

## TEMA XII del Programa Docente

**Apuntes Complementarios al Capítulo 12 de Ozsu & Valduriez.**

### *Fiabilidad en los Sistemas de Información Distribuidos: Recuperación frente a Fallos y Protocolo 2PC*

#### ÍNDICE

<b>1.- Fiabilidad en los Sistemas de Bases de Datos Distribuidos.</b>	<b>2</b>
1.1.- Protocolos de Fiabilidad Local.	2
1.2.- Consideraciones sobre la Arquitectura.	2
1.3.- Información a Recuperar por Fallos del Sistema.	3
1.4.- Gestor de Recuperación frente a fallos del Sistema.	6
1.5.- Fallos del Sistema: LRM y Checkpoint.	7
<b>2.- Protocolo COMMIT de Dos Fases, 2PC, encargado de la Distribución.</b>	<b>10</b>
2.1.- Introducción al Protocolo 2PC.	10
2.2.- Razones para Sincronizar todos los SGBDs Locales.	11
2.3.- Pasos del Protocolo 2PC en ausencia de fallos.	12
2.4.- Acciones del Protocolo 2PC.	13
2.5.- Diagrama de Transición de Estados del Protocolo 2PC.	14
<b>3.- Estructuras de Comunicación en el Protocolo 2PC.</b>	<b>15</b>
3.1.- Comunicación Centralizada del 2PC.	15
3.2.- Comunicación Lineal del 2PC.	16
3.3.- Comunicación Distribuida del 2PC.	16

## 1.- Fiabilidad en los Sistemas de Bases de Datos Distribuidos.

### 1.1.- Protocolos de Fiabilidad Local.

La parte del software de los SGBD locales encargada de gestionar la recuperación local, de cada lugar participante en una BDD, se llama LRM (Local Recovery Manager). Sus funciones mantienen la propiedades de *atomicidad y persistencia de las transacciones locales*. Estas funciones son las relativas a la ejecución de los comandos que entran al LRM: **bot** (begin-of-transaction), **read**, **write**, **commit**, **abort** y **eot** (end-of-transaction).

Existen, además, nuevas instrucciones en el repertorio del LRM encargadas de iniciar las acciones de recuperación despues de que aparezca un fallo. Primero describiremos la ejecución de estos comandos en un entorno centralizado, despues se abordarán los aspectos adicionales para BDD.

### 1.2.- Consideraciones sobre la Arquitectura.

La figura 1 presenta un modelo abstracto de un SGBD centralizado, cuya arquitectura describe cuatro módulos encargados de la gestión de transacciones y la gestión de los datos de las BD. Este modelo no se corresponde fielmente con alguna arquitectura de los SGBD comerciales. Su descripción interesa porque, pedagógicamente, es importante separar limpiamente los problemas de **control de concurrencia y recuperación del resto de las funciones de un SGBD**.

A continuación, en la figura 2, se describe la interfaz específica entre el Gestor de Recuperación Local y el Gestor de Buffers de la BD. Este último es responsable de todos los accesos a la BD (operaciones read y write).

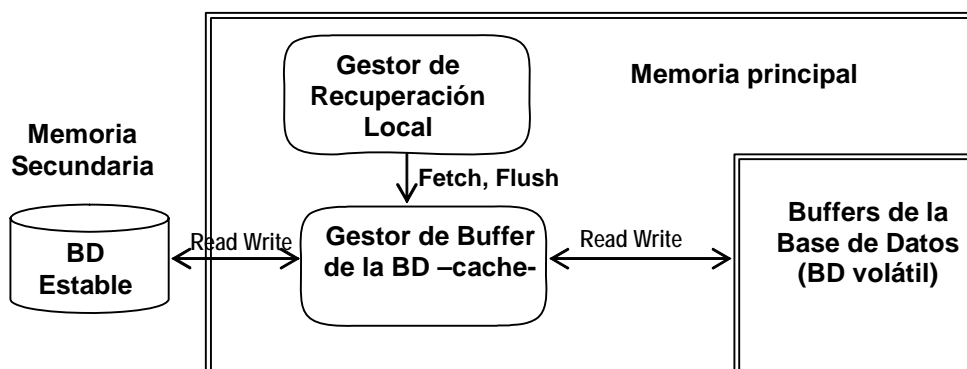


Fig. 2.- Interfaz entre el Gestor de Recuperación Local y el Gestor de Buffers.

Se supone que la BD estable está almacenada permanentemente en memoria secundaria [Larson and Sturgis, 1976]. La estabilidad de este medio de almacenamiento se debe a su robustez frente a fallos. En un dispositivo de almacenamiento estable las expectativas de fallo son mucho menos frecuentes que en uno no-estable, como lo es la memoria principal. En la actual tecnología, se considera estable a los discos magnéticos que almacenan copias duplicadas de datos, que siempre se guardan mutuamente consistentes. A la versión de la BD guardada en un dispositivo estable, se la llama BD estable. Una página de disco es la unidad de almacenamiento y de acceso a una BD estable.

El gestor del buffer de la BD guarda, en memoria principal, los datos recientemente accedidos desde el disco, para mejorar así los tiempos de acceso a disco. Es típico que el buffer esté dividido en páginas, cada una del mismo tamaño que las páginas del disco. La parte de la BD que reside en los buffers se la llama **BD volátil**. Es importante destacar que el Gestor de Recuperación Local ejecuta las operaciones que le envían las transacciones sólo sobre la BD volátil, donde residen los nuevos valores de los datos, los cuales, un tiempo después, serán grabados en la BD estable.

Cuando el Gestor de Recuperación Local necesita leer una página de datos (o un bloque) pedidos por alguna operación de alguna transacción, envía un comando **fetch** indicando la página que quiere leer. El gestor del buffer examina si la página está en su área (debido a previos comandos fetch de otra transacción), si es así deja la página disponible para esa transacción; si no, lee la página desde la BD estable en un buffer vacío. Hay varios algoritmos distintos por los que el gestor del buffer elige la página a ser sustituida por una nueva.

