U.N.L.P.

Facultad de Informática

Trabajo de Grado

"Estudio de Recuperación de errores en Bases de Datos Distribuidas"

Director: Lic. Rodolfo Bertone

Co-Director: Lic. Hugo Ramón

Sebastián Mariano Ruscuni

Diciembre – 2000

Agradecimientos

Primeramente quiero agradecer profundamente a mis padres, ya que sin su apoyo cotidiano, no hubiese conseguido nada.

Voy a comenzar recordando a los miembros del LIDI (Laboratorio de Investigación y Desarrollo en Informática) que, con solo la demostración diaria de las grandes personas que son, produjeron que los últimos años de mi carrera tengan otro color. Y en particular a los becarios y amigos/as que ingresamos juntos al laboratorio: Diego, Ivana y Laura.

Al grupo del ingreso 1994, que nunca voy a olvidar los grandes momentos pasados juntos. Y de ellos, principalmente a Ivana Miaton, con la cual realicé casi toda la carrera, y de la que guardo grandes recuerdos.

Por último, quiero agradecer profundamente a mi Director de Tesis, Rodolfo Bertone, el cual formó parte de todos mis emprendimientos, ya sea en el área de investigación como en la docente. Fue y es muy importante para mi, y me atrevo calificarlo como el principal responsable de que mis días en el LIDI hayan sido mucho más divertidos.

Sebastián Ruscuni, 4 de Diciembre de 2000

ÍNDICE GENERAL

Objetivos

El trabajo que se presenta es la última evolución conseguida hasta el momento de un ambiente de simulación donde se modelizan e implementan situaciones en la que una BDD debe mantener la integridad de la información ante fallos producidos durante la ejecución de transacciones, enfocándose en la utilización y posterior comparación de diferentes protocolos de commit atómicos (ACP).

La implementación se realizó en Java debido a diferentes aspectos como facilidad de trabajo, portabilidad, etc.

Las situaciones evaluadas en el ambiente se basan en la recuperación de fallos de transacciones en un entorno de datos distribuidos, permitiendo seleccionar, para el desarrollo de la simulación, entre los protocolos dos fases, de tres fases, el protocolo de presumed commit y presumed abort. A partir del desarrollo y los resultados obtenidos se presentan una serie de conclusiones respecto de las evaluaciones registradas.

CAPITULO 1

INTRODUCCIÓN

1.1 Sistemas Distribuidos. Conceptos generales

Se define un sistema distribuido como una colección de computadoras autónomas interconectadas mediante una red, con un software diseñado para brindar facilidades integradas de cómputo. [COUL95]. Dicho software habilita a las computadoras a coordinar sus actividades y compartir los recursos del sistema (hardware, software y datos). Los usuarios de un sistema distribuido correctamente diseñado deberían percibirlo como una unidad simple e integrada a pesar que pueda estar implementada sobre muchas computadoras, repartidas en diferentes ubicaciones.

Los sistemas distribuidos están implementados sobre plataformas de hardware que varían en su tamaño y conexión. Desde unas pequeñas workstations conectadas a través de una red de área local hasta Internet – millones de computadoras conectadas a través de redes de área local y global.

Las características claves que un sistema distribuido presenta son: soportes para compartir recursos, concurrencia, escalabilidad, tolerancia a fallos y transparencia.

- ✓ El *compartir recursos* es la característica fundamental de los sistemas distribuidos y afecta en gran proporción a la arquitectura de software disponible en dichos sistemas. Los recursos pueden ser datos, software o componentes de hardware.

- ✓ Mediante la *concurrencia* se consigue aumentar la performance. En sistemas distribuidos de propósito general se explota la concurrencia entre procesos cliente y servidor, pero nunca lo podríamos tener bajo una aplicación simple, a menos que haya sido implementada como un conjunto de procesos concurrentes.

- ✓ La *escalabilidad* ha sido un tema dominante durante la década pasada y su importancia continúa. La replicación de datos y el balance de carga son las principales técnicas utilizadas para proveerla.

- ✓ La *tolerancia a fallos* puede lograrse de mejor manera en arquitecturas de sistemas distribuidos que en centralizados. La redundancia de hardware puede ser utilizada como tarea principal para migrar procesos de una máquina a otra en caso de algún fallo. Pero la recuperación de fallos de hardware y software sin pérdida de datos requiere un diseño muy cuidadoso.

- ✓ La *transparencia* aduce a la necesidad de los usuarios y programadores de aplicaciones a percibir una colección de computadoras en red como un sistema integrado, produciéndose un ocultamiento de la distribución de los recursos.

El rango de aplicaciones distribuidas es por demás amplio, desde la provisión de facilidades de cómputo para grupos de usuarios, hasta sistemas de comunicación multimedia o bancarios automatizados.

1.2 Downsizing y Rigthsizing

Con el advenimiento de nuevo hardware (workstation más poderosas y de menor precio) y del software de redes robusto y confiable, el uso de grandes computadoras se vio limitado para migrar a redes locales o globales. Asociado a esto, aparece un

concepto nuevo, *downsizing*, que se refiere a migración de aplicaciones existentes para mainframes a redes de workstations. [UMAR93]

Para efectuar el downsize de una aplicación, el diseñador debe elegir migrar la aplicación completa a una estación de trabajo, o dividir manejo de datos, interfaces de usuario y lógica de la aplicación entre diferentes computadoras, utilizando por ejemplo un modelo Client Server. [BERS92].

Por otro lado, *rigthsizing* involucra la migración de aplicaciones a plataformas "correctas"; por ejemplo, a partir de grandes bases de datos ubicadas en mainframes, se traslada el procesamiento de la interface de usuario hacia workstations.

Existen diferentes estrategias de alocación, para lo cual se deben tomar las siguientes decisiones:

- ✓ Donde alocar la interface del usuario
- ✓ Donde alocar los programas de aplicación
- ✓ Donde alocar los datos

La interface del usuario puede llegar a ser alocada en cualquier computadora que posea el software cliente/servidor necesario para conectarse con los servidores de datos. Los programas de aplicación pueden ser alocados por razones similares.

La alocación de los datos trae aparejado nuevas consideraciones. Se basa en varios factores como tamaño de almacenamiento, costo y tiempo para la comunicación bajo lecturas o actualizaciones de los datos, tiempo de respuesta, etc. Por ejemplo, tener los datos en un solo lugar significa un menor tamaño de almacenamiento, un aumento en el tráfico de lecturas y un bajo tráfico de actualizaciones. En cambio si los datos los tuviésemos duplicados, aumentaría el tráfico de actualizaciones pero disminuiría el de lecturas, ya que su costo sería mayor.

Al centralizar los datos en los grandes mainframes, las compañías gastaban millones de dolares cada año; por lo que decidieron muchas de ellas a utilizar sistemas distribuidos de datos, con el objetivo de reducir los costos. En algunos casos, este downsizing produjo una reducción de costos y un aumento en la productividad. En otros, la necesidad de adquirir herramientas cliente/servidor de elevado costo. Además cabe destacar que la distribución de datos trae aparejado un pronunciado aumento en los recursos humanos para el manejo de los mismos. Es decir, las compañías deberán disponer de personal para manejar redes, expertos en comunicaciones para mantener la conectividad entre plataformas, y expertos en seguridad para permitir un control de acceso y mantenerlo a través de todas las plataformas de hardware disponibles.

1.3 Bases de Datos Distribuidas. Conceptos generales

De la misma forma en que el hardware y los soportes de redes fueron evolucionando, se observó el mismo cambio o evolución en los sistemas de bases de datos. En un principio las bases de datos fueron monousuarias, luego esta evolución llevó a bases de datos centralizadas, donde los usuarios acceden a los dados ubicados en una única computadora, puede ocurrir que este usuario esté interconectado a través de una red de área local, o eventualmente que su localidad esté distribuida dentro de una red de área global. La siguiente evolución en bases de datos llevó a distribuir la información, los datos dejaron de estar ubicados en una computadora y se distribuyeron geográficamente en varias workstation distribuidas en una WAN. [KROE96]

Cuando se analiza la división de los datos, teniendo en cuenta las características disponibles con BDD, es posible que la migración de los mismos incluya distribuirlos en varias computadoras de la red.

Una Base de Datos Distribuidas (BDD) puede ser definida como una colección integrada de datos compartidos que están fisicamente repartidos a lo largo de los nodos de una red de computadoras. Un DDBMS es el software necesario para manejar una BDD de manera que sea transparente para el usuario. [BURL94]

En un sistema de base de datos distribuida, los datos se almacenan en varios computadores. Los computadores de un sistema distribuido se comunican entre sí a través de diversos medios de comunicación, tales como cables de alta velocidad o líneas telefónicas. No comparten la memoria principal ni el reloj.

La tecnología general de BDD involucra dos conceptos diferentes, llamados *integración* a través de los elementos que componen una base de datos y *distribución* a través de los elementos de una red, como se observa en la siguiente Figura 1.1. La *distribución* es provista al repartir los datos a través de los diferentes sitios de la red, mientras que la *integración* está dada a partir del agrupamiento lógico de los datos distribuidos haciéndolos aparecer como una simple unidad ante la vista del usuario final de la BDD.

Los procesadores de un sistema distribuido pueden variar en cuanto a su tamaño o función. Pueden incluir microcomputadores pequeños, estaciones de trabajo, minicomputadores y sistemas de computadores grandes de aplicación general. Estos procesadores reciben diferentes nombres, tales como *localidades, nodos, computadores,* dependiendo en el contexto en donde se mencionen.

Un DBMS centralizado es un sistema que maneja una BD simple, mientras que un DDBMS es un DBMS simple que maneja múltiples BD. El término global y local se utiliza, cuando se discute sobre DDBMS, para distinguir entre aspectos que se refieren al sitio simple (local) y aquello que se refiere al sistema como un todo (global). La BD local se refiere a la BD almacenada en un sitio de la red, mientras que la DB global se refiere a la integración lógica de todas las BD locales.

