# Licenciatura en Informática

Tesina de Grado

# Simulador para la evaluación de tiempos de respuesta en transacciones distribuidas y para el estudio de recuperación de errores



Director
Ing. De Giusti, Armando

Co-Director
Lic. Pesado, Patricia

Alumna: Miaton, Ivana Carla

UNLP - Facultad de Informática, Marzo de 2001

Dedico con todo mi amor este esfuerzo a mis padres que me acompañaron y me brindaron lo mejor "siempre", no solo en estos años de la carrera, sino en toda mi vida. A ellos les debo lo que soy.

A mi hermano.

A mis amigas Carlota, Mónica y Elena.

Al LIDI que me recibió con los brazos abiertos cuando apenas terminaba tercer año, donde me formé y sigo haciéndolo... A toda su gente, que me abrió el corazón, especialmente a "Tito" y al "Pampa" que fueron mi guía en todos estos años.

A los chicos de "Chucaro" y "Sintonizo" que quiero y adoro, y con quienes sentí que podía realizarme profesionalmente.

A todo mi grupo de la Facu, con quienes empecé y seguí esta carrera, compartiendo momentos inolvidables. Muy especialmente a Cecilia, Santiago y Fernando que me dieron lo mejor de lo mejor, particularmente en los momentos más difíciles.

Ivana C. Miaton

## Nota aclaratoria

El software resultante del presente trabajo de grado, se encuentra disponible para su prueba y/o utilización en el Laboratorio de Investigación y Desarrollo en Informática (L.I.D.I.), Facultad de Informática, de la U.N.L.P.

No se entrega junto con este trabajo debido a que para poder probarlo es necesario contar con una red de computadoras y que cada una contenga el JDK 1.3 instalado y configurado apropiadamente.

# Indice

# 1. Objetivo

Un sistema distribuido de tiempo real debe interactuar con el mundo real, en puntos fisicamente distantes, en períodos de tiempo que vienen determinados por el contexto o las restricciones de la especificación (en muchos casos a partir de una activación asincrónica). [HAT88]

La evolución tecnológica en el tratamiento de señales (locales y remotas) y en los sistemas de comunicaciones ha impulsado enormemente esta área temática, sobre todo en los aspectos de planificación, desarrollo y verificación de software para Sistemas Distribuidos de Tiempo Real [LAP88].

Algunas de las dificultades principales del desarrollo de software para sistemas de tiempo real son[SHU92]:

- Procesar mensajes que arriban en forma asincrónica, con diferentes velocidades y diferentes prioridades
- Detectar y controlar condiciones de falla. Prever diferentes grados de recuperación del sistema.
- Modelizar condiciones de concurrencia en un conjunto apropiado de procesos.
- Manejar las comunicaciones inter-procesos e inter-procesadores.
- Proteger datos compartidos por procesos concurrentes.
- Organizar (schedule) y despachar la atención de procesos.
- Manejar las restricciones de tiempo y performance.
- Relacionarse con uno o múltiples relojes de tiempo real.
- Testear y poner a punto un sistema que normalmente está distribuido en diferentes procesadores.
- Elaborar herramientas de software que permitan simular o emular dispositivos o eventos de hardware no disponibles en el desarrollo.
- Reducir y estructurar los requerimientos.
- Seleccionar la estructura de hardware adecuada.

En este contexto de estudio de SDTR y en particular de transacciones con restricciones de respuesta en tiempo, interesa modelar e implementar un ambiente de simulación para el mantenimiento y recuperación de datos en un sistema de BDD, donde se puedan estudiar, monitorear, medir y comparar tiempos de respuesta al ejecutar transacciones distribuidas (concurrentes o no) considerando la posibilidad de fallos de cualquiera de las localidades involucradas en la misma.

Otro aspecto importante dentro del objetivo, es generar de manera parametrizable diferentes trazas de ejecución de transacciones, representando cada una de ellas entornos diferentes de distribución de información.

## 2. Procesamiento Distribuido

Un **sistema de cómputos distribuidos** es un número de elementos de procesamiento autónomos (no necesariamente homogéneos) que están interconectados por una red de computadoras y que cooperan para realizar las tareas que les han sido asignadas. En esta definición, *elemento de procesamiento*, es un dispositivo de computación que puede ejecutar un programa sobre sí mismo.

¿Qué se está distribuyendo? Una de las cosas que puede ser distribuida es la lógica de procesamiento. De hecho la definición de un sistema de computación distribuida asume, implícitamente, que la lógica de procesamiento o elementos de procesamiento están distribuidos. Otra posible distribución es de acuerdo a la *función*. Varias funciones de sistemas de computación podrían ser delegadas a varias piezas de hardware o software. Una tercera opción de distribución es de acuerdo a los datos. Los datos usados por un número de aplicaciones podrían distribuirse en un número de sitios de procesamiento. Finalmente, se puede distribuir el *control*. El control de la ejecución de varias tareas pueden distribuirse en vez de ser realizadas por un sistema de computadoras. Desde el punto de vista de los sistemas de bases de datos distribuidas, esos modos de distribución son todos necesarios e importantes.

Los sistemas de computación distribuidos pueden ser clasificados desde distintos puntos de vista o criterios. Algunos de los criterios son los siguientes: grado de acoplamiento, estructura de interconexión, independencia de componentes, sincronización entre componentes.

El grado de acoplamiento se refiere a la medida que determina cuán estrechamente están conectados los elementos de procesamiento entre sí. Esto podría ser medido como la relación de transformación de la cantidad de intercambio de datos al costo de proceso local que se realizó en ejecutar una tarea. Si la comunicación está hecha sobre una red de computadoras existe un *acoplamiento débil* entre elementos de procesamiento. Sin embargo, si los componentes están compartidos se está hablando de *acoplamiento fuerte*. Los componentes que se comparten pueden ser tanto memoria primaria como dispositivos de almacenamiento secundario. Para la *estructura de*

*interconexión,* podemos hablar sobre aquellos casos que tienen interconexión punto-a-punto entre elementos de procesamiento, en contraposición a aquellos donde el uso del canal de interconexión es común. Los elementos de procesamiento pueden depender entre sí *fuertemente* en al ejecución de la tarea o puede ser mínima mediante el pasaje de mensajes en el comienzo de la ejecución y reportando los resultados al final de la misma. La *sincronización* entre los elementos de procesamiento pueden ser mantenidos sincrónica o asincrónicamente.

Para una perspectiva global, se puede afirmar que la razón fundamental detrás del procesamiento distribuido es para poder solucionar mejor los complejos problemas que se presentan hoy en día, usando una variación de la regla "divide y vencerás". Si se pudiera desarrollar el soporte del software necesario para el procesamiento distribuido, sería posible solucionar estos complejos problemas dividiéndolos en pequeñas piezas y asignándolos a diferentes grupos de software, que trabajan en diferentes computadoras. Esto produce un sistema que corre sobre múltiples elementos de procesamiento, y que puede trabajar eficientemente en la ejecución de una tarea en común.

Este enfoque tiene dos ventajas fundamentales desde el punto de vista económico. En primer lugar, la computación distribuida provee un método económico de aumentar la potencia de cómputo mediante el empleo óptimo de múltiples elementos de procesamiento. La segunda razón es que se pueden atacar estos problemas dividiéndolos en pequeños grupos trabajando más o menos en forma autónoma, puede que esto sea posible para disciplinar del costo del desarrollo del software. De hecho es bien conocido que el costo del software crece en contraposición al costo del hardware.

# 3. Bases de Datos Distribuidas

## 3.1 Sistemas de Bases de Datos Distribuidas

Una **base de datos distribuida** puede ser definida como una *colección de datos compartidos integrados lógicamente que físicamente están distribuidos en nodos de una red de computadoras*. Los dos términos importantes en esta definición son: "integrados lógicamente" y "distribuido en una red de computadoras".

Un **Sistema de Manejo de Bases de Datos Distribuido** (DDBMS) en consecuencia, es el software que permite manejar una base de datos distribuida de manera que los aspectos de distribución sean transparentes al usuario.

Un DBMS centralizado es un sistema que maneja una sola base de datos, mientras que un DDBMS es un solo DBMS que maneja múltiples bases de datos. Los términos *local* y *global* se utilizan a menudo al discutir sobre DDBMS (y sobre los sistemas distribuidos en general) para distinguir entre aspectos que se refieren a un único sitio (local) de los que se refieren al sistema en su totalidad (global). Por ejemplo, si se habla de una base de datos local se está haciendo referencia a la base de datos almacenada en un sitio de la red, mientras que una base de datos global se refiere a la integración lógica de todas las bases de datos locales.

Muchas veces se ha asumido que la distribución física de los datos no es el asunto más significativo. Sin embargo es muy importante. Esto crea problemas que no existían cuando la base de datos estaba en una misma computadora. La distribución física no solamente implica que los sistemas de computadoras estén geográficamente muy apartados, ya que podrían estar en una misma habitación, simplemente implica que la comunicación entre ellas se hace sobre una red en vez de hacerlo mediante memoria compartida, con la red como único recurso compartido.

Un sistema multiprocesador es considerado generalmente un sistema donde dos o más procesadores comparten cierta forma de memoria, si es primaria se le llama *memoria compartida* (Figura 3.1) y si es memoria secundaria, se le llama *disco compartido*. (Figura 3.2)

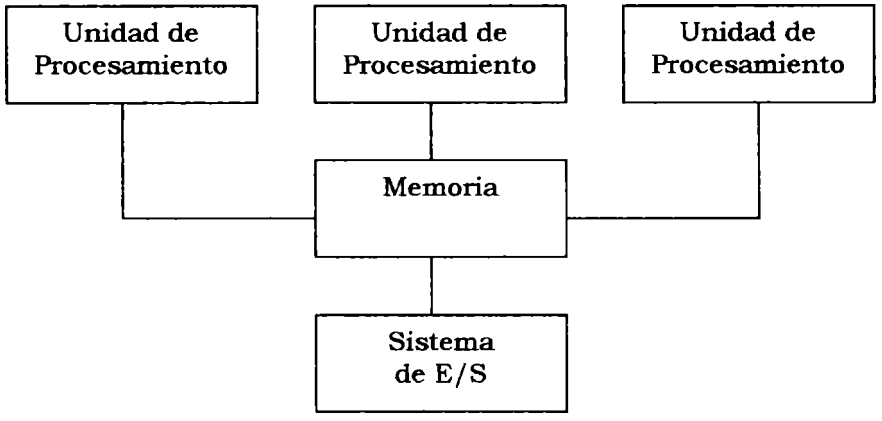



Figura 3.1: Multiprocesador de Memoria Compartida

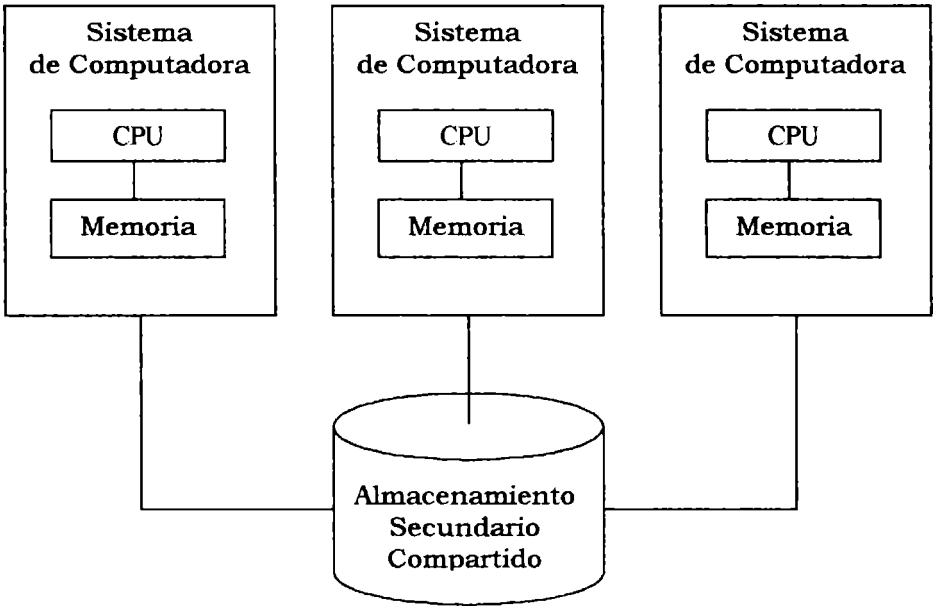



Figura 3.2: Multiprocesador de Disco Compartido

## 3.2 Fundamentos de las Bases de Datos Distribuidas

### 3.2.1 Reglas de Date

La mejor definición de base de datos distribuida fue desarrollada por C. J. Date, quién indicó 12 especificaciones para una base de datos distribuida ideal:

1. Autonomía local.
2. Ninguna confianza en un sitio central.
3. Operación continua.
4. Independencia de la ubicación.
5. Independencia de la fragmentación.
6. Independencia de la replicación.
7. Procesamiento distribuido de consultas.
8. Manejo distribuido de la transacción (proceso de actualización)
9. Independencia del hardware.
10. Independencia del sistema operativo.
11. Independencia de la red
12. Independencia de la base de datos.

Cada una de las especificaciones hace referencia a:

1. Autonomía local. Significa que todos los datos en la red distribuida se manejan localmente.
2. Ninguna confianza en un sitio central. Idealmente, todos los sitios son igualmente "remotos", y no hay ningún sitio que tiene autoridad sobre otro nodo. Cada sitio retiene sus propios diccionarios de datos y seguridad.
3. Operación continua. Mientras cada sitio mantiene su única identidad y control, funcionan como una parte de una federación unificada, tales que otros sitios remotos pueden acceder a información desde el sitio. Cada nodo está disponible a todo el sistema las 24 horas del día, siete

días a la semana. Para lograr esta meta, los sitios remotos pueden tener utilidades "flying dump" o "hot backup" para realizar backups de la base de datos mientras está disponible para actualizaciones.

4. Independencia de la ubicación. Los usuarios finales no necesariamente conocen, o les interesa, la ubicación física de las bases de datos que abarca el sistema. Los datos son recuperados sin ninguna referencia específica a la ubicación física.

5. Independencia de la fragmentación. Se refiere a la capacidad del usuario final de almacenar lógicamente información relacionada en diferentes ubicaciones físicas. Normalmente se utiliza por razones de performance, disponibilidad y confiabilidad. Además, la fragmentación puede reducir los efectos negativos de la replicación. Cada réplica no es la relación en su totalidad, sino solo un subconjunto de la misma; de esta manera se reduce el espacio requerido y son menos los ítems de datos que necesitan ser manejados.

   Hay dos alternativas para realizar la fragmentación: una es llamada *fragmentación horizontal* y la otra *fragmentación vertical*:

   Con la fragmentación horizontal, una relación es particionada en un conjunto de sub-relaciones cada uno de los cuales tiene un subconjunto de tuplas (filas) de la relación original. Esto es usado normalmente por organizaciones que tienen varias sucursales, cada una con un conjunto de estructuras de tablas idénticas.

   Fragmenación vertical es aquella donde cada sub-relación se define en subconjuntos de atributos (columnas) de la relación original.

   Cuando objetos de una base de datos son fragmentados, tenemos que distribuir el problema de manejos de consultas del usuario que fueron especificadas sobre la totalidad de la relación pero ahora se tiene que procesar sobre sub-relaciones. En otras palabras, la cuestión es encontrar una estrategia de procesamiento de consultas basadas en fragmentos en vez de sobre relaciones, aunque las consultas se hayan especificado sobre estas últimas (relación).

6. Independencia de la replicación. Replicación es la capacidad de una base de datos de crear copias de una base de datos maestra en sitios remotos. Estas copias, a veces llamadas *snapshots*, pueden contener la

base de datos completa o cualquier componente de ella. Existen varias razones por las cuales es deseable distribuir datos de manera replicada en las máquinas de una red, entre ellas por performance, disponibilidad, confianza. Por ejemplo, los datos que son normalmente accedidos por un usuario pueden ser ubicados en la máquina local del mismo así como en la máquina de otro usuario con el mismo requerimiento de acceso. Esto incrementa los lugares de referencia. Además, aunque una de las máquinas falle, habrá copias disponibles en otra máquina de la red. Por supuesto, esta es una simple descripción de la situación. De hecho la decisión de replicar o no y cuántas copias de objetos de bases de datos deben tener, depende de las aplicaciones del usuario. Cabe destacar que la replicación puede causar problemas en la actualización de bases de datos, por consiguiente, si las aplicaciones del usuario son en su mayoría orientadas a la actualización, no es una buena idea tener demasiadas copias de bases de datos.

Asumiendo que los datos están replicados, el tema de la transparencia está en si los usuarios deberían ser concientes tanto de la existencia de estas copias como del sistema que maneja la administración de estas copias, actuando el usuario como si tuviese una única copia de los datos (cabe destacar que no se hace referencia a la ubicación de las copias sino solo a su existencia). Desde la perspectiva de los usuarios la respuesta es obvia. Es preferible no involucrarse con el manejo de copias y tener que especificar el hecho de que ciertas acciones puedan y/o deban ser tomadas de múltiples copias. Desde el punto de vista de los sistemas, sin embargo, la respuesta no es tan simple. Cuando la responsabilidad de especificar una acción que necesita ser ejecutada en múltiples copias es delegada al usuario, hace del Transaction Manager (Manejador de Transacciones) simple para DBMSs distribuidos. Por otra parte haciendo esto, los resultados perderán flexibilidad. No es el sistema el que decide si tener copias o no, y cuántas hay que tener, sino la aplicación del usuario. Cualquier cambio en estas decisiones debido a la variación de consideraciones, afecta a la aplicación del usuario y por lo tanto reduce la independencia de datos considerablemente.

Teniendo en cuenta todas estas consideraciones, es deseable que la transparencia de replicación sea provista como una característica standard de los DBMSs. Hay que recordar que la tranparencia de la replicación se refiere solo a la existencia de réplicas, no de su actual ubicación. También cabe destacar que la distribución de esas réplicas a través de la red en una manera transparente es dominio de la *transparencia de red.*

7. Procesamiento distribuido de consultas. Es más que la capacidad de ejecutar una consulta sobre más de una base de datos. En varios motores de bases de datos, la consulta es ejecutada en el nodo en donde está el usuario, mientras que otras bases de datos parten la consulta distribuida en subconsultas, ejecutando cada una en el procesador del host.

8. Manejador de transacciones distribuidas (procesamiento de actualización). Se refiere a un sistema que puede manejar una actualización, una inserción o eliminación en múltiples bases de datos con una simple consulta. Muchas veces se utiliza el protocolo de Commit de Dos Fases (two-phase commit).

9. Independencia del hardware. Se refiere a la capacidad de una consulta que consulte y actualice información sin importar la plataforma de hardware en la que residen los datos.

10. Independencia del sistema operativo. Una consulta no debe ser dependiente del sistema operativo.

11. Independencia de la red. En los sistemas de bases de datos centralizados, el único recurso disponible que necesita ser protegido del usuario son los datos. En los de bases de datos distribuidas, sin embargo, existe un segundo recurso que necesita ser administrado en más de una forma: la red. Preferentemente, el usuario no debería conocer los detalles operacionales de la red. Incluso, si es posible, es deseable ocultar la existencia de la red. Entonces no habría diferencia entre aplicaciones de base de datos que corren en una base de datos centralizada y las que corren en bases de datos distribuidas.

    Podemos considerar la independencia de la red desde los distintos puntos de vista tanto de los servicios provistos como de los datos.

Desde la anterior perspectiva, es deseable tener una definición uniforme a través de los cuales se accede a los servicios. Desde el punto de vista del DBMS, la independencia de la red requiere que los usuarios no tienen que especificar dónde están ubicados los datos.

12. Independencia de la base de datos. Con la independencia de la base de datos, es posible recuperar y actualizar información desde diferentes bases de datos y arquitecturas de bases de datos.

### 3.2.2 Integridad de Datos

La integridad de datos se refiere a la capacidad de las bases de datos de manejar actualizaciones concurrentes de datos que están en varias ubicaciones físicas y de asegurar que todos ellos sean física y lógicamente correctos. En una única base de datos, la integridad de datos es manejada de manera muy efectiva, mientras que en las distribuidas es mucho más complejo.

Se pueden dividir los problemas en tres grupos principales:

1) Inconsistencias entre las restricciones de integridad locales
2) Dificultades en especificar las restricciones globales de integridad.
3) Inconsistencias entre restricciones locales y globales.

Restricciones de Integridad Local. Las restricciones de integridad local son especificadas y aplicadas por el DBMS.