El gestor del buffer también proporciona la interfaz por la cual el Gestor de Recuperación Local puede forzarle a re-escribir alguna de las páginas del buffer. Esto se puede conseguir mediante el comando **flush**, que indica las páginas del buffer que el GRL quiere que se re-escriban. No todas las implementaciones de los GRL tienen esta modalidad de escritura forzada por el GRL. Este asunto será tratado posteriormente.

### **1.3.- Información a Recuperar por Fallos del Sistema.**

Vamos a suponer que sólo habrá fallos debidos al sistema. Más adelante se tratarán las técnicas para la recuperación de fallos debidos a los medios. Puesto que trataremos de la recuperación en bases de datos centralizadas, los fallos debidos a la comunicación tampoco serán aplicables.

Cuando aparece un fallo del sistema, la BD volátil se pierde. Por tanto, el SGBD tiene que mantener alguna información sobre su estado en el momento del fallo para poder llevar a la base de datos al estado que tenía cuando apareció el fallo. Llamaremos a esta información la **información a recuperar**.

La información a recuperar que mantiene el sistema depende de los métodos con los que se ejecuten las actualizaciones. Hay dos posibilidades para ello, actualización in-situ o fuera-del-lugar.

La actualización in-situ cambia físicamente los valores de la BD en la BD estable, y, por tanto los valores anteriores se pierden. Por otro lado, la actualización fuera-de-lugar no cambia los valores de los datos en la base de datos estable, sino que mantiene los nuevos valores separadamente. Por supuesto, periódicamente, estos nuevos valores tienen que ser integrados en la base de datos estable. Los estudios de fiabilidad son más simples si no se usa la actualización in-situ. Sin embargo, la mayoría de los SGBD utilizan actualizaciones in-situ porque se mejora el rendimiento.

**Información a Recuperar con Actualización in-situ.** Debido a que la actualización in-situ produce que los valores previos de los ítems de datos afectados se pierdan, resulta necesario guardar la información suficiente sobre los cambios de estado de la BD para facilitar la recuperación de la BD a un estado consistente después del fallo.

Esta información se mantiene típicamente en una base de datos, llamada **log**. De esta forma, cada actualización no sólo cambia la BD sino también se queda registrada en la **BD log** (véase fig. 3).

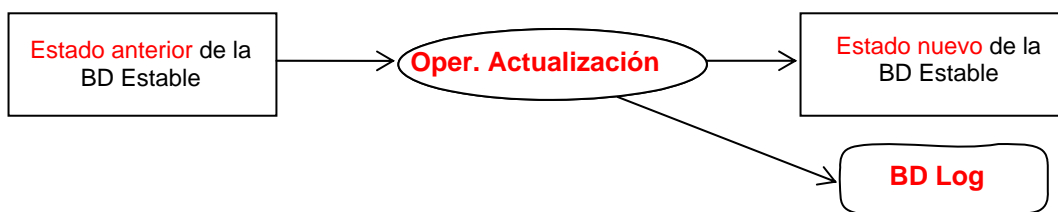


Fig. 3.- Ejecución de una operación de actualización

Consideremos el siguiente escenario. El SGBD empezó a funcionar en el tiempo 0 y, en el tiempo  $t$  aparece un fallo del sistema. Durante el periodo  $[0, t]$ , dos transacciones se han iniciado en el SGBD (sean T1 y T2). Una de ellas, T1 está committed, y T2 está activa, como indica la figura 4.

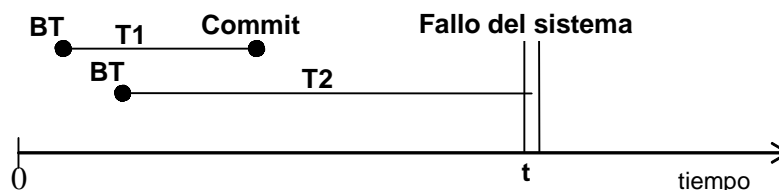


Fig. 4.- Ocurrencia de un fallo del sistema

La propiedad de **persistencia** (*durability*) de las transacciones requiere que los efectos de la transacción T1 queden reflejados en la BD estable. Análogamente, la propiedad de atomicidad requiere que la BD estable no contenga ninguno de los efectos de la transacción T2. Sin embargo, para asegurar esto el SGBD necesita tomar precauciones especiales.

Supongamos que el GRL y los algoritmos del gestor del buffer son tales que las páginas del buffer se re-escriben en la BD estable sólo cuando el gestor del buffer necesita nuevo

espacio en el buffer. Es decir, el comando **flush** no es lanzado por el GRL, y la decisión de re-escribir las páginas en la BD estable se toma a discreción por el gestor del buffer. En este caso, es posible que las páginas en la BD volátil hayan sido actualizadas por T1 pero no se hayan re-escrito en la BD estable en el momento en que aparece el fallo. Por tanto, en la recuperación es muy importante poder *re-hacer* (**redo**) las operaciones de T1. Ello requiere guardar información en el BD log sobre los efectos de T1. Con la información del LOG es posible recuperar la BD desde su "viejo" estado al "nuevo" estado que refleje los efectos de T1, como indica la figura 5.

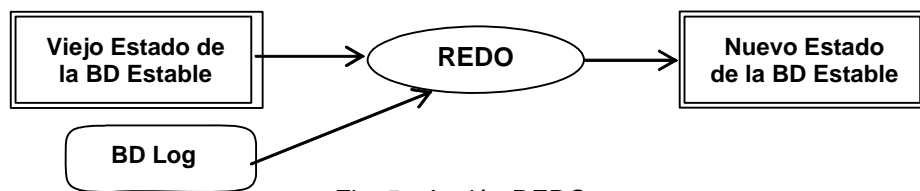


Fig. 5.- Acción REDO

Análogamente, puede que el gestor de buffer haya escrito en la BD estable algunas páginas de la BD volátil que hayan sido actualizadas por T2. Por lo cual, para la recuperación después del fallo es necesario *des-hacer* (**undo**) las operaciones de T2. De forma tal que la información a recuperar deberá incluir datos suficientes para poder deshacer desde el "nuevo" estado de la Bd que contiene los efectos parciales de T2 y recuperar el "viejo" estado que tenía antes de arrancar la transacción T2, como refleja la figura 6.