El propósito de la BD es integrar y manejar los datos relevantes de una empresa o corporación. La motivación para definir esta BD es la de tener todos los datos relevantes de las operaciones de la empresa en un único almacenamiento, de manera que todos los problemas asociados con aplicaciones en una empresa puedan ser servidos de una manera uniforme. La dispersión geográfica de estas empresas hace a un modelo centralizado no aplicable, siendo un modelo distribuido la solución apropiada. [BELL92]

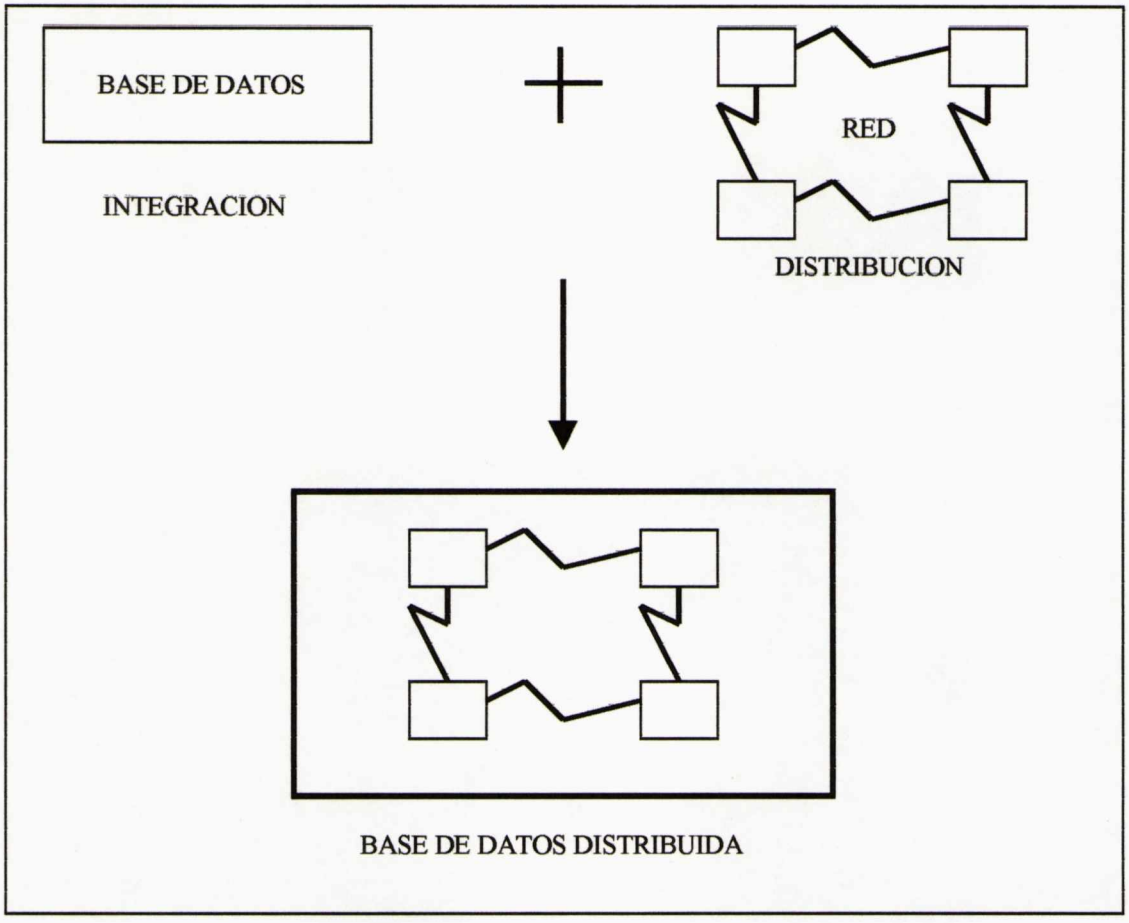


Figura 1.1

1.3.1 Bases de Datos Distribuidas Homogéneas, Heterogéneas y Federativas

Un sistema de datos distribuido se puede dividir en dos tipos separados, basados en filosofías totalmente diferentes, y son diseñados para satisfacer necesidades disímiles:

- ✓ Sistemas de manejo de bases de datos distribuidas Homogéneas
- ✓ Sistemas de manejo de bases de datos distribuidas Heterogéneas

Un *DDBMS homogéneo* posee múltiples colecciones de datos; integra varios recursos de datos. Puede ser dividido en diferentes clases dependiendo si posee autonomía o no, que es la habilidad del sistema local de controlar sus propios destinos. Una BD homogénea, en vez de almacenar todos los datos en un sitio como una BD centralizada, son distribuidos a través de un número de sitios de la red. La Figura 1.2 muestra la arquitectura global de un DDBMS homogéneo.

Por otro lado, un *sistema heterogéneo*, se caracteriza por el uso de diferentes DBMSs en los nodos locales. La *heterogeneidad* en DDBMSs se observa en diferentes niveles del sistema, incluyendo distintos tipos de hardware, sistemas operativos y protocolos de red en diferentes sitios de la red. Del lado de la base de datos, la heterogeneidad se observa cuando existen diferentes DBMSs locales basados en el mismo modelo de datos (por ejemplo INGRES u Oracle), o basados en distintos modelos de BD (relacional, jerárquico o de red). Cabe destacar que el modelo de datos es siempre el mismo, independientemente de los tipos de DBMS o de BD utilizados. [ZANC96]

Un modelo distribuido de datos hace posible la integración de BD heterogéneas proveyendo una independencia global del DBMS respecto de esquema conceptual. Además, es posible implementar una integración tal, que reúna varios modelos de datos, representado cada uno de ellos características propias de empresas diferentes, asociadas para un trabajo conjunto. Este modelo de distribución genera las denominadas *Bases de Datos Federativas*. [LARS 95] [SHET 90]

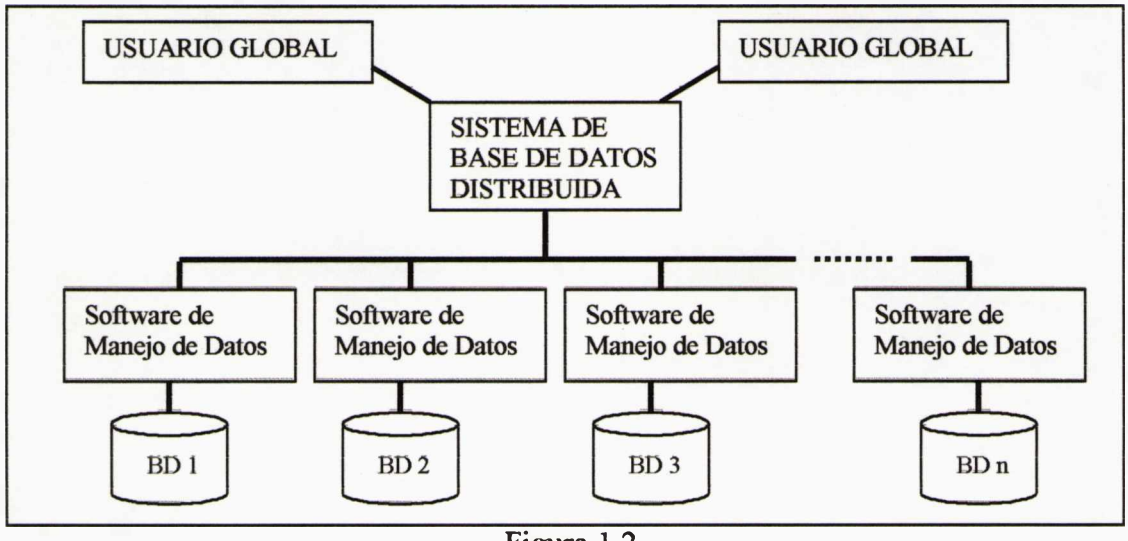


Figura 1.2

Nuevamente está bajo discusión si las BDD son la solución para los problemas que presenta actualmente el mercado. Los conceptos de downsizing y rigthsizing están siendo nuevamente revisados. La globalización existente lleva a que grandes empresas fusionen sus actividades y esto conlleve la necesidad de integrar varios sistemas funcionando, y generando lo que denominamos BDF, las cuales tienen asociados los principales inconvenientes que aparecen en los entornos distribuidos.

Con los modelos distribuidos o federativos de datos, surgen una serie de consideraciones especiales y problemas potenciales que no estaban presente en los sistemas centralizados. La replicación y fragmentación de información hace a que los datos estén más "cerca" y "disponibles" para los usuarios. Estos dos conceptos necesitan nuevos mecanismos para la preservación de la integridad y consistencia de la información que contemplen la ubicación de los datos y las copias existentes de los mismos. [DATE 94] [THOM 90]

1.3.2 Bases de Datos Distribuidas. Ventajas y desventajas

Existen varias razones para crear sistemas de bases de datos distribuidos. Entre ellas está el compartir información, la fiabilidad y disponibilidad y la agilización del procesamiento de las consultas. No obstante, estas ventajas conllevan ciertas desventajas, que incluyen el coste de desarrollo de software, el aumento en la probabilidad de errores y un mayor tiempo extra de procesamiento. La desventaja principal de los sistemas de bases de datos distribuidos es el aumento en su complejidad que se requiere para garantizar una coordinación adecuada entre los sitios en donde se encuentran distribuidos los datos.

La utilización de BDD tiene asociado una serie de beneficios y costos que deben evaluarse en el momento de tomar decisiones respecto de su utilización.[ÖZSU 91].
Algunas de las ventajas mencionables son:

- ✓ Autonomía local: cada sitio de la red en un entorno distribuido debe ser independiente del resto. Cada BD local es procesada por su propio DBMS.

- ✓ Mejoras en la performance.

- ✓ Mejoras en la disponibilidad de la información: Si se produce un fallo en una localidad en un sistema distribuido, es posible que las demás localidades puedan seguir trabajando. En particular, si los datos se repiten en varias localidades, una transacción que requiere un dato específico puede encontrarlo en más de una localidad.

- ✓ Economía: el equipamiento necesario para montar una red para distribuir la información, es comparativamente mucho menor que tener un sitio central con gran capacidad de procesamiento. Está asociado con el concepto de downsizing.

- ✓ Sistemas fácilmente expansibles.

- ✓ Compartir información.

- ✓ Agilización del procesamiento de consultas: Si una consulta comprende datos de varias localidades, puede ser posible dividir la consulta en varias subconsultas que se ejecuten en paralelo en distintas localidades.

Entre los inconvenientes y desventajas que surgen están:

- ✓ Poca experiencia en el desarrollo de sistemas y BD distribuidas

- ✓ Complejidad (sin inherentemente más complejos)

- ✓ Costo: se requiere hardware adicional, software más complejo, y el costo del canal de transmisión, para conectar los nodos distribuidos de la red.

- ✓ Distribución de control: se contradice con una de las ventajas planteadas. Este punto toma preponderancia con los problemas creados por la sincronización y coordinación de tareas.

- ✓ Seguridad: aparecen nuevos problemas, y más complejos relacionados con la seguridad de la información manipulada.

- ✓ Mayor tiempo extra de procesamiento: El intercambio de mensajes y los cálculos adicionales que se requieren para coordinar las localidades son una forma de tiempo extra que no existe en los sistemas centralizados.

Al escoger el diseño de una base de datos, el diseñador debe hacer un balance entre las ventajas y las desventajas de la distribución de los datos.

1.3.3 Reglas para los DDBMS

En 1987 Date, propuso un conjunto de reglas que un DBMS debe cumplir [BERS 92]:

- ✓ Autonomía local: Significa que todos los datos en una red distribuida son manejados localmente.

- ✓ No es necesario que exista un sitio central: Idealmente todos los sitios son igualmente "remotos", y ninguno tiene autoridad sobre otro. Cada sitio almacena su propio diccionario de datos e información de seguridad.

- ✓ Operación continua: Mientras cada sitio mantiene su propia y única identidad y control, funciona como parte de una federación unificada.