Restricciones de Integridad Global. Puede verse con un ejemplo, la necesidad de tener restricciones de integridad globales. Supóngase una organización con varias departamentos semi-autónomos, cada uno corriendo sobre su DBMS el manejo de sus propios empleados. Si un individuo trabaja part-time en más de un departamento de la organización, para mantener el status de part-time ese empleado no debería trabajar más de 20 horas por semana. Es obvio que no es suficiente aplicar las restricciones locales ya que

cada uno restringe a 20 la cantidad de horas y sin embargo, sumando las horas trabajadas en todos los departamentos, podría sumar más de 20 horas.

Inconsistencias entre restricciones de integridad locales y globales. Si se permiten tanto las restricciones de integridad locales como las globales, debe haber una manera de resolver las inconsistencias entre ellas. En el caso de los empleados anteriormente citado, por ejemplo, se asume que la base de datos local y la global, tienen ambas especificado un límite máximo de 20 horas. Qué sucede si una base de datos local ($BD_1$), quiere ingresar un valor de 15 horas a un empleado $X$ y otra base de datos local ($BD_2$) quiere ingresar 10 (para la misma semana en el mismo momento). Si $BD_2$ rechaza la transacción, se estaría violando su autonomía; pero si la aceptan las dos, se estaría violando la integridad global. Probablemente si esta situación ocurriera en el mundo real, ambos departamentos serían notificados y tomarían la acción correcta en conjunto. En un ambiente de bases de datos distribuidas, por lo tanto, necesitamos la facilidad de definir triggers globales que puedan permitir que se realice la actualización de $BD_2$, pero notificando tanto a $BD_1$ como a $BD_2$.

La resolución a conflictos de integridad locales y globales, dependerán principalmente del grado de integración de los varios sitios que participan de la base de datos distribuida. Si existe un fuerte control global, es probable que tengan prioridad sobre las restricciones locales. Por otro lado, si la base de datos está participando en una pobre federación, con un control virtualmente no global, entonces las restricciones de integridad local dominarán a las globales.

El diseño de las restricciones globales forman una parte importante del esquema de integración. Por ejemplo el tema de chequear que el estado actual de una base de datos local no viole una nueva especificación de restricción de integridad global, queda en manos del **management policy level** (nivel de la política de administración).

La recuperación en un ambiente de una base de datos distribuida involucra que la transacción en su totalidad sea completada de manera exitosa antes de que cometa para cada componente de la transacción en su totalidad.

## 3.3 Control de Concurrencia

### 3.3.1 Transacciones – Conceptos Básicos

Una transacción se define como una serie de acciones, realizado por una aplicación del usuario, la cual debe ser tratada como una unidad indivisible. Las transacciones convierten la base de datos de un estado consistente a otro consistente, a pesar de que la consistencia haya sido violada durante la ejecución de la transacción. Si durante la ejecución de la transacción ocurre una falla, el Manejador de Recuperación del DBMS debe asegurar que todas las transacciones activas en el momento de la falla hagan un **rollback**. El efecto de la operación rollback es para recuperar la base de datos al estado en que se encontraba antes de comenzar la transacción y así quedar en un estado consistente. Hay cuatro propiedades básicas de una transacción (también llamadas A.C.I.D.):

- Atomicidad: Es la propiedad de "todo o nada". Una transacción es una unidad indivisible.
- Consistencia: las transacciones transforman la base de datos de un estado consistente a otro.
- Independencia: Las transacciones se ejecutan independientemente de otra transacción; es decir que los efectos parciales de una transacción incompleta no son visibles al resto de las transacciones.
- Durabilidad (o persistencia): Los efectos de una transacción que terminó satisfactoriamente (cometida) son almacenados permanentemente en la base de datos y no pueden ser cancelados.

El *Transaction Manager* es el encargado de supervisar la ejecución de las transacciones y de coordinar las peticiones de la base de datos que realiza la transacción. Por otro lado el *Scheduler* implementa una estrategia particular para la ejecución de las transacciones. El objetivo del *scheduler* es maximizar la concurrencia sin permitir la ejecución concurrente de transacciones que interfieran en otra, y así comprometer la integridad o la consistencia de la base de datos. El *Transaction Manager* y el *scheduler* están claramente relacionados ya que la respuesta del *Transaction Manager* a una petición de la base de datos de una aplicación dependerá del *scheduler* utilizado.

### 3.3.2 Transacciones Distribuidas

En un ambiente de bases de datos distribuidas, las transacciones pueden acceder a datos almacenados en más de un lugar. Cada transacción es dividida en un número de sub-transacciones, una por cada lugar en donde los datos accedidos por al transacción están almacenados. Estas sub-transacciones están representadas por **agentes** en los distintos sitios.

La indivisibilidad de toda la transacción global es fundamental, pero además cada sub-transacción, o *agente,* de la transacción global debe ser tratada como una transacción indivisible en el sitio donde está ejecutándose. Las sub-transacciones de la transacción global no sólo deben ser sincronizadas con otras transacciones concurrentes locales, sino que además deben sincronizarse con otras transacciones globales que se ejecutan concurrentemente en el sistema. Por lo tanto la distribución agrega una nueva dimensión de complejidad para el problema de control de concurrencia.

### 3.3.3 Interferencia entre Transacciones Concurrentes

Hay varias formas en que la ejecución de una transacción concurrente puede interferir en otra, y comprometer la integridad y consistencia de la base de datos:

- Problemas de actualizaciones perdidas. Sucede cuando una operación de actualización de un usuario aparentemente completa es sobreescrita por otro usuario.
- Violación a las restricciones de integridad. Puede suceder cuando se le permite a dos transacciones que ejecuten concurrentemente sin haber sido sincronizadas.
- Problema de recuperación inconsistente. La mayor parte del trabajo del control de concurrencia se concentra en las transacciones que están actualizando la base de datos, puesto que su interferencia puede corromper la base de datos. Sin embargo, las transacciones que sólo están leyendo de la base de datos pueden obtener resultados inexactos si

se permiten leer resultados parciales de las transacciones incompletas que están actualizando simultáneamente la base de datos. Esto normalmente se lo llama *dirty read* (lectura sucia).

### 3.3.4 Técnicas de Control de Concurrencia

Básicamente hay tres técnicas de control de concurrencia que permiten que las transacciones se ejecuten en paralelo con seguridad conforme a ciertas restricciones:

1- Métodos de Bloqueo.

2- Métodos de *TimeStamp*

3- Métodos Optimistas.

Estos métodos han sido desarrollados principalmente para DBMS centralizados y luego fueron extendidos para el caso distribuido. Tanto el Bloqueo como el *TimeStamp*, son enfoques básicamente conservadores que causan que las transacciones sean retrasadas en caso de que puedan estar en conflicto con otras transacciones en algún momento. Los métodos optimistas se basan en la premisa de que el conflicto es inusual y permite que las transacciones procedan asincrónicamente, y sólo controla si hay conflictos al final, cuando la transacción ya cometió.

#### 3.3.4.1 Métodos de Bloqueo

Los métodos de bloqueos son los más usados para el control de concurrencia en los DBMSs. Hay varias diferencias, pero todas comparten la misma característica fundamental: que una transacción deba reclamar un bloqueo de lectura (compartida) o escritura (exclusiva) sobre los datos antes de la ejecución de la correspondiente operación de lectura o escritura de los datos.

Dado que las operaciones de lectura no producen conflictos, se permite que más de una transacción sostenga los bloqueos de lectura

simultáneamente sobre el mismo dato. Por otro lado, un bloqueo de escritura da un acceso exclusivo al dato. De esta manera, mientras la transacción mantenga el bloqueo de escritura sobre el dato, ninguna otra transacción puede leer o actualizar dicho dato. Una transacción mantiene el bloqueo hasta que sea liberado explícitamente y solo a partir de ese momento la operación de escritura se hará visible a otras transacciones. Por consiguiente pueden haber varios bloqueos de lectura al mismo tiempo sobre un mismo dato, pero la existencia de un bloqueo de escritura sobre un dato imposibilita la existencia de otros bloqueos de lectura o escritura sobre ese dato.

Si una transacción mantiene un bloqueo de lectura sobre un dato, entonces puede promocionar ese bloqueo a uno de escritura si es la única transacción que mantiene un bloqueo sobre el dato. Esto efectivamente permitirá que la transacción primero examine el dato y luego decida si quiere o no actualizarlo. Cuando la promoción no está permitida, una transacción debe mantener los bloqueos de escritura sobre todos los datos que puede llegar a decidir actualizar durante la ejecución de la transacción, por consiguiente, se reduciría potencialmente el nivel de concurrencia del sistema. Asimismo, si una transacción mantiene un bloqueo de escritura sobre un dato, puede degradar ese bloqueo a uno de lectura, una vez que haya terminado de actualizarlo. Esto permite mucha mayor concurrencia.

La magnitud de la granularidad de los datos que pueden ser bloqueados en una única operación tendrá un efecto significativo en toda la performance del algoritmo de control de concurrencia. Considérese una transacción que está actualizando un dato en una única tupla de una relación. En un extremo, el algoritmo de control de concurrencia podrá permitir que la transacción bloquee solo esa tupla, mientras que en el otro extremo, puede tener que bloquear la base de datos en su totalidad. En el primer caso, la magnitud de granularidad para el bloqueo es de una única tupla; mientras que en el segundo caso, es toda la base de datos y prevendría la ejecución de cualquier otra transacción hasta que el bloqueo sea liberado (esto es sin dudas indeseado). Por otro lado, si la transacción va a actualizar el 90% de las tuplas de la relación, entonces sería más eficiente permitir que bloquee la relación entera antes que forzarla a bloquear cada tupla individual por separado. Idealmente el DBMS debe soprtar una granularidad mixta con niveles de bloqueo de tuplas, páginas y relaciones. Muchos sistemas automáticamente

promueven bloqueos de tuplas/páginas a bloqueos de relación, si una transacción particular está bloqueando más de un determinado porcentaje de tuplas/páginas.

El protocolo de bloqueo más común es conocido **como "Bloqueo de Dos Fases" (2PL).** Es llamado así porque el bloqueo se realiza en dos fases distintas: una fase de **crecimiento (upgrading)**, durante la cual la transacción adquiere bloqueos; y una fase de **decrecimiento (downgrading)** durante la cual se liberan esos bloqueos.

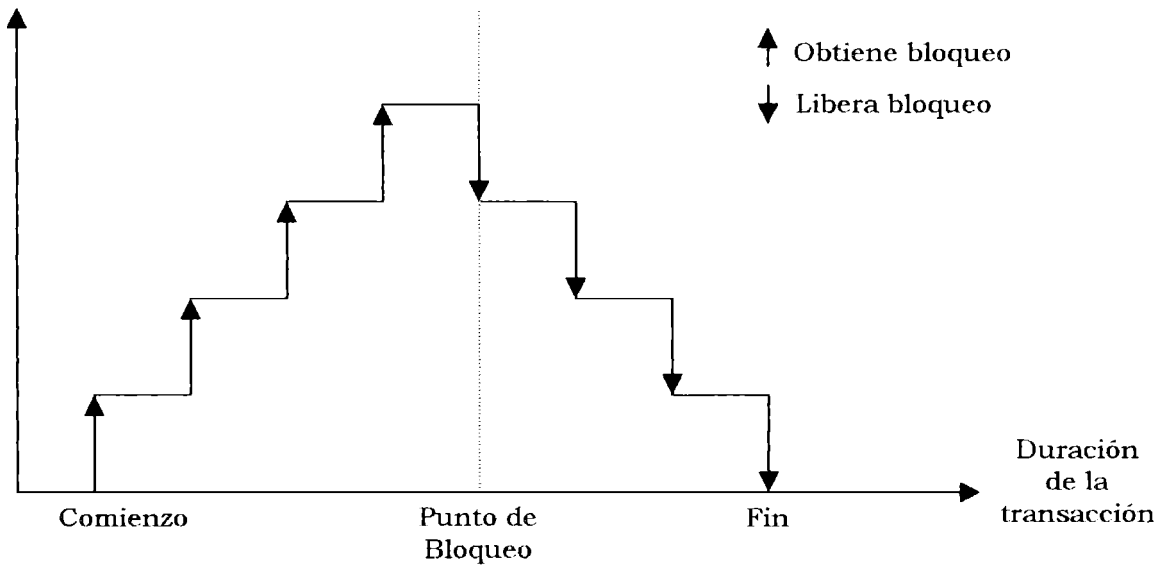



Figura 3.3: Bloqueo de Dos Fases

Las reglas para las transacciones que siguen el Bloqueo de Dos Fases son:

1- Las transacciones están "bien formadas", es decir, una transacción debe adquirir un bloqueo sobre un dato antes de operar sobre él, y todos los bloqueos sostenidos por una transacción deben ser liberados cuando la transacción termina.

2- Se observan las reglas de compatibilidad para el bloqueo, es decir se mantienen los bloqueos que están en conflicto (los conflictos de escritura-ecritura y lectura-escritura están prohibidos).

3- Una vez que la transacción libera el bloqueo, no puede adquirir nuevos bloqueos.

4- Todos los bloqueos de escritura son liberados cundo la transacción comete.

Es necesaria la condición cuatro para asegurar la atomicidad de la transacción, de lo contrario otras transacciones podrán ver resultados parcialmente "no-cometidos" de la transacción; pero hay que notar que los desbloqueos solo están permitidos en la fase de decrecimiento.

El objetivo es producir una planificación correcta, permitiendo que las operaciones de la transacción sean interpoladas de una manera tal que no interfieran con otro y no comprometan la integridad de la base de datos. La seriabilidad es tomada como una prueba de correctitud. Sin embargo, debido a la complejidad de decidir si una planificación es serializable (y por lo tanto correcta) o no, es preferible diseñar solamente planificadores que garanticen la producción de planificadores serializables. La seriabilidad se toma como prueba de la corrección. Sin embargo, dada la complejidad de decidir si un determinado planificador es serializable, y por lo tanto correcto, o no, es preferible diseñar solo planificadores serializables.

El problema principal con el bloqueo de dos Fases en un ambiente distribuido, es la enorme sobrecarga de mensajes. Consideremos, por ejemplo, una transacción global que genera agentes en $n$ sitios. La ejecución exitosa de esa transacción requerirá un mínimo de $4n$ mensajes:

- n mensajes de comienzo de la transacción, desde el coordinador a los agentes,
- n mensajes de listo para cometer la sub-transacción, desde el coordinador.
- n mensajes de cometer la transacción global, a los agentes.
- n mensajes de haber cometido la sub-transacción local, al coordinador.

De hecho, probablemente hay más de 4n mensajes intercambiados porque la mayoría de los mensajes requieren normalmente un acuse de recibo que deben ser enviados del receptor al remitente, duplicando el número de mensajes. Si hay muchas transacciones que involucran múltiples sitios, la

sobrecarga de mensajes sería inaceptable. Afortunadamente, uno de los principales objetivos para el uso de la tecnología de bases de datos distribuidas, es el almacenamiento en el sitio de los datos más frecuentemente utilizados. En otras palabras muchas de las actualizaciones involucrarán a datos puramente locales, haciendo del 2PL distribuido una opción realista. Todos los mensajes de cada uno de los "rounds" generalmente pueden enviarse en paralelo. De este modo la distribución de datos entre sitios es una consideración importante.

Hay que considerar también el impacto de la replicación de datos en el algoritmo de control de concurrencia 2PL. Si un dato es actualizado, entones es deseable que todas las copias del dato sean actualizadas 'simultáneamente'. De este modo la actualización de datos replicados, resulta en una transacción global cuyo propósito es propagar la actualización a todos los sitios que contienen réplicas del dato. En el caso de los protocolos de bloqueo como el 2PL, cada Transaction Manager local debe adquirir un bloqueo de escritura sobre la copia del dato almacenado en su sitio. Un enfoque simple y más eficiente es diseñar una copia como copia primaria para cada dato replicado. Todas las réplicas son referidas como copias esclavas. Así, sólo es necesario un bloqueo de escritura sobre la copia primaria mientras los datos tienen que ser actualizados. Una vez que la copia primaria ha sido alterada, la actualización puede ser propagada a las copias esclavas. La propagación debe ser hecha tan pronto como sea posible para prevenir que alguna transacción lea versiones de datos viejas. Sin embargo, no tiene que ser realizado como una transacción global atómica. Se garantiza que sólo la copia maestra tiene un valor actualizado y consistente. De esta manera tanto las transacciones que desean actualizar un dato y los que requieren una copia consistente y actualizada, dirigirán sus peticiones a la copia principal de todos modos.

#### 3.3.4.2 Métodos de *TimeStamp*

Los métodos de control de concurrencia con *TimeStamp* no involucran bloqueos y por lo tanto no tendrán "dead-lock". Los métodos de bloqueo generalmente involucran transacciones que hacen conflictiva la espera del requerimiento. Con los métodos *TimeStamp*, no hay espera: las transacciones involucradas en conflictos simplemente son deshechas y reiniciadas.

El objetivo fundamental de los métodos *TimeStamp* es para ordenar globalmente las transacciones, de manera tal que las transacciones envejecidas, las de menor *TimeStamp*, tengan prioridad cuando haya conflictos. Si una transacción intenta leer o escribir un dato, se le permitirá realizar sólo si la última actualización fue realizada sobre ese dato; y se le da un nuevo *TimeStamp*. Cabe destacar que el nuevo *TimeStamp* debe ser asignado para reiniciar la transacción para prevenir que tengan continuamente denegado el *"commit"*. En la ausencia de nuevos *TimeStamp*, una transacción con un *TimeStamp* viejo (tanto porque realmente tenía ese tiempo de vida o porque había sido reiniciado muchas veces) no pueden cometer debido a que transacciones más jóvenes están cometiendo. Los métodos *TimeStamp* producen planificadores serializables, que son equivalentes a los planificadores definidos por el *TimeStamp* de las transacciones que cometieron satisfactoriamente.

Los TimeStamps son usados para ordenar transacciones con respecto a otras. A cada transacción se le asigna un único TimeStamp cuando es iniciada, por lo que dos transacciones nunca tendrán el mismo TimeStamp. En los DBMSs centralizados, pueden ser generados simplemente con el reloj del sistema. De esta manera la transacción tendrá el valor del reloj del sistema de cuando comenzó. Alternativamente, se puede usar un simple contador global o un generador de números secuenciales. Cuando inicia una transacción, se le asigna el siguiente valor del contador de transacciones y luego es incrementado. Para eludir la generación de TimeStamps muy extensos, el contador periódicamente se puede poner a cero.

Consideraciones de los *TimeStamps* en un ambiente distribuido. En un sistema distribuido no hay un reloj del sistema global (o un contador centralizado), más bien nodos en una red que tiene cada uno su propio reloj y no hay garantía de que esos relojes estén sincronizados con el resto. Más aún, en un ambiente centralizado, dos eventos normalmente no pueden ocurrir al mismo tiempo; en particular el *Transaction Manager* puede iniciar sólo una transacción a la vez. Por el contrario, en un sistema distribuido dos o más transacciones pueden comenzar simultáneamente en diferentes sitios de la red. Usando serializabilidad como garantía de la corrección de la planificación, debe haber un mecanismo por el que estos dos acontecimientos "simultáneos" puedan ser ordenados uno con respecto a otro.

Una simple aproximación para soportar la noción de tiempo global en un sistema distribuido es definir un **tiempo global** como la concatenación de reloj local del sistema con el identificador del sitio, esto es:

<reloj del sitio, identificador del sitio>

Mientras es más fácil pensar en relojes en términos del sistema físico, se utiliza generalmente un contador global que se puede controlar. En particular, a veces es necesario adelantar un reloj local para asegurarse de que un acontecimiento en un sitio ocurra después de un acontecimiento en otro sitio, lo cual no sería posible si se utiliza el reloj del sistema. En la práctica, normalmente se utiliza el término **reloj** ("clock") para referirse al contador global en sí mismo y **timestamp** para indicar el valor del contador cuando ocurre un evento particular.

Las siguientes reglas son suficientes para asegurar el orden de los eventos tanto respecto a otros eventos locales, como con respecto a eventos que ocurren en todos los sitios:

1- El reloj local del sitio está adelantado en una unidad para cualquier evento que ocurra en ese sitio; los eventos de interés son los comienzos de transacciones y el envío y recepción de mensajes.

2- A los mensajes entre sitios les da el timestamp el emisor; cuando un sitio B recibe un mensaje de un sitio A, B adelanta el reloj local a:

Max(timestamp del mensaje, reloj B)

Donde reloj B es el valor del reloj local del sitio B.

