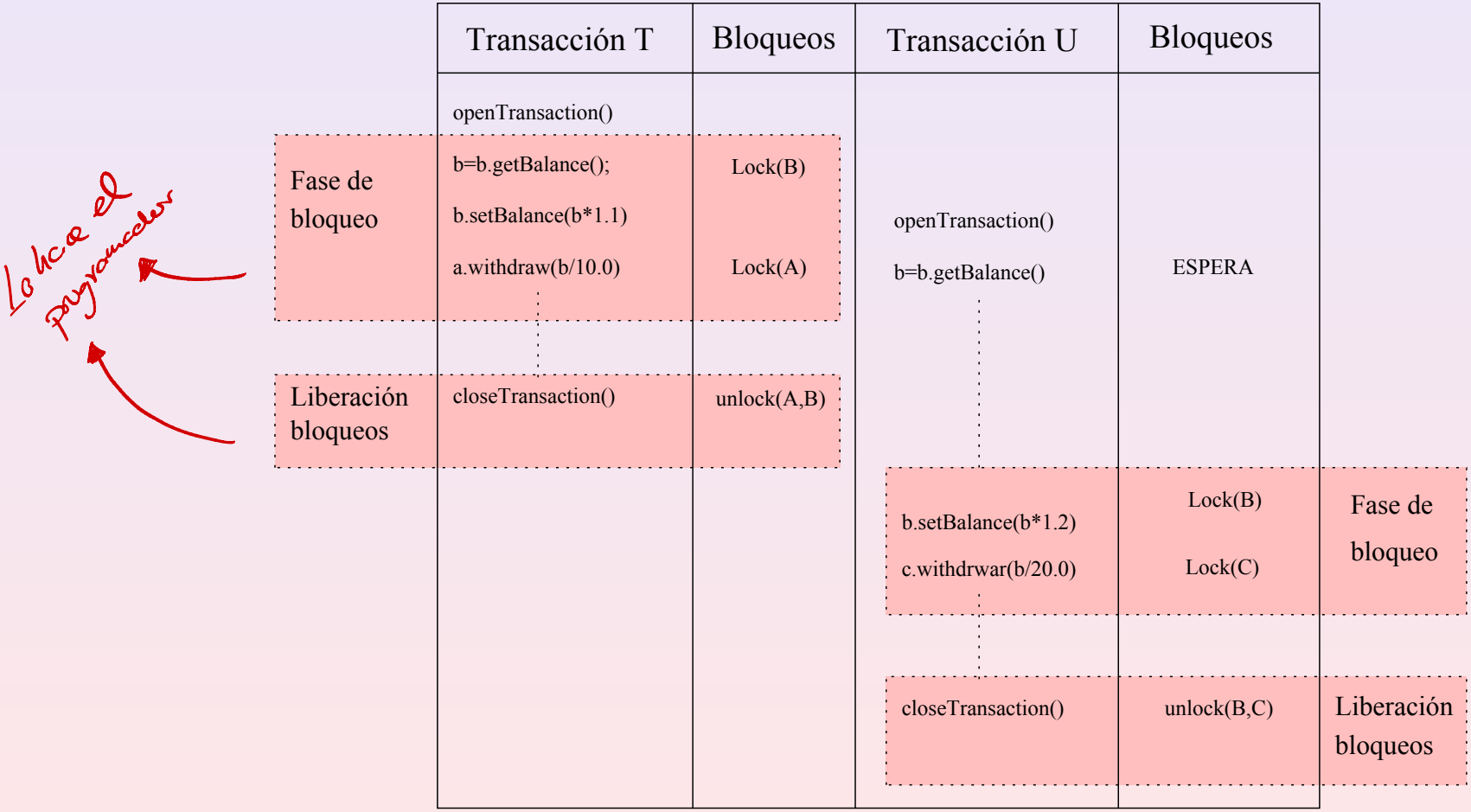
Mejor solución:

- Adquisición de bloqueos de recursos conforme se necesiten, asumiendo el riesgo de *deadlocks*.
- **Criterio:** No se pueden adquirir nuevos bloqueos una vez liberado algún bloqueo.
- Ejecución en 2 fases:
  1. Adquisición de bloqueos.
  2. Liberación de bloqueos.
- Puede haber DEADLOCKS.

PROTOCOLO DE BLOQUEO DE 2-FASES

Transacción T	Bloqueos	Transacción U	Bloqueos
openTransaction()			
b=b.getBalance();	Lock(B)		
b.setBalance(b*1.1)		openTransaction()	
a.withdraw(b/10.0)	Lock(A)	b=b.getBalance()	ESPERA
⋮		⋮	
closeTransaction()	unlock(A,B)		
		b.setBalance(b*1.2)	Lock(B)
		c.withdrwar(b/20.0)	Lock(C)
		⋮	
		closeTransaction()	unlock(B,C)

PROTOCOLO DE BLOQUEO DE 2-FASES



PROTOCOLO DE BLOQUEO DE 2-FASES

- El protocolo de bloqueo de 2-fases no evita las lecturas sucias o escrituras prematuras.
- CAUSA: Un recurso liberado por una transacción puede ser usado por otra, pero en caso de aborto de la primera recupera un valor anterior, y la segunda tendrá que ser abortada.

PROTOCOLO DE BLOQUEO DE 2-FASES

PREGUNTA EXAMEN

- El protocolo de bloqueo de 2-fases no evita las lecturas sucias o escrituras prematuras.
- CAUSA: Un recurso liberado por una transacción puede ser usado por otra, pero en caso de aborto de la primera recupera un valor anterior, y la segunda tendrá que ser abortada.

PROTOCOLO DE BLOQUEO DE 2-FASES

- Para evitar lecturas sucias o escrituras prematuras: mantener los bloqueos hasta la terminación de la transacción $\rightarrow$ Protocolo de 2-fases estricto.
- No se desbloquean los recursos u objetos hasta haber escrito en memoria permanente (estable) toda la información modificada por la transacción sobre los mismos.
- Modelo muy restrictivo en el nivel de concurrencia permitido, al mantener el bloqueo cuando ya no se usa el recurso.

SERIALIZACIÓN

- El protocolo de 2-fases es una mejora sobre el protocolo de bloqueo inicial de todos los recursos.
- Pero sigue teniendo problemas: *deadlocks* y restricción importante del nivel de concurrencia.
- Se puede mejorar distinguiendo dos tipos de bloqueo, según se use el recurso sólo para lecturas o para lecturas/escrituras.

BLOQUEOS DE LECTURA/ESCRITURA

Uso de dos tipos de bloqueo:

- **read locks**: Para las operaciones de lectura.

  No pueden obtenerse si hay establecido un bloqueo de escritura sobre el recurso u objeto. Sí pueden obtenerse varios bloqueos de lectura sobre el recurso simultáneamente por varias transacciones.

- **write locks**: Para las operaciones de escritura.

  No pueden obtenerse si hay establecido un bloqueo de lectura o escritura sobre el recurso u objeto.

BLOQUEOS DE LECTURA/ESCRITURA

Uso de dos tipos de bloqueo:

- **read locks**: Para las operaciones de lectura.

  No pueden obtenerse si hay establecido un bloqueo de escritura sobre el recurso u objeto. Sí pueden obtenerse varios bloqueos de lectura sobre el recurso simultáneamente por varias transacciones.

- **write locks**: Para las operaciones de escritura.

  No pueden obtenerse si hay establecido un bloqueo de lectura o escritura sobre el recurso u objeto.

BLOQUEOS DE LECTURA/ESCRITURA

Bloqueos R/W con el protocolo de 2-fases estricto:

- Se adquieren los bloqueos según las reglas indicadas:
  - 2-fases: adquisición y liberación.
  - Control de bloqueos de lectura/escritura.
- Versión estricta: Sólo cuando se confirma o aborta la transacción se liberan todos los bloqueos que haya ido obteniendo.
  - Sigue existiendo el riesgo de *deadlocks*.
  - No hay lecturas sucias o escrituras prematuras.
  - Restricción de la concurrencia: mejora al permitir lecturas simultáneas, pero los recursos siguen bloqueados hasta la confirmación.

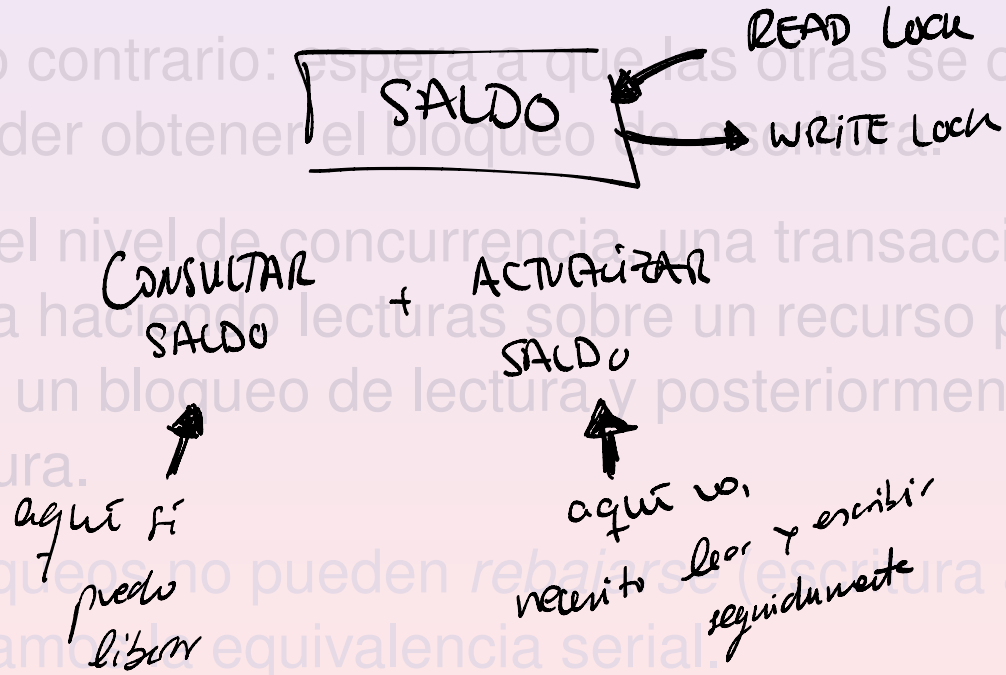
BLOQUEOS DE LECTURA/ESCRITURA

Bloqueos R/W con el protocolo de 2-fases estricto:

- Se adquieren los bloqueos según las reglas indicadas:
  - 2-fases: adquisición y liberación.
  - Control de bloqueos de lectura/escritura.
- Versión estricta: Sólo cuando se confirma o aborta la transacción se liberan todos los bloqueos que haya ido obteniendo.
  - Sigue existiendo el riesgo de *deadlocks*.
  - No hay lecturas sucias o escrituras prematuras.
  - Restricción de la concurrencia: mejora al permitir lecturas simultáneas, pero los recursos siguen bloqueados hasta la confirmación.

AMPLIACIÓN DE BLOQUEOS

- Una transacción puede ampliar su bloqueo de lectura a uno de escritura si no hay otros bloqueos de lectura simultáneos por otras transacciones.
- En caso contrario: espera a que las otras se confirmen, para poder obtener el bloqueo de escritura.
- Mejora el nivel de concurrencia: una transacción que empieza haciendo lecturas sobre un recurso pedirá primero un bloqueo de lectura y posteriormente lo amplía a escritura.
- Los bloqueos no pueden *rebajarse* (escritura a lectura): perderíamos la equivalencia serial.