- ✓ Independencia

    - Respecto de la fragmentación de los datos: Habilidad de los usuarios finales de almacenar información lógicamente relacionada en diferentes sitios físicos. Existen dos tipos de independencia de la fragmentación: vertical y horizontal. La primera permite que diferentes tuplas de una misma tabla sean almacenadas en distintos sitios remotos. En cambio, la vertical, es la habilidad de un sistema distribuido de fragmentar la

información a través un grupo de columnas de una misma tabla, las cuales son distribuidas en diferentes lugares de la red.

- Respecto de la replicación: Significa crear copias de la base de datos en sitios remotos. Dichas copias, a veces llamadas *"snapshots"*, pueden contener la base de datos entera, o un grupo de componentes de la misma.

- Respecto de la ubicación de los datos: Los usuarios finales no tienen conocimiento, o no deben tenerlo, acerca de la ubicación física de la base de datos que están utilizando. Los datos deben ser recuperados sin ninguna referencia específica de los sitios físicos.

- Respecto del hardware, sistema operativo, red o base de datos utilizada: Significa poder manipular la BDD sin tener en cuenta la plataforma de hardware, el sistema operativo, el tipo de red o la arquitectura de la base de datos en donde la información reside.

✓ Procesamiento de consultas distribuidas: Ejecución de una consulta en más de un sitio de la BDD. En algunos DBMS, la consulta es ejecutada en el nodo que el usuario especificó, mientras que en otros, se particiona la consulta en subconsultas, ejecutando cada una de éstas en un sitio particular, dependiendo de la información que necesita obtener.

✓ Manejo de transacciones distribuidas: Habilidad que posee el sistema de manejar un Insert, Delete o Update sobre más de una base de datos desde una simple consulta. Para satisfacer éstos requerimientos, se suele utilizar el protocolo de commit de dos fases; el cual asegura que todas las bases de datos remotas han completado satisfactoriamente sus sub-actualizaciones antes que la transacción entera realice el commit. Un fallo en algunas de las BD remotas podría causar que la transacción entera falle. Sin embargo, algunas técnicas permiten commits parciales, que permiten que la transacción pueda finalizar correctamente, a pesar de los fallos.

1.4 Transacciones

Una *transacción* se define como una serie de acciones las cuales deben ser tratadas como una unidad indivisible. Es una unidad de ejecución de un programa que accede y posiblemente actualiza varios elementos de datos. Se debe exigir que las transacciones no violen ninguna restricción de integridad de la base de datos. Es decir, si la base de datos era consistente cuando comenzó la transacción, debe seguir en el mismo estado cuando culmine. La transferencia de fondos de una cuenta bancaria a otra es un ejemplo, en donde destacamos dos operaciones básicas: el débito en la cuenta origen y la acreditación en la cuenta destino. Entonces, dichas operaciones deben ser efectuadas en su totalidad o, en caso de algún fallo, la transacción no se ejecuta en absoluto. Esto significa que, ante un problema, los datos deben quedar de la misma manera que al comenzar la transacción.

Se debe tener en cuenta que la consistencia de la base de datos puede ser violada durante la ejecución de la transacción. En el ejemplo de la transferencia de fondos, si ocurriese un fallo en el período que abarca el débito de una cuenta y el crédito de la otra, la base de datos quedaría en un estado inconsistente. Esta es la tarea del *encargado de la recuperación* del DDBMS que asegura que todas las transacciones activas al producirse un fallo son *rolled back*, o *undone*. El efecto de la operación de *roollback* es volver al estado que poseía la base de datos antes de comenzar la transacción. Es decir, a través de algún método, se almacenan los cambios que se fueron llevando a cabo durante la ejecución de la misma. Por ejemplo, se podrían guardar el valor viejo y el valor nuevo de cada dato que se modifica. Por lo tanto, al ejecutar la operación de *rollback*, se vuelven a almacenar en la BD los valores viejos de cada dato que se intentó actualizar.

Las cuatro propiedades básicas de una transacción, denominadas *A.C.I.D.*, son:

- ✓ Atomicidad: a una transacción se la debe tratar como una unidad indivisible; o sea, se deben ejecutar todas las operaciones involucradas en la misma o ninguna de ellas.

- ✓ Consistencia: las transacciones transforman la base de datos de un estado consistente a otro también consistente

- ✓ Aislamiento (*Isolation*): las transacciones se ejecutan independientemente una de las otras (los efectos parciales de una transacción incompleta no son visibles para el resto de las transacciones)

- ✓ Durabilidad (también llamada persistencia): los efectos de transacciones que ya fueron completadas (cometidas) son almacenados permanentemente en la base de datos y no pueden ser desechos.

La ejecución de un programa de aplicación sobre un ambiente de base de datos puede ser vista como una serie de transacciones atómicas. Es el rol del manejador de transacciones dirigir la ejecución de las transacciones y coordinar los requerimientos a la base de datos que parten de dichas transacciones. Por otro lado, el scheduler, implementa una estrategia particular para la ejecución de transacciones. Su objetivo es maximizar la concurrencia de ejecución de transacciones sin que la ejecución de una no interfiera con otra, comprometiendo así a la integridad o consistencia de la base de datos.

1.4.1 Estados de una Transacción

En ausencia de fallos, todas las transacciones se completan con éxito. Sin embargo, como se destacó anteriormente, una transacción puede que no siempre termine su ejecución de la mejor forma. Una transacción de este tipo se la denomina *abortada*. Si se pretende asegurar la propiedad de atomicidad, una transacción abortada no debe tener efecto sobre el estado de la base de datos. Así, cualquier cambio que haya hecho la transacción abortada sobre la base de datos debe deshacerse. Una vez que se han deshecho los cambios efectuados por la transacción abortada, se dice que la transacción se ha *retrocedido* (*roolled back*).

Una transacción que termina con éxito se dice que está *cometida* (*commited*). Dicha transacción lleva a la base de datos a un nuevo estado consistente, que permanece incluso si hay un fallo en el sistema.

Se establece por tanto un modelo simple abstracto de transacción, la cual puede estar en uno de los siguientes estados, que se observan en la Figura 1.3:

- ✓ Activa, el estado inicial; la transacción permanece en este estado durante su ejecución.
- ✓ Parcialmente cometida, después de ejecutarse la última instrucción.
- ✓ Fallada, tras descubrir que no puede continuar la ejecución normal.
- ✓ Abortada, después de haber retrocedido la transacción y restablecido la base de datos a su estado anterior al comienzo de la transacción.
- ✓ Cometida, tras completarse con éxito.

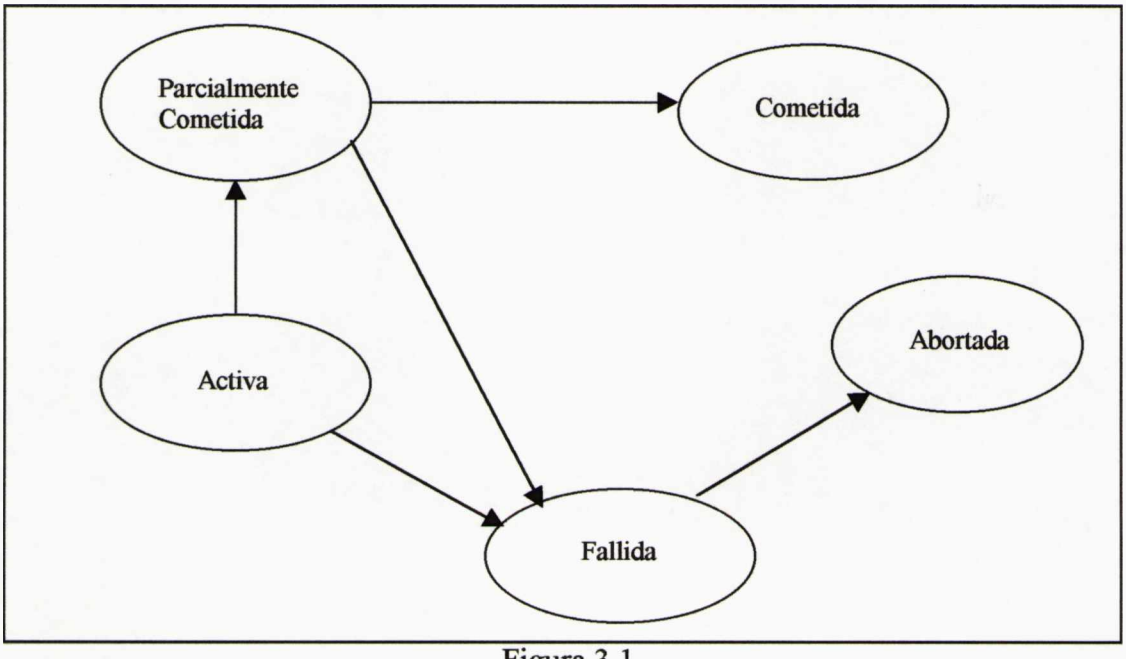


Figura 3.1

Una transacción comienza en el estado activa. Cuando acaba su última instrucción pasa al estado de parcialmente cometida. En éste punto ha terminado su ejecución, pero es posible que aún tenga que ser abortada, puesto que los datos actuales pueden estar todavía en memoria principal y puede producirse un fallo en el hardware antes de que se complete con éxito.

El sistema de base de datos escribe en disco la información suficiente para que, incluso al producirse un fallo, puedan reproducirse los cambios hechos por la transacción al reiniciar el sistema tras el fallo. Cuando se termina de escribir esta información, la transacción pasa al estado cometida.

Una transacción pasa al estado fallida luego que el sistema determine que dicha transacción no puede continuar su ejecución normal (por ejemplo, a causa de errores de hardware o lógicos). Una transacción de este tipo se debe retroceder. Después pasa al estado abortada. En este punto, el sistema tiene dos opciones, o bien reiniciar la transacción, o bien cancelarla.

Los algoritmos para asegurar la consistencia de la BD, utilizando las transacciones, incluyen: [SILB 98]

- ✓ Acciones tomadas durante el procesamiento normal de las transacciones que aseguran la existencia de información suficiente para asegurar la recuperación de fallos.

- ✓ Acciones tomadas a continuación de un fallo, para asegurar la consistencia de la BD.

Existen dos mecanismos utilizados para asegurar la atomicidad de las transacciones:

- ✓ Basado en bitácora

- ✓ Doble paginación

1.4.2 Bitácora

La estructura más ampliamente utilizada para grabar modificaciones de la base de datos es la bitácora. Cada registro de la bitácora describe una única estructura de la base de datos. Todas las operaciones que se ejecutan sobre una base de datos son registradas en la bitácora. Existe una bitácora por cada DBMS, la cual es compartida por todas las transacciones que están bajo el control del mismo DBMS. En un ambiente de BDD, cada sitio posee su propia bitácora. Se crea una nueva entrada en la bitácora local de un sitio cada vez que uno de los siguientes comandos es ejecutado por la transacción:

- ✓ begin transaction
- ✓ write (incluye inserciones, eliminaciones o actualizaciones)
- ✓ commit transaction
- ✓ abort transaction

Cabe destacar que la bitácora también es utilizada para otros propósitos como por ejemplo monitoreos y auditorías. En estos casos se maneja información adicional en la bitácora como lecturas de BD, logins de usuarios, logouts, etc.