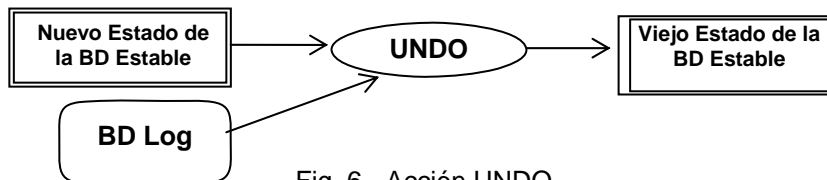


Fig. 6.- Acción UNDO

Las acciones REDO y UNDO son equipotentes, es decir, su aplicación reiterada sobre una transacción produce el mismo efecto que si se ejecuta sólo una vez. Undo y Redo constituyen la base de distintos métodos de ejecución de los comandos **Commit**.

El contenido del **Log** difiere de unas implementaciones a otras. Sin embargo, el **Log** debe contener, al menos, la siguiente información mínima por cada transacción:

- un registro begin-transaction,
- el valor del item del dato antes de la actualización (llamado imagen anterior),
- el valor del dato actualizado (imagen posterior),
- un registro de terminación indicando la condición del fin de la transacción (commit, abort).

La granularidad de las imágenes anterior y posterior puede ser distinta, se puede guardar en el Log páginas enteras o unidades de información más pequeñas.

#### 1.4.- Gestor de Recuperación frente a fallos del Sistema.

Al igual que la BD volátil, el **Log** también se mantiene en buffers de memoria principal (llamado *buffers del log*) y se re-escrive en el almacenamiento estable (llamado log estable) análogo a como se hace con las páginas del buffer de la BD (ver figura 7).

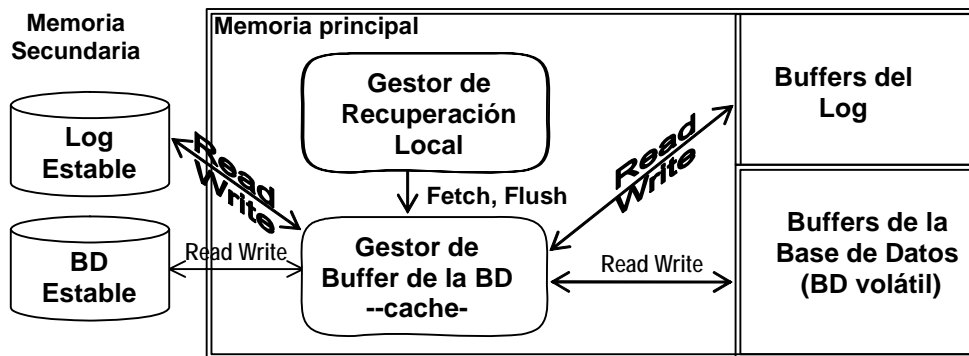


Fig. 7.- Interfaces de los Logs.

Las páginas del Log pueden guardarse en el log estable de dos formas. Pueden ser escritas de forma síncrona, (conocido como **forzar un log**) en la cual, cada vez que se añade un registro al log es necesario llevar un registro del log en memoria principal a la memoria estable. También, las páginas del Log pueden guardarse de forma *asíncrona*, donde el log se graba en disco bien por intervalos de tiempo periódicos, bien porque se llene el buffer. Con la forma síncrona, se suspende la ejecución de la transacción hasta que se complete la escritura, lo que retarda el tiempo de respuesta de los datos. Por otro lado, si el fallo aparece después de haber forzado una escritura, resulta relativamente fácil la recuperación de la BD a un estado consistente.

Bien se escriba el log de forma síncrona o asíncrona, es muy importante que el mantenimiento de los logs se regule por algún protocolo.

Consideremos el caso donde las actualizaciones a la BD se escriben en la BD estable antes de haber modificado el Log estable para reflejar el nuevo valor. Si aparece un fallo antes de escribir en el log, la BD estará actualizada pero no el Log lo que haría imposible recuperar la BD a un estado consistente y actualizar su estado. Esto se conoce como el protocolo **write-ahead-logging, -WAL-** [Gray, 1979] y se especifica como se indica a continuación:

- 1.- *Antes de actualizar la BD estable* (quizá debido a las acciones de una transacción activa y no committed), *la imagen anterior deberá ser guardada en el log estable*. Esto facilita el UNDO.
- 2.- *Cuando una transacción alcanza el estado committed*, *la imagen anterior tiene que ser grabada en el log estable antes de actualizar la BD estable*. Esto facilita el REDO.

En definitiva, el Log es como un mini-base-de-datos que vaguardando los cambios habidos durante un día. El Log estable reside en el disco duro y suelen generarse unos 200 MB. Por motivos de espacio y seguridad, el Log suele guardarse también en cinta.

La parte del Log generada más recientemente, reside en un bloque de Memoria Principal y se llama **Log Activo**, como representa la figura 8.

El Gestor del Log (Log Manager, LM) hace el volcado a disco y a cinta cuando el bloque del Log Activo se llena. Este bloque lleno, sólo tiene transacciones que se han ejecutado correctamente y completamente. Si una transacción activa no cabe en el Log Activo, es fallada por el sistema y pasa al estado *aborted*.

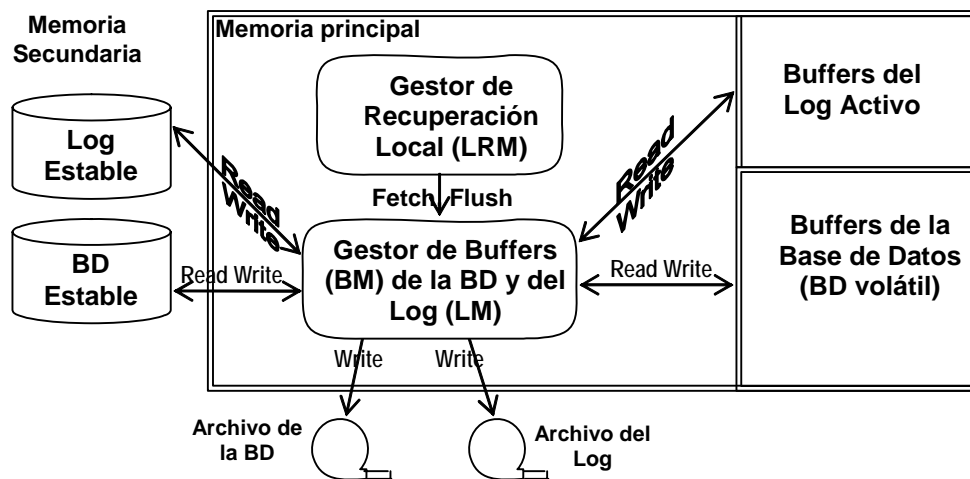


Fig. 8.- Jerarquía completa de memorias gestionadas por el LRM y el BM.

