# Capítulo 7
# Transacciones y Control de Concurrencia



*Sistemas Distribuidos*

Universidad Nacional de Asunción
Facultad Politécnica
Ingeniería Informática

***Ing. Fernando Mancía***

## Transacciones

**Transacción:** define una secuencia de operaciones que se realiza por el servidor y se garantiza por el mismo que es atómica, ya sea en presencia de múltiples usuarios e incluso de caídas del servidor.

Los servidores deben garantizar que se realiza completamente y que los resultados se almacenan en memoria permanente o no.

Cada transacción de un cliente es considerada invisible desde el punto de vista de otras transacciones, estas no pueden observar los efectos parciales.

## Sincronización simple (sin transacciones)

Sistemas cuidadosamente diseñados (que implican mayor costo y esfuerzo), aseguran que las operaciones realizadas en nombre de diferentes clientes no interfieren entre sí. En caso de no contemplarlo, esta interferencia puede resultar en valores incorrectos en los objetos. Se detalla cómo se pueden sincronizar las operaciones sin recurrir al manejo de transacciones.

**Operaciones atómicas en el servidor.**

- La utilización de múltiples hilos es beneficioso para las prestaciones.
- Permite que se ejecuten concurrentemente las operaciones de varios clientes y aún accediendo, posiblemente, a los mismos objetos.

## Sincronización simple (sin transacciones)

Ejemplo: si los métodos ***deposita*** *(deposit)* y ***extrae*** *(withdraw)* no están diseñados para su utilización en un programa multihilo, es posible que las acciones de dos o más ejecuciones concurrentes del método *puedan* entremezclarse arbitrariamente y tener efectos extraños en las variables de instancia de los objetos ***cuenta****.*

En Java, los métodos "`syncronized`" aseguran que solo puede acceder a un objeto un hilo cada vez. Si un hilo invoca un método sincronizado de un objeto, entonces el objeto es bloqueado efectivamente, y otro hilo que invoque uno de sus métodos sincronizados será bloqueado hasta que el bloqueo anterior sea liberado.

```
public synchronized void deposit(int amount) throws RemoteException{
    // adds amount to the balance of the account
}
```

## Sincronización simple (sin transacciones)

Esta forma de sincronización fuerza a que la ejecución de los hilos sea separada en el tiempo y asegura que las variables instancia de un único objeto sean accedidas de forma consistente.

Sin sincronización, dos invocaciones ***deposita*** separadas podrían leer el balance antes de que ninguna lo hubiera actualizado, lo que produciría un valor incorrecto. Cualquier método que acceda a una variable instancia que pueda variar debe ser sincronizada.

Las operaciones que están libres de interferencia de operaciones concurrentes que se están realizando en otros hilos se llaman ***operaciones atómicas***.

## Sincronización simple (sin transacciones)

**Mejora de la colaboración del cliente mediante sincronización de las operaciones del servidor.** Los clientes pueden utilizar un servidor como un medio de compartir algunos recursos.

Esto se consigue por algunos clientes utilizando operaciones para actualizar los objetos del servidor y otros utilizando operaciones para acceder a ellos.

El esquema anterior para acceso sincronizado a objetos proporciona todo lo que se precisa en muchas aplicaciones, previene que los hilos interfieran unos con otros. Sin embargo, algunas aplicaciones necesitan una forma para que los hilos se comuniquen unos con otros.

Por ejemplo, puede surgir una situación en la que la operación requerida por un cliente no puede completarse hasta que se haya realizado otra operación requerida por otro usuario.

## Sincronización simple (sin transacciones)

Esto puede ocurrir cuando algunos clientes son **productores** y otros **consumidores**, los consumidores deben esperar hasta que un productor haya proporcionado algún elemento más del *artículo en* cuestión. Puede también ocurrir cuando los clientes comparten un recurso, pueden necesitar esperar hasta que otros clientes lo liberen.

Los métodos ***wait*** (espera) y ***notify*** (notifica) de Java permiten que los hilos se comuniquen con los otros de una manera que resuelva los problemas anteriores.

## Sincronización simple (sin transacciones)

Deben utilizarse dentro de los métodos sincronizados de un objeto.

Un hilo llama a *wait* en un objeto para suspenderse él mismo y permitir a otro hilo ejecutar un método en ese objeto.

Un hilo llama a *notify* para informar que cualquier hilo que esté esperando en el objeto que ha cambiado algunos de sus datos.

El acceso a un objeto es todavía atómico cuando un hilo espera por otro: un hilo que llama a *wait* activa su bloqueo y se suspende como una acción atómica, cuando el hilo es activado después de ser notificado y adquiere un nuevo bloqueo en el objeto y recupera la ejecución del *wait.*

```
public class BlockingQueue<T> {

    private Queue<T> queue = new LinkedList<T>();
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void put(T element) throws InterruptedException {
        while(queue.size() == capacity) {
            wait();
        }
        queue.add(element);
        notify();
    }

    public synchronized T take() throws InterruptedException
    {
        while(queue.isEmpty()) {
            wait();
        }

        T item = queue.remove();
        notify();
        return item;
    }
}
```

## Operaciones de la interfaz Cuenta

*deposit(amount)*
deposit amount in the account
*withdraw(amount)*
withdraw amount from the account
*getBalance() -> amount*
return the balance of the account
*setBalance(amount)*
set the balance of the account to amount

---

Operations of the Branch interface

*create(name) -> account*
create a new account with a given name
*lookUp(name) -> account*
return a reference to the account with the given name
*branchTotal() -> amount*
return the total of all the balances at the branch

## Una transacción bancaria de un cliente

Un cliente, que realiza una secuencia de operaciones en una cuenta *particular* del banco por encargo de un usuario, realizará primero una búsqueda *(**busca**)* de la cuenta por nombre, después *aplicará* las *operaciones* de depositar *(deposita),* extraer *(extrae)* y obtener balance *(obténBalance)* directamente sobre la cuenta considerada.

En nuestros ejemplos, utilizamos cuentas con nombres *A, B* y *C.* El *cliente* las *localiza* y almacena las *referencias* a ellas en las variables *a, b* y *c* de *tipo* cuenta *(Cuenta).*

## Una transacción bancaria de un cliente

*Transaction T:*
*a.extrae(100);*
*b.deposita(100);*
*c.extrae(200);*
*b.deposita(200);*

La Figura muestra un ejemplo de la transacción de un cliente que especifica una serie de acciones relacionadas en las que están implicadas las cuentas *A, B* y C. La primera acción transfiere 100$ de *A* a B y la segunda 200$ de C a B. Un cliente consigue una operación de transferencia realizando un reintegro en una cuenta seguido de un depósito en la otra.

Las transacciones provienen de los sistemas de gestión de bases de datos. En ese contexto una transacción es la ejecución de un programa que accede a la base de datos.

## Transacciones

Las transacciones pueden venir dadas como una parte del middleware. En estos contextos una transacción se aplica a objetos recuperables y está pensada para ser atómica. Hay dos aspectos de atomicidad:

**Todo o nada:** Una transacción o finaliza correctamente, y los efectos de todas sus operaciones son registrados en los objetos, o (si falla o es abortada deliberadamente) no tiene ningún efecto. Este efecto *todo o nada* tiene otros dos aspectos en sí mismo:

*Atomicidad de fallo:* los efectos son atómicos aun en el caso de ruptura del servidor.