AMPLIACIÓN DE BLOQUEOS

- Una transacción puede ampliar su bloqueo de lectura a uno de escritura si no hay otros bloqueos de lectura simultáneos por otras transacciones.
- En caso contrario: espera a que las otras se confirmen, para poder obtener el bloqueo de escritura.
- Mejora el nivel de concurrencia, una transacción que empieza haciendo lecturas sobre un recurso pedirá primero un bloqueo de lectura y posteriormente lo amplía a escritura.
- Los bloqueos no pueden *rebajarse* (escritura a lectura): perderíamos la equivalencia serial.

AMPLIACIÓN DE BLOQUEOS

- Una transacción puede ampliar su bloqueo de lectura a uno de escritura si no hay otros bloqueos de lectura simultáneos por otras transacciones.
- En caso contrario: espera a que las otras se confirmen, para poder obtener el bloqueo de escritura.
- Mejora el nivel de concurrencia, una transacción que empieza haciendo lecturas sobre un recurso pedirá primero un bloqueo de lectura y posteriormente lo amplía a escritura.
- Los bloqueos no pueden *rebajarse* (escritura a lectura): perderíamos la equivalencia serial.

AMPLIACIÓN DE BLOQUEOS

POSIBLE PREGUNTA → EJEMPLO de cuando se puede ampliar o no el bloqueo.

- Una transacción puede ampliar su bloqueo de lectura a uno de escritura si no hay otros bloqueos de lectura simultáneos por otras transacciones.
- En caso contrario: espera a que las otras se confirmen, para poder obtener el bloqueo de escritura.
- Mejora el nivel de concurrencia, una transacción que empieza haciendo lecturas sobre un recurso pedirá primero un bloqueo de lectura y posteriormente lo amplía a escritura.
- Los bloqueos no pueden *rebajarse* (escritura a lectura): perderíamos la equivalencia serial.

BLOQUEO EN TRANSACCIONES ANIDADAS

Reglas:

1. **Independencia entre árboles de transacciones**:
   Cada conjunto de transacciones anidadas (árbol/subárbol) es una entidad que no debe verse afectada por los efectos parciales de cualquier otro conjunto de transacciones anidadas.

2. **Independencia entre las transacciones de un árbol**:
   Cada transacción dentro de un conjunto de transacciones anidadas (árbol/subárbol) no debe verse afectada por los efectos parciales del resto de transacciones en el conjunto.

BLOQUEO EN TRANSACCIONES ANIDADAS

- Regla 1 (Independencia entre conjuntos de transacciones):
  - Los bloqueos adquiridos por una subtransacción se *pasan* a su progenitora cuando termina. A su vez, ésta lo pasará a su antecesora, y así sucesivamente hasta la raíz.
  - La transacción de alto nivel es la responsable de liberar los bloqueos.
  - Los bloqueos nuevos obtenidos por una subtransacción que aborta son sencillamente descartados. Sólo los bloqueos heredados pasan hacia arriba.
  - Se garantiza así la independencia entre árboles de transacciones, sólo las transacciones raíz tienen la potestad de liberar bloqueos, una vez confirmadas.

BLOQUEO EN TRANSACCIONES ANIDADAS

- Regla 2 (Independencia dentro del conjunto):
  - Cada progenitora se bloquea hasta que terminen sus hijas, que se ejecutan concurrentemente.
  - Si una transacción progenitora tiene un bloqueo sobre un objeto, ésta retiene el bloqueo mientras las hijas están en ejecución. Las hijas pueden usar estos bloqueos, pero han de serializar el acceso a los recursos comunes (usando p.e. el protocolo de 2-fases).

BLOQUEO EN TRANSACCIONES ANIDADAS

Reglas de bloqueo en operaciones de lectura/escritura:

1.- Una subtransacción sólo puede adquirir un bloqueo de lectura sobre un objeto o recurso si no hay otra transacción activa con un bloqueo de escritura sobre dicho objeto o recurso. Sólo pueden tener *retenidos* bloqueos de escritura sus antecesoras.

2.- Una subtransacción sólo puede adquirir un bloqueo de escritura sobre un objeto o recurso si no hay otra transacción activa con un bloqueo de lectura o escritura sobre dicho objeto o recurso. Sólo pueden tener *retenidos* bloqueos de lectura o escritura sus antecesoras.

BLOQUEO EN TRANSACCIONES ANIDADAS

Reglas de bloqueo en operaciones de lectura/escritura:

1.- Una subtransacción sólo puede adquirir un bloqueo de lectura sobre un objeto o recurso si no hay otra transacción activa con un bloqueo de escritura sobre dicho objeto o recurso. Sólo pueden tener *retenidos* bloqueos de escritura sus antecesoras.

2.- Una subtransacción sólo puede adquirir un bloqueo de escritura sobre un objeto o recurso si no hay otra transacción activa con un bloqueo de lectura o escritura sobre dicho objeto o recurso. Sólo pueden tener *retenidos* bloqueos de lectura o escritura sus antecesoras.

BLOQUEO EN TRANSACCIONES ANIDADAS

Reglas de bloqueo en operaciones de lectura/escritura (II):

3.- Cuando se confirma provisionalmente una subtransacción, su progenitora adquiere (*hereda*) sus bloqueos.

4.- Cuando aborta una subtransacción se descartan sus bloqueos, pero si la progenitora los tenía retenidos, ésta mantendrá estos bloqueos.

- **Estas reglas garantizan la serialización en el acceso a los recursos por parte de las subtransacciones, cuando éstas usan los bloqueos retenidos por sus progenitoras.**

BLOQUEO EN TRANSACCIONES ANIDADAS

Reglas de bloqueo en operaciones de lectura/escritura (II):

3.- Cuando se confirma provisionalmente una subtransacción, su progenitora adquiere (*hereda*) sus bloqueos.

4.- Cuando aborta una subtransacción se descartan sus bloqueos, pero si la progenitora los tenía retenidos, ésta mantendrá estos bloqueos.

- Estas reglas garantizan la serialización en el acceso a los recursos por parte de las subtransacciones, cuando éstas usan los bloqueos retenidos por sus progenitoras.

BLOQUEO EN TRANSACCIONES ANIDADAS

Reglas de bloqueo en operaciones de lectura/escritura (II):

3.- Cuando se confirma provisionalmente una subtransacción, su progenitora adquiere (*hereda*) sus bloqueos.

4.- Cuando aborta una subtransacción se descartan sus bloqueos, pero si la progenitora los tenía retenidos, ésta mantendrá estos bloqueos.

- **Estas reglas garantizan la serialización en el acceso a los recursos por parte de las subtransacciones, cuando éstas usan los bloqueos retenidos por sus progenitoras.**

BLOQUEOS MUTUOS

- Las técnicas anteriores no evitan el problema potencial de que se produzcan DEADLOCKS.
- Tenemos que ver cómo tratarlos.

BLOQUEOS MUTUOS

- El uso de bloqueos sobre objetos o recursos puede generar *deadlocks*.
- Construcción del *grafo de esperas*, como en los sistemas centralizados:

  $T \rightarrow U$ si y sólo si T espera que U libere un bloqueo

- Un bucle implica que existe un bloqueo mutuo entre las transacciones del ciclo: $T \rightarrow T_1 \rightarrow T_2 \ldots T_{n-1} \rightarrow T_n = T$.
- Solución: eliminación de algunas transacciones, hasta que no haya bucles.

BLOQUEOS MUTUOS

- El uso de bloqueos sobre objetos o recursos puede generar *deadlocks*.
- Construcción del *grafo de esperas*, como en los sistemas centralizados:

  $T \rightarrow U$ si y sólo si T espera que U libere un bloqueo

- Un bucle implica que existe un bloqueo mutuo entre las transacciones del ciclo: $T \rightarrow T_1 \rightarrow T_2 \ldots T_{n-1} \rightarrow T_n = T$.
- Solución: eliminación de algunas transacciones, hasta que no haya bucles.

BLOQUEOS MUTUOS

- El uso de bloqueos sobre objetos o recursos puede generar *deadlocks*.
- Construcción del *grafo de esperas*, como en los sistemas centralizados:

  $T \rightarrow U$ si y sólo si T espera que U libere un bloqueo

- Un bucle implica que existe un bloqueo mutuo entre las transacciones del ciclo: $T \rightarrow T_1 \rightarrow T_2 \ldots T_{n-1} \rightarrow T_n = T$.
- Solución: eliminación de algunas transacciones, hasta que no haya bucles.

BLOQUEOS MUTUOS

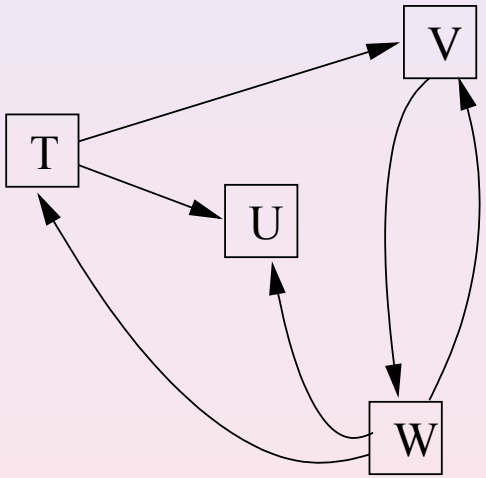
- El uso de bloqueos sobre objetos o recursos puede generar *deadlocks*.
- Construcción del *grafo de esperas*, como en los sistemas centralizados:

  $T \rightarrow U$ si y sólo si T espera que U libere un bloqueo

- Un bucle implica que existe un bloqueo mutuo entre las transacciones del ciclo: $T \rightarrow T_1 \rightarrow T_2 \ldots T_{n-1} \rightarrow T_n = T$.
- Solución: eliminación de algunas transacciones, hasta que no haya bucles.

BLOQUEOS MUTUOS

Podemos tener varios arcos desde una misma transacción: bloqueos de lectura del objeto sobre el que pretende escribir:



T, U, V: Comparten bloqueo de lectura sobre un objeto.

T : Intenta obtener bloqueo de escritura sobre ese objeto.

W: Intenta obtener un bloqueo de escritura sobre ese objeto también.

V : Intenta obtener un bloqueo sobre un objeto que tiene bloqueado W.

BLOQUEOS MUTUOS

Tratamiento de bloqueos mutuos:

- **Prevención**: como en sistemas centralizados, negando una de las 4 condiciones de bloqueo: Exclusión mutua, Retener y esperar, No-expropiación y Espera circular.
  - Negación de la espera circular: Ordenación de las peticiones, todas las transacciones piden los recursos en el mismo orden.

    Problemas: Hay que reservar algunos recursos anticipadamente para mantener el orden, y no siempre es fácil saber con certeza los recursos que nos harán falta.
- **Evitación**: es inaplicable en sistemas distribuidos, dada la enorme cantidad de información que habría que transmitir para poder tomar las decisiones en cada momento.