La regla 1asegura que si un evento $e_i$ ocurre antes que un evento $e_j$ en el sitio A, entonces, llamando **ts** al timestamp:

$$ts(e_i) < ts(e_j)$$

donde $ts(e_i)$ y $ts(e_j)$ son valores del reloj del sitio A cuando los eventos $e_i$ y $e_j$ ocurren respectivamente. La regla 2 efectivamente mantiene el grado de sincronización de los relojes locales entre dos sitios que se están comunicando, de manera tal que si el evento del sitio $e_i$ A está enviando un mensaje al sitio B en el momento $t_i$, entonces se puede asegurar que el evento $e_k$, el receptor del mensaje en el sitio B, ocurre en el momento $t_k$ siendo $t_k > t_i$. Si no hay comunicación entre dos sitios, entonces sus relojes se alejarán, pero

esto no importa ya que al no existir dicha comunicación, no hay necesidad de sincronización entre los sitios.

Consideraciones respecto de la atomicidad de las transacciones con *TimeStamp*. El propósito de la operación ***commit*** realizada por una transacción es para hacer que la actualización realizada por la misma sea permanente y visible a otras transacciones (para asegurar la atomicidad y durabilidad de la transacción).

Las transacciones que han sido cometidas nunca pueden ser deshechas. Con los protocolos basados en bloqueo, la atomicidad de la transacción estaba garantizada por los bloqueos de escritura sobre todos los registros hasta el momento de cometer, y particularmente en 2PL, todos los bloqueos se liberan en ese momento. Con los protocolos de *TimeStamp*, sin embargo, no existe la posibilidad de evitar que otras transacciones vean actualizaciones parciales puesto que no hay bloqueos. Por lo tanto se debe adoptar un enfoque distinto que efectivamente oculte las actualizaciones parciales. Esto se hace usando **pre-escrituras** (**actualizaciones diferidas**). Las actualizaciones de las transacciones no cometidas no se escriben en la base de datos, en lugar de eso se escribe en un conjunto de buffers que solo pasará a la base de datos cuando la transacción cometa. Este enfoque tiene la ventaja de que cuando la transacción es abortada y reiniciada, no se necesitan realizar cambios físicos a la base de datos. Este no es el único método, existen otras variaciones como el método Conservador.

### 3.3.4.3 Métodos Optimistas

Los métodos de control de concurrencia anteriores son por naturaleza pesimistas. En otras palabras, asumen que los conflictos entre transacciones son muy frecuentes, y no permiten que una transacción acceda a un dato si hay conflictos en el acceso al mismo. De esta manera la ejecución de cualquier operación de una transacción sigue la secuencia de estas fases: validación (V), lectura (R), computación (C), escritura (W) (Figura 3.4)

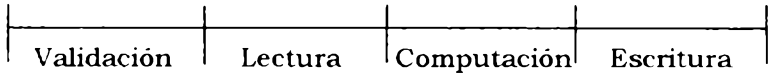



Figura 3.4: Fases Pesimistas de la ejecución de una transacción

Los métodos optimistas de control de concurrencia están basados en la premisa de que son poco comunes los conflictos y el mejor acercamiento es permitir que las transacciones procedan sin obstáculo por métodos complejos de la sincronización y sin tener que esperar. Cuando una transacción desea cometer, el sistema controla los conflictos y cuando ocurre uno, reinicia la transacción. Con optimismo se permite que la transacción proceda lo más que sea posible. Para asegurar la atomicidad de la transacción todas las actualizaciones de los datos son realizadas en copias locales y sólo son propagadas a la base de datos cuando no se hayan detectado conflictos. En el evento con conflicto, la transacción es deshecha y reiniciada. La sobrecarga involucrada en la reiniciación de la transacción es claramente considerable, ya que efectivamente debe ser hecha nuevamente toda la transacción. Esto se puede tolerar sólo si sucede muy poco frecuentemente. Las transacciones se ejecutan en tres pasos:

1- **Fase de lectura**: esta fase representa el cuerpo de la transacción a ser cometida (no hay escrituras en la base de datos durante esta fase).

2- **Fase de validación**: los resultados (actualizaciones) de la transacción son examinados para ver si hay conflictos.

3- **Fase de escritura**: si la transacción es validada, entonces las actualizaciones se propagan desde la copia local a la base de datos. Si durante la fase de validación se encontraron conflictos que pueden resultar en una pérdida de integridad, entonces la transacción es deshecha y reiniciada.

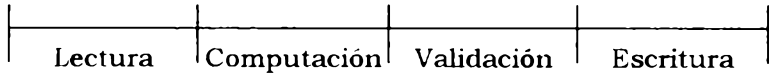



Figura 3.5: Fases Optimistas de la ejecución de una transacción

Los ambientes de baja argumentación o los ambientes donde predominan las transacciones de solo lectura, hacen que el control optimista de concurrencia sea atractivo, ya que permite que la mayoría de las transacciones procedan sin obstáculos por la sobrecarga de la sincronización. Sin embargo, ninguno de los prototipos principales de DDB utilizan este método.

Las fases de lectura y escritura de los métodos de control de concurrencia optimistas son directos, excepto que las escrituras durante la fase de lectura son internas de la transacción. La fase de escritura involucra el *"commit"* y *"rollback"* estándar.

La ventaja de los métodos de control de concurrencia optimistas es su potencial para permitir un alto nivel de concurrencia. El mayor problema con estos métodos es el alto costo de almacenamiento. Para validar una transacción tiene que almacenar las lecturas y los conjuntos de escrituras de varias otras transacciones. Específicamente, las lecturas y escrituras de las transacciones terminadas que estaban corriendo cuando la transacción $T_{ij}$ llegó al sitio $j$ necesitan ser almacenadas para validar $t_{ij}$. Obviamente esto incrementa el costo de almacenamiento.

### 3.3.5 Concurrencia en Bases de Datos Replicadas

La replicación de datos en bases de datos distribuidas es común por razones tanto de performance como por la tolerancia a fallos. Cuando un DDBMS soporta replicación, el sistema debe asegurar que todas las copias de los datos replicados, sean consistentes. Esto se puede hacer tratando la actualización de los datos replicados como una transacción atómica distribuida, que puede ser manejada por el planificador de manera normal. Tal enfoque requeriría que todas las réplicas estén almacenadas para ser

operacionales y para estar conectadas a la red. En el caso de una falla de la red no sería posible actualizar dichas copias replicadas en todos los sitios. Claramente este enfoque se ejecutaría opuestamente al objetivo de la tolerancia a fallas de la replicación. En consecuencia, es común adoptar el método del consenso, por el cual si la mayoría de los sitios opta por aceptar la actualización de una copia replicada, entonces el planificador global da instrucciones a todos los sitios de actualizar (cometer). Los sitios que son inaccesibles durante la actualización, tanto por una falla local como por una falla de la red, simplemente pueden ser notificados de la actualización cuando se reincorporen a la red.

Uno de los métodos de control de concurrencia optimistas más fáciles para las bases de datos replicadas, es el desarrollado por Thomas. Como en el resto de los métodos de control de concurrencia optimistas, las transacciones se ejecutan en tres fases:

1- **Fase de lectura** durante la cual las actualizaciones se realizan únicamente en la copia de los datos local;

2- **Fase de validación** durante la cual la actualización propuesta es evaluada para ver si hay conflictos en todos los sitios

3- **Fase de escritura** durante la cual se comete la transacción.

El método de validación de Thomas está basado en timestamp para las transacciones y los datos en bases de datos totalmente replicadas. Las transacciones se ejecutan en su totalidad en un sitio. Junto con cada copia de cada dato se guarda también el timestamp de la última transacción que actualizó con éxito ese dato. Para tener una consistencia global , el valor del timestamp debe ser el mismo para todas las copias de un determinado dato.

En resumen, el método procede como sigue: al entrar el sitio S en la fase de validación, la transacción *Ts* envía los detalles de lo que leyó, escribió y su correspondiente *timestamp* al resto de los sitios. Cada sitio entonces valida *Ts* comparando con su estado local y luego opta si la acepta o rechaza. Si la mayoría "acepta", entonces *Ts* comete y se le notifica a todos los sitios. Para validar *Ts*, el sitio I chequea el *timestamp* del conjunto de lecturas con el *timestamp* de las copias locales de los datos correspondientes. Si fueran los mismos, significa que si la actualización realizada por *Ts* fuera propagada al sitio I, no produciría inconsistencias en el resultado. Si el timestamp para un

único dato fuera diferente, indicaría que *Ts* ha leído datos inconsistentes. También es necesario validar $T_s$ para cada sitio I el resto de las transacciones pendientes (concurrentes) en el sitio *I.* Si una transacción pendiente, $T_{ij}$ que se encuentra en conflicto es más joven que *Ts*, entonces el sitio I rechaza $T_s$; si es más viejo la validación para *Ts* en I es diferida hasta que el conflicto pedido por $T_{ij}$ sea resuelto. Esto evita deadlocks por asegurar que transacciones jóvenes siempre esperen por las transacciones viejas. Si son aceptados por la mayoría entonces *Ts* es aceptada y la validación tiene éxito, de lo contrario es rechazada y la validación falla.

3.3.5.1 Efectos del Particionado de la Red

Muchos de los problemas que se pueden tener en las bases de datos replicadas son el particionado de la red y las fallas del sitio o en la comunicación. Los datos en una partición pueden ser actualizados por una transacción, mientras las copias de los datos de otra partición pueden ser sometidas a una actualización diferente de otra transacción. Ambas transacciones se ejecutan de manera totalmente independiente, ya que como las dos particiones de la red no tienen comunicación entre ellas, tampoco se pueden comunicar los dos sitios. Por lo tanto los datos replicados pueden diferir, resultando en un problema de inconsistencia. fall

Los métodos para resolver inconsistencias cuando las particiones se juntan, se dividen en dos grupos: aquellos que adoptan un enfoque **optimista** y aquellos que adoptan un enfoque **pesimista**.

Los **Métodos Optimistas**, como los métodos de control de concurrencia optimistas en general, se basan en la premisa de que los conflictos no ocurren con frecuencia. Se pone énfasis en la disponibilidad de los datos y en el costo de la consistencia. En caso de falla, se estarían permitiendo actualizaciones independientes en varias particiones. Por lo tanto se requiere una estrategia muy compleja para detectar y resolver conflictos cuando las particiones se reintegren.

Los **Métodos Pesimistas** para soportar actualizaciones de datos replicados durante el particionado de la red, adoptan un enfoque conservador. Están basados en la premisa de que la integridad de datos es más importante

que la disponibilidad de la información y en consecuencia se esperan conflictos. Por lo tanto sacrifican un grado de disponibilidad para garantizar la consistencia. Ellos evitan la posibilidad de conflictos cuando las particiones sean reintegradas, confinando las actualizaciones hechas durante la partición a particiones **distinguidas** o de la **mayoría**. Las actualizaciones realizadas en la partición distinguida simplemente son propagadas a otras particiones cuando se reintegre la red.

Alternativas disponibles que limitan la actualización de una sola partición:

1- Cada dato tiene una copia (en un sitio) designada como copia primaria, el resto de las réplicas son copias esclavas. Las actualizaciones son dirigidas solo a la copia primaria y después propagadas a las copias esclavas. Además, todas las lecturas deben adquirir primero un bloqueo de lectura sobre la copia primaria antes de leer la copia esclava. En el caso del particionado de la red, solo están disponibles las copias primarias, asumiendo por supuesto que ellas están accesibles. Si el sitio primario fallara, es posible promover una de las copias esclavas y designarla como copia primaria. Esto generalmente se lleva a cabo usando la estrategia de votos, pero esto requiere que el sistema pueda distinguir entre sitios y fallas de red. No se puede elegir una nueva copia primaria si la red estuviera partida debido a la falla en las comunicaciones, pues el sitio primario original podría todavía ser operacional pero el sistema no tendría ninguna manera de saberlo.

2- En la estrategia de **votación** (también llamado quórum por consenso), se permite que una transacción actualice un dato solo si tiene acceso a él y puede bloquear la mayoría de las copias del dato. La mayoría es conocida como **quórum de escritura**. En el caso de que la transacción obtenga la mayoría, todas las copias son actualizadas a la vez como una unidad y los resultados son propagados a los otros sitios. Un sistema similar, basado en el **quórum de lectura**, opera para las lecturas, para prevenir que las transacciones lean versiones viejas de los datos. Si se requiere una lectura consistente, entonces el quórum de lectura debe representar la mayoría. Por lo tanto es frecuente el caso en que el quórum de

escritura es igual al de lectura. Sin embargo, si las aplicaciones pueden tolerar versiones de datos que son ligeramente anticuados, entonces para tener una disponibilidad de datos más alta, se puede reducir el quórum de lectura.

3- Mientras la estrategia de votación provee una gran disponibilidad de los datos de la copia primaria en el caso de falla, su disponibilidad se alcanza, durante una operación normal, a expensas del chequeo de quórums de lecturas y escrituras para cualquier operación de lectura o escritura. La estrategia de **carencia de escrituras** revierte esta situación implicando mucha menos sobrecarga durante la operación normal a expensas de tener costos más altos cuando las cosas van mal. Con esta estrategia las transacciones operan en uno de estos modos:

   - Modo normal, cuando todas las copias están disponibles, y
   - Modo de falla, cuando uno o más sitios pueden tener fallas.

   Para detectar las fallas se utilizan timeouts, de manera que cada transacción en un modo normal, que emite una escritura a un sitio del cual no puede recibir un acuse de recibo, se cambia al modo de falla. Este cambio puede ser hecho tanto dinámicamente, si es posible, como deshaciendo la operación y reiniciándola en el modo de falla. Durante el modo de falla se usa la estrategia de votos que se mencionó más arriba.

4- Se puede utilizar un **análisis de clases de conflictos** como una estrategia de control de concurrencia y no está restringida a las bases de datos replicadas.

## 3.4 Recuperación de Bases de Datos Distribuidas

La posibilidad de asegurar la consistencia de la base de datos en presencia de fallas imprevisibles tanto en componentes de hardware como de software es una característica esencial de cualquier DBMS. El Manejador de Recuperaciones tiene el rol de recuperar la base de datos y llevarla a un

estado consistente después de la falla que lo haya llevado tanto a la inconsistencia como a la sospecha de la misma.

El Manejador de Recuperaciones tiene que asegurarse que en la recuperación de una falla, o todos los resultados de la transacción involucrada están almacenados permanentemente en la base de datos o ninguno de ellos.

### 3.4.1 Archivo Bitácora

Todas las operaciones realizadas por todas las transacciones en la base de datos, se almacenan en la bitácora. Existe una única bitácora por cada DBMS que es compartida por todas las transacciones que están bajo el control del DBMS. En un ambiente de base de datos distribuida, cada sitio tendrá su propia bitácora. Se escribe una entrada en la bitácora local de cada sitio, cada vez que ocurre uno de los siguientes comandos son pedidos por la transacción (incluyendo transacciones puramente locales, y sub-transacciones de una transacción global):

- *Begin-transaction*
- *Write* (abarca el borrado, actualización e inserción)
- *Commit transaction*
- *Abort transaction*

Cada registro en la bitácora contiene la siguiente información (no se necesita toda la información en todas las operaciones):

1- Identificador de la transacción: es necesario para todas las operaciones.

2- Tipo de registro: cuál de los comandos detallados anteriormente está grabando. Es necesario en todas las operaciones.

3- Identificador del dato afectado por la acción de la base de datos. Necesario para las operaciones de actualización, borrado e inserción.

4- Imagen previa del dato. El valor antes de ser actualizado. Utilizado para las actualizaciones y borrados.

5- Imagen posterior del dato. El valor obtenido luego de haber sido actualizado. Utilizado en las operaciones de actualización e inserción.

6- Información administrativa de la bitácora como por ejemplo, el puntero al registro previo. Utilizado en todas las operaciones.

La bitácora es vital para el proceso de recuperación y por lo tanto normalmente está duplicado (se mantienen dos copias separadas). En el caso de que una copia se pierda o se dañe, se puede utilizar la segunda.

La bitácora es tratada por los sistemas operativos como un archivo más en almacenamiento secundario. En consecuencia, los registros de bitácora deben escribirse en buffers que son periódicamente almacenados en almacenamiento secundario. Las bitácoras pueden escribirse de manera sincrónica o asincrónica. Con la escritura **sincrónica**, cada vez que se quiere escribir un registro en la bitácora, se fuerza a enviarlo a almacenamiento estable. Con la escritura **asincrónica** los buffers se bajan o bien periódicamente (por ejemplo cuando comente una transacción), o bien hasta que se llenen.

La escritura sincrónica impone un retardo en todas las operaciones de las transacciones. Esto puede ser inaceptable. La bitácora es un "cuello de botella" potencial y la velocidad de escribir en ella puede ser un factor crucial de la performance del DBMS. Sin embargo, el retraso ocasionado por el método sincrónico puede compensarse con la obvia ventaja de tener más registros actualizados cuando hay que recuperarse.

Es esencial que los registros de bitácora (o al menos cierta parte de ellos) sean escritos antes que su correspondiente escritura en la base de datos. Si las actualizaciones se hicieran primero en la base de datos y ocurriera una falla antes de que pueda escribir en la bitácora, el Manejador de Recuperaciones no tiene forma de rehacer o deshacer la operación. En cambio, escribiendo antes, el Manejador de Recuperaciones puede asumir tranquilamente que si no hay una entrada de "commit" en la bitácora para una determinada transacción, entonces esta se encontraba activa en el momento de la falla, y por lo tanto debe ser deshecha.

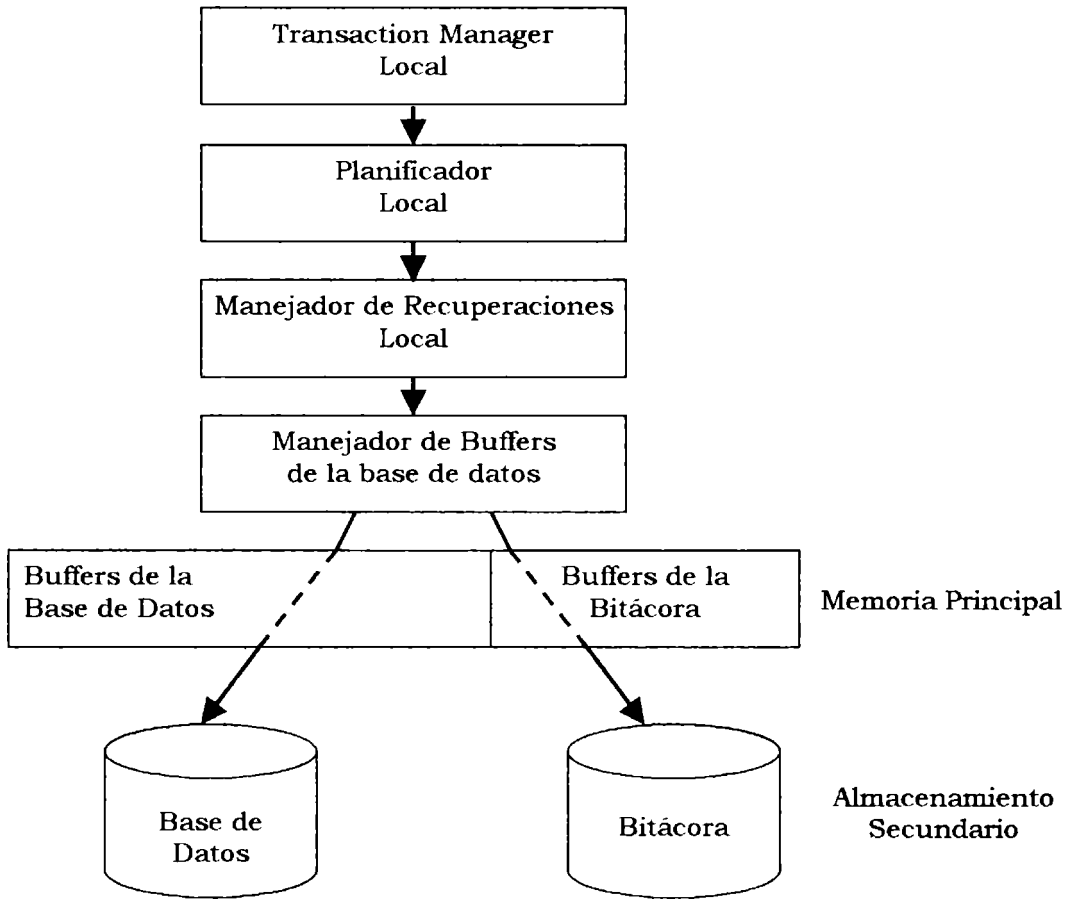



Figura 3.6: Manejador de Recuperaciones Local y sus interfaces

La figura 3.6 muestra la interacción entre el Manejador de Recuperaciones, que supervisa la recuperación local en el sitio; el Manejador de Buffers, que maneja los buffers; el buffer de la bitácora, que realiza la comunicación desde y hacia la bitácora; y los buffers de la base de datos, para la comunicación desde y hacia la base de datos.