*Durabilidad:* después que una transacción ha finalizado con éxito, todos sus efectos son guardados en almacenamiento permanente. Utilizamos el término "almacenamiento permanente" para referirnos a archivos que se mantienen en disco o cualquier otro soporte de datos guardados en disco sobrevivirán incluso en el caso de ruptura del servidor.

## Transacciones

**Aislamiento:** Cada transacción debe ser realizada sin interferencia de otras transacciones, es decir, los efectos intermedios de una transacción no deben ser visibles para los demás.

Para dar soporte al requisito de atomicidad de fallo y durabilidad, los objetos deben ser recuperables, cuando un proceso servidor cae inesperadamente, debido a un fallo hardware o a un error software, los cambios debidos a todas las transacciones completadas deben estar disponibles en el almacenamiento permanente de forma que cuando el servidor sea reemplazado por un nuevo proceso, se pueden recuperar los objetos para reflejar el efecto *todo o nada.*

Cada vez que un servidor reconoce la finalización de una transacción del cliente, todos los cambios de la transacción en los objetos deben haber sido registrados en almacenamiento permanente.

## ACID properties

Härder and Reuter [1983] suggested the mnemonic 'ACID' to remember the properties of transactions, which are as follows:

**A**tomicity: a transaction must be all or nothing;

**C**onsistency: a transaction takes the system from one consistent state to another consistent state;

**I**solation;

**D**urability.

## Transacciones

Se pueden añadir recursos a transacciones a los servidores de objetos recuperables.

Cada transacción es creada y gestionada por un administrador, que implementa la interfaz *Coordinador* mostrada en la Figura siguiente.

El coordinador da a cada transacción un identificador, o TID. El cliente invoca el método *abreTransacción* del coordinador para introducir una nueva transacción, se asigna un TID, y se devuelve. Al final de una transacción, el cliente invoca el método *cierraTransacción* para indicar su fin; todos los objetos recuperables accedidos durante la transacción debieran ser guardados.

Si, por alguna razón el cliente quiere abortar la transacción, invoca el método *abortaTransacción*, y todos los efectos debieran ser eliminados.

## Figura 9.3
## Operaciones en la interfaz Coordinador

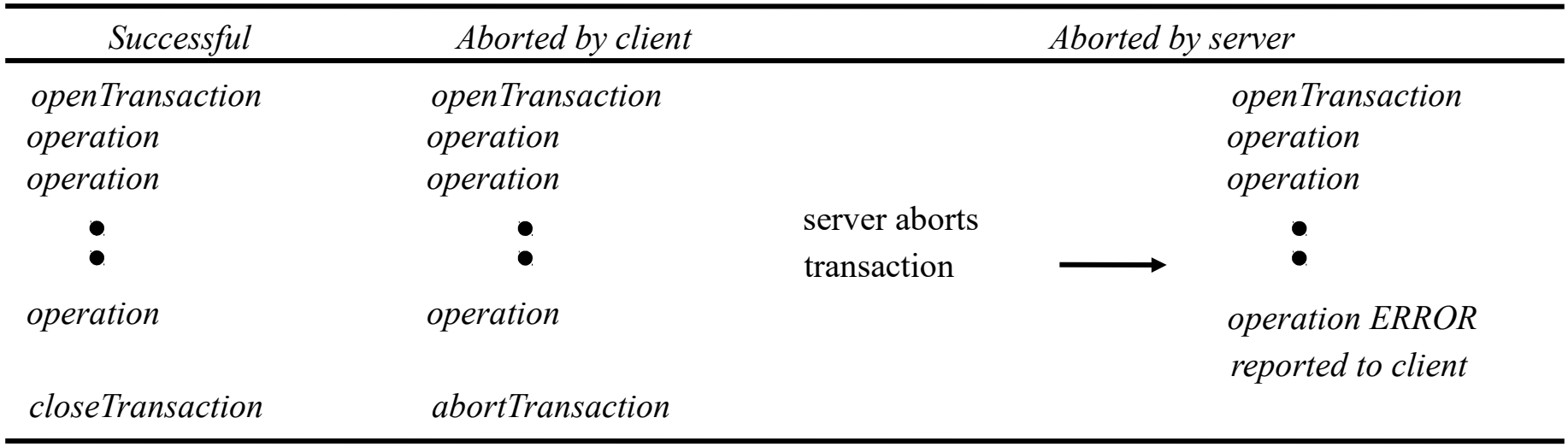
*openTransaction() -> trans;*
Comineza una nueva transacción y proporciona un TID *trans*. Este identificador será utilizado en el resto de las operaciones de la transacción.

*closeTransaction(trans) -> (commit, abort);*
Finaliza una transacción: el retorno de un valor commit significa que la transacción fue consumada. *Abort,* significa que la transacción a abortado.

---

*abortTransaction(trans);*
Aborta la transaccónn.

# Figura 9.4
# Historias de vida de una transacción



**El problema de las actualizaciones perdidas.** Este problema se muestra mediante el siguiente par de transacciones sobre las cuentas bancarias *A, B* y C, cuyos balances iniciales son 100$, 200$ y 300$ respectivamente. La transacción *T* transfiere cierta cantidad desde la cuenta *A* a la cuenta *B.* La transacción *U* transfiere otra cantidad desde la cuenta C a la *B.* En ambos casos, la cantidad transferida se calcula para incrementar el balance de B en un 10%. Los efectos netos sobre la cuenta B al ejecutar las transacciones *T* y *U* debieran hacer incrementar el balance de la cuenta *B* en un 10% dos veces, por lo que el valor final será 242$.

Consideremos ahora los 20$. El resultado es incorrecto, al incrementar el balance de efectos de permitir que las transacciones *T* y *U* se ejecuten concurrentemente, como en la Figura 9.5. Ambas transacciones obtiene el balance de B como 200$ y después depositan la cuenta B en 20$ en lugar de 42$. Esto es un ejemplo del problema de las «actualizaciones perdidas». La actualización de *U* se pierde porque *T* escribe sin mirar. Ambas transacciones han leído el valor previo antes de escribir el nuevo valor.

## Figura 9.5
## El problema de las actualizaciones perdidas

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *balance = b.getBalance();* | | *balance = b.getBalance();* | |
| *b.setBalance(balance*1.1);* | | *b.setBalance(balance*1.1);* | |
| *a.withdraw(balance/10)* | | *c.withdraw(balance/10)* | |
| *balance = b.getBalance();* | $200 | | |
| | | *balance = b.getBalance();* | $200 |
| | | *b.setBalance(balance*1.1);* | $220 |
| *b.setBalance(balance*1.1);* | $220 | | |
| *a.withdraw(balance/10)* | $80 | | |
| | | *c.withdraw(balance/10)* | $280 |

**Recuperaciones inconsistentes.** La Figura *9.6* muestra otro ejemplo relacionado con una cuenta bancaria en la que *la* transacción *V* transfiere una suma desde la cuenta *A* hasta la B y la transacción *W* invoca el método *totalSucursal* para obtener la suma de los balances de todas las cuentas del banco. Los balances de dos cuentas bancarias, *A* y *B,* son ambos inicialmente 200$. El resultado de *totalSucursal* incluyen la suma de *A* y *B* como 300$, que es incorrecto.