BLOQUEOS MUTUOS

Tratamiento de bloqueos mutuos:

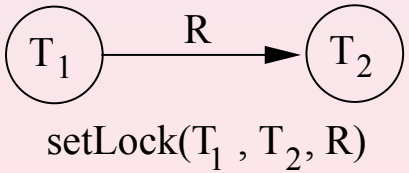
- **Prevención**: como en sistemas centralizados, negando una de las 4 condiciones de bloqueo: Exclusión mutua, Retener y esperar, No-expropiación y Espera circular.
  - Negación de la espera circular: Ordenación de las peticiones, todas las transacciones piden los recursos en el mismo orden.

    Problemas: Hay que reservar algunos recursos anticipadamente para mantener el orden, y no siempre es fácil saber con certeza los recursos que nos harán falta.
- **Evitación**: es inaplicable en sistemas distribuidos, dada la enorme cantidad de información que habría que transmitir para poder tomar las decisiones en cada momento.

BLOQUEOS MUTUOS

Tratamiento de bloqueos mutuos:

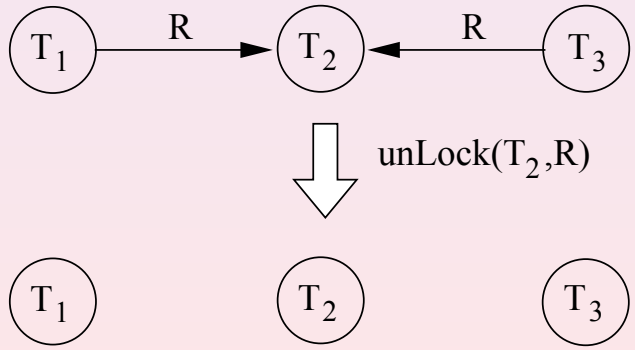
- **Detección y curación**:
  Se construye el grafo de esperas, y se ejecuta un algoritmo de detección de bucles. El Gestor de bloqueos es el responsable de mantener el grafo:
  - Dos operaciones soportadas: setLock y unLock.
  - Bloqueo de una transacción: se invoca a *setLock* para añadir un arco (anotado con el recurso afectado).



BLOQUEOS MUTUOS

Tratamiento de bloqueos mutuos:

- **Detección y curación**:
  - Liberación de bloqueo de un recurso: se invoca a *unLock* para eliminar los arcos incidentes sobre la transacción que estén anotados con ese recurso.



BLOQUEOS MUTUOS

¿Cómo detectamos los bucles en el grafo de esperas?:

- Comprobación de bucle: cada vez que se añade un arco, o bien periódicamente, para evitar sobrecargar el sistema con estas comprobaciones.

- Ejemplo: algoritmo de recorrido en profundidad (DFS): usa una pila con los nodos ya visitados a través de una búsqueda recursiva, y se busca un nodo que ya se encuentre en la pila para detectar un bucle. Una vez detectado para.

  Complexity: $O(|V|+|E|)$.

CONSTRUCCIÓN DEL GRAFO DE ESPERAS

Versión centralizada:

- Coordinador: recoge la información de bloqueos de los diferentes objetos y gestiona centralizadamente el grafo.
- Cuando detecta un bucle, el coordinador decide qué transacciones deben abortarse.
- Problemas típicos del servidor central:
  - Menor tolerancia a fallos (debe preverse el reemplazo del servidor en caso de fallo).
  - Se convierte en un cuello de botella.

CONSTRUCCIÓN DEL GRAFO DE ESPERAS

Versión centralizada:

- Coordinador: recoge la información de bloqueos de los diferentes objetos y gestiona centralizadamente el grafo.
- Cuando detecta un bucle, el coordinador decide qué transacciones deben abortarse.
- Problemas típicos del servidor central:
  - Menor tolerancia a fallos (debe preverse el reemplazo del servidor en caso de fallo).
  - Se convierte en un cuello de botella.

CONSTRUCCIÓN DEL GRAFO DE ESPERAS

Versión centralizada:

- Coordinador: recoge la información de bloqueos de los diferentes objetos y gestiona centralizadamente el grafo.
- Cuando detecta un bucle, el coordinador decide qué transacciones deben abortarse.
- Problemas típicos del servidor central:
  - Menor tolerancia a fallos (debe preverse el reemplazo del servidor en caso de fallo).
  - Se convierte en un cuello de botella.

BLOQUEOS FANTASMAS

- Los algoritmos de detección de bloqueos mutuos en sistemas distribuidos a veces producen falsos positivos.
- Debido a los retrasos (variables) en la transmisión de mensajes y a la ausencia de un reloj global: no podemos definir estados globales exactos.
- A veces se producen bucles en el grafo de esperas que no corresponden a bloqueos mutuos reales.
- Ejemplo:

BLOQUEOS FANTASMAS

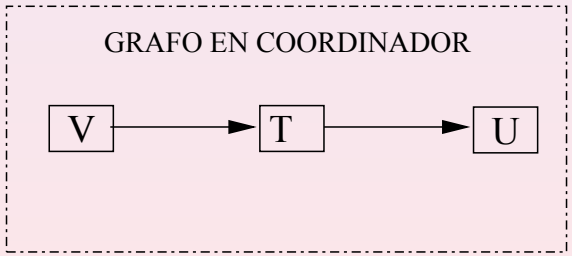
- Los algoritmos de detección de bloqueos mutuos en sistemas distribuidos a veces producen falsos positivos.
- Debido a los retrasos (variables) en la transmisión de mensajes y a la ausencia de un reloj global: no podemos definir estados globales exactos.
- A veces se producen bucles en el grafo de esperas que no corresponden a bloqueos mutuos reales.
- Ejemplo:

BLOQUEOS FANTASMAS

- Los algoritmos de detección de bloqueos mutuos en sistemas distribuidos a veces producen falsos positivos.
- Debido a los retrasos (variables) en la transmisión de mensajes y a la ausencia de un reloj global: no podemos definir estados globales exactos.
- A veces se producen bucles en el grafo de esperas que no corresponden a bloqueos mutuos reales.
- Ejemplo:

BLOQUEOS FANTASMAS

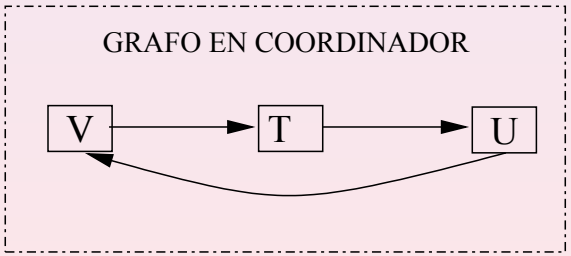
- Los algoritmos de detección de bloqueos mutuos en sistemas distribuidos a veces producen falsos positivos.
- Debido a los retrasos (variables) en la transmisión de mensajes y a la ausencia de un reloj global: no podemos definir estados globales exactos.
- A veces se producen bucles en el grafo de esperas que no corresponden a bloqueos mutuos reales.
- Ejemplo:



U libera el objeto que quiere T y pide uno que tiene V.

BLOQUEOS FANTASMAS

- Los algoritmos de detección de bloqueos mutuos en sistemas distribuidos a veces producen falsos positivos.
- Debido a los retrasos (variables) en la transmisión de mensajes y a la ausencia de un reloj global: no podemos definir estados globales exactos.
- A veces se producen bucles en el grafo de esperas que no corresponden a bloqueos mutuos reales.
- Ejemplo:

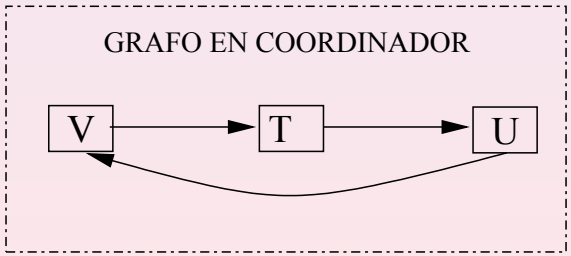


U libera el objeto que quiere T y pide uno que tiene V.

El mensaje de petición llega primero al coordinador, y el de liberación está en tránsito.

BLOQUEOS FANTASMAS

- Los algoritmos de detección de bloqueos mutuos en sistemas distribuidos a veces producen falsos positivos.
- Debido a los retrasos (variables) en la transmisión de mensajes y a la ausencia de un reloj global: no podemos definir estados globales exactos.
- A veces se producen bucles en el grafo de esperas que no corresponden a bloqueos mutuos reales.
- Ejemplo:



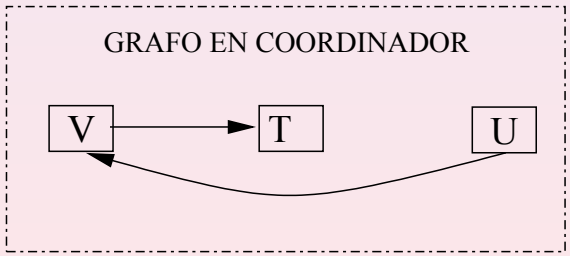
U libera el objeto que quiere T y pide uno que tiene V.

El mensaje de petición llega primero al coordinador, y el de liberación está en tránsito.

SE PODRIA ABORTAR INNECESARIAMENTE UNA TRANSACCION

BLOQUEOS FANTASMAS

- Los algoritmos de detección de bloqueos mutuos en sistemas distribuidos a veces producen falsos positivos.
- Debido a los retrasos (variables) en la transmisión de mensajes y a la ausencia de un reloj global: no podemos definir estados globales exactos.
- A veces se producen bucles en el grafo de esperas que no corresponden a bloqueos mutuos reales.
- Ejemplo:



U libera el objeto que quiere T y pide uno que tiene V.

El mensaje de petición llega primero al coordinador, y el de liberación está en tránsito.

SI HUBIERA LLEGADO PRIMERO EL DE LIBERACION NO HABRIA BUCLE

CAZA DE ARCOS

ALGORITMO DISTRIBUIDO DE DETECCION DE BLOQUEOS

[Edge chasing, Sinha & Natarajan, 1985]

- No es necesario construir el grafo de esperas en un nodo centralizado.
- Cada nodo contiene su grafo de esperas local.
- La unión de estos grafos locales de esperas es el grafo global, pero no se construye.
- Para comprobar si hay bucles se envían mensajes a otros nodos (*pruebas*): corresponden a caminos en el grafo global.

CAZA DE ARCOS

¿ Cuándo se envían las pruebas?

- Al insertar un arco en el grafo local, que pudiese cerrar un bucle en el hipotético grafo completo.
- IDEA: Al insertar localmente un arco $X \xrightarrow{R} Y$:

  Si Y está a su vez bloqueada esperando un recurso en el nodo N, se envía una *prueba* a N.
- La respuesta será si existe un bucle o no.

CAZA DE ARCOS

¿Quiénes se encargan de gestionar el estado de las transacciones y de los recursos?

- Coordinador transacciones: mantiene su estado (bloqueada o activa).

  Puede haber un coordinador central o un coordinador por transacción (nodo en el que se inició, normalmente).

- Gestor de objeto: En el nodo del objeto. Informan a los coordinadores de transacciones cuando éstas quedan bloqueadas a la espera del objeto, o cuando lo obtienen.

CAZA DE ARCOS

¿Quiénes se encargan de gestionar el estado de las transacciones y de los recursos?

- Coordinador transacciones: mantiene su estado (bloqueada o activa).