### 3.4.2 Puntos de Control (Checkpoints)

Una de las dificultades que debe afrontar el Manejador de Recuperaciones después de una falla importante, es saber cuán lejos ir a buscar en la bitácora para encontrar las transacciones que pudieran tener que

rehacerse (que cometieron antes de la falla) o que deshacerse (aquellas que estaban activas en el momento de la falla). Para limitar esa búsqueda el Manejador de Recuperaciones, toma **checkpoints** periódicos, y ante una recuperación sólo se tiene que buscar el último checkpoint. Existen dos formas de realizarlos: una **sincrónica** y otra **asincrónica**. En el caso asincrónico, se permite que el sistema continúe procesando; en cambio en el sincrónico, no se acepta ninguna nueva transacción hasta que todas hayan terminado para poder marcar el checkpoint.

En el caso de los checkpoints asincrónico, se llevan a cabo las siguientes acciones:

1- Se escribe en la bitácora una lista de todas las transacciones actualmente activas. Esta información está disponible en el Transaction Manager y se necesita para el control de concurrencia.

2- La dirección del checkpoint en la bitácora, es escrito en un archivo especial conocido como **archivo de reinicio**.

3- Se fuerza a escribir todos los buffers tanto en la bitácora como en la base de datos.

Para los checkpoints sincrónicos, la primera acción no es necesaria. Cabe destacar que le simplifica mucho la tarea al Manejador de Recuperaciones ya que no sería necesario rehacer ninguna transacción. Sin embargo tiene la desventaja de la sobrecarga de retardar el comienzo de nuevas transacciones hasta que el checkpoint se complete.

### 3.4.3 Casos de Fallas

Existen diferentes causas de fallas, pero para el propósito de este trabajo de la recuperación de DDBMS, se pueden clasificar de cuatro formas:

1- Falla de una transacción local.

2- Fallas de sitios.

3- Fallas de medios

4- Fallas de la red.

A continuación se describen cuáles son las acciones que debería realizar el Manejador de Recuperaciones después de ocurridas cada una de las fallas; cabe destacar que se asume que el sistema se detiene cuando ocurre una falla o error.

### 3.4.3.1 Fallas de una Transacción Local

La falla de una transacción individual puede ser causado por varias razones:

1- Aborto inducido por una transacción. Es cuando una transacción maneja una excepción que solo la afectará a ella.

2- Fallas imprevistas de una transacción. Surgen de los defectos de los programas de aplicación, como por ejemplo el error aritmético de la división por cero. En este caso el sistema detecta que la transacción ha fallado e informa al Manejador de Recuperaciones para que deshaga la transacción. Aquí tampoco son afectadas otras transacciones.

3- Aborto inducido por el sistema. Este ocurre por ejemplo cuando el Transaction Manager explícitamente aborta una transacción por los conflictos con otra transacción, o por entrar en deadlock. El Manejador de Recuperaciones anuncia explícitamente que deshará la transacción y no se afecta a otras transacciones, excepto que deba realizar el desbloqueo.

En un DBMS centralizado, la falla de transacciones locales es una operación relativamente directa, que involucra deshacer cualquier cambio que la transacción haya hecho en la base de datos en un almacenamiento estable. Esto se realiza restableciendo la imagen previa de los datos actualizados por la transacción desde la bitácora. En los DDBMSs, sin embargo, la falla de un agente local de una transacción global, requiere que el resto de los agentes, de esa misma transacción global, aborten y deshagan lo realizado para garantizar la atomicidad global. Ninguna otra transacción, tanto local como global, será afectada.

3.4.3.2 Fallas del Sitio

Estas fallas pueden ocurrir como resultado de una falla de la CPU local o un incidente en la fuente de alimentación de energía (dando como resultado la caída del sistema). Todas las transacciones de la máquina son afectadas. El contenido de la memoria principal (volátil), incluyendo todos los buffers, se pierden. Sin embargo, se asume que la base de datos tiene los datos almacenados de manera persistente y que la bitácora no fue dañada.

En un ambiente de bases de datos distribuidas, como los sitios operan uno independientemente del otro, es perfectamente posible que algunos sitios permanezcan operables mientras que otros han fallado. Si se cayeran todos los sitios, se le llamaría **falla total**. Si sólo algunos han fallado, se le llama **falla parcial**. La dificultad principal con las fallas parciales es para los sitios que están trabajando tratando de saber cuál es el estado de otro sitios (por ejemplo cuando habían fallado o están operacionales). Como resultado de una falla parcial, también es posible causar un **bloqueo**, y que sea imposible proceder. Esto puede suceder cuando el sitio que ejecuta el agente de una transacción global falla en el medio de la transacción. Otros agentes de la transacción global pueden dudar de si cometer o deshacer. Se debe recordar que una de las ventajas principales de la distribución de los datos es que éstos (o la mayoría de ellos) permanezcan disponibles a los usuarios aún cuando algún sitio se haya caído. Por lo tanto que la caída de un sitio no cause un bloqueo operacional es el objetivo de mayor importancia.

Para la recuperación de la caída de un sitio, el Manejador de Recuperaciones antes que nada debe descubrir el estado del sistema local en el momento de la falla, en especial qué transacciones estaban activas. El objetivo es recuperar la base de datos y llevarla a un estado consistente rehaciendo o deshaciendo las transacciones de acuerdo al estado en que estaban en el momento de la falla. Esto se hace aplicando las imágenes previas o posteriores de la bitácora. El último conocimiento del estado está almacenado en el último checkpoint. Si se usan checkpoint sincrónicos, el último checkpoint marcará un estado consistente. Si en cambio se usa del tipo asincrónico, el último checkpoint marcará un *conocimiento* el estado, no necesariamente consistente.

Cuando después de una caída se recuperan los sitios, se pasa el control al Manejador de Recuperaciones para ejecutar el proceso de recuperación o reinicio. Durante el mismo, el DBMS no acepta ninguna transacción hasta que la base de datos no haya sido reparada. La primer tarea del Manejador de Recuperaciones es ubicar el registro del checkpoint en la bitácora y luego trabaja a través del registro y va armando una lista con las transacciones que tienen que ser deshechas y otra con las que tiene que ser hechas de nuevo.

Al reinicio que sigue luego de una falla del sistema se le llama **reinicio de emergencia**. Existen otros tipos de reinicios dependiendo de cómo fue que el sistema se detuvo.

Con un **arranque en frío**, el sistema se reinicia desde un archivo. Esto sucede cuando los archivos de la bitácora y/o de reinicio se han corrompido después de una falla importante y catastrófica. Cabe recordar que tanto los archivos de bitácora como los de reinicio están duplicados o triplicados, así que las chances de perder todas las copias es realmente muy pequeña. Todas las actualizaciones realizadas desde que fue hecha la copia del archivo, probablemente se perderán. El arranque frío también es necesario cuando se inicia el sistema.

Un **arranque caliente** sigue la bajada normal controlada por el sistema. El manejador del sistema emite un comando "shutdown" después del cual no se comienza ninguna nueva transacción. El sistema espera a que todas las transacciones activas hayan terminado completamente en el momento del shutdown, y luego pasa tanto los buffers como la bitácora a un almacenamiento estable, y luego si se termina a sí mismo. En ese momento no hay más actividad en el sistema, por lo que no hay necesidad de que el Manejador de Recuperaciones haga nada cuando se reinicie.

#### 3.4.3.3 Fallas del Medio

Es una falla que resulta en que varias porciones de la base de datos se corrompen. Una causa habitual de esta falla es la ruptura del cabezal del disco. En estos casos el objetivo del Manejador de Recuperaciones sería recuperar los últimos valores cometidos de todos los datos. Existen dos formas de encararlo: **archivando** o **espejando**.

Es algo muy común que en todas las instalaciones de procesamientos de datos se hagan periódicamente backups del almacenamiento estable. Estos backups forman los **archivos** de la base de datos.

Es deseable que el backup sea realizado cuando el sistema está inactivo, de lo contrario el backup contendrá actualizaciones parciales que harán más complicada la recuperación. Dichos backups se realizan normalmente después de una reorganización de la base de datos o después de haber controlado una parada normal del sistema. Para recuperarse usando archivos, el Manejador de Recuperaciones carga a la base de datos la copia del archivo más reciente y rehace todas las transacciones cometidas desde la bitácora.

En muchos ambientes de bases de datos, realizar el backup de toda la base de datos tardaría mucho tiempo. Algunos sistemas permiten que los backups se hagan de algunas porciones de la base de datos, permitiendo mientras tanto que otras partes permanezcan on-line mientras se hace el proceso de backup. Otros sistemas soportan una **descarga incremental**, donde se almacenan sólo los únicos cambios que han sido realizados desde el último backup. Ambos enfoques son particularmente útiles en el caso de que algunas porciones de la base de datos sean más frecuentemente accedidas que otros, y por lo tanto necesitan que se les haga un backup con más frecuencia.

En los ambientes donde no se puede detener el sistema, la tolerancia a fallas es importantísima. En estos casos se puede utilizar el **espejado**. Esto permite que dos copias completas de la base de datos se mantengan on-line en diferentes dispositivos de almacenamiento estables. Para incrementar la seguridad, los dispositivos deberían conectarse a distintos controladores de disco y ubicarse en distintos lugares físicos (Figura 3.7).

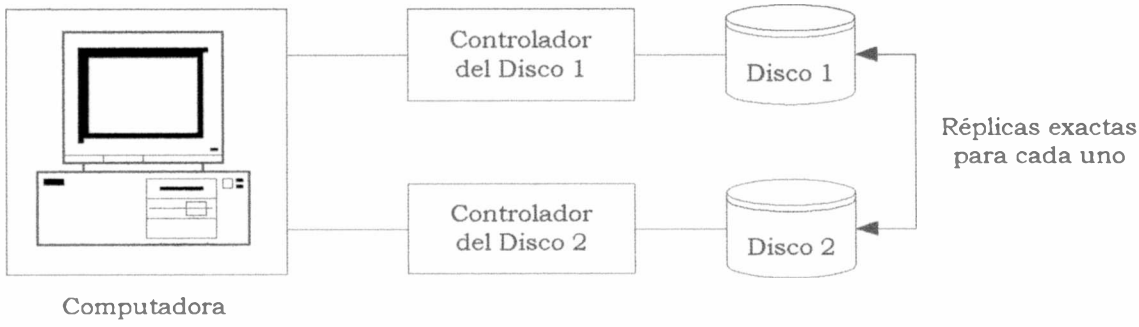



Figura 3.7: Espejado

De esta manera, las lecturas pueden dirigirse a cualquiera de los discos, pero las escrituras deben aplicarse a ambos. Si un disco fallara, todas las lecturas y escrituras se dirigirían a la copia espejo, y el sistema puede seguir operativo sin interrupciones. Una vez que la falla ha sido reparada, debe copiarse las base de datos actualizada en el disco reparado.

Una alternativa que provee el mismo grado de disponibilidad confiabilidad, es dividir cada disco en dos particiones: el **área primaria** y el **área fallback** (de retraso) (Figura 3.8). La base de datos en sí misma se distribuye en todas las particiones primarias y las copias espejo en la particiones fallback, de manera que el mismo segmento de la base de datos sea almacenado en distintos discos. De esta manera, si A y B están almacenados en el área primaria del disco 1, entonces las copias espejo pueden ser almacenadas en el área fallback del disco 2 y viceversa. El controlador del disco tiene múltiples conexiones a los discos. En una operación normal, el DBMS utiliza ambos controladores de disco, para realizar búsquedas en los discos en forma paralela; este paralelismo no es posible en el espejado standard.

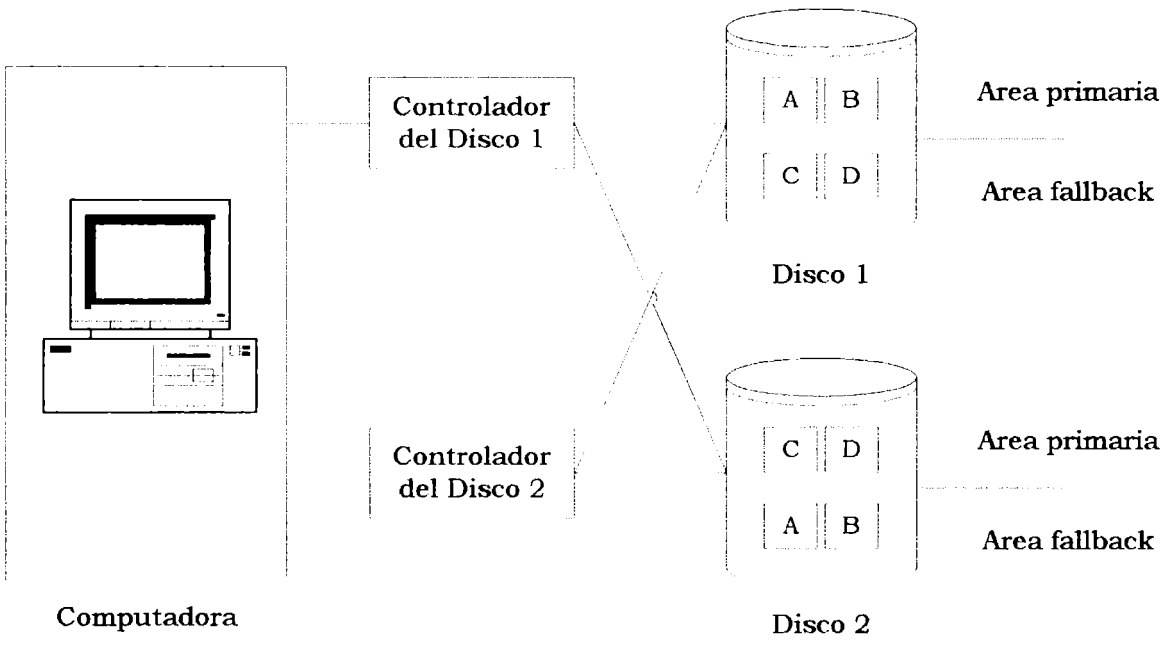



Figura 3.8: Espejado usando particiones primarias y fallback

3.4.3.4 Fallas de Red

El éxito de las operaciones del DBMS depende de que todos los sitios tengan una comunicación confiable con otros sitios. La mayoría de las redes son muy confiables. Los protocolos garantizan una correcta transmisión de mensajes y en un orden también correcto. En el caso de que hayan fallas en al línea, muchas redes soportan un nuevo ruteo de los mensajes. Sin embargo las fallas en las comunicaciones pueden ocurrir. Estas fallas pueden ocurrir cuando la red fue particionada en una o más sub-redes. Los sitios que están en la misma partición se pueden comunicar con cualquiera de ellos, pero no con sitios de otras particiones.

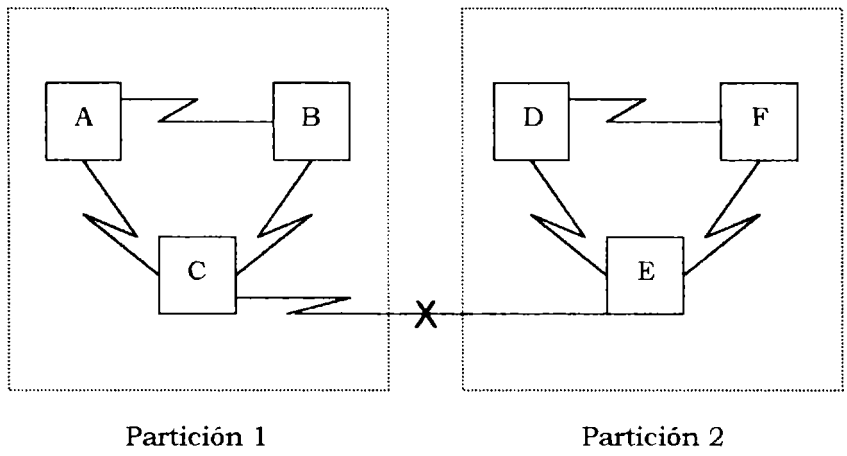



Figura 3.9: Particionado de Red.

En la figura 3.9, luego de una falla en la línea que comunica C con E, los sitios de la partición {A, B, C} quedan aislados de los sitios {D, E, F} de la otra partición. Una de las dificultades de operar en un ambiente distribuido es conocer cuándo y dónde ocurrió una falla en la comunicación o en un sitio. Por ejemplo, suponiendo que es sitio C de la figura 3.9 envía al sitio E un mensaje, y no recibe el acuse de recibo de E durante cierto período de tiempo, llamado timeout. El tema es cómo puede decidir C si ha fallado el sitio E o a si se ha partido la red debido a un incidente de comunicación de una manera que C y E estén en particiones separadas y por lo tanto no pueda comunicarse una con la otra. De hecho, a la única conclusión a la que puede llegar C, es que no puede enviar ni recibir mensajes de E. Es muy complicado elegir un

buen valor para el timeout que determinará si ocurrió una falla. Debería ser menor o igual al tiempo máximo posible para la ida-vuelta del mensaje, mas el reconocimiento, más el tiempo de procesamiento en E.

Suponiendo que se está ejecutando una transacción global, que hay agentes en C y en E, y que ocurre una falla de red haciendo que tanto C como E queden en distintas particiones, podría suceder que C decidiera, junto con otros sitios de su partición, cometer la transacción global; mientras que E y otros sitios de su partición podrían decidir abortarla. Si todo esto ocurriera, se estaría violando la atomicidad de la transacción global.

Un protocolo no-bloqueante es uno que no bloquea sitios operacionales en caso de fallas. Los sitios que pueden seguir procesando, siguen haciéndolo sin tener que esperar a que otros sitios fallados se recuperen. Las técnicas de recuperación que serán utilizadas luego de que haya sido partida la red, dependen de la estrategia determinada para el control de concurrencia. Los métodos se clasifican en optimistas y pesimistas.

Los **protocolos de commit optimistas** anteponen la disponibilidad al estado de consistencia, y van con el enfoque optimista del control de concurrencia para particiones de red mencionado anteriormente en 3.3.4, en donde se permite que las actualizaciones procedan independientemente en varias particiones. De esta manera, cuando los sitios son reconectados, es muy probable encontrarse con inconsistencias. La resolución de estas inconsistencias depende en gran medida de la semántica de las transacciones. Generalmente el Manejador de Recuperaciones no puede restablecer el estado de consistencia por sí solo, sino que necesita ser asistido por el usuario.

Una vez que se logra establecer que hay un problema de consistencia, el sistema puede hacerle frente mediante tres posibilidades:

1- Deshaciendo una (o más) de las transacciones ofensivas. Esto podría tener un efecto en cascada sobre otras transacciones de la misma partición.

2- Aplicar una **transacción de compensación**, que implique deshacer una de las transacciones y notificar a cualquier agente externo afectado, que se ha realizado la corrección.

3- Aplicar una **transacción de corrección**, que implique corregir la base de datos para que refleje todas las actualizaciones.

Los **protocolos pesimistas de fusión** eligen la consistencia por sobre la disponibilidad y siguen los enfoques pesimistas del control de concurrencia sobre redes particionadas. Utilizando este enfoque, la recuperación es más directa, ya que las actualizaciones estarían confinadas a una sola partición. La recuperación o reconexión de la red simplemente involucra la propagación de todas las actualizaciones a todos los sitios.

### 3.4.4 Protocolos de Recuperación Local

Se ha planteado en varias oportunidades la necesidad de recuperación. Existen distintos algoritmos que se adecuan a diferentes casos. Los algoritmos principales son cuatro:

1- Undo/redo (deshacer/rehacer)

2- Undo/no-redo (deshacer/no rehacer)

3- No undo/redo (no deshacer/ rehacer)

4- No-undo/no-redo (no deshacer/ no rehacer)

Estos cuatro algoritmos especifican cómo el Manejador de Recuperaciones maneja las diferentes operaciones: *begin-transaction, read, write, commit, abort* y *restart.*

De acuerdo con el título de los algoritmos, éstos difieren de acuerdo a si tienen que rehacer y/o deshacer las actualizaciones hechas en la base de datos luego de la falla. Esta decisión depende de cuándo el buffer de la base de datos fue guardada en un almacenamiento permanente. Si fueron almacenados sincrónicamente durante la ejecución de la transacción, los cambios pueden tener que ser deshechos. Si en cambio, los buffers se almacenan sólo después de una escritura forzada por el Manejador de Recuperaciones, entonces puede no ser necesario deshacer. El rehacer una transacción depende de si los buffers fueron pasados sincrónicamente al commit, en cuyo caso no se necesita el *redo*, o asincrónicamente de acuerdo a las necesidades del manejador del buffer, en cuyo caso se puede requerir un *redo.*

3.4.4.1 Undo/Redo

Son los algoritmos más complejos. Tienen la ventaja de que el manejador del buffer puede decidir cuándo liberarlo, haciendo que se reduzca la sobrecarga de entrada/salida. Son de una eficiencia máxima durante la operación normal a expensas de una gran sobrecarga si tiene que recuperarse. El Manejador de Recuperaciones debe realizar distintas acciones dependiendo de las distintas operaciones:

*Begin-transaction*: Acciona algunas funciones del administrador del DBMS tales como agregar la nueva transacción a la lista de las transacciones actualmente activas. Conceptualmente también se debería guardar en la bitácora, aunque por razones de eficiencia, normalmente se pospone hasta la primer escritura de la transacción.