Cuando una transacción quiere leer una página de datos, indicada en el comando **Fetch**, el LRM envía un **Fetch** al **BM**; primero accede a Memoria Principal, si no está allí, va a disco duro a modo de **Caché** o memoria virtual (véase figura 8).

El LRM envía un **Flush** al **BM**; para forzar una escritura desde Memoria Principal a disco duro y a cinta de un número de páginas en los Buffers. El **BM** es quien decide cuándo se van a escribir dichas páginas.

### 1.5.- Fallos del Sistema: LRM y Checkpoint.

Un fallo del sistema supone parar todos los procesos y arrancar de nuevo. Las transacciones que estuvieran activas en el momento del fallo hay que abortarlas ya que no habían finalizado cuando vino el fallo. Ello supone realizar una operación UNDO desde el Log Estable.

El LRM podría abortar las transacciones leyendo el Log y buscando las que tienen **Begin-Transaction** pero no tienen **Commit** o **Abort**. Pero, si se hace así, se gasta mucho tiempo. Para agilizar, se usa la técnica de los **Checkpoint**.

El LRM, cada cierto tiempo (5 min. o cuando el Log Activo tiene ya una cantidad de datos) toma un **Checkpoint** que adopta la forma de registro y lo graba en el Log Estable ( estos registros **Checkpoint** hacen las veces de las *piedras que Pulgarcito dejaba por el camino para saber volver* por lo ya andado).

Grabar un registro **Checkpoint** supone seguir los tres pasos siguientes:

Paso 1.- Grabar en el Log Activo un registro **Checkpoint (Begin-Check)**

Paso 2.- Forzar a que el contenido de los Buffers de la BD salga hacia la BD Estable

Paso 3.- Escribir en un fichero de re-arranque la dirección del Log Activo donde se ha grabado el último **Checkpoint** y escribir un **End-Check** en el Log Activo.

Si aparece un fallo entre el Paso 1 y el Paso 3, el LRM –al rearrancar- no considera válido el último **Checkpoint** e iría a buscar el anterior  $\Rightarrow$  **atomicidad de la operación checkpoint**.

Cada registro **Checkpoint** grabado en el Log contiene la siguiente información:

- Una lista con todas las transacciones activas en el momento del **Checkpoint**.
- La dirección que tiene cada transacción en el Log, que son las más recientemente incorporadas en él.

Al rearrancar, el LRM crea las dos listas siguientes, y efectúa las siguientes acciones:

- Lista-UNDO
- Lista-REDO

A.- Consulta el fichero de re-arranque para leer en qué direcciones del Log están los registros **Checkpoint** más recientes.

B.- Localiza el registro **Checkpoint** más reciente en el Log

C.- Explora el Log hacia delante y hasta el final.

Inicialmente, la Lista-UNDO contiene todas las transacciones grabadas en el registro **Checkpoint** y la Lista-REDO está vacía.

El LRM, en el paso C.-, explora el Log desde el último **Checkpoint** hasta el final, como representa la figura 9; y lleva a cabo las siguientes acciones, de forma que:

- 1.- Si encuentra un registro **Begin-transaction** para una transacción dada, añade esa transacción a la Lista-UNDO.
- 2.- Si encuentra un registro **Commit** para una transacción dada, lleva esa transacción de la Lista-UNDO a la Lista-REDO.

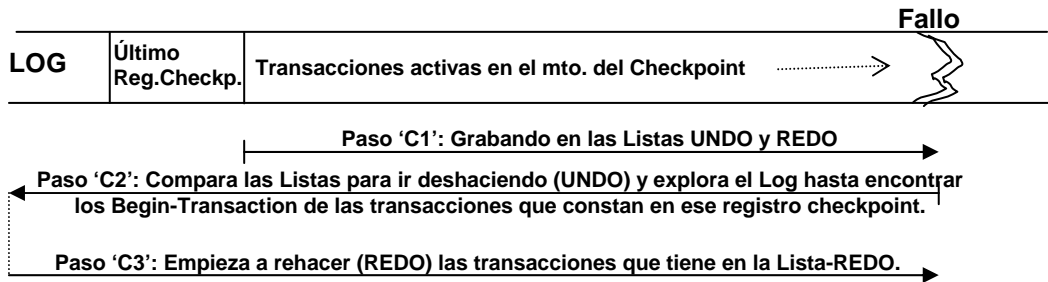


Fig. 9.- Acciones UNDO y REDO del LRM.

La figura 10 muestra un ejemplo de las acciones llevadas a cabo por el LRM, para un determinado contenido de un registro **Checkpoint**, en el momento que surge un fallo.