A continuación se detalla la información que posee un registro de la bitácora, indicando para cada una en que operaciones de una transacción puede ser incluida:

- ✓ Identificador de la Transacción (para todas las operaciones);
- ✓ Tipo de registro, ésto es, cual de las acciones listadas anteriormente, sobre la base de datos, es registrada (para todas las operaciones);
- ✓ Identificador del dato afectado por la acción sobre la BD (para las operaciones de Insert, Delete o Update);

- ✓ El valor viejo del dato, esto es, el valor que poseía antes de la actualización (para las operaciones Update o Delete);

- ✓ El valor nuevo del dato, esto es, el valor que posee luego de la actualización (para las operaciones Update o Insert);

- ✓ Información de manejo del registro como por ejemplo el puntero al registro anterior de la misma transacción (para todas las operaciones).

La bitácora es vital para el proceso de recuperación y generalmente es duplicada, o sea son mantenidas dos copias de la misma bitácora. Si una de ellas se pierde o daña, la segunda copia es utilizada. Tradicionalmente estos archivos son almacenados en discos espejos o en algún otro dispositivo seguro.

Los registros de la bitácora pueden ser almacenados de forma sincrónica o asincrónica. Con la escritura *sincrónica*, cada vez que un registro es almacenado en la bitácora, se fuerza a registrarlo también en almacenamiento estable. Mientras que una escritura *asincrónica*, el buffer es almacenado de forma periódica (por ejemplo, ante el commit de la transacción). La escritura sincrónica impone un delay sobre todas las operaciones de la transacción, por lo que se considera inaceptable.

Siempre que una transacción realice una escritura, es fundamental que se cree el registro de bitácora para esa escritura antes de que se modifique la base de datos. Una vez que exista el registro de bitácora podemos sacar la modificación de la base de datos si esto es deseable. También tenemos la posibilidad de *deshacer* una modificación que ya se ha escrito en la base de datos. Esto se realiza usando el campo del valor antiguo en los registros de la bitácora.

Para que los registros de bitácora sean útiles en la recuperación de fallos del sistema y del disco, el registro debe residir en memoria estable.

1.4.3 Doble Paginación

Una alternativa a las técnicas de recuperación de caídas basadas en bitácora es la *doble paginación*. Bajo ciertas circunstancias, es posible que la doble paginación requiera menos accesos a disco que los métodos de bitácora que ya se analizaron. Sin embargo, el enfoque de bloques dobles tiene ciertas desventajas, como se explicará más adelante.

La base de datos se divide en cierto número de bloques de longitud fija, que se denominan *páginas*. El término *página* se asimila al de los sistemas operativos, ya que se utiliza un esquema de paginación para gestionar la memoria. Supongamos que existen n páginas numeradas de 1 a n. La forma de encontrar la página i-ésima de la base de datos para cualquier i dado se logra utilizando la *tabla de paginación*. Dicha tabla posee n entradas, una por cada página de la base de datos. Cada entrada posee un puntero a una página del disco.

La idea clave en que se basa la técnica de paginación es mantener *dos* tablas de paginación durante la vida de una transacción, la tabla de paginación *actual* y la tabla de paginación *doble*. Cuando comienza la transacción, las dos tablas de paginación son idénticas. La tabla de paginación doble no se modifica en ningún momento durante la transacción. La tabla de paginación actual puede cambiarse cuando la transacción realiza una operación de escritura sobre la base de datos.

El enfoque de recuperación de doble paginación consiste en almacenar la tabla de paginación en memoria no volátil de forma que el estado de la base de datos, antes de ejecutarse la transacción, pueda recuperarse en el caso de una caída o aborto de una transacción. Cuando la transacción termina, la tabla de paginación actual se escribe en memoria no volátil de acción, y se convierte en la nueva tabla de paginación doble, permitiéndose así el comienzo de una nueva transacción. Es importante que la tabla de paginación doble se almacene en memoria no volátil, ya que es la única forma de localizar las páginas de la base de datos. La tabla de paginación actual puede conservarse en memoria principal (almacenamiento volátil). No es de importancia si la tabla de paginación actual se pierde en una caída, puesto que el sistema se recupera utilizando la tabla de paginación doble.

Por la forma en que se definieron las operaciones de escritura, se puede garantizar que la tabla de paginación doble apunta a las páginas de la base de datos que corresponden al estado anterior a cualquier transacción que estuviera activa en el momento de la caída.

La doble paginación ofrece varias ventajas con respecto a las técnicas basadas en bitácora. Se elimina el tiempo extra requerido para grabar registros, y la recuperación de las caídas es bastante más rápidas. Sin embargo, al técnica de doble paginación también tiene desventajas:

- ✓ Fragmentación de los datos: como la técnica de doble paginación hace que las páginas de la base de datos cambien de posición cuando se actualizan, se pierde la propiedad de localización de las páginas

- ✓ Recolección de basura: cada vez que se comete una transacción, las páginas de la base de datos que contiene la versión anterior de los datos que la transacción modificó se vuelven inaccesibles. Tales páginas se consideran *basura*, ya que no son parte del espacio libre y no contienen información útil. Periódicamente es necesario localizar todas las páginas de basura y añadirlas a la lista de páginas libres.

1.4.4 Actualización de la Base de Datos

La mayoría de los DBMSs utilizan la denominada *modificación inmediata* que escribe directamente en la base de datos a través del buffer. La ventaja es que las páginas actualizadas ya se encuentran en su lugar en el momento en que la transacción realiza el commit y ninguna operación adicional es requerida. Por otro lado sufre la desventaja que, ante un fallo en la transacción, las actualizaciones deben ser desechas.

La técnica de actualización inmediata permite que las modificaciones de la base de datos se graben mientras la transacción está todavía en estado activo. Las modificaciones de datos que escriben las transacciones activas se denominan *modificaciones no cometidas*. En el caso de que ocurra una caída o un fallo de una

transacción, el campo del valor antiguo de los registros de bitácora debe utilizarse para restaurar los datos modificados al valor que tenían antes de que comenzara la transacción.

Para contrarrestar esto, existe la *modificación diferida*. Con este método las actualizaciones son almacenadas en un lugar separado de la BD en almacenamiento estable y los índices de la BD no son actualizados hasta el momento del commit de la transacción. Las versiones viejas de las páginas son utilizadas por el proceso de recuperación y efectivamente formarán parte de la bitácora.

Esta técnica garantiza la atomicidad de la transacción grabando todas las modificaciones de la base de datos en la bitácora, pero aplazando la ejecución de todas las operaciones de escritura hasta que la transacción se comente parcialmente.

Cuando una transacción está parcialmente cometida, la información en la bitácora asociada a la transacción se usa en la ejecución de las escrituras diferidas. Si el sistema se cae antes de que la transacción termine su ejecución, o si la transacción aborta, entonces simplemente se ignora la información en la bitácora.

Cabe destacar que la técnica de modificación diferida solamente requiere el nuevo valor del dato. Así pues, podemos simplificar la estructura general de la bitácora que se explicó anteriormente, omitiendo el campo del valor antiguo.

1.4.5 Casos de Fallo

Existen diferentes casos de fallo que se pueden clasificar de la siguiente manera:

- ✓ Fallos en la transacción
- ✓ Fallo de un medio

1.4.5.1 Fallo en una transacción

Un fallo en una transacción individual puede ser producido por las siguientes causas:

- ✓ La Transacción aborta: esto se produce cuando, por alguna operación o resultado erróneo, la aplicación decide abortar la transacción. El encargado de la recuperación hará explícitamente un "rollback" de la misma. Ninguna otra transacción del sistema es afectada.

- ✓ Fallo en la Transacción no forzado: se produce ante un bug en la aplicación (como por ejemplo un error aritmético como una división por cero). En este caso, el sistema detecta que la transacción ha fallado e informa al encargado de la recuperación para que realice el "rollback". Aquí también ninguna otra transacción es afectada.

- ✓ Fallo inducido por el sistema: Ocurre, por ejemplo, cuando el manejador de las transacciones explícitamente aborta una transacción porque existe un conflicto con otra, o ha ingresado en deadlock. Nuevamente el encargado de la recuperación es explícitamente invocado para que realice el "rollback" de la transacción.

En un DBMS centralizado el fallo en una transacción local involucrará el volver atrás cualquier cambio realizado sobre la base de datos utilizando la bitácora.

1.4.5.2 Fallo de un medio

Un fallo de un medio se produce cuando alguna porción de la base de datos ha sido corrompida. La principal causa de este tipo de fallos es la rotura de la cabeza del disco. El objetivo del encargado de la recuperación es obtener los últimos valores cometidos de todos los objetos de datos. Existen dos posibles soluciones: *archivar* o *espejar*.

Es una práctica común en todos los procesos de instalación realizar periódicos backups en almacenamiento estable. Estos backups forman el *archivo* de la base de datos. Tradicionalmente las bases de datos fueron archivadas en cintas magnéticas porque era un medio barato y fácil de utilizar. La mayoría de las instalaciones actuales son enviadas a discos ópticos ya que son una mejor y más eficiente alternativa.

Es deseable que la copia del backup se realice cuando el sistema se encuentra ocioso; de otra forma el backup podría contener actualizaciones parciales que complicarían la recuperación.

Para realizar la recuperación a través del archivo, se carga la base de datos con la copia más reciente del archivo y luego se graba la información de todas las transacciones cometidas utilizando la bitácora.

En ambientes que no pueden detenerse, en donde la operación de tolerancia a fallos es importante, se utiliza una técnica denominada *espejo*. Esto permite que dos copias completas de la base de datos sean mantenidas on-line bajo diferentes dispositivos de almacenamiento estable. Las lecturas son realizadas sobre algunos de los discos, mientras que las escrituras deben hacerse sobre ambos dispositivos. Si uno de los discos llegase a fallar, entonces todas las lecturas y escrituras son efectuadas directamente sobre la copia espejo y el sistema seguiría trabajando sin ninguna interrupción. Una vez que la falla ha sido reparada, las actualizaciones que se realizaron deben ser pasadas al disco que acaba de repararse.

CAPITULO 2

Bases de Datos Distribuidas. Integridad de datos

2.1 Transacciones distribuidas

En un ambiente de base de datos distribuida, una transacción puede acceder a datos que están almacenados en más de un sitio de la red. Cada transacción es dividida en una serie de *sub-transacciones*, una por cada sitio en donde existan datos que la transacción original necesita procesar. Estas *sub-transacciones* son implementadas por *agentes* en cada uno de los sitios que sea necesario.