Esto es un ejemplo del problema de las «recuperaciones inconsistentes». Las recuperaciones de *W* son inconsistentes porque *V* había realizado *sólo* la parte de extracción de su transferencia mientras se calculaba la suma.

## Figura 9.6
## El problema de las recuperaciones inconsistentes

| **Transaction *V*:**<br>*a.withdraw(100)*<br>*b.deposit(100)* | | **Transaction *W*:**<br>*aBranch.branchTotal()* | |
|---|---|---|---|
| *a.withdraw(100);* | $100 | | |
| | | *total = a.getBalance()* | $100 |
| | | *total = total+b.getBalance()* | $300 |
| | | *total = total+c.getBalance()* | |
| *b.deposit(100)* | $300 | ⋮ | |

**Equivalencia secuencial.** Si se sabe que cada una de las distintas transacciones tiene el efecto correcto cuando se realiza ella sola, podemos inferir que si estas transacciones se realizan una cada vez en el mismo orden, el efecto combinado también será correcto.

Un solapamiento de las operaciones de las transacciones en las que el efecto combinado es el mismo que si las transacciones hubieran sido realizadas una cada vez en el mismo orden es un solapamiento secuencialmente equivalente.

Cuando decimos que dos transacciones diferentes tienen el mismo efecto, significa que las operaciones devuelven los mismos valores y que las variables de instancia de objetos tienen los mismos valores al final.

## Control de concurrencia

*El* uso de la equivalencia secuencial como un criterio para la ejecución concurrente previene la ocurrencia de actualizaciones perdidas y recuperaciones inconsistentes.

El problema de las actualizaciones perdidas ocurre cuando dos transacciones leen el valor antiguo de una variable y la utilizan para calcular el nuevo valor. Esto no puede ocurrir si una transacción se realiza antes que otra, porque la última transacción leerá el valor escrita por la anterior.

Como un solapamiento secuencialmente independiente de dos transacciones produce el mismo efecto de una secuencial, podemos resolver el problema de las actualizaciones perdidas mediante la equivalencia secuencial

**Solapamiento 1:** La Figura 9.7 muestra dicho solapamiento en el que las operaciones que afectan a la cuenta compartida *B,* son realmente secuenciales, puesto que la transacción *T* realiza todas las operaciones antes que las realice la transacción *U.*

Otro solapamiento de *T* y *U* que tenga esta propiedad es uno en el que la transacción *U* finaliza sus operaciones en la cuenta *B* antes de que comience la transacción *T.*

## Figura 9.7
## Un solapamiento de T y U secuencialmente equivalente

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *balance = b.getBalance()* | | *balance = b.getBalance()* | |
| *b.setBalance(balance*1.1)* | | *b.setBalance(balance*1.1)* | |
| *a.withdraw(balance/10)* | | *c.withdraw(balance/10)* | |
| *balance = b.getBalance()* | $200 | | |
| *b.setBalance(balance*1.1)* | $220 | | |
| | | *balance = b.getBalance()* | $220 |
| | | *b.setBalance(balance*1.1)* | $242 |
| *a.withdraw(balance/10)* | $80 | | |
| | | *c.withdraw(balance/10)* | $278 |

**Solapamiento 2:** Equivalencia secuencial con relación al problema de las recuperaciones inconsistentes. En la que la transacción V está transfiriendo una suma de desde la cuenta *A* a la cuenta *B* y la transacción *W está* obteniendo la suma de todos los balances.

El problema de las recuperaciones inconsistentes puede ocurrir cuando una transacción de recuperación se ejecuta concurrentemente con una transacción de actualización. No puede ocurrir si la transacción de recuperación se realiza antes o después de una transacción de actualización.

Un solapamiento secuencialmente equivalente de una transacción de recuperación y una de actualización, por ejemplo como en la Figura 9.8, impedirá que ocurran las recuperaciones inconsistentes.

## Figura 9.8
## Un solapamiento de V y W secuencialmente equivalente

| **Transaction *V*:** | | **Transaction *W*:** | |
|---|---|---|---|
| *a.extrae(100);*<br>*b.deposita(100)* | | *unaSucursal.totalSucursal()* | |
| *a.extrae(100);* | $100 | | |
| *b.deposita(100)* | $300 | | |
| | | *total = a.getBalance()* | $100 |
| | | *total = total+b.getBalance()* | $400 |
| | | *total = total+c.getBalance()* | |
| | | ... | |

## Figura 9.9
## Reglas de conflicto en las operaciones lee y escribe

| *Operaciones de Diferentes transac.* | | *Conflicto* | *Causa* |
|---|---|---|---|
| *read* | *read* | No | Porque el efecto de un par de operaciones de lectura no depende del orden en que se ejecutan. |
| *read* | *write* | Yes | Porque el efecto de una operación de lectura y una de escritura dependen del orden en que se ejecutan. |
| *write* | *write* | Yes | Porque el efecto de un par de operaciones de escritura depende del orden en que se ejecutan. |

Para cualquier par de transacciones, es posible determinar el orden de pares de operaciones conflictivas sobre objetos accedidos por ambas.

La equivalencia secuencial puede definirse en términos de las operaciones conflictivas como sigue:

- Para que dos transacciones sean *secuencialmente equivalentes,* es necesario y suficiente que todos los pares de operaciones conflictivas de las dos transacciones se ejecuten en el mismo orden sobre los objetos a los que ambas acceden.

Consideremos como ejemplo las transacciones *T* y *U,* definidas como sigue:

*T:* x = lee(i); escribe(i, 10); escribe(j, 20);
*U:* y = lee(j); escribe(j, 30); z = lee(i);

Consideremos ahora el solapamiento de sus ejecuciones, mostrado en la Figura 9.10 . El acceso de c/ transacción a los objetos *i* y *j* es secuencial con respecto al otro, *T* realiza todos sus accesos a *i* antes de que lo haga *U* y *U* hace todos sus accesos a j antes de que lo haga *T.*

Pero la ordenación no es secuencialmente equivalente, porque los pares de operaciones conflictivas no se hacen el mismo orden en ambos objetos. Las ordenaciones secuencialmente equivalentes requieren una de las dos condiciones siguientes:

1. *T* accede a *i* antes de *U* y *T* accede a *j* antes de *U.*
2. *U* accede a *i* antes de *T* y *U* accede a *j* antes de *T.*

Figura 9.10
Un solapamiento de las operaciones de las transacciones T y U no secuencialmente equivalente

| **Transaction *T*:** | **Transaction *U*:** |
|---|---|
| *x = lee(i)* | |
| *escribe(i, 10)* | |
| | *y = lee(j)* |
| | *escribe(j, 30)* |
| *escribe(j, 20)* | |
| | *z = lee(i)* |

La equivalencia secuencial se utiliza como un criterio para la obtención de protocolos de control de concurrencia.

Estos protocolos intentan secuencializar las transacciones en sus accesos a los objetos. Para el control de concurrencia se utilizan normalmente tres aproximaciones:

- bloqueo,
- control de concurrencia optimista, y
- ordenación por marca de tiempo.

## Recuperabilidad de transacciones abortadas

Los servidores deben registrar los efectos de todas las transacciones finalizadas y ninguno de los efectos de las transacciones abortadas. Deben considerar, por tanto, el hecho de que una transacción pueda ser abortada en previsión de que afecte a otras transacciones concurrentes si es el caso.