  Puede haber un coordinador central o un coordinador por transacción (nodo en el que se inició, normalmente).

- Gestor de objeto: En el nodo del objeto. Informan a los coordinadores de transacciones cuando éstas quedan bloqueadas a la espera del objeto, o cuando lo obtienen.

CAZA DE ARCOS

¿Quiénes se encargan de gestionar el estado de las transacciones y de los recursos?

- Coordinador transacciones: mantiene su estado (bloqueada o activa).

  Puede haber un coordinador central o un coordinador por transacción (nodo en el que se inició, normalmente).

- Gestor de objeto: En el nodo del objeto. Informan a los coordinadores de transacciones cuando éstas quedan bloqueadas a la espera del objeto, o cuando lo obtienen.

CAZA DE ARCOS

Fases del algoritmo de caza de arcos:

1.- **Iniciación**: T pide un objeto O en nodo N, que posee U.

Se incluye $T \xrightarrow{O} U$ en grafo local de N. Entonces:

- Si U no está bloqueada: no hay bloqueo.
- Si U bloqueada sobre O' (en N), comprobar bucle en grafo local de N.
- Si U bloqueada sobre O' (en nodo N'), desde gestor de objeto O en N se envía la prueba $T \xrightarrow{O} U$ al nodo N'.

Bloqueos compartidos: la prueba se envía a todos los nodos N_i afectados. ▸ Ver Ejemplo Anterior

CAZA DE ARCOS

2.- **Detección**: Gestor de O' en N' recibe una prueba $T_1 \xrightarrow{O_1} T_2 \ldots \xrightarrow{O_{n-1}} T_n$:

- Si T_n no bloqueada, responde OK (no bloqueo).
- Si T_n está bloqueada sobre O' local, añade arco $T_n \xrightarrow{O'} T_{n+1}$ a la prueba (T_{n+1} posee O'):
  - Si se genera un bucle, responde indicándolo.
  - Si no hay bucle, y T_{n+1} no bloqueada, responde OK.
  - Si no hay bucle, y T_{n+1} bloqueada, propaga la prueba extendida al nodo correspondiente.
- Bloqueos compartidos: se propaga la prueba a todos los nodos afectados: T_n espera sobre varias T^j_{n+1}.

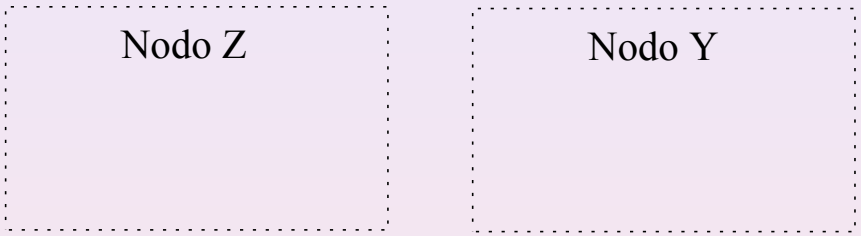
CAZA DE ARCOS

Fases del algoritmo de caza de arcos:

3.- **Resolución**: Detectado un bucle, se elige una transacción y se aborta:

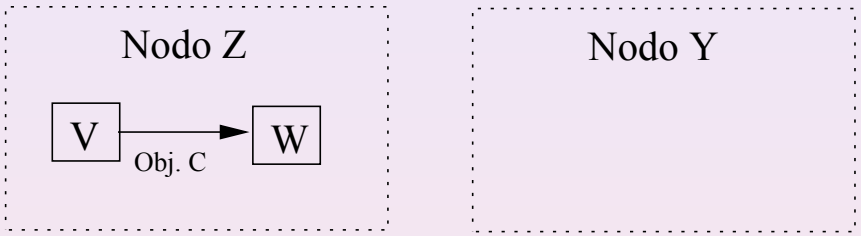
- Su coordinador informa a todos los nodos afectados, para que eliminen sus bloqueos sobre los objetos que posee, y eliminen los arcos correspondientes de sus grafos locales.

EJEMPLO DE CAZA DE ARCOS



V, W : No bloqueadas.

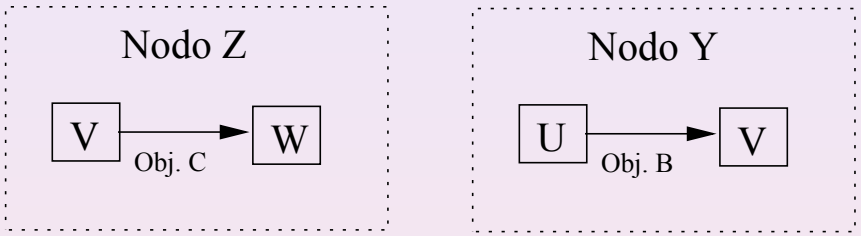
EJEMPLO DE CAZA DE ARCOS



V, W : No bloqueadas.

1) V pide C, que posee W. Se inserta arco V –> W sin problemas

EJEMPLO DE CAZA DE ARCOS



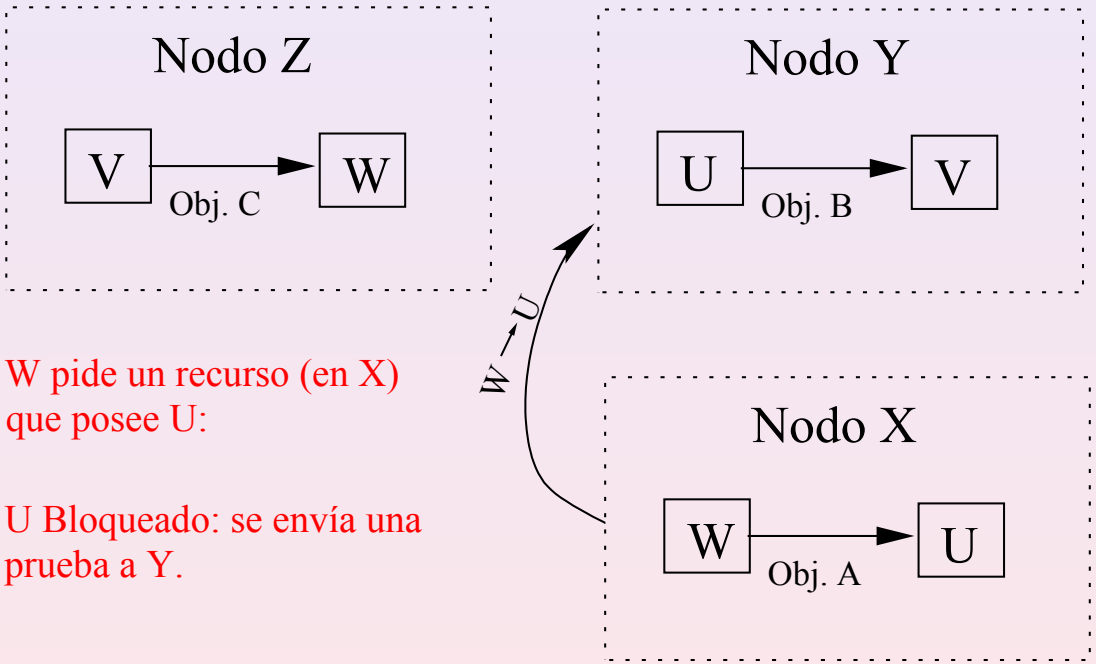
V bloqueada, W no bloqueada.

1) V pide C, que posee W. Se inserta arco V −> W sin problemas

2) U pide B, que posee V. Al insertar U −> V, V bloqueado ==> se envia la prueba la Z

3) Se construye U −> V −> W, no hay bloqueo.

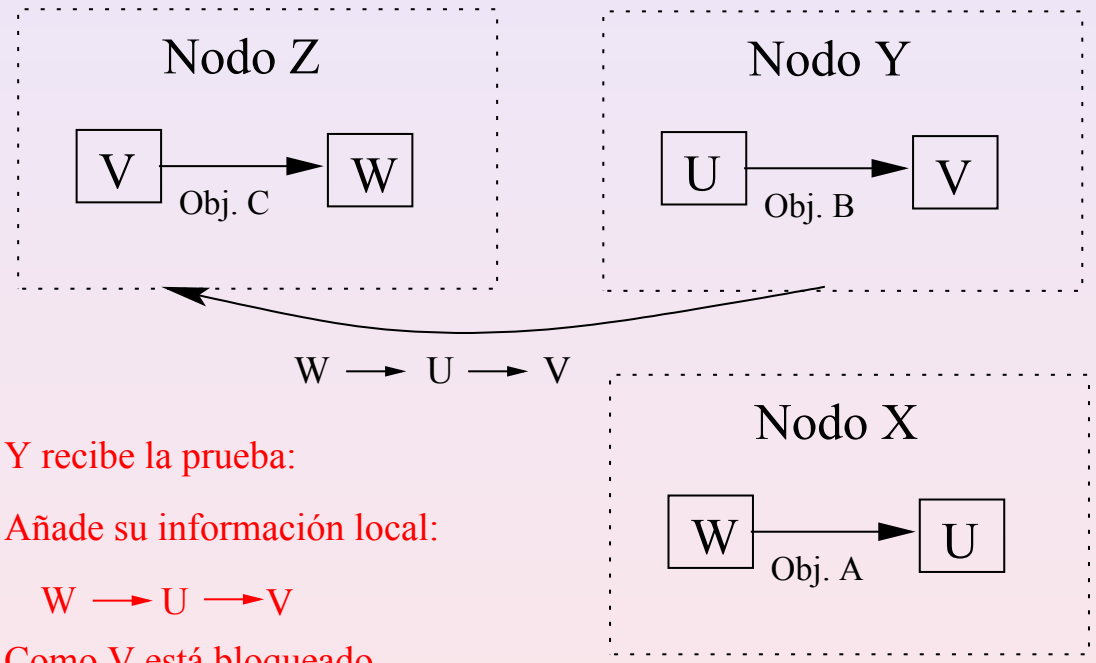
EJEMPLO DE CAZA DE ARCOS



W pide un recurso (en X) que posee U:

U Bloqueado: se envía una prueba a Y.

EJEMPLO DE CAZA DE ARCOS



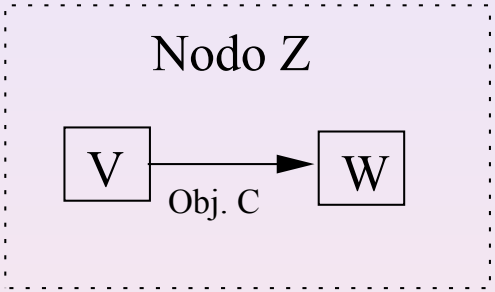
Y recibe la prueba:

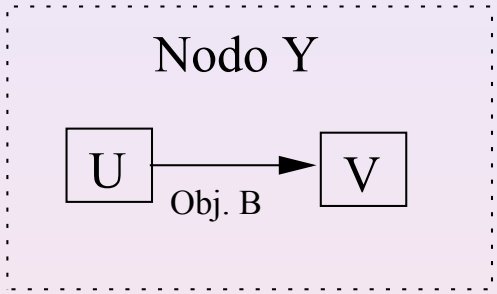
Añade su información local:

W → U → V

Como V está bloqueado,
envía la prueba a Z.