*Read*: El objeto de datos requerido para leer se busca en los buffers que pertenecen a la transacción, si no está, se continúa la búsqueda en los otros buffers de la base de datos. Normalmente no se necesita que se registre una lectura en la bitácora, pero a veces si.

*Write*: Los objetos de datos serían actualizados en los buffers de la transacción, si los datos están allí; y si no, se realiza una búsqueda en la base de datos y se los actualiza. En la bitácora se guardan la imagen previa y posterior a la escritura.

*Commit*: Se escribe en la bitácora un registro de "commit".

*Abort*: El Manejador de Recuperaciones tiene que deshacer la transacción. Si se utiliza una actualización inmediata, se tienen que actualizar todos los buffers de la base de datos y recuperar de la bitácora los valores con la imagen previa de los datos.

*Restart*: El Manejador de Recuperaciones trabaja sobre la bitácora para ir rehaciendo todas las transacciones que cometieron y deshaciendo aquellas que no tienen su respectivo "commit". Es necesario realizar el redo porque los cambios de las transacciones que cometieron después del último checkpoint, se escribieron en los buffers y no en la base de datos.

Una variación del Undo/Redo es que el Manejador de Recuperaciones mantenga listas de transacciones activas, abortadas y cometidas. Con el *begin-transaction* una transacción es agregada a la lista de activas. La lista de abortos contiene una lista de todas las transacciones abortadas, mientras que la lista de cometidas contiene todas las transacciones que fueron cometidas. Cuando se hace un *restart*, el Manejador de Recuperaciones simplemente utiliza esas listas para decidir qué transacciones deben ser rehechas (lista de cometidas) y cuáles deshechas (lista de abortadas). Obviamente estas listas tienen que estar en un almacenamiento estable y generalmente forman parte de la bitácora.

#### 3.4.4.2 Undo/No-Redo

Usando este algoritmo, los buffers de la base de datos se pasan a un almacenamiento estable cuando comete la transacción. De esta manera nunca hay necesidad de rehacer la transacción cuando se reinicia y por lo tanto tampoco hay que guardar la imagen previa en la bitácora. El detalle de las acciones es el que sigue:

*Begin-transaction*, *Read*, *Write* y *Abort*: Sirven para el Undo/Redo.

*Commit*: Todos los buffers de la base de datos son guardados en un almacenamiento estable y se escribe un registro de "commit" en la bitácora. Si se usara una lista de cometidas, simplemente se agrega el identificador de la transacción a dicha lista.

*Restart*: El Manejador de Recuperaciones debe realizar un undo global.

#### 3.4.4.3 No-Undo/Redo

Utilizando este algoritmo, el Manejador de Recuperaciones no escribe transacciones no cometidas en la base de datos estable. El manejador de buffers está forzado a retener los registros en los buffers hasta que la transacción cometa. A esto se lo conoce como **fijación de los buffers**. Alternativamente, las actualizaciones pueden ser escritas en la bitácora en vez de escribirse en los buffers.

El detalle de las acciones requeridas son las que siguen:

*Begin-transction* y *Read*: Sirven para el Undo/Redo.

*Write:* Si se está utilizando la bitácora para guardar las actualizaciones, cuando llega un pedido de escritura de una transacción, el Manejador de Recuperaciones agrega la imagen posterior de los datos en la bitácora.

*Commit:* O bien el manejador de buffers anuncia que van a vaciar los buffers y ponerlos en un almacenamiento estable; o bien si las actualizaciones fueron escritas en la bitácora, las imágenes diferidas de los registros actualizados se escriben en la base de datos a través de los buffers.

*Abort:* Si las actualizaciones fueron escritas en al bitácora, el Manejador de Recuperaciones simplemente escribe un registro de "abort" en la bitácora o agrega el identificador de la transacción a la lista de abortos. Además esto facilita la tarea del recolector de basura de la bitácora.

*Restart:* El Manejador de Recuperaciones debe realizar un undo global.

3.4.4.4 No-Undo/No-Redo

Para no tener que deshacer transacciones, el Manejador de Recuperaciones debe asegurarse de que no se escriba ninguna actualización en la base de datos estable antes de cometer. Y para evitar tener que rehacer las transacciones, el Manejador requiere que todas las actualizaciones se hayan escrito a la base de datos estable antes de que cometan. Esta aparente paradoja puede resolverse escribiendo en la base de datos estable una acción en el commit. Para hacer esto, el sistema utiliza la escritura **shadowing**. Mediante este mecanismo las actualizaciones se escribirían en un lugar separado de la base de datos estable, en un almacenamiento secundario, y los índices de la base de datos no apuntan a las páginas actualizadas hasta que la transacción haya cometido.

El detalle de las acciones son las siguientes:

*Begin-transction* y *Read*: Sirven para el Undo/Redo.

*Write*: Las actualizaciones, a través de los buffers de la base de datos, son escritos en una ubicación no utilizada del almacenamiento estable y sus direcciones son almacenadas en la *lista de direcciones shadow.*

*Commit*: Los buffers de la base de datos son almacenados en la sección *shadow* de la base de datos estable. Los índices de la base de datos son actualizados de manera que apunten al área shadow y se agrega un registro de "commit" en la bitácora (o se agrega el identificador de la transacción a la lista de cometidas).

*Abort*: Se eliminan las transacciones de la lista de direcciones *shadow*, de esta manera, las actualizaciones de la transacción efectivamente quedan inalcanzables. Como en el caso de No-Undo/Redo, se escribe en la bitácora un registro de aborto (o se agrega a la lista de abortos).

*Restart*: Se pasa el Recolector de Basura a la lista de direcciones shadow, que contiene todas las transacciones activas en el momento de la falla, dejando los índices de la base de datos como estaban.

### 3.4.5 Protocolos de Recuperación de Base de Datos Distribuidas

La recuperación en los sistemas de bases de datos distribuidas es complicada por el hecho de que se necesita atomicidad tanto para las sub-transacciones locales como para las transacciones globales.

Se necesita modificar el procesamiento de commit y abort para que una transacción global no cometa o aborte hasta que todas las sub-transacciones hayan cometido satisfactoriamente o abortado. La posibilidad de fallas en las comunicaciones y en los sitios complica aún más la situación. Un objetivo importante de las técnicas de recuperación para DBMS Distribuidos (DDBMS) es que la falla de un sitio no afecte el procesamiento en otro sitio. En otras palabras, los sitios no deben quedar bloqueados.

Existen dos protocolos de commit para bases de datos distribuidas muy comunes. Estos son el Protocolo de Commit de Dos Fases (2PC) y el Protocolo de Commit de Tres Fases (3PC). Se verá en detalle el 2PC ya que es el usado en este trabajo.

Se asume que cada transacción global tiene un sitio que actúa como coordinador de esa transacción (normalmente es el sitio al que está sometida la transacción). A los sitios en los cuales la transacción global tiene agentes se los llama participantes. Se asume que el coordinador conoce la identidad de todos los participantes, y cada participante conoce la identidad del coordinador.

### 3.4.5.1 Protocolo de Commit Dos Fases (2PC)

Como su nombre lo indica, 2PC opera en dos fases: una **fase de votación** y otra de decisión. La idea básica es que el coordinador pregunta a todos los participantes si están o no preparados para cometer la transacción. Si un participante vota por un aborto, o falla al responder por un timeout, el coordinador le indica a los participantes que aborten la transacción. Si todos votan para cometer, entonces se les indica a todos los participantes que cometan dicha transacción. Este protocolo asume que cada sitio tiene su propia bitácora local y que por lo tanto puede cometer o deshacer confiadamente la transacción.

Las reglas de votación son las siguientes:

1- Cada participante tiene un voto que puede ser "cometer" o "abortar".

2- Una vez que votaron, los participantes no pueden cambiar su voto.

3- Si un participante vota "abortar", está libre de abortar la sub-transacción inmediatamente. Este tipo de aborto es llamado **aborto unilateral**.

4- Si un participante vota "cometer", entonces debe esperar a que el coordinador haga un broadcast tanto del "cometer global" como del "abortar global".

5- Si todos los participantes votaron "cometer", entonces la decisión global del coordinador debe ser de "cometer".

6- La decisión global debe ser adoptada por todos los participantes.

El Protocolo de Commit de Dos Fases involucra varias esperas de mensajes de otros sitios. Para que el proceso no se bloquee innecesariamente, se utilizan timeouts.

Al principio, los participantes esperan por una instrucción de "preparar" por parte del coordinador. Como se permiten abortos unilaterales, el participante está libre de abortar en cualquier momento hasta que decida su voto. Si fallara al recibir la instrucción de "preparar", se pasaría el timeout y estaría abortando antes de votar.

El Coordinador debe esperar a recibir los votos de todos los participantes. Si un sitio fallara al votar, el coordinador asume un voto de "abortar" y se le envía el mensaje "abortar" a todos los participantes. Cuando se reinicia la localidad que falló, ésta invoca un **protocolo de reinicio** (que se explicará más adelante).

Los participantes esperan por una instrucción de "aborto global" o "commit global" del coordinador. Si el participante falla al recibir la instrucción del coordinador, entonces asume que el coordinador ha fallado y se invoca a un **protocolo de terminación**. Este protocolo es llevado a cabo sólo por los sitios operacionales, aquellos que han fallado siguen el **protocolo de reinicio** en la recuperación.

El **protocolo de terminación** es para dejar bloqueado el proceso del participante hasta que la comunicación con el coordinador se reestablezca para luego poder ser informado de la decisión de "commit global" o de "aborto global" y continuar el proceso.

Hay casos en los que no es necesario bloquear. Supóngase que C es el proceso Coordinador de una transacción con los participantes $P_i$ y $P_j$; y que el Coordinador falla después de haber tomado una decisión y de haberle notificado a $P_i$, pero antes de comunicarle la decisión a $P_j$. Si $P_j$ conoce la identidad de $P_i$, entonces puede preguntarle cuál fue la decisión global y actuar en función de ésta. Una forma sencilla de decirle a los participantes quiénes son el resto de los que participan de la misma transacción, es que el Coordinador les envíe la lista de participantes al mandar el "preparar". A esto se lo conoce como **protocolo cooperativo de terminación**.

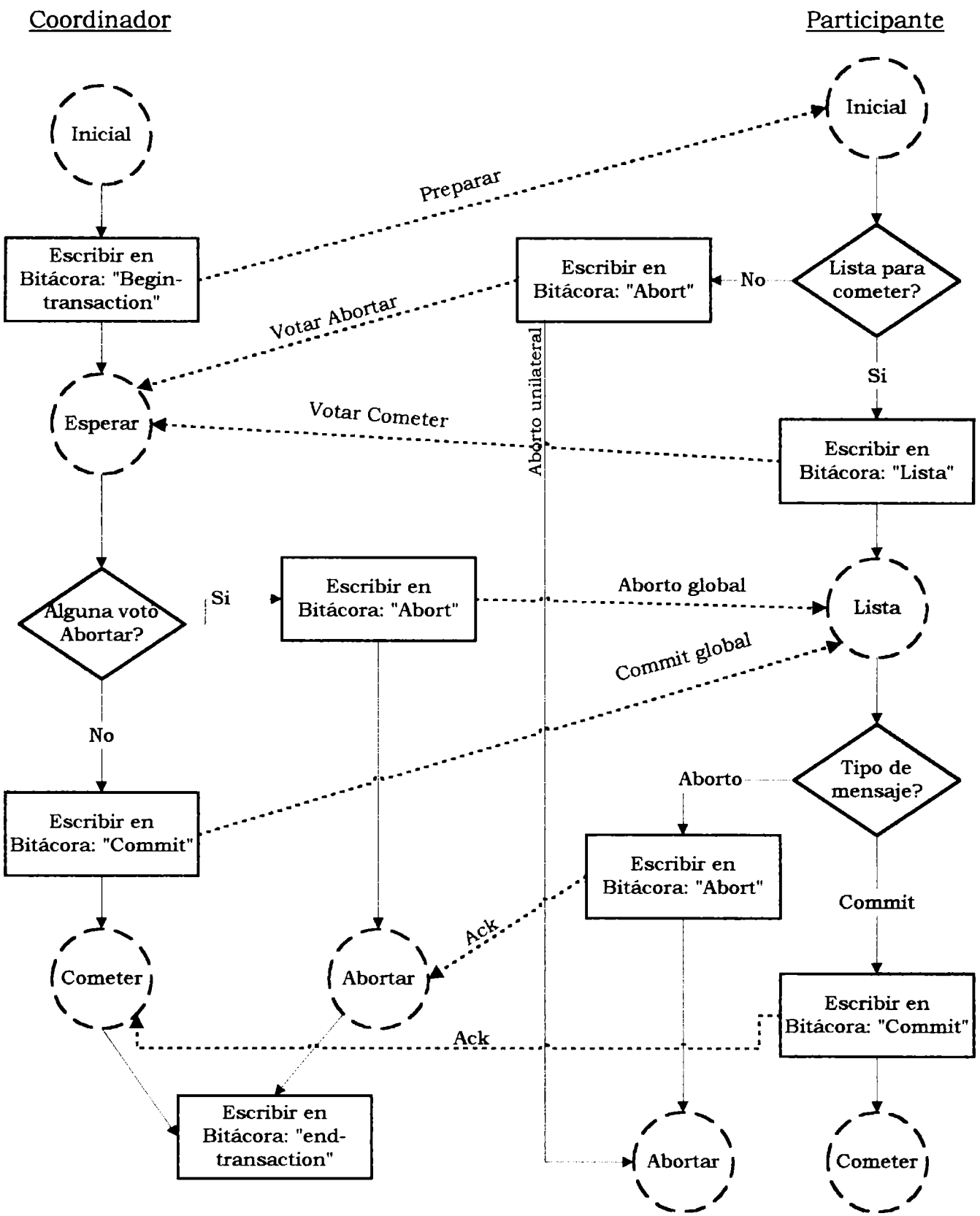



Figura 3.10: Acciones del Protocolo de Commit de Dos Fases

Otra forma es elegir un nuevo coordinador y así desbloquearse. El **protocolo de elección** es sencillo. Todos los sitios involucrados en la elección (todos los que están operacionales) acuerdan ordenarse linealmente. El primer sitio de ese ordenamiento es el elegido como nuevo Coordinador. Este orden se facilita utilizando la identificación de los sitios.

El **protocolo de reinicio** requerido por un participante luego de una falla, depende del estado en el que se encontraba en el momento de la falla. El estado de procesamiento de los participantes puede ser determinado examinando la bitácora local. El objetivo es asegurar que un proceso participante realice las mismas acciones que el resto de los participantes y que lo pueda hacer de manera independiente (sin necesidad de consultarle al Coordinador o a otros participantes).

Sea $P_r$ un proceso participante que intenta reiniciarse después de una falla. Si $P_r$ no había votado a causa de la falla, puede abortar unilateralmente y recuperarse independientemente. También puede recuperarse si había recibido una decisión global ("commit global" o "aborto global") antes de la falla. Sin embargo si $P_r$ había votado "cometer" y no se le informó de la decisión global, no se puede recuperar de manera independiente; tendrá que preguntarle al Coordinador (o a otros participantes) cuál fue la decisión global.

## 3.5 Replicación de Datos

Para superar muchos de los problemas inherentes a la comunicación, los datos se copian y almacenan en múltiples base de datos, incluso muchas veces en diferentes plataformas de hardware. Las razones principales para la réplica de información, es mejorar la confiabilidad y maximizar la velocidad de acceso. En un ambiente cliente/servidor muchas veces es difícil obtener todos los datos requeridos por todos los usuarios. También es complicado balancear el procesamiento de requerimientos entre usuarios "livianos" (por ejemplo sistemas on-line de procesamiento de transacciones) y usuarios "pesados" (por ejemplo sistemas de soporte de decisiones para marketing).

También es deseable tener replicación de datos, cuando se necesita acceder rápidamente a determinada información que justo en ese momento

está siendo sometida a un procesamiento de E/S intensivo. Si hubiera una falla en la base de datos, la información continuaría disponible a través de la base de datos replicada. Este tipo de replicación de datos normalmente se lo llama **distribución de datos**. No hay que confundirla con las bases de datos distribuidas. Con la distribución de datos, la información es copiada redundantemente en otras bases de datos, mientras que en las bases de datos distribuidas la información no está replicada, los datos residen en distintas bases de datos .

### 3.5.1 Replicación Master-Slave (Maestro-Esclavo)

Con este tipo de replicación se utiliza una base de datos Maestra para realizar las actualizaciones, y luego de un tiempo *x* se refrescan múltiples bases de datos de consultas desde la Maestra.

Para crear una configuración Maestro-Esclavo, se deben realizar los siguientes pasos:

1- Definir y poblar la base de datos esclava. Usando copias de las descripciones la tabla del catálogo de tablas del Maestro.

2- Crear una rutina de propagación en el host de la base de datos y establecer puertas de acceso (host gateways) en las bases de datos esclavas.

Para ello, hay varios problemas que deben ser solucionados. Uno de ellos es el tiempo. Las bases de datos replicadas se utilizan porque una compañía no puede permitirse la sobrecarga de la actualización inmediata de la base de datos.

Algunos permiten que las actualizaciones ocurran en un intervalo de tiempo predefinido (por ejemplo, cada hora): las actualizaciones ocurrirán cuando se alcanza dicho intervalo del tiempo.

Otros basan las actualizaciones a los esclavos en el nivel de actividad de los mismos. Se podría definir un umbral de actividad, de manera que las bases de datos esclavas sean actualizadas cuando su actividad está por debajo de ese umbral. Este método permite que los esclavos sean actualizados

cuando no están ocupados, pero los usuarios de las bases de datos esclavas nunca estarían seguros de si sus datos son los actuales.

Un tercer acercamiento hace que cada base de datos esclava haga consultas a la base de datos Maestra automáticamente cuando no está ocupado, para saber si hay algunas actualizaciones que aguardan la propagación.

Cuando hay que mantener la Integridad Referencial, los retrasos de propagación a las bases de datos esclavas pueden crear condiciones en que se violen las reglas. Una forma de solucionarlo, es que refresque periódicamente en forma sincrónica la base de datos esclava desde la base de datos Maestra; de esta manera se tienen todas las bases de datos actualizadas. Hay que tener en cuenta la sobrecarga generada.

### 3.5.2 Actualizaciones: Consistencia Fuerte vs. Consistencia Débil

Con la consistencia fuerte, se utiliza la arquitectura del protocolo de Commit Dos Fases para asegurar que todas las copias están en sincronización todo el tiempo. Bajo el enfoque débil, se utiliza una base de datos "primaria" como Maestra, y las actualizaciones son propagadas a las bases de datos esclavas. Dicha propagación puede suceder de varias formas, como por ejemplo, cada un intervalo de tiempo dado.

**Consistencia fuerte**

Ventaja: Todas las bases de datos están en sincronismo con el tiempo real.

Desventaja: se pueden perder accesos por fallas de comunicación.

**Consistencia débil**

Ventaja: Los datos pueden ser cargados desde snapshots. Se permite la replicación asincrónica.

Desventaja: Los esclavos no están en exacto sincronismo con el Maestro.

# 4. Java

## 4.1 Características Generales

Java es un lenguaje simple, orientado a objetos, distribuido, interpretado, robusto, seguro, de arquitectura neutral, portable, multithreading, de alta performance y dinámico.

- **Simple**: El objetivo de los diseñadores de Java fue crear un lenguaje de programación que fuera fácil de aprender. Para esto Java adoptó una sintaxis similar a la de C/C++ por su gran popularidad, y quitó aquellas características que son fuente de confusión (por ejemplo los punteros).
- **Orientado a Objetos**: Posee todas las características necesarias (polimorfismo, herencia, encapsulamiento, binding dinámico).
- **Distribuido**: Soporta aplicaciones en la red. Es posible utilizar protocolos como HTTP y FTP para acceder a archivos remotos de manera simple.
- **Interpretado**: El compilador de Java genera "byte-codes" para la JVM (Java Virtual Machine). Estos "byte-codes" hacen que Java pueda ejecutarse en cualquier plataforma que tenga la JVM implementada. La naturaleza interpretada de Java acelera el ciclo de desarrollo de software.
- **Robusto**: Fue diseñado para la creación de software altamente confiable. Provee un chequeo en tiempo de compilación, seguido por un segundo nivel de chequeo en tiempo de ejecución. Además provee un modelo de manejo de memoria extremadamente simple: no hay punteros fingidos explícitamente y provee una recolección de basura automática (Garbage Collection), de esta manera se eliminan fuentes de errores, roturas de sistemas y baja performance.
- **Seguro**: Tiene que ver con varios aspectos. Entre ellos, el intérprete de Java realiza un chequeo de los archivos .class que vienen en la red. Con respecto al mapa de memoria y la alocación de memoria, en Java la reserva de memoria se realiza en ejecución y depende de las

características del software y hardware de la estación de trabajo donde se ejecuta el programa, no existe el concepto de puntero, tiene un modelo de alocación de memoria transparente al programador (controlado íntegramente por la JVM).