Existen dos problemas asociados con transacciones abortadas. Estos problemas se conocen como «***lecturas sucias***» y «***escrituras prematuras***», y ambas pueden aparecer en presencia de ejecuciones secuencialmente equivalentes.

Se consideran dos categorías de operaciones, operaciones de lectura *(lectura)* y de escritura *(escritura).* En nuestras ilustraciones, *obténBalance* es una operación de lectura y *ponBalance* es una de escritura.

## Recuperabilidad de transacciones abortadas

**Lecturas sucias.** La propiedad de aislamiento de las transacciones requiere que éstas no vean el estado no finalizado de las demás. El problema de la «lectura sucia» está causado por la interacción entre una operación de lectura en una transacción y una operación temprana de escritura en otra transacción sobre el mismo objeto.

Considérense las trazas mostradas en la Figura 9.11, donde *T* consigue el balance de la cuenta *A* y le añade 10$ más, a continuación *U* consigue el balance de *A* y le añade 20$ más, y las dos ejecuciones son secuencialmente equivalentes.

Supongamos ahora que la transacción *T* **aborta** después de que *U* ha finalizado. Entonces la transacción *U* habrá visto un valor que nunca ha existido, puesto que se restaurará el valor original en *A.* En este caso se dice que la transacción *U* ha realizado una *lectura sucia.* Como ha finalizado no puede ser desecha.

Figura 9.11
Una lectura sucia cuando se aborta T.

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *a.getBalance()* | | *a.getBalance()* | |
| *a.setBalance(balance + 10)* | | *a.setBalance(balance + 20)* | |
| *balance = a.getBalance()* | $100 | | |
| *a.setBalance(balance + 10)* | $110 | | |
| | | *balance = a.getBalance()* | $110 |
| | | *a.setBalance(balance + 20)* | $130 |
| | | *commit transaction* | |
| *abort transaction* | | | |

## Recuperabilidad de transacciones abortadas

**Recuperación de transacciones.** Si una transacción (como *U)* ha finalizado después de que ha visto los efectos de una transacción que posteriormente ha sido abortada, la situación no es recuperable.

Para asegurar que tales situaciones no se planteen, cualquier transacción (como *U*) que esté en peligro de tener una lectura sucia rechaza la finalización de su operación. La estrategia para la recuperación es retrasar la finalización hasta después de que haya finalizado cualquier otra transacción cuyo estado no finalizado haya sido observado.

En el ejemplo, *U* retrasa su finalización hasta después que se produzca la finalización de *T.* En el caso en que *T* sea abortada, *U* debe abortar también.

## Recuperabilidad de transacciones abortadas

**Abortos en cascada.** En la Figura 9.11, suponemos que la transacción *U* retrasa su consumación hasta después que *T* aborte.

Como hemos dicho, *U* debe abortar también. Desafortunadamente, si algunas otras transacciones han visto los efectos debidos a *U,* deberían también ser abortadas. El aborto de estas últimas transacciones pude causar que se aborten todavía más transacciones. Tales situaciones se conocen como *abortos en cascada.*

Para evitar los abortos en cascada solo se permite a las transacciones leer objetos que fueron escritos por transacciones consumadas. Para asegurar que esto es así, debe retrasarse cualquier operación de *lectura* hasta que haya sido consumada o abortada cualquier otra transacción que haya realizado una operación de *escritura* sobre el mismo objeto. Evitar los abortos en cascada es una condición más fuerte que la recuperabilidad.

**¿Retrasar la finalización o abortar transacciones en cascada?**

## Recuperabilidad de transacciones abortadas

**Escrituras prematuras.** Consideremos que una transacción pueda abortar. Esta relacionada con las interacciones entre operaciones de ***escritura*** *en* el mismo objeto que se realizan en diferentes transacciones. Como ilustración, consideremos dos transacciones *InicializaBalance T* y *U* sobre la cuenta A, como se muestra en la Figura 9.12.

Antes de las transacciones el balance de la cuenta A era de 100$.
Las dos ejecuciones son secuencialmente equivalentes, con *T* poniendo el balance a 105$ y *U* a 110$.
Si la transacción *U* aborta y *T* se consuma, el balance debería ser 105$.

Algunos sistemas de bases de datos implementan la acción *aborta* restableciendo las «***imágenes anteriores***» de todas las *escrituras* de una transacción. En nuestro ejemplo, *A* es 100$ inicialmente, que es la «imagen anterior» de la *escritura* de *T;* de forma similar 105$ es la «imagen anterior» de la *escritura* de *U.*

## Recuperabilidad de transacciones abortadas

Por tanto si *U* aborta, conseguimos el balance correcto de 105$.

Ahora consideremos el caso en que *U* se consuma y después *T* aborta. El balance debería estar a 110$, pero la imagen anterior de la *escritura* de *T* es 100$, por lo que conseguimos el balance incorrecto de 100$.

De forma similar si *T* aborta y después *U* aborta, la «imagen anterior». *escritura* de *U* es 105$ por lo que conseguimos el balance erróneo de 105$, el balance debería proporcionar 100$.

Para garantizar los resultados correctos en un esquema de recuperación que utiliza las imágenes anteriores, las operaciones de *escritura* deben ser retrasadas hasta que hayan sido consumadas o abortadas las transacciones anteriores que actualizaron los mismos objetos.

## Figura 9.12
## Reescritura en valores no consumados

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *a.setBalance(105)* | | *a.setBalance(110)* | |
| | $100 | | |
| *a.setBalance(105)* | $105 | | |
| | | *a.setBalance(110)* | $110 |

## Recuperabilidad de transacciones abortadas

**Ejecuciones estrictas de las transacciones.** Generalmente, se requiere que las transacciones retrasen sus operaciones de *lectura* y *escritura* lo suficiente como para impedir tanto las "lecturas sucias" como las "escrituras prematuras".

Las ejecuciones de las transacciones se llaman *estrictas* si el servicio de las operaciones de *lectura* y de *escritura* sobre un objeto se retrasa hasta que todas las transacciones que previamente escribieron el objeto han sido consumadas o abortadas.

La ejecución estricta de las transacciones hace cumplir la deseada propiedad de aislamiento.

## Recuperabilidad de transacciones abortadas

**Versiones provisionales.** Para que un servidor de objetos recuperables participe en las transacciones, debe estar diseñado de forma que las actualizaciones de los objetos puedan ser eliminadas si transacción aborta.

Para esto, todas las operaciones de actualización realizadas durante una transacción se hacen sobre versiones provisionales de los objetos en memoria volátil.

A cada transacción se le proporciona su propio conjunto privado de versiones provisionales de cualquiera de los objetos que ella ha alterado.