EJEMPLO DE CAZA DE ARCOS



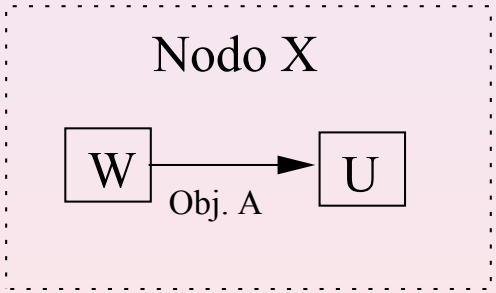


Z recibe la prueba:

Añade su información local:

W → U → V → W

Deadlock detectado en Z.



BLOQUEOS MUTUOS

¿ Qué transacción debe ser eliminada?

- Puede atenderse a diversos criterios, p.e. edad de la transacción (tiempo de ejecución).
  - Eliminar la más joven en principio tiene la ventaja de que deshacer sus efectos parciales será menos costoso.
  - Eliminar la más vieja: para beneficiar a las transacciones cortas.
- Por prioridades, abortando la de menor prioridad.
  Puede darse una ordenación total de prioridades en base a algún criterio.

BLOQUEOS MUTUOS

Detección de bloqueos mutuos con time-outs:

- En la práctica queremos soluciones rápidas.
- Cuando se bloquea un recurso se le asocia un tiempo máximo de utilización (lease). Sólo puede mantenerse el bloqueo por ese tiempo. En caso de deadlock obviamente el recurso no es liberado y vencerá ese tiempo.
- Podrían comprobarse únicamente si hay otras transacciones demandando el recurso:

  Si ha vencido el tiempo se libera el recurso y la transacción es abortada.

BLOQUEOS MUTUOS

Inconvenientes del uso de time-outs:

- Pueden abortarse transacciones sin que exista ningún bloqueo mutuo: una transacción puede necesitar un recurso más tiempo que el asociado al *time-out*.
- Se puede renovar el *lease* para resolver el problema anterior.
- Es difícil elegir adecuadamente el tiempo máximo de bloqueo de un recurso u objeto, sobre todo cuando el sistema está muy cargado.

INDICE:

VISIÓN OPTIMISTA DE LA CONCURRENCIA

[Kung & Robinson, 1981]

La visión optimista se introduce debido a los problemas siguientes derivados del uso de bloqueos:

- La gestión de los bloqueos de recursos representa una carga adicional al sistema (control de bloqueos).
- Los bloqueos de recursos provocan *deadlocks*. Las soluciones a los mismos son costosas.
- Los bloqueos sólo se liberan al final de las transacciones para evitar continuos abortos de transacciones $\Rightarrow$ reducción severa del nivel de concurrencia.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

Visión optimista:

- La probabilidad de interferencias entre transacciones normalmente es baja, no suelen utilizar simultáneamente los mismos recursos.
- IDEA: Dejar proceder a las transacciones hasta llegar al punto de confirmación sin ocuparnos de bloquear recursos.
- CONFIRMACIÓN: Si se detecta que ha existido algún conflicto, una transacción es abortada (quizás distinta de la confirmada).

VISIÓN OPTIMISTA DE LA CONCURRENCIA

Examen → lectura sobre qué copia (confirmada? / tentativa) + en qué fase

Tres fases en la ejecución de las transacciones:

FASE DE TRABAJO	FASE DE VALIDACION	FASE DE ACTUALIZACION
LECTURAS DE RECURSOS ESCRITURAS EN COPIAS TENTATIVAS	¿HAY CONFLICTOS CON OTRAS TRANSACCIONES?	ESCRITURA SOBRE ORIGINALES DE RECURSOS

PASO DEL TIEMPO

COPIA CONFIRMADA

VISIÓN OPTIMISTA DE LA CONCURRENCIA

- *Fase de trabajo:*

  Se genera una versión tentativa con la primera operación de escritura sobre un recurso (usando su último valor confirmado):

  - Operaciones de lectura: sobre la versión tentativa, si ya existe. Si no, sobre la última versión confirmada del objeto.
  - Operaciones de escritura: sobre la versión tentativa, si existe. Si no, se crea la versión tentativa.

  Nota: podemos tener varias versiones tentativas de un mismo recurso (una por transacción).

VISIÓN OPTIMISTA DE LA CONCURRENCIA

closeTransaction:

- *Fase de validación:* Se analiza si hay conflictos con otras transacciones. Si los hay, alguna transacción debe ser abortada, y sus copias tentativas descartadas.

- *Fase de actualización:* una vez validada, se vuelcan (escrituras) los cambios de las versiones tentativas sobre los objetos originales. Así sólo podemos tener conflictos en la etapa de actualización.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

closeTransaction:

- *Fase de validación:* Se analiza si hay conflictos con otras transacciones. Si los hay, alguna transacción debe ser abortada, y sus copias tentativas descartadas.

- *Fase de actualización:* una vez validada, se vuelcan (escrituras) los cambios de las versiones tentativas sobre los objetos originales. Así sólo podemos tener conflictos en la etapa de actualización.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

- Cada transacción tiene sus propias copias tentativas de los objetos o recursos:
  - Desde el punto de vista de la serialización las escrituras en copias tentativas no pueden estar en conflicto con otras transacciones (son “recursos” diferentes).
  - El resultado de la ejecución de la transacción individualmente es el mismo, es como diferir las escrituras sobre los objetos a la parte final de su ejecución (fase de actualización).
- Se crea un historial de cada transacción (lecturas y escrituras realizadas) para poder realizar la validación.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

- Cada transacción tiene sus propias copias tentativas de los objetos o recursos:
  - Desde el punto de vista de la serialización las escrituras en copias tentativas no pueden estar en conflicto con otras transacciones (son “recursos” diferentes).
  - El resultado de la ejecución de la transacción individualmente es el mismo, es como diferir las escrituras sobre los objetos a la parte final de su ejecución (fase de actualización).
- Se crea un historial de cada transacción (lecturas y escrituras realizadas) para poder realizar la validación.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

HISTORIAL POR TRANSACCIÓN

Log transacciones

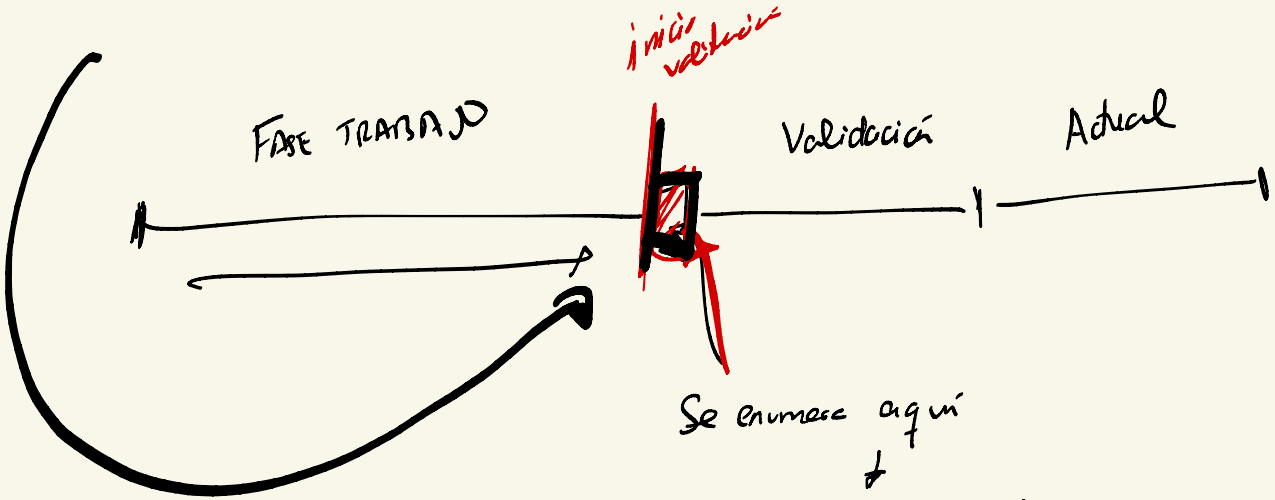
- *Read-set*: Conjunto de objetos leidos por la transacción.
- *Write-set*: Conjunto de objetos sobre los cuales ha escrito. Un objeto sobre el que sólo ha escrito no estará en su *Read-set*.

  NOTA: No puede haber lecturas sucias, porque las transacciones solo leen valores confirmados (o copias de ellos actualizadas por ellas mismas).

Log Tr → Read Set = Todos Obj que se están leyendo
→ Write Set = Todos " " escribiendo

∩ interacción entre estos = Conflictos

¿Cuándo se enumeran?



Se enumera aquí

Tare 1, .. 2, .. N
cada una tarda
X, Y, ...

esto garantiza orden

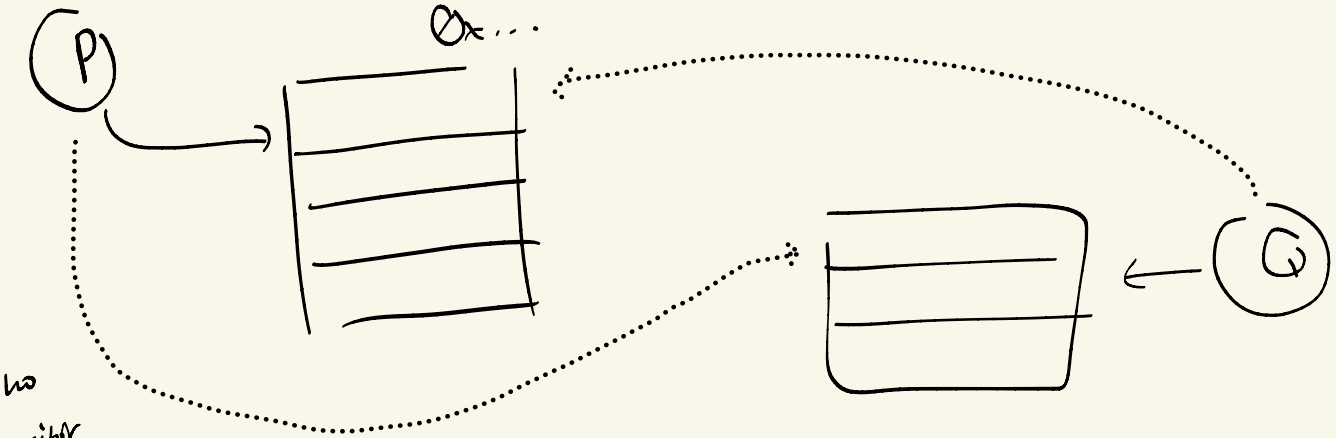
VISIÓN OPTIMISTA DE LA CONCURRENCIA

- Las transacciones se numeran al entrar en la fase de validación. Se confirman siguiendo este orden.
- Se asigna al entrar en la validación porque si se hiciese antes (al inicio) sería ineficiente: una transacción que hubiese acabado su fase de trabajo tendría que esperar su turno.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

Validación de T_v:

- Se comprueba el historial de las T_i que se solapan con ella durante su ejecución.
- Las fases de validación y actualización suelen requerir poco tiempo:
  - La actualización en concreto puede basarse en la reasignación de punteros o referencias a objetos (de la copia tentativa al original).
- Simplificación: se unen y supondremos una sola transacción realizándolas en exclusión mutua.
- Consecuencia: todas las ejecuciones serán estrictas (no hay lecturas sucias ni tampoco escrituras prematuras).