- **Arquitectura neutral**: Java fue diseñado para soportar aplicaciones que se ejecutan en ambientes de redes heterogéneos, independientemente de la plataforma de hardware y de software. Por este motivo, las aplicaciones Java se ejecutan en una amplia variedad de arquitecturas de hardware y sobre múltiples sistemas operativos. Todo éstos es gracias al formato de "byte-codes", ya que es un formato intermedio de arquitectura neutral que permite transportar eficientemente código entre múltiples plataformas de hardware y software.
- **Portable**: La arquitectura neutral dada por los "byte-codes" es el paso más importante hacia la portabilidad de los programas.
- **Multithreading**: Permite mejorar la interactividad y la performance del sistema. Será explicado con mayor detalle más adelante.
- **Alta performance**: Como el código Java es chequeado en compilación y en ejecución, el intérprete puede correr a alta velocidad sin necesidad de hacer chequeos. Además el Garbage Collection corre como un thread de baja prioridad (aprovechando los tiempos muertos del usuario), mejorando la disponibilidad de memoria.
- **Dinámico**: Java es dinámicamente extensible ya que las clases se linkean a medida que se necesitan y pueden ser cargadas dinámicamente a través de la red.

## 4.2 La Plataforma Java

Una plataforma es un ambiente de software o hardware sobre el que se ejecuta un programa. La plataforma Java es sólo una Plataforma de Software, que se ejecuta por encima de otras plataformas de software (Sistemas

Operativos). La plataforma Java es la Máquina Virtual de Java (JVM) y una porción de la API (*Application Programming Interface*) Java que se denomina *API Core*. La API Core es el subconjunto mínimo de la API Java que debe ser admitido por la plataforma Java. Todas las demás clases e interfaces de la API Java que no están en la API Core están en la *API Standard Extension* [JAVA00].

Un programa escrito en Java, se compila a byte-codes y éste puede correr en cualquier lugar donde esté presente la plataforma Java, sin importar el Sistema Operativo subyacente. La Java Virtual Machine asegura esta portabilidad.

El entorno de tiempo de ejecución de Java o JRE es la implementación de la plataforma Java y consta de la JVM, la API Core y archivos de apoyo. Aunque el JRE ha sido llevado a muchas plataformas de sistemas operativos, Sun sólo distrubuye el JRE para Win32 (Windows NT, Windows 95, Windows 98) y Solaris (en sus versiones Sparc e Intel) y Linux.

Existen una serie de temas relevantes que se deben mencionar cuando se habla de Java y que son de mucha importancia en la implementación del simulador. Estos son el uso del AWT, del JDBC, el paquete java.net y todo lo relacionado con la comunicación cliente/servidor.

## 4.3 AWT

Ofrece la posibilidad de crear programas GUI independientes de la plataforma que se utilice, y contribuye en gran medida a la popularidad de Java. El AWT no sólo es una API mejor para el desarrollo de las aplicaciones de Windows, sino que también lo es para programar aplicaciones basadas en ventanas en plataformas que van desde Motif hasta OS/2.

## 4.4 JDBC - Acceso a las Bases de Datos con Java

Permite acceder a bases de datos desde Java. Para acceder a las bases de datos remotamente, los usuarios necesitan un *cliente de base de datos*.

Ofrece al usuario la posibilidad de acceder a la base de datos. Los clientes de base de datos utilizan controladores de base de datos para enviar instrucciones SQL a servidores de base de datos y recibir conjuntos de resultados y otras respuestas de los servidores. Los controladores JDBC son usados por los applets y aplicaciones de Java para comunicarse con servidores de base de datos. Oficialmente Sun dice que JDBC es un acrónimo que no significa nada. No obstante, se le asocia con la "conectividad de base de datos de Java".

Muchos servidores de base de datos utilizan protocolos específicos de las marcas. Esto implica que un cliente de base de datos tiene que aprender un lenguaje nuevo para hablar con un servidor de base de datos diferente. No obstante Microsoft ha establecido una norma común para comunicarse con las bases de datos, llamada *Conectividad Abierta de Base de Datos (ODBC)*. Hasta ODBC, la mayoría de clientes de bases de datos eran específicos del servidor. Los controladores ODBC se separan de los protocolos específicos de la marca, proporcionando una interfaz de programación para los clientes de bases de datos. Escribiendo sus clientes de bases de datos en la API ODBC, lo que se está haciendo es permitir que sus programas puedan acceder a más servidores de bases de datos.

JDBC ofrece una API común de programación de bases de datos para programas de Java. Sin embargo los controladores JDBC todavía no se comunican directamente con tantos productos de bases de datos como los controladores ODBC. En lugar de ello, muchos controladores JDBC se comunican con las bases de datos a través de la ODBC. De hecho, uno de los primeros controladores JDBC fue el controlador puente JDBC-ODBC desarrollado por JavaSoft e Intersolv.

¿Por qué JavaSoft creó JDBC? ¿Qué había de malo en ODBC? Existen una serie de razones que explican porqué JDBC era necesario, lo cual demuestra que JDBC es una solución mejor para los applets y aplicaciones de Java:

- ODBC es una API con lenguaje C, y no una API Java. Java está orientado a objetos y C no. C emplea punteros y otros constructores de programación "peligrosos" que Java no admite. Una versión Java de ODBC requería volver a escribir en gran medida la API ODBC.

- Los controladores ODBC se deben instalar en máquinas cliente. Esto implica que el acceso de applets a bases de datos estaría limitado por la exigencia de tener que descargar e instalar un controlador JDBC. Una solución que sólo pasa por Java permite descargar e instalar automáticamente los controladores JDBC junto al applet. Esto significa en gran medida el acceso a bases de datos por parte de los usuarios de applets.

JavaSoft creó el controlador puente Java-ODBC que traduce la API JDBC a la API ODBC y se utiliza con un controlador ODBC. Dicho puente no constituye una solución elegante, pero permite a los programadores de Java utilizar los controladores ODBC existentes.

## 4.5 Paquete java.sql

El propósito de este paquete es el de permitir ejecutar instrucciones SQL desde Java. Para ejecutarlas se utilizan interfaces del API JDBC. La interfaz *Statement* define los métodos utilizados para interactuar con las bases de datos a través de la ejecución de instrucciones SQL. Estos métodos también admiten el procesamiento de los resultados de las consultas que se devuelven a través de objetos *ResulSet* y controlan la mecánica del prosesamiento de consultas. La interfaz *PreparedStatement* amplía la interfaz *Statement* para definir los métodos que se usan para trabajar con instrucciones SQL precompiladas. El uso de este tipo de instrucciones proporciona una forma más eficiente de ejecutar las instrucciones SQL que más frecuentemente se utilicen. Los objetos *PreparedStatement* se pueden utilizar con instrucciones SQL parametrizadas. [JAVAW1]

## 4.6 Sockets y Comunicaciones Cliente/Servidor

Los clientes y servidores establecen *conexiones* y se comunican a través de *sockets*. Las conexiones son vínculos de comunicación que se crean en

Internet por medio del TCP. Ciertas aplicaciones cliente/servidor se construyen también entorno al UDP (sin conexión). Estas aplicaciones también utilizan sockets para comunicarse.

Los sockets son los puntos finales de la comunicación en Internet. Los clientes crean sockets y los conectan a sockets de servidor. Los sockets están asociados a una dirección de host y a una dirección de puerto. La dirección del host es la dirección IP del host en la que está ubicada el programa cliente o servidor. La dirección de puerto es el puerto de comunicación que utiliza el programa cliente o servidor [JAVAW2].

Un cliente se comunica con un servidor estableciéndose una conexión la socket del servidor. Cliente y servidor intercambian entonces los datos por la conexión. La comunicación orientada a la conexión es más fiable que la comunicación sin conexión, ya que el TCP subyacente proporciona reconocimiento de mensajes, detección de errores y servicios de recuperación de errores.

Cuando se utiliza un protocolo sin conexión, el cliente y el servidor se comunican enviando datagramas a los sockets de cada uno. El UDP se utiliza en protocolos sin conexión. No admite la comunicación fiable, como en el caso de TCP.

El paquete java.net ofrece varias clases que admiten la comunicación cliente/servidor basada en sockets. Son de interés particular las clases *Socket* (implementa los sockets de cliente basados en conexión) y *SeverSocket* (implementa un socket del servidor TCP).

## 4.7 Multithreading

Un Thread es un flujo de control secuencial dentro de un programa. Java provee múltiples Threads en un programa, ejecutándose concurrentemente y llevando a cabo tareas distintas. El soporte multithread de Java lo proporciona la clase Thread del paquete java.lang. incluye primitivas de sincronización para cuando múltiples threads que se están ejecutando concurrentemente comparten recursos.

# 5. Trabajo Desarrollado

## 5.1 Comportamiento

La simulación comienza desde una de las máquinas donde se encuentra la clase *Manejador*. Allí se pueden configurar cuáles son las transacciones que se quieren ejecutar, dónde se desea que se inicie cada una de ellas (en qué sitio) y qué máquinas tendrán fallas y en qué momento. Una vez que se inicia la simulación se le envía dicha información a los sitios correspondientes. (Cabe destacar que cuando se habla de máquina y sitio, se hace referencia a lo mismo).

En cada sitio participante de la simulación debe estar corriendo un proceso llamado *ProcesadorDeServidor* que acepte peticiones para ejecutar consultas. Él recibirá del *Manejador* los pedidos para que coordine alguna transacción o recibirá de otro *ProcesadorDeServidor* la petición de tener un agente que ejecute una sub-transacción.

El funcionamiento de un servidor implica la escucha de conexiones, su aceptación, el procesamiento de solicitudes por las conexiones y su finalización una vez que todas las solicitudes hayan sido procesadas. El manejo de conexiones múltiples se ejecuta por medio de múltiples Threads. Por lo tanto, como el *ProcesadorDeServidor* puede llegar a recibir peticiones de ejecución de varios procesos (pedidos para ejecutar algún Coordinador de transacciones globales, y Participantes de otras transacciones globales). Para que no hayan esperas, se creo otro proceso (Thread) llamado *ClienteDeServidor* que se encarga de procesar uno de esos pedidos (existe uno por pedido), logrando que puedan satisfacerse varios al mismo tiempo.

El *ClienteDeServidor* de acuerdo a lo que se le pida, creará un proceso *Coordinador* o uno Participante (en la aplicación este proceso se llama *Localidad*).

En cada sitio existe un mapa que indica en qué lugares se encuentran las distintas tablas de la base de datos, ya que como existe replicación de datos, y se está utilizando la actualización **sincrónica,** deben actualizarse todas las bases de datos al mismo tiempo.

Para la ejecución de las transacciones globales se sigue el protocolo de Commit de dos Fases. Por lo tanto es de vital importancia la comunicación que habrá entre el Coordinador y los Participantes de una transacción global.

## 5.2 Modelo

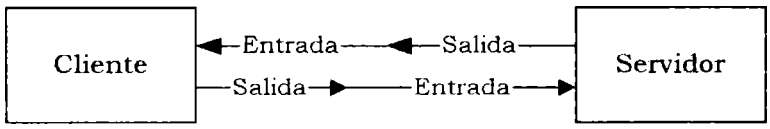
Se implementaron tres paquetes. Uno para el manejo de Sockets (PackageSockets), otro propio de la simulación (PackageTesis) y un tercero que tiene clases que son útiles a las clases de los otros dos paquetes (PackageUtiles).

### 5.2.1 PackageSockets

El manejo de sockets en Java es de un nivel bastante bajo de programación. Por lo tanto se decidió crear clases de manera que el manejo de los mismos esté a más alto nivel, facilitando la programación de las comunicaciones. Este paquete es totalmente independiente de la aplicación, de manera que puede ser utilizado por cualquier otra donde se necesite intercambiar información entre computadoras a través de los sockets de Java.

Se crearon las clases *SocketServidor* y *SocketCliente* para representar a los sockets servidor y cliente respectivamente, para que encapsulen las operaciones que pueden realizarse sobre ellos.

Todos los sockets, ya sean clientes o servidores, intercambian información a través de un canal de entrada y uno de salida. En la comunicación punto a punto, los datos disponibles en el canal de entrada del Cliente serán los enviados por el Servidor por su canal de salida, y los datos enviados por el canal de salida del Cliente serán recibidos por el canal de entrada del Servidor.



Se agruparon las operaciones posibles sobre estos canales y se implementaron dos nuevas clases: *CanalEntrada* y *CanalSalida*.

**Clase CanalEntrada**

Como se mencionó anteriormente, encapsula las operaciones posibles sobre el canal de entrada al socket (ya sea Cliente o Servidor). Algunas de las operaciones implementadas son:

- *leer()*: Retorna en un String el mensaje recibido en el canal de entrada al socket
- *cerrar()*: Cierra el canal de entrada.

Esta clase es utilizada por las clases *SocketCliente* y *SocketServidor* que se verán más adelante.

**Clase CanalSalida**

Encapsula las operaciones posibles sobre el canal de salida del socket (ya sea Cliente o Servidor). Algunas de las acciones posibles sobre ella son:

- *enviar(String msg)*: Envía el mensaje "msg" a través del canal de salida del socket.
- *cerrar()*: Cierra el canal de salida.

Esta clase es utilizada por las clases *SocketCliente* y *SocketServidor* que se verán a continuación.

**Clase SocketCliente**

Representa a un socket que es cliente de un servidor y encapsula todas las operaciones posibles sobre él. Tiene como variable de instancia un objeto de la clase java.net.Socket. Se comunica con el servidor a través de un *CanalEntrada* y un *CanalSalida* (ambas son sus variables de instancia).

A continuación se detallan las operaciones de mayor relevancia:

- *enviarMensaje(String msg)*: Envía el mensaje "msg" al servidor. Esta tarea es realizada por su *CanalSalida*.

- *leerMensaje()*: Retorna en un String el mensaje recibido desde el servidor. Es realizada por su *CanalEntrada*.
- *cerrar()*: Cierra la conexión con su servidor. Esta operación cierra el *CanalEntrada*, el *CanalSalida* y el socket en sí mismo.
- *isReadyTimeOut(int tiempo)*: Al no existir ningún método para esperar algún mensaje en el canal de entrada por un tiempo determinado, se implementó uno que espera un mensaje durante un "tiempo" determindo. Si pasado ese límite no se recibió ningún mensaje retornará un String vacío, de lo contrario retorna el mensaje recibido. Este punto merece una explicación que será realizada más adelante cuando se vea el paquete PackageUtiles.

**Clase SocketServidor**

Esta es la clase más compleja del paquete. Representa a un socket Servidor y encapsula todas las operaciones posibles sobre él. Tiene como variable de instancia un objeto de la clase java.net.ServerSocket. Como todo servidor, puede atender a varios clientes al mismo tiempo, mantiene un registro de todos sus clientes y se encarga de la comunicación con ellos, teniendo varios *CanalEntrada* y *CanalSalida* (uno por cada cliente).

Las operaciones más importantes son:

- *aceptarCliente()*: Acepta un nuevo cliente y lo guarda en su vector de clientes. Además agrega un nuevo *CanalSalida* y un *CanalEntrada* para comunicarse con ese cliente.
- *enviarMensajeA(int index, String msg)*: Cada cliente tiene un identificador, de manera que cuando el servidor le quiere enviar un mensaje, envía el "msg" a través del correspondiente CanalSalida al cliente identificado con el número "index".
- *enviarMensajeATodos(String msg)*: Hace el broadcast del mensaje "msg". Se lo envía a todos sus clientes a través de los *CanalSalida* correspondientes.
- *leerMensajeDe(int index)*: lee un mensaje del cliente identificado con el número "index" y lo retorna en un String.

- *isReadyTimeOut(int index, int tiempo):* Intenta leer un mensaje del cliente identificado con el número "index" por un "tiempo" y retorna lo leído. Si el tiempo expirara y no se recibió nada, se retorna un string vacío.
- *Cerrar():* Cierra todos los canales y la conexión propia.

### 5.2.2 PackageUtiles

Este paquete contiene clases que son utilizadas por los otros dos paquetes, pero también actúa como repositorio de clases que son necesarias en la aplicación pero que no tienen importancia relevante para ser ubicadas en el paquete principal.

Las clases comunes utilizadas por ambos paquetes son: *ManejoArchivo*, *Dormilon* y *Activo*.

#### Clase ProcesadorDeMensajes

Contiene la codificación de todos los mensajes que pueden ser enviados. Para que exista comunicación entre unidades, se debe establecer un protocolo común de comunicación. Esta clase es quien establece dicho protocolo.

Para reducir la sobrecarga en las comunicaciones, se permiten enviar varios datos en un mismo mensaje. Cada uno de ellos tiene la forma <tipo|valor>. Aquí están definidos además los separadores de dicha datos.

Es la encargada de armar y desarmar la mayoría de los mensajes, reduciendo la posibilidad de errores de interpretación en la comunicación, ya que al ser una misma clase la que realiza esta tarea no hay errores en la codificación de los mismos. La comunicación es uno de los ejes fundamentales de la aplicación, en varios puntos de la misma se arman y desarman mensajes. Si no existiera esta clase, la probabilidad de errores sería muy alta ya que los tipos de mensajes deben ser codificados para reducir el tráfico en la red. Además facilita la tarea del programador al no necesitar recordar la codificación de cada mensaje. Esto permite la modificación de los códigos sin que las comunicaciones se vean afectadas.

**Clase Consulta**

Contiene los datos necesarios para poder realizar una consulta: el código de la consulta, y los valores de los parámetros. Actúa como un simple repositorio de información. Solo encapsula los datos de una consulta. Una de las operaciones principales es:

toString(): Este método se encarga de codificar la clase en un formato de String, de manera que pueda ser enviada a través de un canal de comunicación y ser interpretada cuando llega a destino. Para esto requiere la ayuda del *ProcesadorDeMensajes*.

**Clase Global**

Mantiene valores globales de la aplicación, como lo son el puerto de comunicación por el que se comunicarán los procesos servidores, los tiempos del timeout, etc.

**Clase ManejoArchivo**

En Java no existe una clase que permita escribir en archivos de una forma fácil. Por lo tanto se implementó ésta para poder realizar las operaciones de escritura deseadas, sin que el programador deba preocuparse por cómo lo realiza.

La mayoría de las clases de la aplicación la utilizan para almacenar el seguimiento de lo sucedido. Es por ello que esta clase sólo tiene utilidad para los archivos de texto.

Las operaciones que están disponibles son:

- *ManejoArchivo(String nomArchivo):* Es el mensaje creador y genera un archivo físico con el nombre establecido en el parámetro.
- *escribirln(String cadenaSt):* Escribe en el archivo el string "cadenaSt", terminando con un retorno de carro.
- *escribir(String cadenaSt):* Escribe en el archivo el string "cadenaSt", sin un retorno de carro.

**Clase ManejoBD**

Se encarga del manejo de la base de datos local para una sub-transacción. Cabe destacar que se utiliza el puente JDBC-ODBC que se mencionó con anterioridad para conectarse con la base de datos local.

Las operaciones más importantes que contiene son:

- *ejecutarConsulta(Consulta consulta)*: Ejecuta la consulta en la base de datos local. Recibe como parámetro un objeto de la clase que se describió con anterioridad para encapsular una consulta de base de datos. La consulta es ejecutada utilizando las clases y métodos del paquete "java.sql" visto en el punto 4.5
- *cometer()*: Tiene como precondición que se haya ejecutado con anterioridad el método 'ejecutarConsulta'. Realiza la acción que nombra: comete la consulta ejecutada.
- rollback(): Tiene el efecto contrario al mensaje anterior. Pero coinciden en la precondición de que tiene que haberse ejecutado con anterioridad el método 'ejecutarConsulta' para poder deshacer la misma.

**Clase Bitacora**

Esta es una clase sencilla, sirve para almacenar el estado de un agente, de una localidad que está ejecutando una sub-transacción. Más adelante se verá que un objeto de esta clase será compartido por las clases *Localidad* y *ReceptorMensajesVecinas* (ambas del paquete PackageTesis).