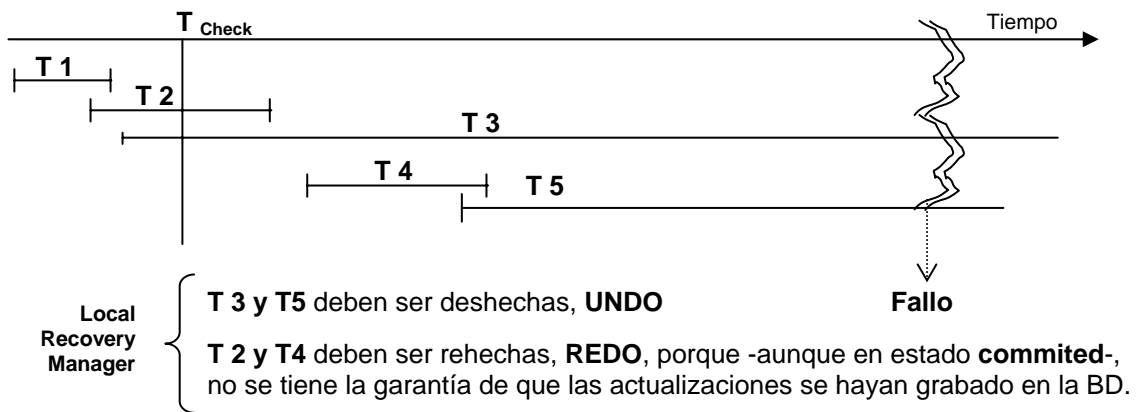


Fig. 10.- Ejemplo de las Acciones UNDO y REDO del LRM.

## 2.- Protocolo COMMIT de Dos Fases, 2PC, encargado de la Distribución.

### 2.1.- Introducción al Protocolo 2PC.

El protocolo *Two Phase Commit*, **2PC**, controla la ejecución de una transacción distribuida. La transacción distribuida se inicia en un determinado nodo (flecha número 1 de la figura 11) que posee un SGBD participante en la distribución -llamado el **Coordinador**-. Para la ejecución de la transacción, se precisa emitir o difundir partes de las operaciones a otros nodos donde residen otros SGBDs -llamados **Participantes**- que son los encargados del acceso a los datos ubicados en cada localidad (flechas con número 2 en la figura 11). Cada Participante responde a la solicitud del Coordinador, indicando si está o no dispuesto para llevar a cabo su correspondiente transacción local (flechas con núm. 3 en la figura 11). Con esta información de control, recibida de los Participantes, el Coordinador adopta una decisión unánime de si ha de llevarse a cabo la transacción o no, y se la difunde a todos los Participantes (flechas con número 4 en la figura 11).

El papel de Coordinador o de Participante que puede jugar un determinado nodo, es eventual en cada ejecución. Será Coordinador el nodo que vaya a hacerse cargo de dirigir la ejecución de dicha transacción, y será Participante aquel nodo encargado del acceso a los datos de su localidad participante.

La figura 11 representa el protocolo de interacción, indicado mediante flechas numeradas.

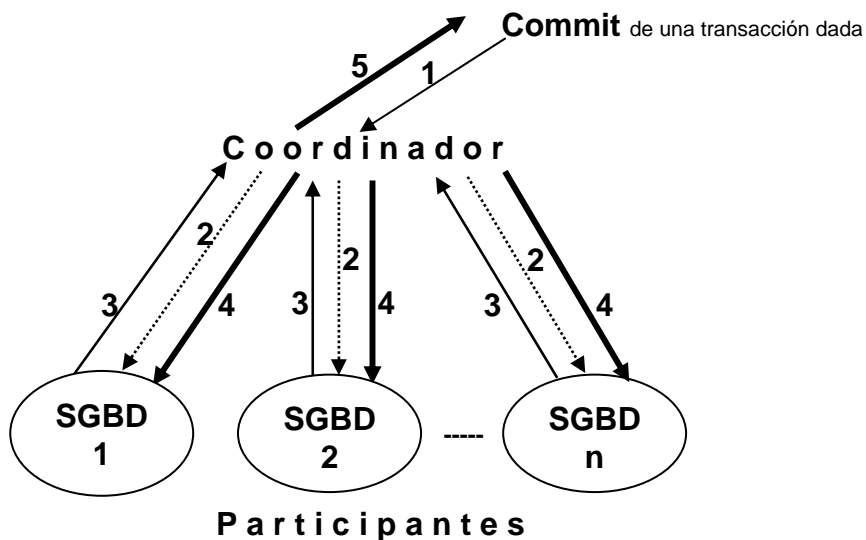


Fig. 11.- Protocolo COMMIT de dos Fases, 2PC.

2PC es un protocolo muy elegante y simple que asegura la atomicidad de las transacciones distribuidas.

### ¿ Cuándo es distribuida una Transacción?

Cuando, en su ejecución, participan varios **SGBDs** en distintos sitios.

### ¿ En qué nivel de distribución funciona este protocolo?

2PC se aplica por igual a las distintas formas de distribución que existen hoy en la tecnología de Bases de Datos. Este protocolo funciona en:

LA DISTRIBUCIÓN DE LOS PROCESOS (islas interconectadas, interoperables mediante aplicativos, cuyas transacciones obtienen datos globales), y en  
LA DISTRIBUCIÓN DE LOS DATOS de una BD Distribuida propiamente dicha.

### ¿ En qué se diferencia el 2PC Distribuido del Commit Local?

Extiende los efectos del COMMIT local atómico a las Transacciones Distribuidas.

### ¿ En qué consiste su funcionamiento general?

El protocolo 2PC cuida que todos los lugares (participantes en una transacción) se **COMPROMETAN POR UNANIMIDAD**, antes de hacer permanentes los efectos de la transacción.

## 2.2.- Razones para Sincronizar todos los SGBDs Locales.

**1ª Razón:** algunas localidades pueden generar 'Schedulers' usando un tipo de algoritmo para el control de la concurrencia que NO Garantice, a la ejecución de sus transacciones locales, Alcanzar Siempre un estado 'COMMITTED'.

Ahora bien, si se usa el protocolo 2PL estricto, entonces no se permite que un item 'x' actualizado por 'T<sub>j</sub>' sea leído por otra ' T ' hasta que 'T<sub>j</sub>' finalice. Esto se conoce como la **condición de recuperabilidad**.

### Concepto Importante:

Una ejecución es RECUPERABLE si, por cada ' T<sub>i</sub> ' que hace 'commit', le preceden todos los 'commit' de las transacciones desde las que ' T<sub>i</sub> ' ha leído.

Por tanto, una 'T<sub>j</sub>' no puede hacer 'commit' hasta saber que todas las ' Ts ' que han escrito valores leídos por 'T<sub>j</sub>' no han abortado y todas las 'Ts' están 'committed'.