Cambio de puntero, no regiones escribir en memoria

Fase de Trabajo Concurrentemente?

Sí, se debería

Validación

No ....

Pregunta examen

VISIÓN OPTIMISTA DE LA CONCURRENCIA

REGLAS DE VALIDACIÓN PARA T_v :

IMPORTANTE

T_v y T_i se solapan en su ejecución, aunque no pueden solaparse en su validación/actualización. Tenemos que comprobar las 3 reglas de conflictos entre lecturas y escrituras:

(R1) T_i no puede leer un objeto r escrito por T_v:

$$\nexists r \in Write\_set(T_V) \text{ tal que } r \in Read\_set(T_i)$$

(R2) T_v no puede leer un objeto r escrito por T_i:

$$\nexists r \in Write\_set(T_i) \text{ tal que } r \in Read\_set(T_v)$$

CONTROL OPTIMISTA DE LA CONCURRENCIA

REGLA 3: ¿EXISTE CONFLICTO WRITE/WRITE?

No, desde el punto de vista de la serialización, el conflicto sólo se puede dar con dos recursos r, r' y dos escrituras realizadas en orden inverso sobre ellos. Ejemplo:

$$write(T_i, r); write(T_v, r); write(T_v, r'); write(T_i, r')$$

Esta situación no puede ocurrir, debido a la exclusión mutua en la fase de validación/actualización.

Debe ser algo así:

$$write(T_i, r); write(T_i, r'); termina(T_i); write(T_v, r); write(T_v, r'); termina(T_v)$$

CONTROL OPTIMISTA DE LA CONCURRENCIA

- Sólo tenemos que comprobar las reglas 1-2.
- Se puede demostrar que cumplir ambas reglas es condición suficiente para que las computaciones de las transacciones sean serializables.
- Nota: para dos transacciones que sólo realizan escrituras sobre los mismos recursos (ninguna lectura), el orden de validación establece el orden de la correspondiente serialización, y por tanto, el resultado final de los recursos (como si sólo se hubiese realizado la última de ellas).

CONTROL OPTIMISTA DE LA CONCURRENCIA

- Sólo tenemos que comprobar las reglas 1-2.
- Se puede demostrar que cumplir ambas reglas es condición suficiente para que las computaciones de las transacciones sean serializables.
- Nota: para dos transacciones que sólo realizan escrituras sobre los mismos recursos (ninguna lectura), el orden de validación establece el orden de la correspondiente serialización, y por tanto, el resultado final de los recursos (como si sólo se hubiese realizado la última de ellas).

CONTROL OPTIMISTA DE LA CONCURRENCIA

- Sólo tenemos que comprobar las reglas 1-2.
- Se puede demostrar que cumplir ambas reglas es condición suficiente para que las computaciones de las transacciones sean serializables.
- Nota: para dos transacciones que sólo realizan escrituras sobre los mismos recursos (ninguna lectura), el orden de validación establece el orden de la correspondiente serialización, y por tanto, el resultado final de los recursos (como si sólo se hubiese realizado la última de ellas).

VISIÓN OPTIMISTA DE LA CONCURRENCIA

Técnicas de comprobación reglas 1-2:

- Hacia atrás (backwards): mirando las que entraron antes en la validación.
- Hacia delante (forward): mirando las que aún no han entrado en la validación.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Validación hacia atrás de T_v:

- Tenemos que comprobar los conflictos con transacciones T_i ya validadas cuando T_v inicia su fase de validación/actualización:
  - T_i ya está confirmada y todas las operaciones de lectura de T_i se hicieron antes de iniciar la validación de T_v (cuando se hacen las escrituras en los recursos originales) $\Rightarrow$ la regla 1 no causa problemas.

    El orden serial equivalente es: $T_i; T_v$.
  - Sólo hay que comprobar la regla 2.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Validación hacia atrás de T_v:

- Tenemos que comprobar los conflictos con transacciones T_i ya validadas cuando T_v inicia su fase de validación/actualización:
- T_i ya está confirmada y todas las operaciones de lectura de T_i se hicieron antes de iniciar la validación de T_v (cuando se hacen las escrituras en los recursos originales) $\Rightarrow$ la regla 1 no causa problemas.

  El orden serial equivalente es: $T_i; T_v$.
- Sólo hay que comprobar la regla 2.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Validación hacia atrás de T_v:

- Tenemos que comprobar los conflictos con transacciones T_i ya validadas cuando T_v inicia su fase de validación/actualización:
- T_i ya está confirmada y todas las operaciones de lectura de T_i se hicieron antes de iniciar la validación de T_v (cuando se hacen las escrituras en los recursos originales) $\Rightarrow$ la regla 1 no causa problemas.

  El orden serial equivalente es: $T_i; T_v$.
- Sólo hay que comprobar la regla 2.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Validación hacia atrás de T_v:

- ¿Cómo comprobamos la regla 2?

  Conflicto: si tenemos un recurso r leído por T_v y escrito por T_i.

- Podemos tener en el log marcas de tiempo asociadas a las operaciones de lectura/escritura:

  En ese caso, observamos que las operaciones de lectura de T_v realizadas después de validar T_i no causan problemas (si no son sobre copias tentativas) $\Rightarrow$ sólo comprobamos que para un mismo recurso no haya operaciones de lectura de T_v (no realizadas sobre copias tentativas) anteriores que la escritura (actualización) de ese recurso por T_i.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Validación hacia atrás de T_v:

- ¿Cómo comprobamos la regla 2?

  Conflicto: si tenemos un recurso r leído por T_v y escrito por T_i.

- Podemos tener en el log marcas de tiempo asociadas a las operaciones de lectura/escritura:

  En ese caso, observamos que las operaciones de lectura de T_v realizadas después de validar T_i no causan problemas (si no son sobre copias tentativas) $\Rightarrow$ sólo comprobamos que para un mismo recurso no haya operaciones de lectura de T_v (no realizadas sobre copias tentativas) anteriores que la escritura (actualización) de ese recurso por T_i.

CONTROL OPTIMISTA DE LA CONCURRENCIA

No hay problema porque no hay lecturas sucias.

Validación hacia atrás de T_v:

- OBSERVACIÓN: la validación es inmediata para aquellas transacciones T_v que solo hagan operaciones de escritura (ninguna lectura).

CONTROL OPTIMISTA DE LA CONCURRENCIA

Validación hacia delante de T_v:

- Debemos comprobar aquellas transacciones T_i que están todavía en su fase de trabajo.
- La regla 2 no causa problemas: las operaciones de escritura (actualización de originales) de T_i son posteriores a las lecturas de T_v (mismos recursos), y T_v ya está confirmada.

  Orden serial equivalente: $T_v; T_i$.
- Sólo hay que comprobar la regla 1.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Validación hacia delante de T_v:

- Debemos comprobar aquellas transacciones T_i que están todavía en su fase de trabajo.
- La regla 2 no causa problemas: las operaciones de escritura (actualización de originales) de T_i son posteriores a las lecturas de T_v (mismos recursos), y T_v ya está confirmada.

  Orden serial equivalente: $T_v; T_i$.
- Sólo hay que comprobar la regla 1.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Validación hacia delante de T_v:

- Debemos comprobar aquellas transacciones T_i que están todavía en su fase de trabajo.
- La regla 2 no causa problemas: las operaciones de escritura (actualización de originales) de T_i son posteriores a las lecturas de T_v (mismos recursos), y T_v ya está confirmada.

  Orden serial equivalente: $T_v; T_i$.
- Sólo hay que comprobar la regla 1.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Validación hacia delante de T_v:

- ¿Cómo comprobar la regla 1?

  Conflicto: T_i lee un recurso r escrito por T_v (fase de actualización).

- Con marcas de tiempo en las operaciones:

  Si una operación de lectura de T_i es anterior a una escritura de T_v (mismo recurso), la validación falla. Las operaciones de lectura de T_i posteriores a la validación de T_v no son problemáticas si no se hacen sobre copias tentativas (se lee un valor confirmado).

CONTROL OPTIMISTA DE LA CONCURRENCIA

Resolución de conflictos cuando T_v no puede validarse:

- En validación hacia delante podemos retrasar la validación de T_v, con la esperanza de que las T_i en conflicto aborten (otras causas), y así se resuelva el conflicto.

  No hay garantía de que finalmente podrá validarse.

- En validación hacia delante se puede abortar T_v o las T_i en conflicto con ella.

- En validación hacia delante, una vez abortada T_v, si luego las T_i en conflicto abortasen (otras causas), este aborto de T_v habría sido innecesario.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Resolución de conflictos cuando T_v no puede validarse:

- En validación hacia delante podemos retrasar la validación de T_v, con la esperanza de que las T_i en conflicto aborten (otras causas), y así se resuelva el conflicto.

  No hay garantía de que finalmente podrá validarse.

- En validación hacia delante se puede abortar T_v o las T_i en conflicto con ella.

- En validación hacia delante, una vez abortada T_v, si luego las T_i en conflicto abortasen (otras causas), este aborto de T_v habría sido innecesario.

CONTROL OPTIMISTA DE LA CONCURRENCIA

Resolución de conflictos cuando T_v no puede validarse:

- En validación hacia delante podemos retrasar la validación de T_v, con la esperanza de que las T_i en conflicto aborten (otras causas), y así se resuelva el conflicto.

  No hay garantía de que finalmente podrá validarse.

- En validación hacia delante se puede abortar T_v o las T_i en conflicto con ella.

- En validación hacia delante, una vez abortada T_v, si luego las T_i en conflicto abortasen (otras causas), este aborto de T_v habría sido innecesario.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

Comparativa de ambas técnicas:

- En la validación hacia atrás el fallo sólo se trata abortando T_v.
- Costes de comparación de los conjuntos:
  - Hacia atrás: $|Read\_set(T_v)| \times \sum_i |Write\_set(T_i)|$.
  - Hacia delante: $|Write\_set(T_v)| \times \sum_i |Read\_set(T_i)|$.
- Los conjuntos de lecturas suelen ser significativamente mayores que los de escritura.
- La comparativa depende de estos tamaños y del número de transacciones solapadas con T_v.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

Comparativa de ambas técnicas:

- En la validación hacia atrás el fallo sólo se trata abortando T_v.
- Costes de comparación de los conjuntos:
  - Hacia atrás: $|Read\_set(T_v)| \times \sum_i |Write\_set(T_i)|$.
  - Hacia delante: $|Write\_set(T_v)| \times \sum_i |Read\_set(T_i)|$.
- Los conjuntos de lecturas suelen ser significativamente mayores que los de escritura.
- La comparativa depende de estos tamaños y del número de transacciones solapadas con T_v.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

Comparativa de ambas técnicas:

- En la validación hacia atrás el fallo sólo se trata abortando T_v.
- Costes de comparación de los conjuntos:
  - Hacia atrás: $|Read\_set(T_v)| \times \sum_i |Write\_set(T_i)|$.
  - Hacia delante: $|Write\_set(T_v)| \times \sum_i |Read\_set(T_i)|$.
- Los conjuntos de lecturas suelen ser significativamente mayores que los de escritura.
- La comparativa depende de estos tamaños y del número de transacciones solapadas con T_v.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

Comparativa de ambas técnicas:

- En la validación hacia atrás el fallo sólo se trata abortando T_v.
- Costes de comparación de los conjuntos:
  - Hacia atrás: $|Read\_set(T_v)| \times \sum_i |Write\_set(T_i)|$.
  - Hacia delante: $|Write\_set(T_v)| \times \sum_i |Read\_set(T_i)|$.
- Los conjuntos de lecturas suelen ser significativamente mayores que los de escritura.
- La comparativa depende de estos tamaños y del número de transacciones solapadas con T_v.