En cuanto al control de transacciones distribuidas, existen una serie de pasos que son requeridos al procesar una transacción global:

- ✓ Una transacción global es iniciada en un sitio A a través del manejador global de transacciones (GTM^A)
- ✓ Utilizando la información acerca de la ubicación de los datos (a través de un catálogo o diccionario de datos), el GTM^A divide la transacción global en una serie de agentes para cada sitio relevante.
- ✓ El manejador de comunicación global (GCM^A) en A envía esos agentes a los sitios correspondientes a través de la red.
- ✓ Una vez que todos los agentes han completado su trabajo, el resultado es comunicado al sitio A a través de todos los GCMs.

Cabe destacar que, normalmente, los agentes no se comunican directamente con los otros pertenecientes a la misma transacción global, sino que la comunicación se realiza mediante un *coordinador*.

Se ha especificado un escenario simplificado, ya que se ha ignorado muchos matices como por ejemplo la descomposición de una consulta o la optimización de la misma. Cada agente de la transacción global es, por sí mismo, una transacción atómica, desde el punto de vista del *manejador local de transacciones* (LTM), en el lugar en donde es ejecutada; la cual culminará con la operación de commit o roolback (abort). Por otro lado, la transacción global entera es también una unidad atómica, pero deberá esperar a que se completen el trabajo todos los agentes, antes de tomar la decisión de commit o rollback. Por lo tanto, cada agente representa solo una transacción *parcial* y no puede hacerse visible al resto de las transacciones o agentes. Una vez que los agentes han completado su trabajo, ingresa en un estado de espera y envía un mensaje al GMT, indicando que está en condiciones de cometer, o en el caso de un fallo, de abortar la sub-transacción. Luego, el GMT analiza todas las respuestas de cada uno de los agentes y entonces decide el destino final de la transacción global.

Un sistema de base de datos distribuidos está formado por un conjunto de localidades, cada una de las cuales mantiene un sistema de base de datos local. Una localidad puede procesar *transacciones locales*, es decir, aquellas transacciones que utilizan sólo datos de esa localidad. Además, una localidad puede participar en la ejecución de *transacciones globales*, las cuales utilizan datos de diferentes localidades. La ejecución de transacciones globales requiere comunicación entre localidades.

Para garantizar la atomicidad es necesario que todas las localidades donde se ejecutó una transacción T coincidan en el resultado final de la ejecución, o sea, T debe ser ejecutada o abortada en todas las localidades.

Los sistemas de bases de datos distribuidas implementan un *protocolo de commit* sobre transacciones para asegurar la atomicidad de las mismas. En las últimas dos décadas, investigadores de bases de datos han propuesto una gran variedad de protocolos. Entre los más importantes incluimos al *protocolo de dos fases (2PC)*, dos variaciones de éste llamadas *protocolo presumed commit y presumed abort*, y por

último al *protocolo de tres fases*. Según la funcionalidad de cada uno, los protocolos de commit requieren intercambiar mensajes, a través de distintas fases, entre sitios participantes donde la transacción distribuida es ejecutada. Por lo tanto, son generados varios registros en memoria estable, alguno de los cuales son grabados a disco de manera sincrónica. A causa de estos costos adicionales, el procesamiento de commit puede resultar en un incremento del tiempo de ejecución de la transacción, haciendo que la elección del protocolo de commit sea una importantísima decisión de diseño para sistemas de bases de datos distribuidas.

2.2 Fallos distribuidos

En un sistema distribuido pueden encontrarse los mismos fallos que en un sistema centralizado. En forma adicional existen otros tipos de fallos a los que se debe enfrentar una configuración distribuida, e incluyen el fallo de una localidad, la interrupción de una línea de comunicación, la pérdida de mensajes y la fragmentación de la red.

2.2.1 Fallo de un sitio

Los fallos en sitios se producen como resultado de un fallo de la CPU local o una rotura del sistema. Todas las transacciones de esa máquina serán afectadas. El contenido de la memoria principal (volátil), incluyendo todos los buffers, se pierden. No obstante, se asume que tanto la Base de Datos bajo almacenamiento persistente, como la bitácora no son dañados.

En un ambiente distribuido, desde que los sitios operan independientemente uno del otro, es perfectamente posible que algunos sitios se mantengan operacionales mientras otro se encuentre en fallo. Si todos los sitios del ambiente distribuido se encontraran "en Fallo", estaríamos frente a un *Fallo Total*. Si solo algunos de ellos fallaran, sería un *Fallo Parcial*. La principal dificultad del fallo parcial son los sitios que deben trabajar para conocer el estado de otros (por ejemplo, si están "en Fallo" u

"Operacional"). Como resultado de un fallo parcial es posible que algunos sitios queden *bloqueados* y por tanto inhabilitados de trabajar. Esto puede ocurrir cuando un sitio que ejecuta un agente de una transacción global falla en medio de la misma. Otros agentes participantes de dicha transacción no tendrán certeza de sí deben cometer o abortar. También es importante destacar que una de las principales ventajas del manejo de datos de forma distribuida, es que los mismos continuarán disponibles a los usuarios a pesar de algún fallo local.

Cuando un sitio se debe recuperar seguido de un fallo, el control es pasado al encargado de la recuperación para ejecutar el procedimiento de *recuperación* o de *recomienzo*. Durante estos procedimientos, ninguna otra transacción es aceptada por el DBMS hasta que la Base de Datos haya sido reparada.

2.2.2 Fallo de la red

Los DDBMSs dependen de la habilidad de todos los sitios de la red de estar disponibles y comunicados entre sí de forma segura, para que las operaciones se realicen de forma correcta. La mayoría de las redes actuales son muy seguras. Los protocolos existentes garantizan una correcta transmisión y orden de los paquetes de mensajes enviados. Muchas redes actuales soportan el reruteo automático de mensajes, que se utiliza ante un fallo de la red. Sin embargo, los errores en la comunicación aún siguen ocurriendo, y producen que las redes se *particionen* en dos o más subredes. Los sitios que pertenecen a la misma partición pueden comunicarse entre sí, pero dos sitios que son de distintas particiones no lo pueden hacer.

Supongamos disponer de dos sitios C y D, los cuales pertenecen a distintas particiones (a partir de un fallo de la red) y a una misma transacción global. Puede darse el caso que en partición donde pertenece C se halla decidido "cometer" la transacción mientras que en la partición de E se pretenda "abortar" la misma. Estas decisiones violarían la propiedad de atomicidad de una transacción.

En general, no es posible diseñar un protocolo de commit atómico *no bloqueante* para redes particionadas arbitrariamente. Con un protocolo *no bloqueante* los sitios no

son bloqueados a pesar de algún fallo. Esto significa que serán capaces de seguir su procesamiento sin tener que esperar la recuperación del sitio que falló.

2.3 Protocolos de Commit Distribuidos

Un modelo común para una transacción distribuida se centra en un proceso, llamado *coordinador*, que se ejecuta en el sitio en donde la transacción se crea, y un conjunto de procesos, llamados *localidades*, que se ejecutan en distintos sitios que deben ser accedidos por la transacción.

Para garantizar la atomicidad, es preciso que en todas las localidades en las que se haya ejecutado la transacción distribuida *T* coincidan en el resultado final de la ejecución. *T* debe quedar ejecutada o abortada en todas las *localidades*. Para garantizar esta propiedad, el *coordinador* de transacciones encargado de *T* debe ejecutar un protocolo de commit.

Entre los más comunes y más ampliamente usados se encuentra el *protocolo de dos fases (2PC)*. Existen otras alternativas que derivan del anterior protocolo, en las que podemos nombrar al *protocolo de tres fases (3PC)*, el *protocolo presumed abort, el protocolo presumed commit*, el *protocolo de una fase (1PC)* etc.

El principal requerimiento de los Protocolos de Commit Atómicos es mantener la *atomicidad* de las transacciones distribuidas. Esto significa que, como las transacciones distribuidas se ejecutan en varios sitios de la red, alguno de ellos puede llegar a fallar, pero el efecto de la transacción debe ser "todo o nada", o sea se ejecuta en todos los sitios en que estaba involucrado o en ninguno. Esto se denomina *commit atómico*.

Un protocolo es *no bloqueante* si permite que una transacción culmine en el sitio operacional sin tener que esperar la recuperación de un sitio que falló. Esto afectará notoriamente a la performance de la ejecución de la transacción. Por lo tanto también es necesario que los protocolos de recuperación distribuidos sean *independientes*. Estos protocolos determinan como será la terminación de una transacción que fue ejecutada

en el momento de un fallo de algún sitio sin tener que consultar a ningún otro sitio de la red. La existencia de dichos protocolos reducen el número de mensajes necesarios en el momento de la recuperación.

2.3.1 Protocolo de Dos Fases (2PC)

Como su nombre lo indica, 2PC opera bajo dos fases: una *fase de voto* y una *fase de decisión*. La idea básica es que todos los participantes de la transacción global son consultados por el coordinador acerca de sí están preparados o no para realizar el commit de la transacción. Si alguno de ellos vota "Abort", o falla al producir la respuesta en un período de tiempo especificado, entonces el coordinador instruirá a todos los participantes para que aborten la transacción. El protocolo asume que cada sitio posee su propia bitácora y por lo tanto podrán realizar el commit o rollback de la transacción de forma segura.

Las reglas de "voto" son las siguientes:

- ✓ Cada participante tiene un voto, el cual puede ser "Commit" o "Abort"
- ✓ Luego de haber votado, ningún participante tendrá la chance de volverlo a hacer
- ✓ Si un participante vota "Abort", entonces estará libre de abortar la transacción inmediatamente. En realidad cualquier sitio estará libre de abortar la transacción el cualquier momento hasta recibir el voto de "Commit"
- ✓ Si un participante vota "Commit", deberá esperar hasta que el coordinador realice un "broadcast" del mensaje "Commit" o "Abort"
- ✓ Si todos los participantes votan "Commit", entonces la decisión global del coordinador deberá ser "Commit"

- ✓ La decisión global debe ser tomada por todos los participantes de la transacción

Sea T una transacción que se inició en la localidad L_i, y sea C_i el coordinador de transacciones de esa localidad.

Cuando T termina de ejecutarse, es decir, cuando todas las localidades en las que se ejecutó T informan a C_i que llegó a su término, C_i inicia el protocolo de commit de dos fases.

- ✓ *Fase 1.* C_i añade el registro <**prepare** T> a la bitácora y la graba en memoria estable. Una vez hecho esto envía un mensaje de **prepare** T a todas las localidades en las que se ejecutó T. Al recibir el mensaje, el gestor de transacciones de cada una de esas localidades determina si está dispuesto a ejecutar la parte de T que le correspondió. Si no está dispuesto, éste añade un registro <**no** T> a la bitácora y a continuación enviartá un mensaje **abort** T a C_i. Si la respuesta es afirmativa agregará un registro <T **ready**> a la bitácora y grabará todos los registros de bitácora que corresponden a T en memoria estable. Una vez hecho ésto, responderá a C_i con el mensaje T **ready**.