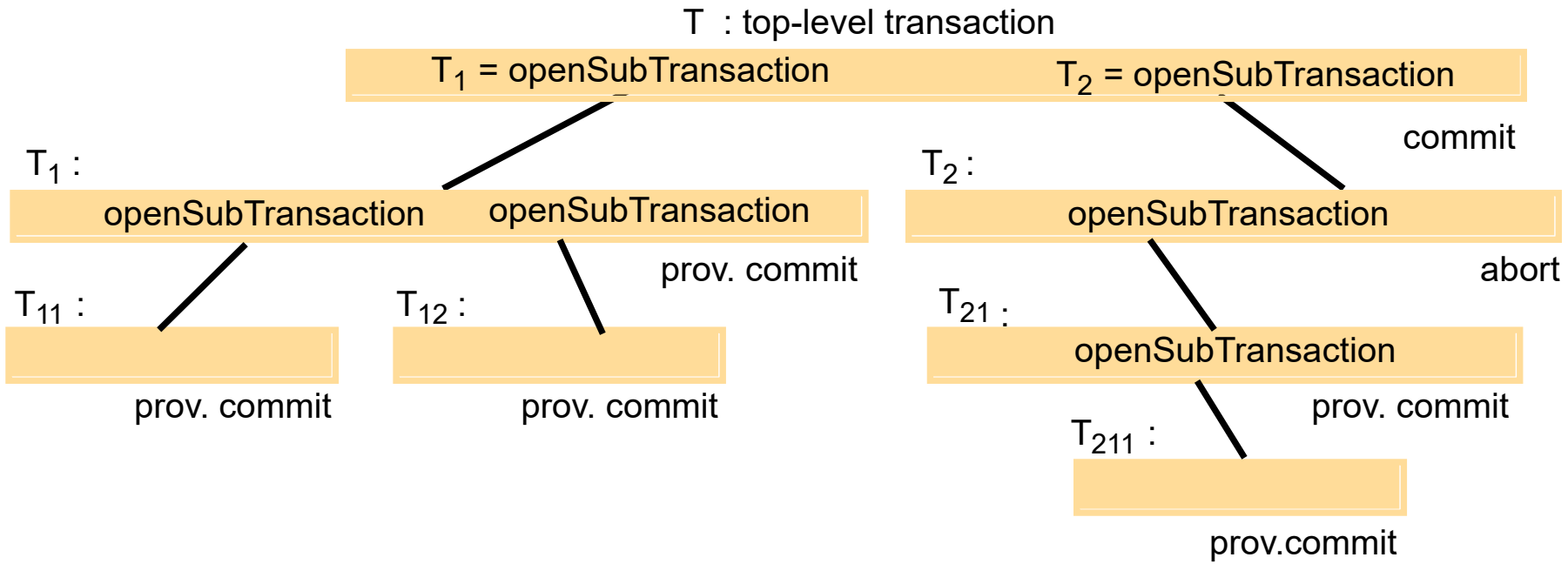
Todas las operaciones de actualización de una transacción almacenan valores de los objetos en el propio conjunto privado de la transacción si es posible, o fallan.

## Recuperabilidad de transacciones abortadas

Las versiones provisionales son transferidas a los objetos sólo cuando una transacción se consuma, en cuyo caso ellas serán también registradas en memoria permanente.

Esto se realiza en un único paso, durante el cual las otras transacciones son excluidas de acceder a los objetos que están siendo alterados. Cuando una transacción aborta, sus versiones provisionales se borran.

# Figura 9.13
# Transacciones anidadas



## Transacciones anidadas

Las transacciones anidadas extienden el modelo anterior de transacción permitiendo que las transacciones estén compuestas de otras. Desde una transacción se pueden arrancar varias transacciones, permitiendo considerarlas como módulos que pueden componerse cuando se precise.

La transacción más exterior en un conjunto de transacciones anidadas se llama la transacción de *nivel superior.* Las transacciones diferentes de la del nivel superior se llaman *sub transacciones.*

Por ejemplo en la Figura 9.13, *T* es una transacción de *nivel-superior,* que inicializa un par de subtransacciones *T*1 y *T* 2. La subtransacción *T*1 inicializa su propio par de transacciones *T*11 y *T*12. Del mismo modo, la transacción *T*2 inicia su propia subtransacción *T*21 que inicia otra subtransacción *T211.*

## Transacciones anidadas

Una subtransacción se presenta como atómica a su padre con respecto a los fallos en la transacción y al acceso concurrente.

Las sub transacciones del mismo nivel, como *T*1 y *T*2 pueden ejecutarse concurrentemente pero sus accesos a objetos comunes son secuenciados, por ejemplo mediante un esquema de bloqueo.

Cada subtransacción puede fallar independientemente de su padre y otras subtransacciones. Cuando la subtransacción aborta, la transacción padre puede elegir una subtransacción alternativa para completar su tarea.

## Transacciones anidadas

Por ejemplo, una transacción para entregar un mensaje de correo a una lista de destinatarios podría estar estructurada como un conjunto de sub transacciones, cada una de las cuales entrega el mensaje a uno de los destinatarios.

Si una o más subtransacciones falla, la transacción padre podría registrar el hecho y después consumarse con el resultado que todas las subtransacciones hijas con éxito se consuman. Podría empezar entonces otra transacción que intentara redirigir los mensajes que no fueron enviados la primera vez.

## Transacciones anidadas

Las transacciones anidadas tienen las siguientes ventajas principales:

- Las sub transacciones en un nivel se ejecutan concurrentemente con otras en el mismo nivel. Permite concurrencia adicional. Cuando las sub transacciones se ejecutan en diferentes servidores, pueden trabajar en paralelo.

- Las sub transacciones se pueden consumar o abortar independientemente. En comparación con una única transacción, un conjunto de sub transacciones anidadas es potencialmente más robusto. El ejemplo anterior de entrega de correo muestra que esto es cierto (con una transacción plana, un fallo en la transacción produciría el que la transacción fuera totalmente reiniciada). De hecho, un padre puede decidir diferentes acciones de acuerdo con si una subtransacción ha abortado o no.

## Transacciones anidadas

**Las reglas para la consumación de transacciones anidadas son bastante sutiles:**

. Una transacción se puede consumar o abortar sólo después de que se han completado sus transacciones hijas.

. Cuando un subtransacción finaliza, hace una decisión independiente sobre si consumarse provisionalmente o abortar. Su decisión de abortar es final.

. Cuando un padre aborta, todas sus sub transacciones son abortadas. Por ejemplo, si *T* 2 aborta entonces *T* 21 y *T* 211 deben abortar también, incluso aunque ellas puedan haber sido consumadas provisionalmente.

## Transacciones anidadas

**Las reglas para la consumación de transacciones anidadas son bastante sutiles:**

. Cuando una subtransacción aborta, el padre puede decidir si abortar o no. En nuestro ejemplo, *T* decide consumarse aunque *T*2 ha abortado.

. Si se consuman las transacciones de alto nivel, entonces todas las sub transacciones que se han consumado provisionalmente pueden consumarse también, proporcionando que ninguno de sus antecesores ha abortado.

En nuestro ejemplo, la consumación de *T* permite que se consumen *T1, T* 11 y *T* 12, pero no *T*21 y *T*211 puesto que su padre *T* 2 ha abortado. Hay que considerar que los efectos de una subtransacción no son permanentes hasta que no se consuma la transacción de nivel superior.

# Bloqueos

La Figura 9.14 ilustra el uso de bloqueos exclusivos. Muestra las mismas transacciones de la figura 9.7, pero con una columna extra para cada transacción que representa el ***bloqueo***, ***la espera*** y el ***desbloqueo***.