VISIÓN OPTIMISTA DE LA CONCURRENCIA

Starvation:

- Las transacciones abortadas normalmente son reiniciadas inmediatamente o al cabo de un cierto tiempo (para evitar repetir los mismos conflictos que han provocado el aborto).
- Pueden ser abortadas reiteradas veces, cayendo en *starvation*.
- Solución: Cuando han sido abortadas N veces, se les da acceso exclusivo a los objetos a los que acceden.

INDICE:

GESTOR DE RECUPERACIÓN

Un gestor de recuperación en cada nodo se encarga de:

- Salvar el estado (confirmado) de los objetos/recursos en dicho nodo sobre un fichero de recuperación (log).
- Restablecer un estado previo de los objetos (desde el fichero de recuperación) en caso de fallo o aborto de transacción *(rollback)*.
- Organizar eficientemente el fichero de recuperación y garantizar su disponibilidad mediante técnicas de almacenamiento estable.

FICHERO DE RECUPERACIÓN

LOG: Histórico de valores de los objetos o recursos:

- Contiene *checkpoints*: valores confirmados de todos los objetos. Se incluyen periódicamente para facilitar la recuperación.
- El log además almacena un registro histórico del estado de los objetos y de las transacciones.
- Las transacciones aparecen en el orden en que han pasado a *preparadas para confirmar*, *confirmadas* o *abortadas* .

FICHERO DE RECUPERACIÓN

LOG: Histórico de valores de los objetos o recursos:

- Contiene *checkpoints*: valores confirmados de todos los objetos. Se incluyen periódicamente para facilitar la recuperación.
- El log además almacena un registro histórico del estado de los objetos y de las transacciones.
- Las transacciones aparecen en el orden en que han pasado a *preparadas para confirmar*, *confirmadas* o *abortadas* .

FICHERO DE RECUPERACIÓN

LOG: Histórico de valores de los objetos o recursos:

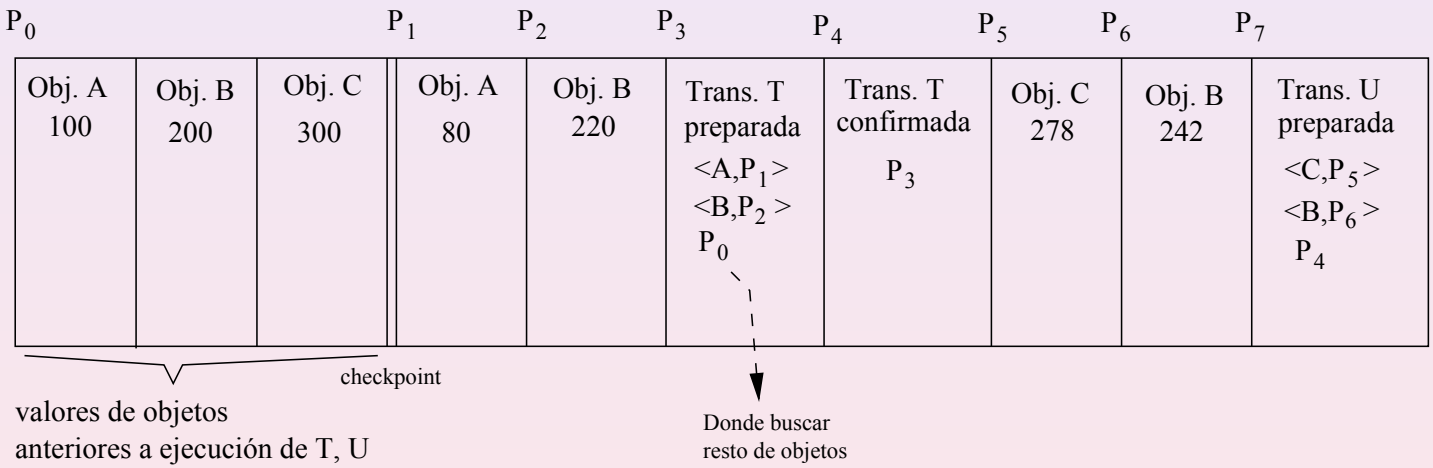
- Contiene *checkpoints*: valores confirmados de todos los objetos. Se incluyen periódicamente para facilitar la recuperación.
- El log además almacena un registro histórico del estado de los objetos y de las transacciones.
- Las transacciones aparecen en el orden en que han pasado a *preparadas para confirmar*, *confirmadas* o *abortadas* .

FICHERO DE RECUPERACIÓN

- Cuando una transacción está preparada para confirmarse se añaden al fichero (en cada nodo afectado por ella):
  - Valores de los objetos que ha modificado.
  - Estado de la transacción (preparada) y referencias (offsets) a los objetos que ha utilizado.
- Transacción confirmada o abortada:
  - Se añade un registro con el nuevo estado de la transacción.

FICHERO DE RECUPERACIÓN

Ejemplo:



FICHERO DE RECUPERACIÓN

- Si *U* abortase incluiríamos una entrada indicándolo.
- Restablecemos los valores de *C* y *B* (los que ha modificado) a su último valor confirmado: yendo hacia atrás, buscando transacciones confirmadas o hasta llegar a un *checkpoint*:

B	Pos. $P_2 => 220$
C	Pos. $P_0 => 300$

RECUPERACIÓN DE OBJETOS

CRASH: Para recuperar los valores de los objetos, o bien:

- Partiendo del último *checkpoint*, actualizando los valores de los objetos para las transacciones que fueron confirmadas, en el orden en que aparecen.
- O leyendo el fichero hacia atrás, desde su final, restaurando los valores confirmados de los objetos, hasta haber recuperado un valor confirmado para todos ellos (una vez recuperado el último valor confirmado de un objeto, éste es el valor válido, se sigue hacia atrás sólo si quedan otros objetos que aún no han sido restaurados).

RECUPERACIÓN

¿Qué hacemos al restaurar los objetos con entradas incompletas?

Para las entradas de transacciones que encuentra marcadas como *preparadas* : incluye al final del fichero entradas de la forma *abortadas*.

SISTEMAS DISTRIBUIDOS

EJERCICIOS DEL TEMA 6

1. Un servidor gestiona una serie de objetos $o_1, o_2, \ldots, o_n$, facilitando a sus clientes dos operaciones para gestionarlos:

   - *read(i)*: Devuelve el valor de o_i.
   - *write(i, valor)*: Asigna el valor actual de o_i.

   Si tenemos las dos transacciones siguientes:

   T: x=read(j); y=read(i); write(j,44); write(i,33);
   U: x=read(k); write(i,55); y=read(j); write(k,66);

   Se pide: proporcionar tres computaciones serializables equivalentes entre sí de T y U.

2. Tomando las dos transacciones del ejercicio anterior, indicar dos computaciones **estrictas** de las mismas, identificando los puntos de confirmación de las transacciones (última operación de las mismas). Recordar que para que sean estrictas las lecturas y escrituras sobre cada recurso deben ser siempre posteriores a la terminación (confirmación) de aquellas otras transacciones que hayan modificado el recurso.

3. Indica, justificadamente, si la computación siguiente es serializable, donde el subíndice indica qué transacción realiza la operación:

   $$C = W_2(X); R_1(X); W_2(Y); R_3(X); W_2(Z); R_1(Y); R_2(X); W_2(Z); W_1(X); W_3(Y).$$

4. Consideremos las dos transacciones siguientes:

   T : *a.withdraw(4); b.deposit(4);*
   U : *c.withdraw(3); b.deposit(3);*

   Y supongamos ahora que cada una de ellas se divide en dos subtransacciones:

   T_1 : *a.withdraw(4)* y T_2 : *b.deposit(4);*
   U_1 : *c.withdraw(3)* y U_2 : *b.deposit(3);*

   Se pide, comparar las computaciones serializables que podemos obtener considerando T y U directamente con las que pueden obtenerse tomando las subtransacciones. ¿Por qué al tomar el modelo de transacciones anidadas se obtiene un mayor número de computaciones serializables?.

5. Consideremos tres transacciones T_1, T_2, T_3, y la siguiente computación de las mismas:

   $$C = open_1; R_1(X); R_1(Y); open_2; R_2(X); open_3; R_3(Y); R_2(X); R_2(Y); W_1(Y); close_1; R_2(Y); close_2; W_3(Y); close_3$$

1. Un servidor gestiona una serie de objetos $o_1, o_2, \ldots, o_n$, facilitando a sus clientes dos operaciones para gestionarlos:

   - *read(i)*: Devuelve el valor de o_i.
   - *write(i, valor)*: Asigna el valor actual de o_i.

   Si tenemos las dos transacciones siguientes:

   T: *x=read(j); y=read(i); write(j,44); write(i,33);*
   U: *x=read(k); write(i,55); y=read(j); write(k,66);*

   Se pide: proporcionar tres computaciones serializables equivalentes entre sí de T y U.

① PASOS → Serializar → todos T antes que U

② → encuentra CONFLICTOS / LECTURAS SUCIAS
                          \ ESCRITURAS

CONFLICTOS → una operación de lectura sucia

ejemplo T: y = read(i) } ¡ conflicto !
        U: write(i,55)

Escrituras Prematuras

ejemplo T: write(i,33); } conflicto !
        U: write(i,55);

1ª Computación

T; U; ① computación.

o/

U; T;

elegir una

X este no es equivalente si cojo el primero

2ª Computación

el truco es leo r(j); r(u) → y ya cojo resto T y resto U

Si son lecturas, los demás no cambia valor y cumple

③ cojo r(u), r(j) → e idem T y U.

2. Tomando las dos transacciones del ejercicio anterior, indicar dos computaciones **estrictas** de las mismas, identificando los puntos de confirmación de las transacciones (última operación de las mismas). Recordar que para que sean estrictas las lecturas y escrituras sobre cada recurso deben ser siempre posteriores a la terminación (confirmación) de aquellas otras transacciones que hayan modificado el recurso.

Solución Agustín pedir foto

Serializable = ejecución misma que serial

3. Indica, justificadamente, si la computación siguiente es serializable, donde el subíndice indica qué transacción realiza la operación:

$$C = W_2(X); R_1(X); W_2(Y); R_3(X); W_2(Z); R_1(Y); R_2(X); W_2(Z); W_1(X); W_3(Y).$$

dicpo 13

Es necesario un modelo de lecturas-escrituras para conocer las situaciones de conflicto entre operaciones de transacciones diferentes:

Op. de T1	Op. de T2	¿Conflicto?	Motivo
Read	Read	NO ok	No se altera el valor leído
Read	Write	SI	El efecto de la escritura afecta a la lectura
Write	Write	SI	Importa el orden de las escrituras

T_1: $R_1(x)$, $R_1(y)$, $W_1(x)$

T_2: $W_2(x)$; $W_2(y)$; $W_2(z)$; $R_2(x)$; $W_2(z)$

T_3: $R_3(x)$; $W_3(y)$

1º Saco Transacciones

CONFLICTOS

WW: en $W_2(x)$ y $W_1(x)$
en $W_2(y)$ y $W_3(y)$

RW: en $R_2(x)$ y $W_2(x)$
en $R_1(y)$ y $W_2(y)$
en $R_3(x)$ y $W_1(x)$

X = ~~1~~ 2 Y = 1 Z = ~~1~~ 2

condiciones para que sea SERIALIZABLE

→ El resultado de la ejecución secuencial es diferente a C, por ejemplo 2 escrituras en X así que no sería serializable.