- ✓ *Fase 2.* Una vez que todas las localidades hayan respondido al mensaje **prepare** T enviado por C_i, o después de un intervalo de tiempo, previamente especificado, C_i puede determinar si la transacción T puede ejecutarse o abortarse. Si C_i recibió el mensaje T **ready** de todas las localidades que participan, la transacción T puede ejecutarse. En caso contrario, la transacción T debe abortarse. Según haya sido el veredicto, se agregará un registro <**execute** T> o <**abort** T> a la bitácora y se grabará en memoria estable. En este punto, el destino de la transacción se habrá sellado. A continuación, el coordinador enviará un mensaje <**execute** T> o <**abort** T> a todas las localidades participantes. Al recibir este mensaje, cada una de las localidades lo registra en la bitácora.

Una de las localidades en las que se ejecutó T puede abortar T incondicionalmente en cualquier momento antes de enviar el mensaje T **ready** al

coordinador. De hecho, el mensaje *T* **ready** es una promesa que hace la localidad de cumplir con la orden del coordinador de ejecutar *T* o abortar *T*. Una localidad sólo puede hacer tal promesa si la información requerida está ya en almacenamiento estable. En caso contrario, si la localidad se cayera después de enviar el mensaje *T* **ready**, no sería capaz de cumplir su promesa.

El destino de una transacción está sellado en el momento en que por lo menos una de las localidades responde **abort** *T*, ya que se requiere unanimidad para ejecutar una transacción. Dado que la localidad coordinadora L_i es una de las localidades en las que se ejecuta *T*, el coordinador puede decidir de forma unilateral abortar *T*. El veredicto final en lo que a *T* concierne queda determinado en el momento en que el coordinador lo registra (ejecutar o abortar) en la bitácora y graba ésta en memoria estable. En algunas implementaciones del protocolo de commit de dos fases, una localidad envía un mensaje de reconocimiento a *T* (**ack** *T*) al coordinador al final de la segunda fase del protocolo. Una vez que el coordinador recibe este mensaje de todas las localidades, añade el registro <*T* **finish**> en la bitácora.

2.3.1.1 Manejo de Fallos

A continuación veremos de manera detallada la forma en que el protocolo de dos fases responde a los distintos tipos de fallos.

- ✓ *Fallo de una localidad participante.* En el momento en que una localidad participante L_k se recupera de un fallo, debe examinar su bitácora para determinar el destino de aquellas transacciones que se estaban ejecutando cuando se produjo el fallo. Sea *T* una de esas transacciones. Consideramos los siguientes casos:
  - La bitácora contiene un registro <**execute** *T*>. En este caso, la localidad ejecuta **redo**(*T*).
  - La bitácora contiene un registro <**abort** *T*>. En tal caso, la localidad ejecuta **undo**(*T*).
  - La localidad contiene un registro <*T* **ready**>. En este caso, la localidad debe consultar con C_i, para determinar el destino de *T*. Si C_i

está activo, notificará a L_k de la ejecución o aborto de T. A continuación, L_k ejecutará **redo**(T) o **undo**(T), según sea el caso. Si C_i está inactivo, L_k debe interrogar a las demás localidades para intentar determinar el destino de T. Esto lo hará enviando un mensaje de **consulta de estado** de T a todas las demás localidades del sistema. Al recibir uno de estos mensajes una localidad, deberá consultar su bitácora para ver si T se realizó allí y, en caso de que así haya sido, si se ejecutó o abortó. A continuación notificará dicho resultado a L_k. Si ninguna de las localidades cuenta con la información apropiada (es decir, si T se ejecutó o abortó), entonces L_k no podrá ejecutar ni abortar T. La decisión con respecto a T se pospondrá hasta que L_k pueda obtener la información necesaria. De esta forma, L_k deberá seguir mandando el mensaje de consulta a las otras localidades de forma periódica hasta que alguna de las localidades que contienen la información requerida se recupere. Se puede hacer mención de que la localidad en la que reside C_i siempre tendrá la información requerida.

- La bitácora no contiene registros de control (**abort, execute, ready**) referentes a T. Esto implica que L_k tuvo un fallo antes de responder al mensaje **prepare** T de C_i. Puesto que el fallo de L_k excluye el envío de esa respuesta, de acuerdo a nuestro algoritmo, C_i debe abortar T. En consecuencia, L_k debe ejecutar **undo**(T).

✓ *Fallo del coordinador.* Si el coordinador falla en la mitad de la ejecución del protocolo de commit de la transacción T, entonces las localidades participantes deben decidir el destino de T. Veremos que en algunos casos las localidades participantes no pueden decidir si ejecutar o abortar T, y, por lo tanto, será necesario que estas localidades esperen a que se recupere el coordinador que falló.

- Si una localidad activa contiene un registro <**execute** T> en su bitácora, entonces T debe ser ejecutada.
- Si una localidad activa contiene un registro <**abort** T> en su bitácora, entonces T debe ser abortada.

- Si alguna localidad activa *no* contiene un registro <T **ready**> en su bitácora, entonces el coordinador que falló C_i no puede haber decidido ejecutar T. Esto se debe a que una localidad que no tiene un registro <T **ready**> en su bitácora no puede haber enviado un mensaje de T **ready** a C_i. Sin embargo, el coordinador puede haber decidido abortar T, y no ejecutar T. Es preferible abortar T antes de esperar a que se recupere el coordinador.
- Si no ocurre ninguno de los casos anteriores, entonces todas las localidades activas deben tener en sus bitácoras un registro <T **ready**>, y ningún registro de control adicional (tales como <**abort** T> o <**execute** T>). Puesto que el coordinador ha fallado, es imposible determinar hasta que el coordinador se recupere si se produjo una decisión, y si la hubo, cuál fue. De esta forma, la localidad activa debe esperar a que C_i se recupere. Debido a que el destino de T continúa en duda, T puede continuar reteniendo recursos del sistema. Por ejemplo, si se utilizan locks, T puede mantener locks sobre los datos en las localidades activas. Esta situación no es deseable puesto que pueden pasar horas o días hasta que C_i esté activo de nuevo. Durante ese tiempo, otras transacciones pueden haber sido forzadas a esperar a T. Como resultado tenemos que los datos no están disponibles no solo en la localidad que falló, sino también en las localidades activas. Cuanto mayor es el tiempo que C_i esté caído, mayor será el número de datos que no estén disponibles. A esta situación se la denomina el problema de *bloqueo* debido a que T está bloqueada, dependiendo de la recuperación del coordinador.

✓ *Fallo de una línea de comunicación.* Cuando una línea de comunicación falla, todos los mensajes que estaban siendo enrutados a través de la línea no llegan a sus destinos intactos. Desde el punto de vista de las localidades conectadas a través de esa línea, parece que el fallo corresponde a las otras localidades. Por tanto, nuestro anterior esquema también se puede aplicar aquí.

- ✓ *Fragmentación de la red.* Cuando se fragmenta la red, caben dos posibilidades.

  - El coordinador y todos sus participantes quedan en un fragmento. En este caso el fallo no tendrá efecto sobre el protocolo de commit.
  - El coordinador y sus participantes quedan distribuidos en varios fragmentos. En este caso se perderán los mensajes entre la participante y el coordinador, situación que es equivalente a la interrupción de líneas de comunicación mencionadas anteriormente.

2.3.2 Protocolo Presumed Abort

Una variante del protocolo de dos fases (2PC), llamado *protocolo presumed abort*, trata de reducir el número de mensajes que se intercambian entre los participantes de la ejecución de una transacción distribuida, utilizando la regla: "en caso de duda, se aborta la transacción". Esto es, si luego de un fallo, un sitio consulta al *coordinador* por el estado de la transacción y no recibe información del mismo, se asume que la transacción abortó. Con esta suposición, no será necesario que las localidades:

- ✓ envíen los mensajes de reconocimiento (ACK) al *coordinador* por los mensajes de ABORT del mismo

- ✓ escriban el registro de ABORT en la bitácora. Y también no será necesario que el *coordinador* escriba el registro de ABORT ni el de FINISH en la bitácora para una transacción que abortó.

En resumen, el protocolo presumed abort es idéntico al protocolo de dos fases para transacciones que terminan cometiendo, pero reduce el número de mensajes y el overhead de almacenamiento en bitácora, para transacciones que terminan abortando.

2.3.3 Protocolo Presumed Commit

Otra variante del protocolo de dos fases se basa en la observación que, en general, el número de transacciones que cometen es mayor al número que abortan. En este protocolo, llamado *protocolo presumed commit*, el overhead se reduce para las transacciones que cometen, en vez de para las que abortan, requiriendo a todos los participantes de la transacción distribuida que cumplan la regla que indica "en caso de duda, cometan". En este esquema, las *localidades* no envían el reconocimiento (ACK) en la decisión global de commit, y no escriben el registro de COMMIT en la bitácora. Por lo tanto, el *coordinador* tampoco escribe el registro FINISH en memoria estable.

Pero este protocolo no actúa como el inverso del anterior. Consideremos el siguiente escenario: el coordinador envía el mensaje "Preparar" a cada una de las localidades y comienza la etapa de recolección de información, pero falla antes de recoger todas las respuestas y tomar la decisión. Por otro lado, las localidades se encuentran esperando por la decisión del coordinador; y al no recibir respuesta deciden "Cometer" de acuerdo al protocolo Presumed Commit. En cambio el coordinador decidirá "Abortar" cuando se recupere produciendo así una *inconsistencia*. Para solucionar este problema el coordinador, previo al envío del mensaje "Preparar" a las localidades, escribe en memoria estable un registro de "Collecting", que contiene los nombres de todas las localidades participantes de la ejecución de la transacción. Por lo tanto, las localidades se verán obligadas a recibir la decisión del coordinador, sea "Commit" o "Abort". En el caso de que sea "Commit", culmina el protocolo porque el coordinador no esperará ningún "Reconocimiento". Si el coordinador cae con el registro de "Collecting" activo, deberá verificar que hicieron el resto de las localidades.

2.3.4 Protocolo de Tres Fases (3PC)

El problema fundamental con los protocolos descriptos anteriormente es que las *localidades* podrían *bloquearse* en el caso de un fallo hasta que el sitio que falló se recupere. Por ejemplo, si el *coordinador* falla luego de iniciar el protocolo pero antes de comunicar su decisión a cada *localidad*, éstas se bloquearán hasta que el *coordinador* se recupere y les informe su decisión. Durante el tiempo en que las *localidades* están bloqueadas, continúan manteniendo recursos del sistema, como por ejemplo *locks* sobre algunos ítems de datos, haciendo que no estén disponibles para otras transacciones.