**Clases ErrorManager, Activo y Dormilon**

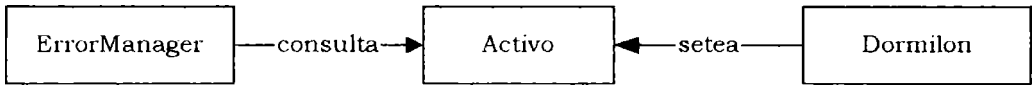
Estas tres clases tienen una actividad en conjunto y requieren una explicación especial para poder comprenderlas.

Existe un único objeto de la clase *ErrorManager* por máquina involucrada en la base de datos distribuida simulada. Es conocido por los procesos involucrados en un mismo sitio, tanto por coordinadores como participantes. Las otras dos (*Activo* y *Dormilon*) son internas y propias de *ErrorManager*.

Sirven para simular las caídas de un sitio dado.

**Clase Activo**

Esta clase guarda el estado de caída, o no, de un sitio. Es un objeto compartido por *ErrorManager* y *Dormilon*. A través de él *ErrorManager* puede saber si terminó el estado de caída.




*Dormilon* setea valores a la clase *Activo* y *ErrorManager* los consulta.

**Clase Dormilon**

La clase *Dormilon* en un Thread cuya única actividad es dormirse por un tiempo determinado. Su ejecución representa la caída de una máquina. De esta manera, cuando empieza a ejecutar, le avisa a *Activo* de esta situación, para que éste se entere de la caída del sitio. Una vez finalizado, vuelve a avisarle a *Activo* par que se entere del reinicio.

**Clase ErrorManager**

Es quien se encarga de manejar la simulación de la falla de un sitio. Como es simulada, necesita la ayuda de otras clases para poder hacerlo. Tiene como variable de instancia a un objeto de la clase *Activo*. Para simular una falla, se instancia un objeto de la clase *Dormilon* para que se duerma por el tiempo de la falla. Cuando lo instancia le da su objeto Activo, ya que es la única forma de mantenerse informado sobre la situación de *Dormilon* y así saber cuándo terminó la falla.

Una vez que se han definido las clases *Activo* y *Dormilon*, se está en condiciones de explicar el funcionamiento de los métodos isReadyTimeOut de las clases *SocketCliente* y *SocketServidor* del paquete *PackageSockets*. Este método espera recibir un mensaje por el canal de entrada por un tiempo determinado. Como esta operación no existe en los sockets de Java, fue

necesario implementarla. Esto fue posible con la ayuda de las clases antes mencionadas. A continuación se presenta el fragmento de código que implementa dicho método:

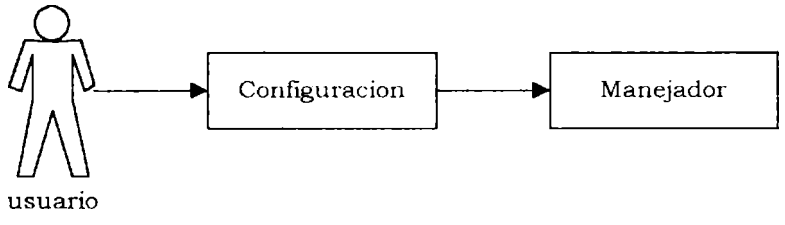
```
public String isReadyTimeOut(int tiempo)
{
   String mensaje = "";
   Activo activo = new Activo();
   activo.setearEstaActivo(false);
    Dormilon dormilon = new Dormilon(activo, tiempo);
   dormilon.start();
   while (!(activo.retornarEstaActivo()) & (mensaje.equals(""))){
     mensaje = this.leerMensaje();
   }
  dormilon.setFin();
  dormilon = null;
return mensaje;
 }
```

### 5.2.3 PackageTesis

Este es el paquete principal del simulador. Aquí están implementados los procesos que atienden pedidos de ejecución, los coordinadores de transacciones, los participantes, etc.

#### Clase Configuración

Implementa la ventana de configuración del simulador. A través de ella se pueden configurar cuáles son las transacciones que se quieren ejecutar, dónde se desea que se inicie cada una de ellas, y qué máquinas tendrán fallas y en qué momento. Además es quien da comienzo a la simulación. Su funcionalidad es solo visual, ya que es un objeto de la clase *Manejador* quien realmente realiza las tareas antes mencionadas. Por lo tanto actúa de intermediaria entre el usuario y el *Manejador.*



**Clase ConfiguracionMain**

Es para poder ejecutar la clase antes mencionada. Solo dispara la ejecución de la ventana *Configuracion*, dándole las medidas iniciales y ubicándola en un lugar predeterminado de la pantalla.

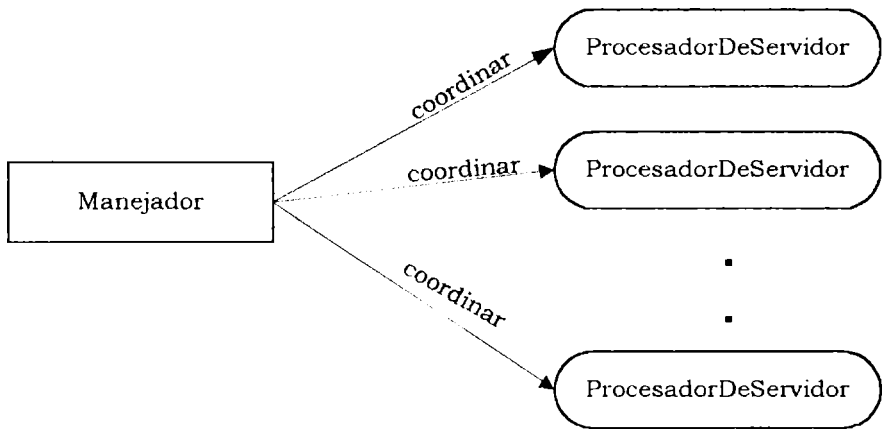
**Clase ObjetoDeConfiguracion**

Es una clase contenedora de información. Sus propiedades son muy similares a la clase *ObjetoDeEjecución* (que se verá más adelante) con la diferencia de que es de uso puramente visual. Le sirve a la clase *Configuracion* para mantener visualmente lo que el usuario predetermina (*Configuracion*, mantiene un vector de instancias de esta clase).

**Clase Manejador**

Es la encargada real de iniciar la simulación.

Para iniciarla efectivamente, una vez realizada la configuración, se le envía a los *ProcesadorDeServidor* que serán coordinadores de transacciones, la información necesaria para que en los sitios donde se encuentren puedan ejecutar un proceso que se encargue de eso. Para poder enviar dicha información, debe crear un *SocketCliente* por cada *ProcesadorServidor*, estableciendo de esta manera una comunicación punto a punto con cada uno de ellos.



**Clase ObjetoDeEjecucion**

Contiene los datos individuales de cada una de las transacciones que hay que ejecutar. En él se almacena la consulta, la dirección IP de la máquina donde se debería iniciar dicha consulta (donde residirá el coordinador de la misma) y un código de error que indica si dicha localidad va a fallar (a los efectos de la simulación).

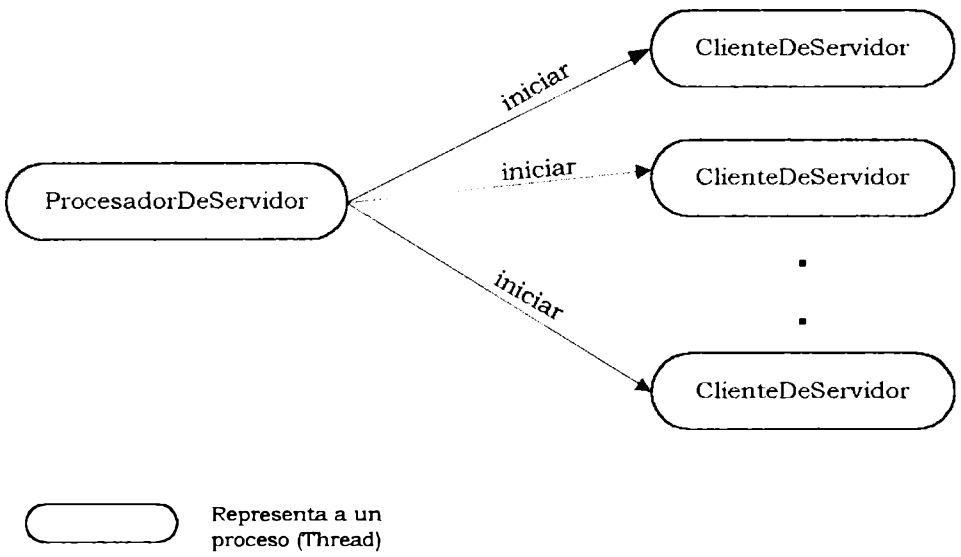
Esta clase es utilizada por el *Manejador* cuando se realiza la configuración y para el comienzo de la simulación.

**Clase ProcesadorDeServidor**

Existe una instancia de esta clase en cada máquina involucrada en la simulación. Tiene un *SocketServidor* a través del cual se comunicarán con él para hacerle los distintos requerimientos.

Se encarga de administrar lo que sucede en la máquina en donde se encuentra con un alcance local.

Cada vez que un cliente se conecta al *SocketServidor*, significa que el *ProcesadorDeServidor* está por recibir una petición. Para evitar retrasos, se crea un objeto de la clase ClienteDeServidor para que se encargue del tratamiento de la misma.

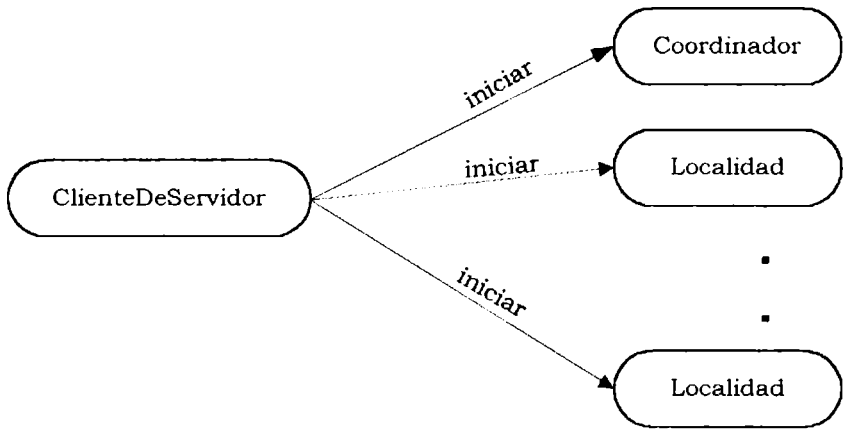


**Clase ClienteDeServidor**

Esta clase es un Thread que se encarga de atender los pedidos que se reciben de uno de los clientes del *ProcesadorDeServidor,* y actúa en base a los mismos.

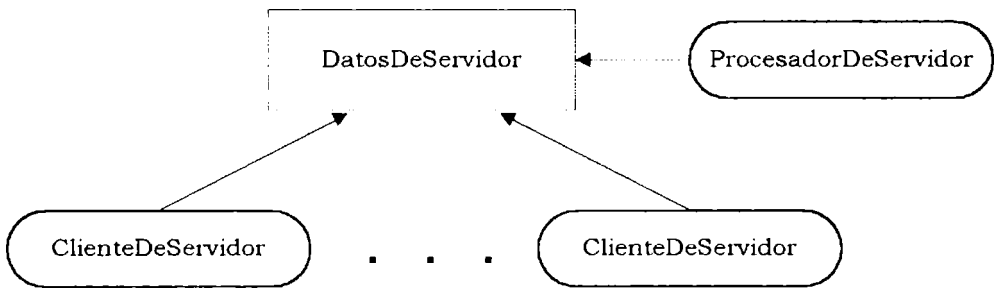
Si recibe un pedido del *Manejador*, normalmente es para que se encargue de coordinar una transacción global. En este caso se crea un proceso Coordinador y se le da comienzo.

Si recibe un pedido de un *Coordinador*, le pedirá que haya un agente que se encargue de ejecutar una sub-transacción. En este caso se creará un proceso *Localidad* que se encargará de realizar dicha tarea. Y también le dará comienzo.




**Clase DatosDeServidor**

Es una clase que contiene información local de cada máquina como lo es su número de IP, los puertos que se están asignando para las comunicaciones entre sockets, el *ErrorManager*, etc.



Existe un objeto de esta clase por máquina y es compartida por *ProcesadorDeServidor* y todos los *ClienteDeServidor* que están en el mismo sitio.

**Clase ReceptorMensajesVecinas**

Durante la ejecución de una transacción global, para mantener la integridad de la base de datos se utiliza el protocolo de Commit de Dos Fases. En el caso que una localidad haya enviado su voto "cometer" al Coordinador, y no recibiera ninguna respuesta de éste (utilizando un timeout), asume que se cayó. Para evitar esperar a que el Coordinador se restablezca, se consulta a las localidades vecinas para averiguar si alguna recibió alguna orden por parte del Coordinador. (Esta situación podría ocurrir si el Coordinador fallaba luego de haber tomado una decisión y haber enviado a alguna de las localidades la decisión final).

Para poder realizar esto, cada localidad debe tener conocimiento de sus vecinas, es decir de las otras localidades que participan de la misma transacción global. Para ello, cuando se le envían los datos de la consulta que tiene que ejecutar, también recibe una lista de los números de IP donde se encuentran las demás.

La clase *ReceptorMensajesVecinas* es un Thread cuya tarea es la de comunicarse con una de esas vecinas. Sirve para responder las consultas de las demás y para recibir las respuestas a las preguntas realizadas por la propia Localidad.

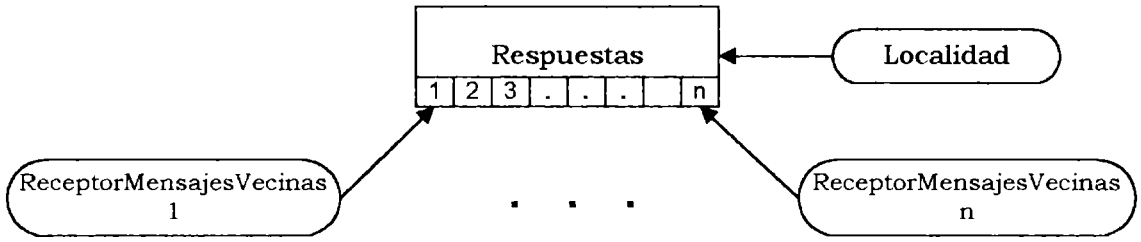
Cada *Localidad* tiene un *SocketServidor* a través del cual puede comunicarse con el resto. Como tiene un *SocketCliente* por cada localidad vecina, se instancia un *ReceptorMensajeVecinas* para que se encargue de ella.

Esta clase, como todos los Threads, tiene vida propia. Es necesaria para no bloquear el funcionamiento de la Localidad y así poder simultáneamente estar recibiendo pedidos de sus vecinas y respondiendo los mismos; así como también estar esperando respuesta por todos los canales a una consulta propia.

**Clase Respuestas**

Es conocida por la Localidad y los *ReceptorMensajesVecinas*. Aquí se almacenan los datos recibidos por el *ReceptorMensajesVecinas*. Es la única forma que tiene la Localidad de saber si alguna de sus vecinas ha respondido a su pedido.

Su contenido más importante es el de un Vector de Strings donde cada posición le pertenece a una Localidad vecina (éstas están identificadas).




**Clase Localidad**

Representa a un agente; un participante de una transacción global que debe encargarse de la ejecución de una sub-transacción.

La clase *Localidad* es un Thread. Contiene un *SocketCliente* a través del cual se comunica con el Coordinador de la transacción. Esta clase es instanciada por *ClienteDeServidor* cuando recibe un pedido de un *Coordinador*. Junto con el pedido, recibe los datos que necesitará la *Localidad* para poder comunicarse con él. Es por ello que conoce la dirección IP de donde se encuentra el *Coordinador* y el puerto por el cual se comunica. Estos son los datos necesarios para poder establecer una comunicación.

También conoce la dirección IP de la máquina donde se encuentra y el puerto por el que se va a comunicar con sus vecinas. Este número de puerto es administrado por *DatosDeServidor*. Como pueden existir varias comunicaciones hacia la misma dirección IP, los distintos clientes se comunicarán por distintos puertos. Suponiendo que en una misma máquina están corriendo tres *Localidad*es distintas, cada una de ellas recibirá pedidos de otras a través de los puertos, por ejemplo, 8202, 8204 y 8206 (uno para cada una).

Comparte con el resto de los procesos que se están ejecutando en el mismo sitio, un *ErrorManager* para enterarse de la simulación de una falla.

Cuando la *Localidad* comienza a ejecutar, se comunica con el *Coordinador* y le envía sus propios datos (dirección IP y puerto por el que se comunicará con las vecinas). Esto es necesario para que luego de obtener esta información de todas las *Localidade*s participantes, el *Coordinador* pueda enviarla a cada una de ellas para que se enteren de quiénes son sus vecinas de ejecución.

Espera recibir la consulta por parte del *Coordinador* y a continuación se encarga de su ejecución a través de *ManejoBD*.

Si no hubieron errores o no hubo una caída, espera a recibir el mensaje "Preparar" del *Coordinador*, siguiendo los pasos del protocolo de Commit de Dos Fases.

A continuación le envía al *Coordinador* su voto "cometer" si no hubieron fallas o el voto "abortar" en el caso contrario. Y se queda esperando una respuesta por parte del *Coordinador* para que le indique la decisión final sobre el destino de la transacción.

Una vez que recibe dicha respuesta, comete o aborta la sub-transacción ejecutada, de acuerdo al mensaje recibido ("commit global" o "aborto global"). A continuación le envía un mensaje de "reconocimiento" para que el *Coordinador* pueda terminar y pueda cerrar todos los canales de comunicación.

Los pasos que se describieron anteriormente son los que corresponden a la ejecución norma de una transacción sin fallas.

A continuación se verá cómo actúa la *Localidad* implementada, en caso de fallas. Los puntos importantes de falla en una *Localidad* son los siguientes:

1. Antes de recibir el mensaje "preparar"
2. Luego de recibir "preparar", pero antes de enviar su voto
3. Luego de enviar su voto, pero antes de recibir la decisión final.
4. Luego de recibir la decisión final.

En el primer caso, el *Coordinador* nunca llega a recibir respuesta de la *Localidad* por lo que asume un aborto. Por lo tanto, cuando la *Localidad* se

recupere de la falla sólo deberá realizar el aborto de lo que antes ejecutó (aborto unilateral).

El segundo caso es muy similar al primero, también el *Coordinador* asumió un aborto, cosa que deberá realizar la *Localidad* cuando se reinicie.

El tercero es el más complejo. Debe consultar al *Coordinador* cuál fue la decisión final. Por lo tanto le envía un mensaje de consulta y se queda esperando la respuesta. Una vez que recibió la decisión, ejecuta la acción correspondiente, le envía un reconocimiento y termina.

En el cuarto caso, ya se conocía el estado de terminación, por lo que sólo debe llevar a cabo dicha acción (si no la llegó a ejecutar antes de la falla), le envía el reconocimiento al *Coordinador* y termina.

Es necesario considerar también la posible caída del *Coordinador.* Existen dos casos de fallas del *Coordinador* en donde la *Localidad* se ve directamente afectada en la ejecución y seguimiento del protocolo de Commit de Dos Fases:

1. El *Coordinador* falla luego de haber enviado "preparar" a las Localidades.
2. El *Coordinador*, luego de haber tomado una decisión (una vez recibidos todos los votos) falla al enviar dicha decisión a las Localidades.

En el primer caso, la *Localidad* se queda esperando durante un tiempo (limitado por un timeout) la respuesta del *Coordinador* (mediante el método isReadyTimeOut() de la clase *SocketServidor* que se describió con anterioridad). Si no lo recibiera, asume una falla del *Coordinador* y aborta la transacción.

El segundo caso es complejo. Si la *Localidad* había votado "abortar", no es importante recibir la respuesta final del *Coordinador* porque ella ya la conoce (sería un "aborto global"). El problema se presenta cuando la *Localidad* había votado "cometer", ya que debería quedarse esperando obligatoriamente una respuesta (porque la acción final depende del resto). Es aquí donde entran en juego las clases *ReceptorMensajesVecinas* y *Respuestas.* Cuando una *Localidad* se da cuenta que el Coordinador está caído (esperó sin éxito una respuesta durante un período de tiempo), le pide ayuda a sus vecinas, porque

pudo haber sucedido que el *Coordinador* antes de fallar haya podido enviarle a alguna de ellas la decisión final. Si alguna le respondiera, conoce la decisión final y actúa en base a ella. Si ninguna lo supiera, se queda bloqueada hasta que el *Coordinador* se recupere.

**Clase Coordinador**

Cuando un *ClienteDeServidor* recibe un pedido de coordinar una transacción, crea una instancia de esta clase.