En este ejemplo se supone que cuando las transacciones *T* y *U* comienzan los balances de las cuentas *A, B* y C no están todavía bloqueados. Cuando la transacción *T* va a utilizar la cuenta *B, B* se bloquea para *T.*

Consecuentemente, cuando la transacción *U* va a utilizar *B* ésta todavía bloqueada para *T,* y la transacción *U* espera. Cuando se consuma la transacción *T, B* es desbloqueado, después de lo cual se reanuda la transacción *U.* El uso del bloqueo en B serializa efectivarnente el acceso a *B.*

Hay que tener en cuenta que si, por ejemplo, *T* ha liberado el bloqueo en B entre sus operaciones *obténBalance* y *ponBalance,* la operación *obténBalance* de la transacción U puede solaparse entre ellas.

## Bloqueos

La equivalencia secuencial precisa que todos los accesos de una transacción a un objeto particular sean secuenciados con respecto a los accesos por otras transacciones. Todos los pares de operaciones conflictivas de dos transacciones debieran ser ejecutados en el mismo orden.

Para asegurarse esto, no está permitido a una transacción ningún nuevo bloqueo después que ha liberado uno.

La primera fase de cada transacción se conoce como «**fase de crecimiento**», durante la cual se adquieren nuevos bloqueos. En la segunda fase, se liberan los bloqueos (una «**fase de acortamiento**» ). Esto se llama bloqueo en dos fases.

## Figura 7.14
## Las transacciones T y U con bloqueos exclusivos

| **Transaction *T*:**<br>*balance = b.getBalance()*<br>*b.setBalance(bal*1.1)*<br>*a.withdraw(bal/10)* | | **Transaction *U*:**<br>*balance = b.getBalance()*<br>*b.setBalance(bal*1.1)*<br>*c.withdraw(bal/10)* | |
|---|---|---|---|
| Operations | Locks | Operations | Locks |
| *openTransaction* | | | |
| *bal = b.getBalance()* | lock *B* | | |
| *b.setBalance(bal*1.1)* | | *openTransaction* | |
| *a.withdraw(bal/10)* | lock *A* | *bal = b.getBalance()* | waits for *T*'s lock on *B* |
| *closeTransaction* | unlock *A*, *B* | • • • | |
| | | | lock *B* |
| | | *b.setBalance(bal*1.1)* | |
| | | *c.withdraw(bal/10)* | lock *C* |
| | | *closeTransaction* | unlock *B*, *C* |

## Bloqueos

Se utilizan dos tipos de bloqueos: ***bloqueos de lectura*** y ***bloqueos de escritura***. Antes de realizar una operación de *lectura* en una transacción, se debe activar un bloqueo de lectura en el objeto. Antes que se realice una operación de escritura, se debe activar un bloqueo de escritura en el objeto.

Cuando es imposible activar un bloqueo inmediatamente, la transacción (y el cliente) debe esperar hasta que es posible hacerlo; tenga en cuenta que nunca se rechaza una solicitud de un cliente.

Como un par de operaciones de lectura, desde transacciones diferentes, no entra en conflicto, un intento de activar un bloqueo de lectura en un objeto que presente un bloqueo de lectura siempre tendrá éxito. Todas las transacciones que leen los mismos objetos comparten su bloqueo de lectura. Por esta razón, los bloqueos de lectura se llaman, a veces, *bloqueos compartidos.*

## Bloqueos

Las reglas de conflicto de la operación nos dicen que:

1, Si una transacción *T* ha realizado ya una operación de lectura en un objeto particular, entonces una transacción concurrente *U* no debe escribir ese objeto hasta la consumación de *T,* o que aborte.

2. Si una transacción *T* ha realizado ya una operación de escritura en un objeto particular, entonces una transacción concurrente *U* no debe leer o escribir ese objeto hasta la consumación de *T,* o que aborte.

## Figura 9.15
## Compatibilidad de bloqueos

| *Para un objeto* | | *Bloqueo solicitado* | |
|---|---|---|---|
| | | *Lectura* | *Escritura* |
| *Bloqueo ya activado* | *Ninguno* | OK | OK |
| | *Lectura* | OK | Espera |
| | *Escritura* | Espera | Espera |

## Figura 9.16
## Uso de los bloqueos en un sistema de bloqueo en dos fases estricto

1. Cuando una operación accede a un objeto en una transacción:
   (a) Si el objeto no estaba ya bloqueado, es bloqueado y comienza la operación.
   (b) Si el objeto tiene activado un bloqueo conflictivo con otra transacción, la transacción debe esperar hasta que esté desbloqueado.
   (c) Si el objeto tiene activado un bloqueo no conflictivo de otra transacción, se comparte el bloqueo y comienza la operación.
   (d) Si el objeto ya ha sido bloqueado en la misma transacción, el bloqueo será promovido si es necesario y comienza la operación. (Donde la promoción está impedida por un bloqueo conflictivo, se utiliza la regla (b).)
2. Cuando una transacción se consuma o aborta, el servidor desbloquea todos los objetos bloqueados por la transacción.

## Figura 9.17
## La clase *Bloqueo*

```
public class Bloqueo {
        private Object objecto;    // El objeto que es protegido por el bloqueo
        private Vector propietarios;          // los TIDs de los propietarios
       private TipoBloqueo tipoBloqueo;    // el tipo actual

       public synchronized void adquiere(TransID trans,   TipoBloqueo unTipoBloqueo){
          while(/*otra transaccion posea en bloqueo en modo confictivo*/) {
                try {
                      wait();
                }catch ( InterruptedException e){/*...*/ }
          }
          if(propietarios.isEmpty()) { // ningun TIDs  posee un bloqueo
                propietarios.addElement(trans);
                      tipoBloqueo = unTipoBloqueo;

          } else if(/*otra trx posee el bloqueo, lo comparte*/ ) ){
                if(/* esta trx no es un poseedor*/) unPropietario.addElement(trans);

          } else if (/* esta trx es un poseedor pero necesita más de un bloqueo exclusivo*/)
                      tipoBloqueo.promote();
          }
}
```

## Figura 9.17
## La clase *Bloqueo*

```
        public synchronized void libera(TransID trans ){
        propietarios.removeElement(trans);      // elimina este poseedor
        // establece el tipo de bloqueo a ninguno
        notifyAll();
    }
}
```

## Figura 9.18
## La clase *Gestorbloqueo*

```
public class GestiorBloqueo {
   private Hashtable losBloqueos;

   public  void ponBloqueo (Object object, TransID trans,  TipoBloqueo tipoBloqueo){
      Bloqueo  bloqueoEncontrado;
      synchronized(this){
                        // busca el bloqueo asociado con el objeto
                  // si no hay ninguno, lo crea y agrega al hash
            }
            bloqueoEncontrado.agrega(trans, tipoBloqueo);
   }

  // sincroniza ya que queremos eliminar todas las entradas
   public synchronized void desbloqueo(TransID trans) {
      Enumeration e = losBloqueos.elements();
      while(e.hasMoreElements()){
                  Lock aLock = (Lock)(e.nextElement());
                  if(/* trans is a holder of this lock*/ ) aLock.libera(trans);
            }
   }
}
```

# Bloqueos indefinidos

El uso de bloqueos puede conducir a bloqueos indefinidos. Consideremos el uso de bloqueos representado en la Figura 9.19. Puesto que los métodos *deposita* y *extrae* son atómicos, los representamos adquiriendo bloqueos de escritura, aunque en la práctica ellos leen el balance y escriben en él. Cada uno de ellos adquiere un bloqueo y se inmoviliza cuando intenta acceder a la cuenta que ha bloqueado el otro. Ésta es una situación de bloqueo indefinido: dos transacciones están esperando y cada una depende de la otra para liberar un bloqueo y poder reanudarse.