Para atacar el problema de bloqueo, fue propuesto el *protocolo de tres fases (3PC)*. Activa la capacidad de no bloquearse agregando una nueva fase: "precommit" entre las dos fases definidas en el protocolo de dos fases. Aquí se obtiene una decisión preliminar, que será comunicada a todos las *localidades* participantes de la transacción, permitiendo que la decisión global del destino de la transacción se genere independientemente de un posible fallo del *coordinador*. Cabe destacar, que el precio de obtener la funcionalidad de no bloquearse, es la existencia de un número mayor de mensajes que se intercambiarán entre el *coordinador* y las *localidades*, ya que existe una fase más. Por lo tanto, también se necesitará escribir registros adicionales en memoria estable, durante la fase de "precommit".

El protocolo de tres fases requiere que:

- ✓ No pueda ocurrir una fragmentación de la red.
- ✓ Debe haber al menos una localidad funcionando en cualquier punto.
- ✓ En cualquier punto, como máximo un numero K de participantes pueden caer simultáneamente (siendo K un parámetro que indica la resistencia del protocolo a fallos en localidades).

El protocolo alcanza esta propiedad de no-bloqueo añadiendo una fase extra, en la cual se toma una decisión preliminar sobre el destino de T. Como resultado de esta

decisión, se pone en conocimiento de las localidades participantes cierta información, que permite tomar una decisión a pesar del fallo del coordinador.

De la misma forma, sea T una transacción iniciada en la localidad L_i y C_i el coordinador de transacciones en L_i.

- ✓ *Fase 1.* Esta fase es idéntica a la fase 1 del protocolo de commit de dos fases.

- ✓ *Fase 2.* Si C_i recibe el mensaje **abort** T de una localidad participante, o si C_i no recibe respuesta dentro de un intervalo previamente especificado de una localidad participante, entonces C_i decide abortar T. La decisión de abortar está implementada de la misma forma que en protocolo de commit de dos fases. Si C_i recibe un mensaje T **ready** de cada localidad participante, tomará la decisión de <<**preexecute**>> T. La diferencia entre **preexecute** y **execute** radica en que T puede ser todavía abortado eventualmente. La decisión de **preexecute** permite al coordinador informar a cada localidad participante que todas las localidades participantes están <<**ready**>>. C_i añade un registro <**preexecute** T> a la bitácora y lo graba en un almacenamiento estable. De acuerdo a ésto, C_i envía un mensaje **preexecute** T a todas las localidades participantes. Cuando una localidad recibe un mensaje del coordinador (ya sea **abort** o **preexecute** T), lo registra en su bitácora, grabando esta información en almacenamiento estable, y envía un mensaje de reconocimiento (**ack** T) al coordinador.

- ✓ *Fase 3.* Esta fase se ejecuta sólo si la decisión en la Fase 2 fue de **preexecute**. Después de que los mensajes **preexecute** T se han enviado a todas las localidades participantes, el coordinador debe esperar hasta que reciba al menos un número K de mensajes de reconocimiento a (**ack** T). Siguiendo este proceso, el coordinador toma una decisión de commit. Añade un registro <**execute** T> en su bitácora y la graba en un almacenamiento estable. De acuerdo a ésto, C_i envía un mensaje **execute** T a todas las localidades participantes. Cuando una localidad recibe el mensaje, lo registra en su bitácora.

Tal y como en el protocolo de commit de dos fases, una localidad en la que se halla ejecutado T puede abortar T incondicionalmente en cualquier momento, antes de

enviar el mensaje *T* lista al coordinador. El mensaje *T* **ready** es, de hecho, una promesa hecha por la localidad para seguir la orden del coordinador de ejecutar *T* o abortar *T*. En contraste con el protocolo de commit de dos fases, en el que el coordinador puede incondicionalmente abortar *T* en cualquier momento antes de enviar el mensaje ejecutar *T*, el mensaje **preexecute** *T* en el protocolo de commit de tres fases, es una promesa hecha por el coordinador para seguir la orden del participante de ejecutar *T*.

Puesto que la fase 3 siempre induce a una decisión de commit, parece no ser muy útil. El papel de la tercera fase toma importancia por la forma en que el protocolo de commit de tres fases maneja los fallos.

En algunas implementaciones del protocolo de commit de tres fases, al recibir una localidad el mensaje **execute** *T*, ésta envía un mensaje de **receive** *T* al coordinador. (Observamos el uso de **receive** para distinguirlo de los mensajes de reconocimiento a (**ack** *T*) usados en la Fase 2.) Cuando el coordinador recibe el mensaje de **receive** *T* de todas las localidades, añade el registro <*T* **finish**> a la bitácora.

2.3.4.1 Manejo de fallos

A continuación veremos de manera detallada la forma en que el commit de tres fases responde a los distintos tipos de fallos.

- ✓ *Fallo de una localidad participante.* En el momento en que una localidad participante L_k se recupera de un fallo, debe examinar su bitácora para determinar el destino de aquellas transacciones que se estaban ejecutando cuando se produjo el fallo. Sea *T* una de esas transacciones. Consideremos los siguientes casos:
  - La bitácora contiene un registro <**execute** *T*>. En este caso, la localidad ejecuta **redo**(*T*).
  - La bitácora contiene un registro <**abort** *T*>. En este caso, la localidad ejecuta **undo**(*T*).
  - La bitácora contiene un registro <*T* **ready**> y ningún <**abort** *T*> o <**execute** *T*>. En este caso, la localidad debe consultar con C_i para

determinar el destino de *T*. Si C_i responde con un mensaje *T* abortada, la localidad ejecutará **undo**(*T*). Si C_i responde con un mensaje **preexecute** *T*, la localidad (como en la fase 2), lo registra en su bitácora y continúa el protocolo enviando un mensaje de reconocimiento a (**ack** *T*) al coordinador. Si C_i responde con un mensaje que se ha ejecutado *T*, la localidad ejecutará **redo**(*T*). En el caso de que C_i falle al responder dentro de un intervalo previamente especificado, la localidad ejecutará un protocolo de fallo del coordinador (que se describe a continuación).

- ✓ *Fallo del coordinador.* Si por cualquier razón una localidad participante falla al recibir una respuesta del coordinador, ésta ejecuta el protocolo coordinador de fallos. Este protocolo resulta en la selección de un nuevo coordinador. Cuando el coordinador que falló se recupera, toma el papel de una localidad participante. No actuará como coordinador por más tiempo. Es más, deberá definir la decisión que haya sido tomada por el nuevo coordinador.

2.3.4.2 Protocolo de fallo del coordinador

El protocolo de fallo del coordinador es provocado por una localidad participante que falla en la recepción de una respuesta del coordinador dentro de un intervalo previamente especificado. Puesto que consideramos que no hay fragmentación en la red, la única causa posible para esta situación es un fallo del coordinador.

- ✓ Las localidades participantes activas eligen un nuevo coordinador utilizando un protocolo de selección (Ver Selección del Coordinador).
- ✓ El nuevo coordinador, C_{nuevo}, envía un mensaje a cada localidad participante pidiendo el estado local de *T*.
- ✓ Cada localidad participante, incluyendo C_{nuevo}, determina el estado local de *T*:
  - *ejecutada.* La bitácora contiene un registro <**execute** *T*>.
  - *abortada.* La bitácora contiene un registro <**abort** *T*>.

- *lista.* La bitácora contiene un registro <*T* **ready**> y ningún registro <**abort** *T*> o <**preexecute** *T*>.
- *preejecutada.* La bitácora contiene un registro <*T* **preexecute**> y ningún registro <**abort** *T*> o <**execute** *T*>.
- *no-lista.* La bitácora no contiene ningún registro <*T* **ready**> ni ningún registro <**abort** *T*>.

✓ Dependiendo de la respuesta recibida, C_{nuevo} decide si ejecutar o abortar *T* o reiniciar el protocolo de commit de tres fases:

- Si al menos una localidad tiene el estado local = **ejecutada**, entonces se ejecuta *T*.
- Si al menos una localidad tiene el estado local = **abortada**, entonces se aborta *T*. (Observamos que no es posible que algunas localidades tengan el estado local = **ejecutada** mientras otras tengan el estado local = **abortada.**)
- Si ninguna localidad tiene el estado local = **abortada**, y ninguna localidad tiene el estado = **ejecutada** pero al menos hay una localidad con el estado = **preejecutada**, entonces C_{nuevo} reanuda el protocolo de commit de tres fases enviando mensajes **preexecute** nuevos.
- En los demás casos, abortar *T*.

El protocolo de fallos del coordinador permite al nuevo coordinador tener conocimiento del estado del coordinador que falló, C_i.

Si alguna localidad tiene un <**execute** *T*> en su bitácora, entonces C_i debe haber decidido ejecutar *T*. Si una localidad tiene <**preexecute** *T*> en su bitácora, entonces C_i debe haber tomado una decisión preliminar de **preexecute** *T*, lo que significa que todas las localidades, incluyendo alguna que puede haber fallado, haber alcanzado estados **lista**. Por tanto, es seguro ejecutar *T*. Sin embargo, C_{nuevo} no ejecuta *T*, puesto que ello crearía un problema de bloqueo, como en el protocolo de commit de dos fases, si C_{nuevo} fallara. Es por esta razón por lo que la fase 3 es reanudada por C_{nuevo}.

Si ninguna localidad ha recibido un mensaje de **preexecute** desde C_i, es posible que C_i decidiera abortar T, antes que causar un bloqueo.

Nuestra consideración de no aceptar una fragmentación de la red es crucial para el tema anterior. La fragmentación de la red podría conducirnos a la elección de dos nuevos coordinadores, cuyas decisiones no podrían coincidir.

La consideración de que no todas las localidades fallen a la vez, es también crucial. Si todas las localidades fallan, sería sólo la última en caer la que puede tomar una decisión. Esto nos llevaría a un problema de bloqueo, puesto que las demás localidades tienen que esperar a que la última se recupere. Además es muy difícil determinar cuál fue la última en caer después del fallo de todas las localidades.

Finalmente es muy importante la elección del parámetro K, puesto si estuvieran activos menos de un número K de participantes, se produciría un bloqueo.

2.3.4.3 Selección del Coordinador

Algunos de los algoritmos que se presentaron requieren de un coordinador. Si el coordinador falla al quedar fuera de servicio, la localidad donde reside, el sistema podrá continuar con la ejecución sólo si activa un coordinador en alguna otra localidad. Esto puede hacerse si se mantiene una copia de seguridad del coordinador que esté preparado para asumir sus responsabilidades si el coordinador fallara. Otra técnica consiste en elegir el nuevo coordinador después de que haya fallado el coordinador original. Los algoritmos que determinan el sitio donde debe activarse una nueva copia del coordinador se denominan algoritmos de *elección*.

2.3.4.3.1 Coordinadores de copia de seguridad (back-up)

Un *coordinador de copia de seguridad* es una localidad que, además de realizar otras tareas, mantiene de manera local la información que le permite asumir el papel de coordinador con una dislocación mínima del sistema distribuido. Tanto el coordinador

como su copia de seguridad reciben todos los mensajes dirigidos al coordinador. El coordinador de copia de seguridad ejecuta los mismos algoritmos y mantiene la misma información de estado interno que el coordinador real. En el caso de un coordinador de concurrencia, dicha información incluiría la tabla de bloqueos. La única diferencia de funciones entre el coordinador y su copia de seguridad es que este último no emprende acciones que afecten a otras localidades. Tales acciones se dejan al coordinador real.