Es un Thread que efectivamente se encarga de coordinar la ejecución de una transacción global, siguiendo el protocolo de Commit de Dos Fases para mantener la integridad de la base de datos.

Se sabe que en cada máquina participante de la base de datos distribuida simulada, existe una tabla Mapa donde figuran todas las tablas de la base de datos distribuida junto con la dirección IP de la máquina donde se encuentran.

Cuando se inicia el *Coordinador*, examina la consulta y establece cuáles son las tablas involucradas. Una vez conocidas, a través del Mapa se entera de dónde deberían haber agentes ejecutando sub-transacciones.

Se conecta con el *ProcesadorDeServidor* de las máquinas necesarias y emite un pedido de ejecución de una Localidad. Al establecer la comunicación, cada *ProcesadorDeServidor* destino crea un *ClienteDeServidor* para que se encargue de él. De esta manera hay un *ClienteDeServidor* por cada máquina involucrada que recibe el pedido del *Coordinador* y se encarga de iniciar una *Localidad*. Lo que sucede con cada *Localidad* ya se ha visto con anterioridad. Ahora se verán en detalle los pasos que sigue el *Coordinador*.

Maneja un *SocketServidor* a través del cual se comunica con las *Localidades* participantes. Una vez comunicado, espera recibir la información de las distintas *Localidades* en los canales de entrada del *SocketServidor*. Cuando las recibe, reenvía envía a todas ellas esa información.

A continuación envía la sub-transcción a ejecutar. Y luego el mensaje "preparar".

Se queda esperando los votos de todas las localidades. Si no ocurrieran errores y los recibiera a todos, está en condiciones de decidir el destino final

de la transacción global. No está demás recordar cómo se decide esto: si alguna de las *Localidades* votó "abortar", entonces la decisión final será "abortar global"; si en cambio todas las *Localidades* votaron "cometer", entonces la decisión será "commit global".

Por último envía por el canal de comunicación que tiene con cada *Localidad*, la decisión final, y se queda esperando los reconocimientos de las mismas para poder cerrar los canales y las conexiones con seguridad.

Los pasos descriptos hasta ahora, son los que ocurren si ninguna falla sucediera. La caída de una máquina es conocida a través de un objeto de la clase *ErrorManager* que es compartida por todos los procesos que se ejecutan en un mismo sitio. A continuación se mencionan los puntos de falla de interés:

1. Al comienzo, antes de enviar cualquier información a algún proceso.
2. Luego de haber enviado "preparar" a las *Localidades*.
3. Luego de haber tomado una decisión (una vez recibidos todos los votos) falla al enviar dicha decisión a las *Localidades*.

En el primer caso, como fue antes de iniciar el protocolo, puede recomenzarlo, ya que ninguna *Localidad* se vio afectada porque no existía.

En el segundo caso, como se describió en la clase *Localidad*, ésta asumirá la falla del Coordinador al no recibir ningún mensaje y abortará la ejecución de la sub-transacción. Por lo tanto el *Coordinador* aborta la ejecución y termina.

En el tercero envía a las *Localidades* la decisión final sobre la ejecución de la transacción, ya que estas se quedaron esperando la respuesta final.

Ahora se verá cómo se ve afectada la ejecución del *Coordinador* ante la caída de alguna de las *Localidades*. Los posibles puntos de falla son:

1. Antes de recibir "preparar"
2. Luego de recibir "preparar", pero antes de enviar su voto
3. Luego de enviar su voto, pero antes de recibir la decisión final.
4. Luego de recibir la decisión final.

En el primer caso, el *Coordinador* nunca recibió respuesta de la Localidad por lo que asumió un aborto. Por lo tanto, cuando la Localidad se

recupere de la falla sólo deberá realizar el aborto de lo que antes ejecutó (aborto unilateral).

El segundo caso es muy similar al primero, también el *Coordinador* asumió un aborto, cosa que deberá realizar la *Localidad* cuando se reinicie.

En el tercer caso el *Coordinador* recibirá una consulta de la *Localidad* que falló y le envía un mensaje con la decisión final

En el cuarto caso, la *Localidad* ya conocía el estado de terminación, por lo que sólo debe llevar a cabo dicha acción (si no la llegó a ejecutar), le envía el reconocimiento al *Coordinar* y termina.

## 5.3 Arquitectura Utilizada

Las computadoras sobre Internet están conectadas a través del protocolo TCP/IP. En la década del 80, ARPA (Advanced Research Projects Agency) del gobierno de EEUU, desarrolló una implementación bajo UNIX del protocolo TCP/IP. Allí fue creada una interface socket. En la actualidad, la interface socket constituye el método más usado para el acceso a una red TCP/IP. [TCPW1]

Un socket es una abstracción que representa un enlace punto a punto entre dos programas ejecutándose sobre una red TCP/IP. Utiliza el modelo Cliente/Servidor. Cuando dos computadoras desean comunicarse, cada una usa un socket. Una de ellas es el "Server" que abre el socket y espera por alguna conexión. La otra es el "Cliente" que llamará al socket server para iniciarla. Además, para establecer la conexión, solo será necesaria la dirección del Server y el número de puerto que se utilizará para la comunicación. Aquí se utiliza principalmente el package java.net. Cuando los dos sockets están comunicados, el intercambio de datos se realiza mediante InputStreams y OutputStreams.

La ventaja de este modelo sobre otros tipos de comunicación se ve reflejada en que el Server no necesita tener ningún conocimiento sobre el sitio en donde reside el cliente. Por otra parte, plataformas como UNIX, DOS, Macintosh o Windows no ofrecen ninguna restricción para la realización de la comunicación. Cualquier tipo de computadora que soporte el protocolo TCP/IP

podrá comunicarse con otra que también lo soporte a través del modelo de sockets.

Las simulaciones efectuadas se realizaron sobre una LAN-WAN. De esta manera cada terminal simula una base de datos que es parte de una base de datos distribuida. Y las comunicaciones se realizan a través de los mencionados sockets.

## 5.4 Resultados Obtenidos

Para demostrar la gran utilidad del simulador, se presentan los resultados obtenidos luego de realizar algunas ejecuciones. El ambiente simulado es de 2 máquinas que se llamarán X e Y. En ambas existen réplicas de las tablas Clientes y Artículos. El Manejador se encuentra en una tercera máquina para no interferir en el funcionamiento de los que serán participantes de la base de datos distribuida.

La tabla de tiempos de ejecución, medidas en milisegundos, muestra en cada máquina cuánto tiempo demoraron los distintos procesos que se ejecutaron. Con esta tabla se pueden obtener varias conclusiones.

Caso 1:

Transacciones ejecutadas:

1. Ejecución de la transacción global: "DELETE FROM Articulos WHERE descripcion = 'televisor' ", en los sitios X e Y, coordinándose en X
2. Ejecución de la transacción global: "DELETE FROM Clientes WHERE nombre = 'Ivana' ", en los sitios X e Y, coordinándose en Y.

Falla simulada:

- Sin fallas

Tabla de tiempos de ejecución:

En la máquina X:

| Actividad | Consulta | Tiempo |
|---|---|---|
| Coordinador | 1 | 5770 |
| Participante | 1 | 1750 |
| Participante | 2 | 5440 |

En la máquina Y:

| Actividad | Consulta | Tiempo |
|---|---|---|
| Coordinador | 2 | 4120 |
| Participante | 2 | 1750 |
| Participante | 1 | 4940 |

Conclusiones:

El que se presentó es el caso de la ejecución normal de transacciones concurrentes en la base de datos distribuida, con replicación de datos. Las dos transacciones involucradas cometen satisfactoriamente. Se puede observar cómo las transacciones que se ejecutan localmente (Participante de la consulta 1 en la máquina X y Participante de la consulta 2 en la máquina Y) son más rápidas que aquellas que tienen que responder a un coordinador que se encuentra en otra máquina (Participante de la consulta 2 en la máquina X y Participante de la consulta 1 en la máquina Y).

El tiempo que tarda en completarse la ejecución global, se ve representado en el tiempo de vida del coordinador. Cabe destacar que algunas diferencias en los tiempos dependen de las características físicas de las máquinas utilizadas.

Caso 2:

Transacciones ejecutadas:

1. Ejecución de la transacción global: "DELETE FROM Articulos WHERE descripcion = 'televisor' ", en los sitios X e Y, coordinándose en X
2. Ejecución de la transacción global: "DELETE FROM Clientes WHERE nombre = 'Ivana' ", en los sitios X e Y, coordinándose en Y.

Fallas simuladas:

- Caída del sitio Y antes de que el Participante de la consulta 1 haya recibido el mensaje de "preparar" del coordinador.

Tabla de tiempos de ejecución:

En la máquina X:

| Actividad | Consulta | Tiempo |
|---|---|---|
| Coordinador | 1 | 2090 |
| Participante | 1 | 1980 |
| Participante | 2 | 710 |

En la máquina Y:

| Actividad | Consulta | Tiempo |
|---|---|---|
| Coordinador | 2 | 490 |
| Participante | 2 | 15380 |
| Participante | 1 | 15870 |

Conclusiones:

Este es el caso de la ejecución de transacciones concurrentes en la base de datos distribuida, con replicación de datos, cuando ocurre

una falla en una de las máquinas involucradas en una transacción global. Ambas transacciones son abortadas.

La caída de la máquina Y cuando el Participante de la consulta 1 estaba en espera de recibir la orden "preparar", tiene varias implicancias.

Por un lado está la ejecución del Coordinador de esa transacción, que se encuentra en X. Éste, envió "preparar", sin saber si el participante lo recibió o no, pero se queda esperando por una respuesta. Al no recibirla, decide abortar y le envía al resto de los participantes esta decisión. Una vez que todas los participantes operantes le mandaron su reconocimiento, termina la transacción global, tardando 2090 milisegundos.

Asimismo, como en Y se produjo un fallo, el coordinador que iba a iniciar la transacción global (de la consulta 2), falla antes de poder enviar "preparar". De esta manera el participante de la consulta 2 en X, al no recibirlo, aborta. El timeout definido es de 6000 miliseg, por eso el tiempo total es de 710 milisegundos.

Los tiempos de duración de los participantes ascienden a 15000 milisegundos debido a que este tiempo es el definido para las caídas durante la simulación.

Caso 3:

Transacciones ejecutadas:

1. Ejecución de la transacción global: "DELETE FROM Articulos WHERE descripcion = 'televisor' ", en los sitios X e Y, coordinándose en X
2. Ejecución de la transacción global: "DELETE FROM Clientes WHERE nombre = 'Ivana' ", en los sitios X e Y, coordinándose en Y.

Fallas simuladas:

- Caída del sitio Y luego de que el participante de la consulta 1 haya enviado el voto de "cometer" al coordinador y antes de que reciba la decisión global.

Tabla de tiempos de ejecución:

En la máquina X:

| Actividad | Consulta | Tiempo |
|---|---|---|
| Coordinador | 1 | 15710 |
| Participante | 1 | 1430 |
| Participante | 2 | 1760 |

En la máquina Y:

| Actividad | Consulta | Tiempo |
|---|---|---|
| Coordinador | 2 | 16590 |
| Participante | 2 | 1820 |
| Participante | 1 | 15550 |

Conclusiones:

Este es el caso de la ejecución de transacciones concurrentes en la base de datos distribuida, con replicación de datos, cuando ocurre otro tipo de falla en una de las máquinas involucradas en una transacción global. Ambas transacciones logran cometerse satisfactoriamente.

El fallo de la máquina Y cuando el Participante de la consulta 1 estaba en espera de la decisión final, tiene varias implicancias.

Una de las actividades que se están llevando a cabo es la ejecución del Coordinador de esa transacción, que se encuentra en X. Éste, envió la decisión final, sin saber si el participante lo recibió o no, pero queda esperando un reconocimiento. En esta espera, también

atiende consultas de participantes que produjeron eventuales fallas, en este caso, luego de que el Participante de la consulta 1 en Y se haya recuperado, le envía una consulta para saber cuál fue la decisión final. El coordinador le responde, y una vez recibidos todos los reconocimientos, finaliza. Es por la falla producida en la máquina Y que el Coordinador tarda 16310 milisegundos para finalizar. Cabe destacar que el participante local (en X) de la misma consulta, no se vio afectada por la falla mencionada en Y.

Otra actividad, producida por el Coordinador de la consulta 2 que se encuentra en la máquina Y que falló, no llega a recibir los reconocimientos de los participantes. Al recuperarse y no recibir ninguna consulta, termina porque ya les había enviado su decisión final sobre el destino de la transacción. Como los participantes de esta transacción no sufrieron fallas, terminaron rápido (1760 milisegundos el participante de la máquina X, y 1820 el de la máquina Y), es el Coordinador, que por su falla, demora para finalizar la transacción satisfactoriamente (16590 milisegundos).

## 5.5 Conclusiones y Líneas de Trabajo Futuro

Se logró dejar operativo un ambiente experimental de especificación y desarrollo de transacciones distribuidas, cuyo soporte de hardware es una red LAN-WAN y el software Java.

Con él se pueden generar pruebas para el estudio del comportamiento de una Base de Datos Distribuida, incluyendo la simulación de fallos sobre el mismo y estudio de performance para recuperación, utilizando el protocolo de dos fases.

Los resultados son de tres trazas de ejecución representativas y se presentan como casos patrones. A partir del modelo de simulación es posible llevar a cabo las trazas de ejecución deseadas y observar el comportamiento del sistema en cada caso.

Las líneas futuras contemplan la generalización del modelo de simulación para poder evaluar el comportamiento de una traza de ejecución con otros protocolos de "commit" conocidos. Entre ellos se pretende utilizar y evaluar el protocolo de "commit" de Tres Fases, el de "Pressumed Commit", el de "Pressumed Abort", y el Optimista.

# 6. Bibliografía


[Aba95] "Especificación y Simulación de Sistemas de Tiempo Real con Redes de Petri Extendidas (RPE) - Derivación de Código a partir de una RPE ". Abasolo, Cantarella, De Giusti. Proceedings CRICTE 95 – UFRGDS – Brasil.

[Andr91] Andrews, "Concurrent Programming", Benjamin/Cummings, 1991.

[ATR93], Atre, "Distributed Databases", McGraw-Hill, 1993.

[BELL92] "Distributed Database Systems". Bell, David; Grimson, Jane. Addison Wesley. 1992

[BHAS92] "The architecture of a heterogeneous distributed database management system: the distributed access view integrated database (DAVID)". Bharat Bhasker; Csaba J. Egyhazy; Konstantinos P. Triantis. CSC '92. Proceedings of the 1992 ACM Computer Science 20th annual conference on Communications, pages 173-179

[BONT95] "Database management, Principles and products". Bontempo, Charles; Maro Saracco, Cynthia. Prentice Hall 1995.

[BURL94] "Managing Distributed Databases. Building Bridges between Database Island". Burleson, Donal. 1994

[BUST88] "Concurrent Program Structures". Bustard, Elder, Welsh, Prentice Hall, 1988.

[CHU79] "Centralized and Distributed Data Base System". Chu, W; Chen, P, IEEE Catalog No, EHO 154-5, 1979.

[COFF92] "Parallel programming- A new approach". M. Coffin. Prentice Hall, Englewood Cliffs, 1992.

[COM93] "Internetworking with TCP/IP Vol III : Client-Server Programming And Applications". BSD Socket Version. Comer D, Stevens D. Prentice Hall, Inc., 1993.

[COU94] "Distributed Systems Concepts and Design". Coulouris G, Dollimore J, Kindberg T. Addison Wesley, 1994.

[DASG87] "Computer Architecture: A modern synthesis". S. Dasgupta. Wiley, 1987.

[DATE 93] "Introducción a los sistemas de Bases deDatos". Date, C.J. Addison Wesley 1993.

[DAV85] "Consistency in partitioned Nerworks". Davison, S.B; García Molina H, Computing Surveys (septiembre 1985)

[DEBL90] "Computer Architecture". Mario de Blasi. Addison-Wesley Publishing Comp., 1990.

[ECK97] "Thinking in Java". Eckel B. MindView Inc. 1997.

[FORT85] "Design and analysis of distributed real-time systems". P. Fortier. McGraw-Hill, 1985.

[GEH86] "Software Specification Techniques". N. Gehani, A. D. McGettrick. Addison Wesley, 1986.

[HAN97] "Diseño y administración de Base de datos". Hansen, G; Hansen J. Prentice Hall, 1997

[HAT88] "Strategies for Real-Time System Specification". Hatley D., Pirbhai I. Dorset House, 1988.

[HEER91] "Parallel Algorithms in Computational Science". D. W. Heermann, A. N. Burkitt. Springer-Verlag, 1991.

[HON94] "Query Processing in Parallel Relational Database Systems ". Hongjun Lu, Beng Chin. Ooi, ian Lee Tan IEEE, 1994

[HWAN84] "Computer architecture and parallel processing". Hwang, Briggs. McGraw Hill, 1984.

[HOA85]"Communicating Sequential Processes". Hoare C; Englewood Cliffs. Prentice-Hall, 1985.

[IEEE] Colección de "IEEE Transactions on Parallel and Distributed Systems", IEEE.

[JAVA00] "Java 1.2 Al descubierto". Jaworski, Jamie. Prentice Hall, 2000.

[JAVA96] "Java FAQ list and Tutorial: a work in progress" , http://sunsite.unc.edu/javafaq/javafaq.html, 1996.

[JAVAW1] "JavaTM 2 Platform, Standard Edition, v 1.3 - API Specification". http://java.sun.com/j2se/1.3/docs/api/index.html

[JAVAW2] "The Java Tutorial - What is a Socket".
http://java.sun.com/docs/books/tutorial/networking/sockets/definition.htm

[KAR92] "Timing Parallel Programs That Use Message Passing". Karonis N. Journal of Parallel and Distributed Computing, 14, 1992.

[KRO'96] "Procesamiento de Bases de Datos". Kroenke D, Prentice Hall, 1996

[LAP88] "Desing and Applications of Real Time Systems". Laplante P. IEEE Press 1988.

[LAWS92] "Parallel processing in industrial real time applications". H. Lawson. Prentice Hall 1992.

[LEIG92] "Introduction to Parallel Algorithms and Architectures". F. T. Leighton. Morgan Kaufmann Publishers, 1992.

[LEV90] "Real Time System Design". Shem-Tov LEVI, Ashok Agrawala. McGraw-Hill Inc, 1990.

[MAR95] "Client Server Databases Enterprise Computing". Martin J, Leben J, Prentice Hall, 1995

[MORS94] "Practical Parallel Computing". H. S. Morse. Ap Professional, 1994.

[PET81] "Petri Nets Theory and the Modeling of Systems". J. Peterson. Prentice-Hall, 1981.

[PRE98] "Ingeniería de Software". Pressman R. McGraw Hill, 1992.

[RAYN87] "Algorítmica del paralelismo". Raynal. Omega, 1987.

[ROM98] "Sistemas Distribuidos de Tiempo Real. Experiencia en el diseño y puesta a punto de un sistema de mensajería para entrenamiento militar". Romero F., Sosa R. F., De Giusti A. Proceedings CACIC 98.

[ROZ88] "Advances in Petri Nets". Rozenberg (Ed.). Springer-Verlag, 1988.

[SHET90] "Federated database systems for managing distributed, heterogeneous, and autonomous databases". Amit P. Sheth; James A. Larson. ACM Computing Surveys.Vol. 22, No. 3 (Sept. 1990), Pages 183-236

[SHU92] "Software specification and design for real-time systems". Shumate K. Wiley 1992.

[SIL95] "Fundamentos de Bases de Datos". Silberchatz, Korth, Mc-Graw Hill 1995

[SUN96] "JDBC: a Java SQL API", Sun Microsystems Inc. 1996.

[TCPW1] "Protocolos TCP/IP". Juan Salvador Miravet Bonet. http://www4.uji.es/~al019803/Tcpip.htm

[THOM90] "Heterogeneous distributed database systems for production use". Thomas, Charles; Glenn R. Thompson; Chin-Wan Chung; Edward Barkmeyer; Fred Carter; Marjorie Templeton; Stephen Fox; Berl Hartman. ACM Computing Surveys. Vol. 22, No. 3 (Sept. 1990), Pages 237-266

[ULL97] A First Course in Data Base Systems. Ullman,J; Windom, J. Prentice Hall, 1997

[UMAR93] A. Umar, "Distributed Computing and Client-Server Systems", Prentice Hall, 1993.

[VAL99] Principles of Distributed Database Systems. M Tamer Ozsu; Patrick Valduriez. Prentice Hall, 1999.

[ZAN97] Análisis de Replicación en Bases de Datos Distribuidas, Zanconi, M, Tesis de Magister en Ciencias de la Computación, Universidad Nacional del Sur, Bahia Blanca, Argentina 1996.