Se dice que 'T<sub>j</sub>' lee 'x' desde 'T<sub>i</sub>' en una ejecución si:

'T<sub>j</sub>' lee 'x' después que 'T<sub>i</sub>' haya escrito en 'x',

'T<sub>i</sub>' no aborta antes que 'T<sub>j</sub>' lea 'x',

Cada 'T', si existe, que escribe 'x' entre el tiempo en que 'T<sub>i</sub>' escribe y 'T<sub>j</sub>' lo lee, aborta antes que 'T<sub>j</sub>' lo lea.

**2ª Razón:** algunos participantes, con un enfoque optimista en sus algoritmos de C.C., quizá no puedan comprometerse a hacer 'Commit'. Los 'Deadlocks' son posibles y

'Deadlock' =====> Abortar la Transacción

Un participante (parado por un 'deadlock') debería poder abortar sin tener que avisar al coordinador de su aborto, y el coordinador por 'timeout' fallaría la transacción. Esta importante posibilidad se conoce como **Aborto Unilateral**.

### 2.3- Pasos del Protocolo 2PC en ausencia de fallos.

Inicialmente, el coordinador:

- escribe un registro 'begin-commit' en su Log
- envía un mensaje 'preparar' a todos los participantes, y
- pasa al estado 'WAIT'.

Cuando cada participante recibe el mensaje 'preparar':

- verifica si puede comprometerse en la ejecución de su transacción local. Si es así, el participante escribe:

1. un registro 'Ready' en su Log,
2. envía un mensaje 'voto-commit' al coordinador y
3. pasa al estado 'READY'

En caso contrario, el participante:

1. escribe un registro 'ABORT' en su Log,
2. envía un mensaje 'voto-abort' al coordinador y
3. pasa al estado 'ABORT', pudiendo olvidarse de esta transacción ya que habrá vetado a todos (por aborto unilateral).

Después que el coordinador haya recibido respuesta de todos los participantes, decide hacer:

'abort', si al menos un participante envió 'abort', entonces:

1. envía un mensaje 'global-abort' a todos los participantes y
2. pasa al estado 'ABORT'.

'commit', si todos envían 'commit':

1. escribe un registro 'Commit',
2. envía un mensaje 'global-commit' a todos los participantes y
3. pasa al estado 'COMMIT'.

Los participantes, según sea el mensaje recibido:

1. *abortan* o comprometen la transacción y
2. devuelven un '*acknowledgement*', en cuyo momento el coordinador finaliza la transacción escribiendo un '*End-Of-Transaction*' en su Log.

#### 2.4.- Acciones del Protocolo 2PC.

La figura 12 representa, con mayor nivel de detalle, las acciones del protocolo Commit de dos Fases, encargado de sincronizar la ejecución de las transacciones distribuidas.

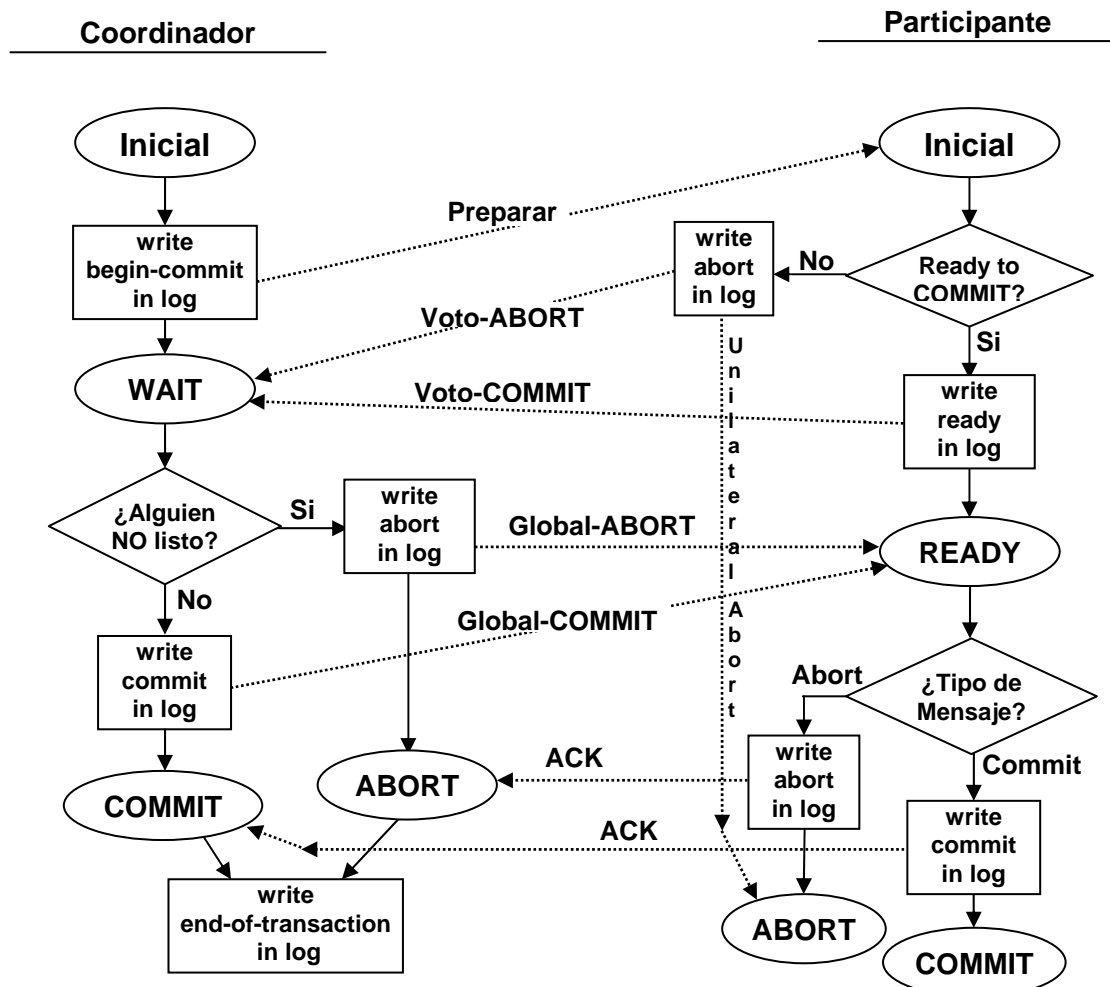


Fig. 12.- Acciones del Protocolo COMMIT de dos Fases, 2PC.