El bloqueo indefinido es una situación particularmente común cuando los clientes están implicados en un programa interactivo, una transacción en un programa interactivo puede durar un período largo de tiempo, produciendo que muchos objetos queden inmovilizados y permanezcan así, impidiendo por tanto que otros clientes los utilicen.

## Figura 9.19
## Bloqueo indefinido con bloqueos de escritura

| Transaction *T* | | Transaction *U* | |
|---|---|---|---|
| **Operations** | **Locks** | **Operations** | **Locks** |
| *a.deposit(100);* | write lock *A* | | |
| | | *b.deposit(200)* | write lock *B* |
| *b.withdraw(100)* | | | |
| • • • | waits for *U*'s | *a.withdraw(200);* | waits for *T*'s |
| | lock on *B* | • • • | lock on *A* |
| • • • | | • • • | |
| • • • | | • • • | |

## Bloqueos indefinidos

Un bloqueo indefinido es un estado en el que cada miembro de un grupo de transacciones está esperando por algún otro miembro para liberar un bloqueo.

Se puede utilizar un *grafo* ***"espera por"*** para representar las relaciones de espera entre las transacciones actuales.

En un grafo *espera por* los nodos representan las transacciones y los arcos representan las relaciones *espera por* entre transacciones, hay un arco del nodo *T* al nodo *U* cuando la transacción *T* está esperando que la transacción *U* libere un bloqueo.

## Bloqueos indefinidos

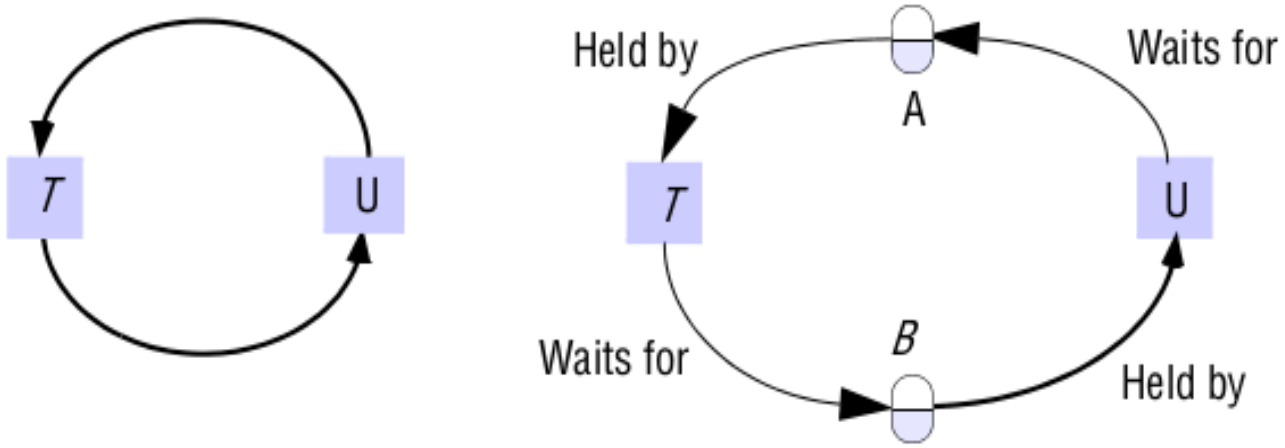
Inspeccionando la Figura 9.20, vemos que el grafo *espera por* correspondiente a la situación de bloqueo indefinido ilustrada en la Figura 9.19.

Un bloqueo indefinido surge porque las transacciones *T* y *U* intentaron cada una adquirir un objeto mantenido por la otra. Por tanto *T* espera por *U* y *U* espera por *T.*

La dependencia entre las transacciones es indirecta, mediante una dependencia de los objetos.

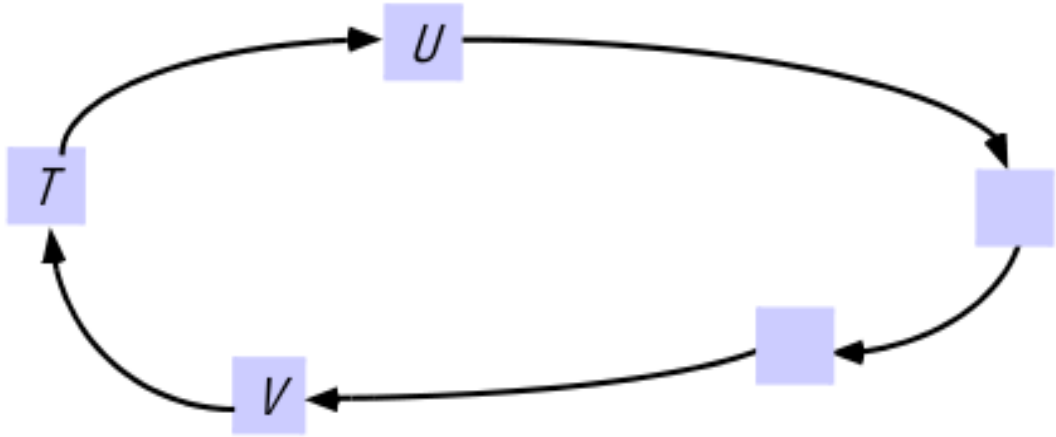
El diagrama de la derecha muestra los objetos mantenidos y esperados por las transacciones *T* y *U.*