FORMA PROFE

más rápido

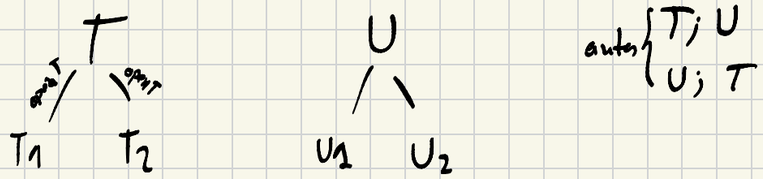
Pares órdenes Y: (2;1 ok, 2;3 ok, 1;3) , X (2;1 ok, 2;3 ok, 2;1 indep, 3;1)

en Z no hay conflictos 1;3 3;1

se invierte el orden así que no serializable

(4)

Un único conflicto a [b] → b.deposita(4)
                        → b.deposita(3)



T1; T2
no se garantiza

(5) Orden (marca tiempo) T1; T2; T3

Escritura W1(y)
↑ se encuentra al final después de haber empezado T2, T3
→ T1 debería haber hecho antes

→ La forma de solucionarlo es haciendo todos antes.

Suponiendo que se utiliza la ordenación por marcas de tiempo, indicar si es posible realizar todas sus operaciones en el orden indicado, y caso contrario, cuáles será posible realizar.

NOP

6. Explicar por qué el protocolo de dos fases garantiza la equivalencia serial. Tomando las dos transacciones siguientes, T, U, indicar una computación de las mismas (identificando los puntos de bloqueo y desbloqueo de los recursos) en la que no se respete el protocolo de 2 fases, tal que no sea serializable.

   $T: x = read_T(i);\ write_T(j, 44);$
   $U: write_U(i, 55);\ write_U(j, 66);$

NOP

7. Consideremos una relajación del protocolo de 2-fases tal que las transacciones que sólo realizan lecturas pueden liberar sus bloqueos anticipadamente, antes de obtener nuevos bloqueos. ¿Podrá el sistema alcanzar estados inconsistentes, no presentes en una ejecución serializada de las transacciones?.

   Para illustrar la respuesta considerar el ejemplo siguiente, basado en un paso a nivel con semáforo y barrera, que va lanzando iterativamente instancias de las transacciones *Monitor*, *C1, C2* (*Control*) y *Alarma*, que se emplean, respectivamente, para monitorizar, controlar y avisar de problemas en la sincronización de barrera y semáforo. Las transacciones gestionan dos recursos, s y b, ambos con valor 0 para indicar semáforo y barrera cerrados, y 1 para indicar que ambos están abiertos.

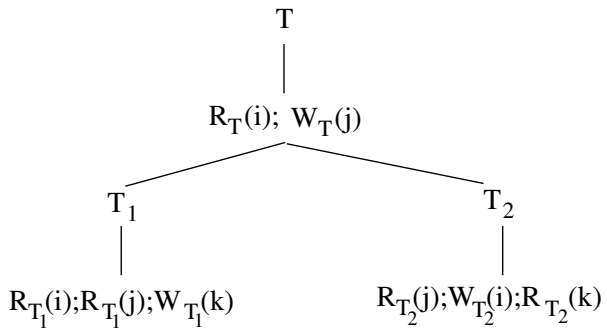
$$
\begin{array}{l}
Monitor = for(;;)\{ \\
\qquad open_M; Lock(s); ms := R_M(s); Lock(b); mb := R_M(b); \\
\qquad Unlock(s); Unlock(b); close_M; \quad \ldots Muestra(ms, mb) \ \ldots \\
\} \\
Control = for(;;)\{ \quad \ldots\ldots\ (Llega\ \ Tren)\ \ \ldots\ldots \\
\qquad open_{C_1}; Lock(s); W_C(s, 0); Lock(b); W_C(b, 0); Unlock(s); Unlock(b); \\
\qquad close_{C_1} \ldots\ldots open_{C_2}; Lock(s); W_C(s, 1); Lock(b); W_C(b, 1); \\
\qquad Unlock(s); Unlock(b); close_{C_2}; \\
\} \\
Alarma = for(;;)\{ \\
\qquad open_A; Lock(s); as := R_A(s); Lock(b); ab := R_A(b); \\
\qquad Unlock(s); Unlock(b); close_A; \\
\qquad if\ ((as == 1)\ \&\&\ (ab == 0)\ ||\ (as == 0)\ \&\&\ (ab == 1))\{ \\
\qquad\quad < LANZAR\ \ ALARMA > \\
\qquad \} \\
\}
\end{array}
$$

   Suponiendo que inicialmente s y b son 1, indicar si estas transacciones cumplen el protocolo de 2-fases y si hay deadlocks. Analizar entonces lo que sucede al cambiar el $Unlock(s)$ de la transacción A (*Alarma*) a la posición inmediatamente anterior a $Lock(b)$. Finalmente, ¿Qué sucede si por ejemplo conmutamos de orden los bloqueos (*Locks*) de M (*Monitor*) ?.

8. Consideremos la transacción siguiente, con las subtransacciones indicadas:

$$T: \quad open_T;\ R_T(i);\ W_T(j);\ open_{T_1};\ R_{T_1}(i);\ R_{T_1}(j);\ W_{T_1}(k);\ close_{T_1};$$
$$open_{T_2};\ R_{T_2}(j);\ W_{T_2}(i);\ R_{T_2}(k);\ close_{T_2};\ close_T;$$

Gráficamente, se corresponde con la estructura siguiente:



Considerando que ambas subtransacciones se inician simultáneamente y que emplean el protocolo de bloqueo de 2 fases estricto con bloqueos de lectura/escritura, indicar una computación terminal (sin *deadlocks*) de T. Debe indicarse en qué puntos se adquieren bloqueos (y su tipo), en qué puntos se producen las esperas por las subtransacciones, en qué puntos pueden continuar su ejecución, y en qué puntos se liberan recursos. Finalmente, determinar si existe alguna computación de T que termine en *deadlock*.

9. Supongamos que tenemos 3 recursos, A, B y C, en tres nodos diferentes de un sistema distribuido, N_A, N_B y N_C. Consideremos 3 transacciones, U, V, T que quieren realizar las siguientes operaciones, en el orden indicado:

$$R_U(A); W_V(B); R_T(A); R_V(A); R_U(C); R_U(A); R_T(C); W_U(A); R_T(B); W_V(C)$$

Donde las R indican lecturas y las W escrituras sobre el recurso indicado entre paréntesis y la transacción que realiza la operación es la indicada mediante un subíndice. Suponiendo que no se libera ningún recurso que se haya ido adquiriendo, que se emplean bloqueos de lectura/escritura, y que se usa la técnica de caza de arcos para la detección de bloqueos mutuos. Se pide: indicar cómo evolucionan los grafos locales de esperas operación a operación, e indicar si algunas de ellas no pueden completarse, así como si se llega en algún momento a un deadlock.

10. Consideremos las tres transacciones siguientes, donde se indican con subíndices los instantes en los que se realizan las operaciones indicadas:

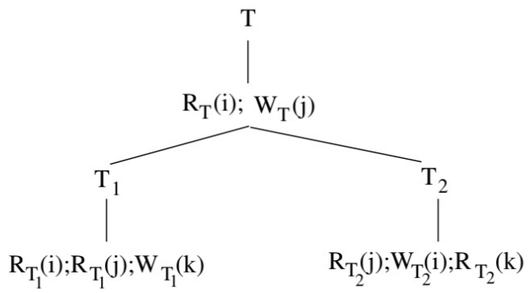
$$T = open_0;\ R_1(X);\ R_7(Y);\ W_8(X);\ close_{10}$$
$$U = open_2;\ R_2(Y);\ W_{11}(Y);\ R_{13}(X);\ close_{15}$$
$$Z = open_9;\ R_{12}(X);\ W_{14}(X);\ close_{17}$$

Donde la operación *close* corresponde a la etapa de validación y confirmación, realizada en exclusión mutua (sólo una transacción puede estar realizándola en cada momento). Aplicar la técnica de validación hacia atrás para determinar si su ejecución es posible conforme a los instantes indicados en las operaciones, o bien alguna de ellas debe ser abortada.

8. Consideremos la transacción siguiente, con las subtransacciones indicadas:

$$T: \; open_T;\ R_T(i);\ W_T(j);\ open_{T_1};\ R_{T_1}(i);\ R_{T_1}(j);\ W_{T_1}(k);\ close_{T_1};\ open_{T_2};\ R_{T_2}(j);\ W_{T_2}(i);\ R_{T_2}(k);\ close_{T_2};\ close_T;$$

Gráficamente, se corresponde con la estructura siguiente:



Considerando que ambas subtransacciones se inician simultáneamente y que emplean el protocolo de bloqueo de 2 fases estricto con bloqueos de lectura/escritura, indicar una computación terminal (sin *deadlocks*) de T. Debe indicarse en qué puntos se adquieren bloqueos (y su tipo), en qué puntos se producen las esperas por las subtransacciones, en qué puntos pueden continuar su ejecución, y en qué puntos se liberan recursos. Finalmente, determinar si existe alguna computación de T que termine en *deadlock*.

BLOQUEO 2 FASES

① Adquisición → Liberar bloqueos

② Liberación → Adquirir bloqueos

cierre transacción

- RW lock
- WR lock

ESTE CAE EN ENERO

T

R_T W_T → se hacen y se bloquean

ACTIVA / BLOQUEADA

→ esta ACTIVA

Tipo Bloqueo	Objeto
R	i
W	i

T_1

① R	i
⑤ R	j
⑥ W	k

⑦ Termina

T_2

② R	j
④ W Sigue	i
⑧ R	k

Orden de explicación

Bloqueo ③

→ Se bloquea hasta que termine bloqueo R(i) en T_1.

→ esto te sigue rellenando cuando terminan T_1 y T_2

Se bloque pero sigue ACTIVA

→ al no acabe de escribir en mem hasta que termine T

T_1, T_2 se acaban provisionalmente.

→ si sólo lecturas no hay

Cómo compruebo que no hay deadlock → y mirando conflictos en mismo objeto

11. Consideremos las dos transacciones siguientes:

    $T : x = read_T(i);\ write_T(j, 44);$
    $U : write_U(i, 55);\ write_U(j, 66);$

    Y una ejecución simultánea de las mismas considerando un control optimista de concurrencia. Indicar la secuencia de operaciones que tienen lugar si:

    a) T pide la confirmación primero, y se usa la validación hacia atrás.

    b) U pide la confirmación primero, y se usa la validación hacia atrás.

    c) T pide la confirmación primero, y se usa la validación hacia delante.

    d) U pide la confirmación primero, y se usa la validación hacia delante.