**Reglas del 'global-commit'** que gobiernan cómo el coordinador decide finalizar la transacción (grabando un EOT):

1. si algún participante emite '*voto-abort*', el coordinador tiene que decidir un '*global-abort*' .
2. si todos los participantes envían '*voto-commit*', el coordinador tiene que decidir un '*global-commit*' .

Observaciones adicionales a la figura 12:

1. 2PC permite que un participante aborte hasta que dicho participante no haya decidido grabar un voto afirmativo.
2. Una vez que el participante haya votado afirmativamente, ya no puede cambiar su voto.
3. Mientras un participante esté en el estado READY, su siguiente estado puede ser ABORT o COMMIT, dependiendo de la naturaleza del mensaje que le envíe el coordinador.
4. El coordinador toma la decisión de una terminación global de acuerdo a las dos reglas del *'global-commit'* dichas.
5. Tanto los procesos del coordinador como los de los participantes están en algún estado, donde han de esperar por los mensajes que se intercambian. Para garantizar que pueden seguir en esos estados, o bien que han de terminar, se usan unos temporizadores. Si el margen del temporizador se sobrepasa, sin haber recibido el mensaje que se esperaba, el proceso invoca a su protocolo *Timeout*.

### 2.5.- Diagrama de Transición de Estados del Protocolo 2PC.

Finalmente, la figura 13 muestra el diagrama de transición de estados del protocolo 2PC, en el Coordinador y en los Participantes.

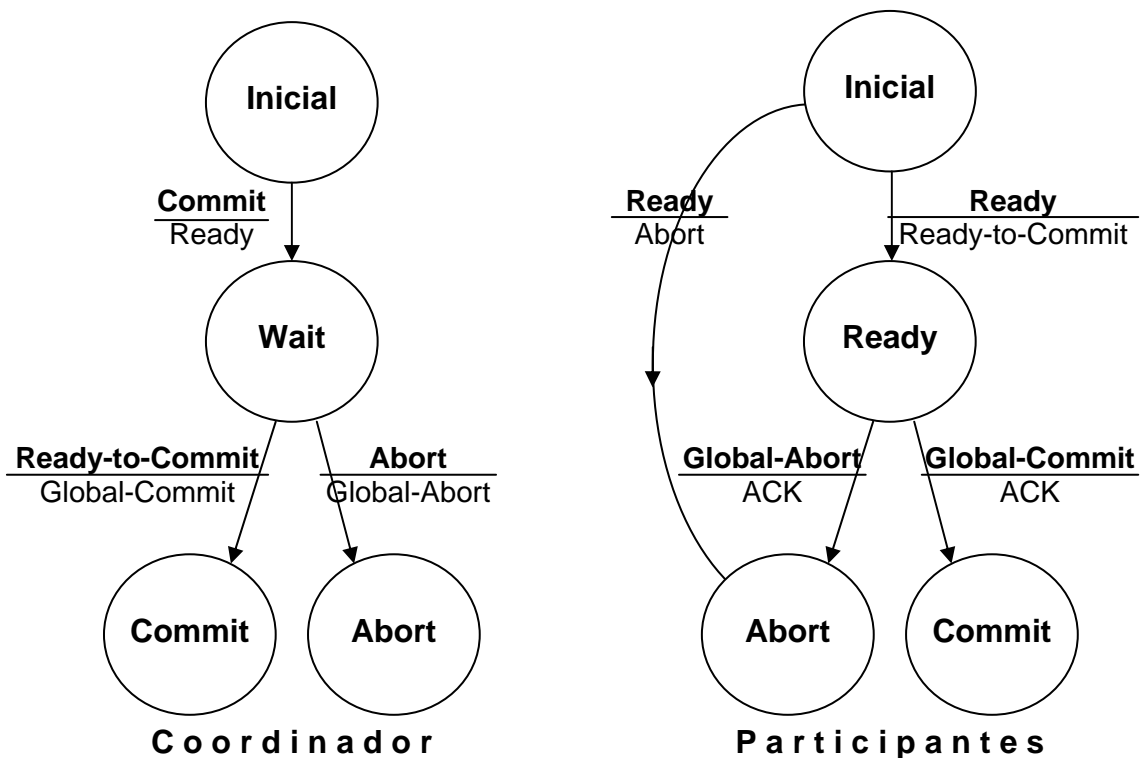


Fig. 13.- Transiciones de Estados en el Protocolo 2PC.

**3.- Estructuras de Comunicación en el Protocolo 2PC.**

**3.1.- Comunicación Centralizada del 2PC.**

La comunicación entre las localidades descrita en la figura 14 se llama *2PC Centralizado*. Sólo se establece comunicación entre el coordinador y los participantes, no la hay entre los participantes.