## Figura 9.20
## El grafo espera por para la Figura 9.19

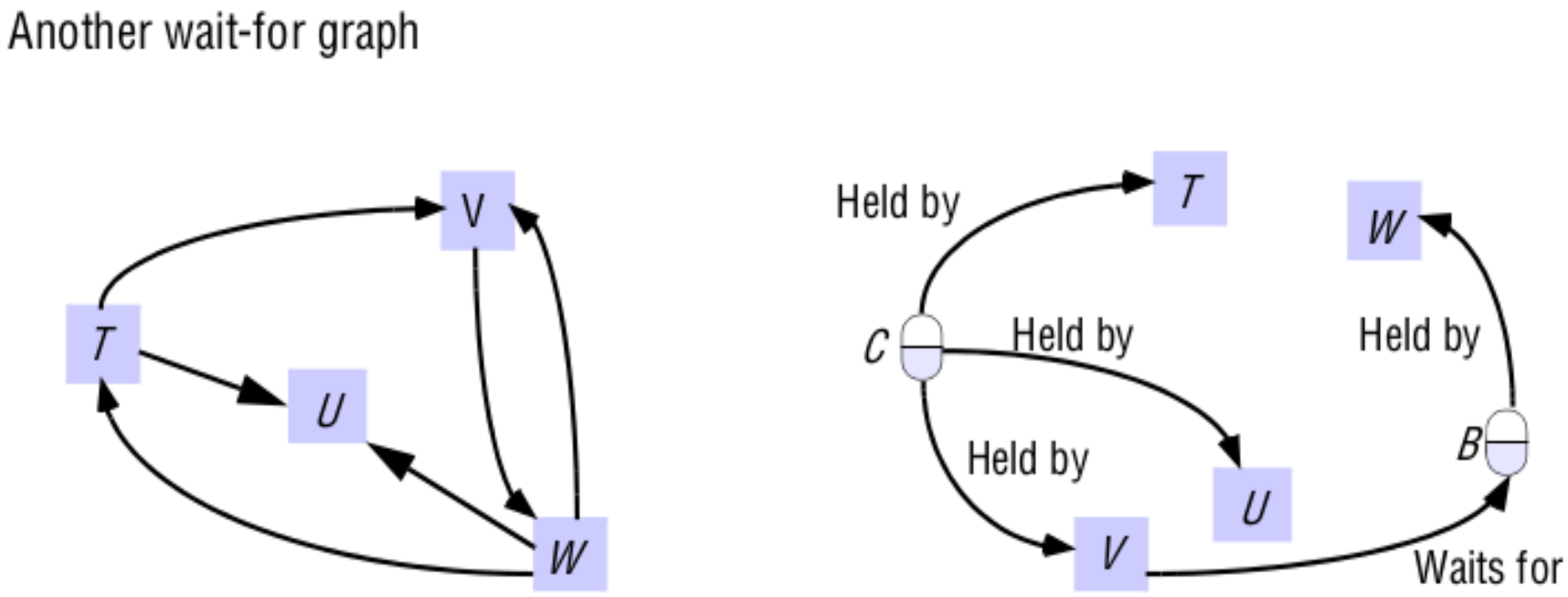


# Figura 9.21
# Un ciclo en el grafo espera por

A cycle in a wait-for graph



# Figura 9.22
# Otro grafo espera por



## Figura 9.23
## Resolución del bloque indefinido en la Figura 9.19

| Transaction T | | Transaction U | |
|---|---|---|---|
| **Operations** | **Locks** | **Operations** | **Locks** |
| *a.deposit(100);* | write lock *A* | | |
| | | *b.deposit(200)* | write lock *B* |
| *b.withdraw(100)* | | | |
| • • • | waits for *U*'s lock on *B* | *a.withdraw(200);* | waits for T's lock on *A* |
| | (timeout elapses) | • • • | |
| *T*'s lock on *A* becomes vulnerable, unlock *A*, abort T | | • • • | |
| | | *a.withdraw(200);* | write locks *A* unlock *A*, *B* |

## Bloqueos indefinidos

Los timeouts de bloqueo son un método para la resolución de bloqueos indefinidos que se utiliza normalmente. A cada bloqueo se le proporciona un período limitado en el que es invulnerable.

Después de este tiempo es vulnerable. Supuesto que ninguna otra transacción está compitiendo por el objeto que está bloqueado, un objeto con un bloqueo vulnerable continúa bloqueado.

Sin embargo, si cualquier otra transacción está esperando para acceder al objeto protegido por un bloqueo vulnerable, el bloqueo se rompe (es decir, el objeto es desbloqueado) y se reanuda la transacción que estaba esperando. La transacción cuyo bloqueo se ha roto normalmente aborta. Hay muchos problemas con el uso de timeouts como un remedio para bloqueos indefinidos: el peor problema es que a veces se abortan transacciones debido a que sus bloqueos llegan a ser vulnerables cuando otras transacciones están esperando por ellas, aunque en realidad no hay bloqueo indefinido.

## Incrementando la concurrencia en esquemas de bloqueo

**Bloqueo de dos versiones.** Esquema optimista que permite que una transacción escriba versiones tentativas de objetos mientras otras transacciones leen de la versión consumada de los mismos objetos.

Esta variación del bloqueo en dos fases estricto utiliza tres tipos de bloqueo:
un bloqueo de lectura,
uno de escritura, y
uno de consumación.

Figura 9.24
Compatibilidad de bloqueos (lectura, escritura, consumación)

| *For one object* | | *Lock to be set* | | |
|---|---|---|---|---|
| | | *read* | *write* | *commit* |
| *Lock already set* | *none* | OK | OK | OK |
| | *read* | OK | OK | wait |
| | *write* | OK | wait | |
| | *commit* | wait | wait | |

# Control optimista de la concurrencia

**Kung y Robinson [1981]** identificaron un número de desventajas inherentes al bloqueo y propusieron una aproximación optimista, alternativa a la secuenciación de transacciones que evita esas desventajas. Desventajas del bloqueo:

- Mantener el bloqueo representa sobrecarga que no está presente en los sistemas que no soportan acceso concurrente a los datos compartidos.

- Puede producir un bloqueo indefinido. Su prevención reduce la concurrencia de forma severa, y por tanto las situaciones de bloqueo indefinido deben ser resueltas o por el uso de *timeouts* o por la detección de bloqueo indefinido. Ninguna de ellas es totalmente satisfactoria para uso en programas interactivos.

- Para impedir abortos en cascada, los bloqueos no pueden ser liberados hasta el final de la transacción. Esto puede reducir significativamente el potencial de concurrencia.

## Control optimista de la concurrencia

La aproximación alternativa propuesta por Kung y Robinson es *optimista* porque se basa en la observación de que, en la mayoría de las aplicaciones, la similitud entre las transacciones de dos clientes que acceden al mismo objeto es baja. Se permite que las transacciones procedan como si no hubiera posibilidad de conflicto con otras transacciones hasta que el cliente complete su tarea y publique una petición *cierraTransacción.*

Cuando aparece un conflicto, habitualmente se abortará alguna transacción y se necesitará reinicializar el cliente.

## Control optimista de la concurrencia

Cada transacción presenta las siguientes fases:

***Fase de trabajo:*** durante esta fase, c/ transacción tiene una versión tentativa de cada uno de los objetos que actualiza. Ésta es una copia de la versión del objeto más recientemente consumada.

Las versiones tentativas permiten a la transacción abortar (sin efecto alguno sobre los objetos), tanto durante la fase de trabajo como si falla su validación debido a otras transacciones en conflicto.

Operaciones de lectura se realizan inmediatamente: si existe una versión tentativa para la transacción, la operación de lectura accederá a ella, de otro modo accederá al valor más recientemente consumado del objeto.

## Control optimista de la concurrencia

La operación de escritura almacena los nuevos valores de los objetos bajo versiones tentativas (las cuales son invisibles al resto de transacciones).

Cuando hay varias transacciones concurrentes, podrán coexistir valores tentativos diferentes sobre el mismo objeto.

Además, se almacenan dos registros para cada objeto al que se accede en la transacción:

- un conjunto de lectura que contiene los objetos leídos por la transacción; y
- un conjunto de escritura que contiene los objetos modificados por la transacción.

## Control optimista de la concurrencia

***Fase de validación***: cuando se recibe la solicitud cierraTransacción, se valida la transacción para establecer si sus operaciones en los objetos entran en conflicto o no con las operaciones en otras transacciones sobre los mismos objetos. Si la validación tiene éxito, entonces se puede consumar la transacción. Si la validación falla, se debe utilizar alguna forma de resolución de conflictos y bien habrá que abortar la transacción actual o, en algunos casos, aquellas con las que entra en conflicto.

***Fase de actualización:*** si una transacción está validada, todos los cambios registrados en su versiones provisionales se hacen permanentes, Las transacciones de sólo lectura pueden consumarse inmediatamente después de pasar la validación. Las transacciones de escritura están dispuestas a consumarse una vez que las versiones provisionales de los objetos se hayan grabado en memoria permanente.