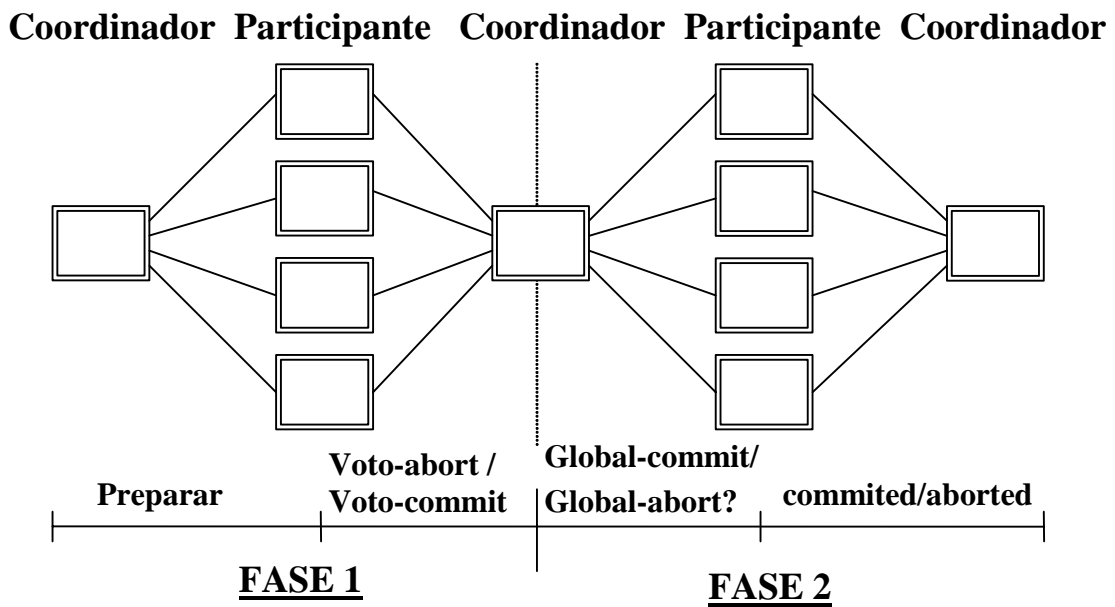


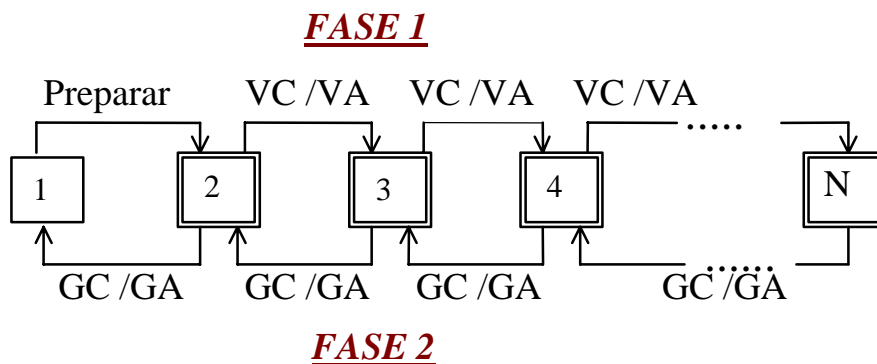
Fig. 14.- Estructura de la comunicación centralizada del Protocolo 2PC.

### 3.2.- Comunicación Lineal del 2PC.

En 2PC Lineal (en Gray, 1979, se llama 2PC Anidado) los participantes pueden comunicarse entre sí, pero en un orden fijo (sea 1, 2, ..., N; estando el coordinador en el primer lugar).

comunicación *hacia adelante* ('forward') desde el n° 1 hasta el N (1ª fase);

comunicación *hacia atrás* ('backward') desde el lugar N hasta el coordinador (2ª fase).



*VC /VA: voto-commit / voto-abort*

*GC /GA: global-commit / global-abort*

Fig. 15.- Estructura de la Comunicación del Protocolo 2PC Lineal

La estructura de comunicación del 2PC lineal no tiene paralelismo alguno y sus tiempos de respuesta son bajos, pero se adecúa para aquellas redes que no pueden hacer emisiones (no broadcasting).

### 3.3.- Comunicación Distribuida del 2PC.

Todos los participantes pueden comunicarse entre sí en la 1ª fase del protocolo. Cada uno de ellos puede decidir terminar la transacción. No se necesita la 2ª fase del protocolo, ya que los participantes pueden decidir por ellos mismos.

El coordinador envía el mensaje *Preparar* a todos los participantes y, cada uno de estos envía lo que ha decidido a todos los demás (participantes y coordinador), mediante un mensaje VA o VC.

Cada participante espera por el mensaje de todos los demás participantes y decide terminar conforme la *regla del commit global*. Por tanto, todos conocen la decisión de todos los participantes al final de la 1ª fase.

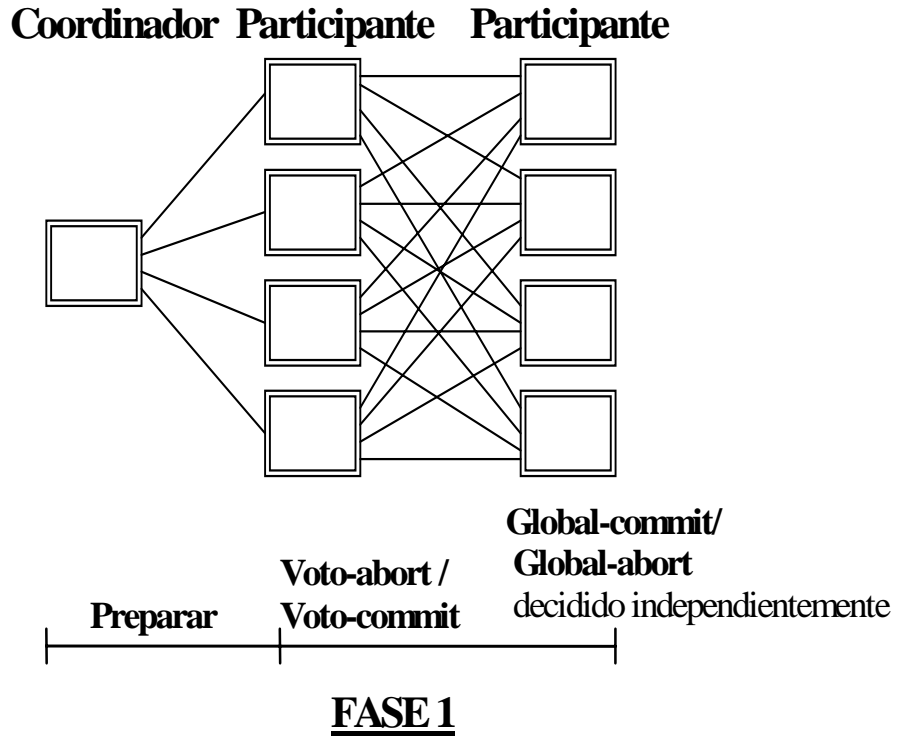


Fig. 16.- Estructura de la Comunicación del Protocolo 2PC Distribuido