Cuando el coordinador de copia de seguridad se da cuenta de que el coordinador está fuera de servicio asume el papel de coordinador. Dado que la copia de seguridad dispone de toda la información de que disponía el coordinador inactivo, el procesamiento puede continuar sin interrupciones.

La ventaja principal de este enfoque es que el procesamiento puede reanudarse inmediatamente. Si una copia de seguridad no estaba lista para asumir la responsabilidad del coordinador, se eligirá un nuevo coordinador, el cual tendrá que pedir información a todas las localidades del sistema para así poder ejecutar las tareas de coordinación. Es frecuente que una parte de la información requerida sólo pueda obtenerse del coordinador inactivo. En este caso, probablemente será necesario abortar varias de las transacciones activas (o todas) y reiniciarlas bajo el control del nuevo coordinador.

Por esto el enfoque de coordinador de copia de seguridad evita un retraso considerable mientras el sistema distribuido se recupera de un fallo del coordinador. Además es necesario que el coordinador y su copia de seguridad se comuniquen de forma regular para garantizar la sincronización de sus actividades.

En resumen, la técnica del coordinador de copia de seguridad implica un tiempo extra durante el procesamiento normal que permite la recuperación rápida después de un fallo del coordinador.

2.3.4.3.2 Algoritmos de elección

Los algoritmos de elección requieren que se asigne un número de identificación único a cada localidad activa del sistema. Para facilitar la notación se supondrá que el número de identificación de la localidad L_i es i. Además, para hacer más sencillo el análisis, se supondrá que el coordinador reside en la localidad que posee el número más alto. El objetivo de un algoritmo de elección es escoger la localidad del nuevo coordinador. Por tanto, cuando un coordinador quede fuera de servicio, el algoritmo deberá escoger la localidad activa que tenga el número de identificación más alto. Dicho número debe comunicarse a todas las localidades activas del sistema. Además, el algoritmo debe incluir un mecanismo para que una localidad que se esté recuperando de un fallo pueda identificar al coordinador actual.

Existen varios algoritmos de elección diferentes. La diferencia radica normalmente en la configuración de la red. Aquí se describe uno de esos algoritmos, denominado algoritmo de *prepotencia (bully)*.

Suponemos que la localidad L_i envía una solicitud al coordinador y éste no responde dentro de un intervalo de tiempo T previamente especificado. En tal situación se da por hecho que el coordinador está fuera de servicio y L_i tratará de elegirse a sí misma como localidad del nuevo coordinador.

La localidad L_i mandará un mensaje de elección a todas las localidades que tengan un número de identificación más alto. A continuación esperará recibir respuesta de alguna de estas localidades dentro de un intervalo de tiempo T.

Si L_i no recibe respuesta en el tiempo T, supondrá que todas las localidades con número de identificación mayor que i están fuera de servicio, pero lo que se eligirá a sí misma como localidad del nuevo coordinador y enviará un mensaje para informar a todas las localidades activas con número de identificación menor que i que L_i es la localidad en la que reside el nuevo coordinador.

Sin embargo, si se recibe una respuesta, L_i inicia el intervalo de tiempo T, esperando recibir un mensaje informándole que ha sido elegida una localidad con un número de identificación mayor. (Alguna otra localidad se está eligiendo a sí misma

como coordinador y deberá reportar los resultados dentro de un tiempo T'). Si no se envía el mensaje dentro del tiempo T', se supondrá que la localidad con número de identificación mayor está fuera de servicio y L_i iniciará de nuevo el algoritmo.

En el momento en que una localidad se recupere de un fallo comenzará de inmediato a ejecutar el mismo algoritmo. Si no existen localidades activas con números de identificación mayores, la localidad que se recuperó obligará a las demás localidades (que tienen números más bajos) a que le permitan ser la localidad coordinadora, aun en el caso en que ya exista un coordinador activo con menor número de identificación. Por esta razón se le llama algoritmo de *prepotencia*.

2.3.5 Protocolo de una fase (1PC)

Las variaciones del 2PC garantizan la atomicidad de la transacción llevando a cabo una fase de Votación y una fase de Decisión. La posibilidad de un participante de votar *No* refleja su capacidad de rechazar una transacción *a posteriori*, o sea, después de que las operaciones de la transacción culminaron. En particular, un participante puede necesitar votar *No* si detecta un riesgo de violar alguna de las propiedades A.C.I.D. de su rama de transacción.

La suposición básica que subyace al 1PC es que un participante no necesita votar. Sin embargo, esto no significa que el resultado de la transacción es conocido por adelantado. El *Commit* se decide si todas las operaciones han sido reconocidas y no ocurre ningún error, y el *Abort* puede ser decidido en caso contrario. Sin embargo, en la mayoría de los casos (o sea, en los casos de commit), el coordinador simplemente tiene que mandar un solo mensaje al participante pidiéndoles el commit.

A diferencia del 2PC, aquí los participantes no necesitan votar. En otras palabras, el 1PC no se encarga de asegurar también las propiedades A.C.I.D. de las ramas locales correspondientes de la transacción. Esto significa que antes de disparar el protocolo de commit, el coordinador se asegura que estas propiedades estén aseguradas localmente en todos los participantes. En otras palabras, el coordinador actúa como un dictador agradable y se asegura que ningún participante pueda tener alguna razón

razonable para votar *No*. Esta observación conduce a comprender mejor las suposiciones que normalmente hacen (explícitamente o implícitamente) los protocolos 1PC. Más precisamente:

- ✓ Los protocolos 1PC suponen que todas las operaciones de las transacciones han sido reconocidas (acknowledged) (o sea, todas las operaciones han sido ejecutadas exitosamente hasta ser completadas) antes que se lance el protocolo. Esto significa que la Atomicidad de todas las ramas de transacción locales (o sea, la Atomicidad local) ya están aseguradas al tiempo del commit.

- ✓ Los protocolos 1PC suponen que las limitaciones de integridad se chequean después de cada operación de actualización y antes de reconocer (acknowledge) la operación. Así, se asegura la Consistencia para todas las ramas de transacción locales en el momento del commit (por ejemplo, se excluye la posibilidad de descubrir, en el momento del commit, que no hay suficiente dinero para hacer una extracción bancaria).

- ✓ Los protocolos 1PC suponen que todos los participantes serializan sus transacciones usando un protocolo de control de concurrencia pesimista que evita que se aborte en cascada (o sea, un loqueo (locking) estrictamente en dos fases). En este contexto, una transacción que ejecuta exitosamente todas sus operaciones no puede abortarse más debido a un problema de serialización. Esto en realidad significa que la serializabilidad (Aislamiento) (Isolation) de todas las ramas de transacción locales ya está asegurada en el momento del commit (por ejemplo, se excluyen los protocolos de control de concurrencia optimistas que chequean la serializabilidad en el momento del commit)

- ✓ Los protocolos 1PC suponen que, en el momento del commit, los efectos de todas las ramas de transacción locales se registran en la bitácora en almacenamiento estable, y de aquí que se asegure la propiedad de Durabilidad. En el protocolo de Bitácora Coordinadora, los participantes no mantienen sus actualizaciones en una bitácora estable local. En vez de esto, envían de vuelta, con el mensaje de reconocimiento (acknowledgment) de cada operación de actualización, todos los registros de bitácora (registros de bitácora de "undo"

(deshacer) y "redo" (rehacer)) generados durante la ejecución de la operación. El coordinador está, así, a cargo de registrar en la bitácora las actualizaciones de transacción antes de realizar el protocolo commit. Para recuperar después de un "crash", un participante le pide al coordinador los registros de bitácora que necesita para restablecer un estado consistente de su base de datos.

El Voto Implícito por Sí es similar, excepto que el registro en la bitácora es una tarea más distribuida. La idea es permitir a los participantes que fallaron que lleven a cabo parte del procedimiento de recuperación (la fase deshacer) independientemente del coordinador, y retomar la ejecución de las transacciones que todavía están activas en el sistema (o sea, transacciones para las que todavía no se tomó una decisión) en vez de abortarlas. Los participantes envían de vuelta sus registros de bitácora de rehacer junto con un *Número de Secuencia de Bitácora* (Log Sequence Number, LSN) cada vez que reconocen (acknowledge) una operación de actualización, y la lista de loqueos de lectura (read locks) otorgados durante la ejecución de una operación de lectura.

Para recuperarse después de un "crash", un participante lleva a cabo la fase deshacer del procedimiento de recuperación y una parte de la fase rehacer usando su bitácora local. Luego el participante le pide al coordinador todos los registros de bitácora de rehacer cuyos LSNs sean mayores que su propio LSN más alto, y para todos los loqueos de lectura (read locks) adquiridos por las transacciones activas. Esto permite que el participante reinstale las actualizaciones que pertenecen a la transacción globalmente encomendada/comprometida (committed) y continúe la ejecución de las transacciones que todavía están activas en el sistema.

Nótese que en el CL y el IYV, se requiere que el coordinador escriba forzado sus registros de bitácora antes de enviar un mensaje de decisión a los diferentes participantes. Al recibir la decisión final, se requiere que los participantes del protocolo IYV escriban forzados un registro de bitácora *Commit* ya que tienen que reconocer / dar cuenta de (acknowledge) esta decisión.

2.3.5.1 Desventajas del protocolo de una fase

Al eliminar la necesidad de votos, los protocolos 1PC de hecho logran mejores performances que los protocolos 2PC. Sin embargo, los protocolos 1PC introducen fuertes suposiciones acerca de la forma en que los participantes manejan las transacciones. Suponer que cada operación tiene que ser reconocida (acknowledged) antes de que se lance el APC no es un requerimiento fuerte ya que la mayoría de los estándares transaccionales como el DTP de X/Open y el OTS de OMG suponen el mismo comportamiento. Por el contrario:

(a) los protocolos de recuperación supusieron un aumento en el tamaño de los mensajes de reconocimiento (acknowledgment) y violan la *autonomía del sitio*[1] al forzar a los participantes a externalizar sus registros de bitácora locales

(b) la suposición del loqueo estricto de dos fases también puede parecer muy fuerte ya que varios sistemas de bases de datos aseguran niveles más bajos de aislamiento estandarizados por SQL2 y no garantizan la seriabilidad.

El algoritmo de recuperación asociado con un protocolo ACP debe garantizar la propiedad de Atomicidad de las transacciones distribuidas incluso en presencia de una falla. Esto significa que una vez que el coordinador ha tomado la decisión de Commit, todas las actualizaciones llevadas a cabo por una rama de transacción fallada pueden recuperarse. En el protocolo tradicional 2PC, cada participante vota *Sí* durante la primera fase del protocolo sólo si puede entrar a un estado *Preparado*, o sea, ha registrado en la bitácora suficiente información como para poder recuperar la rama de transacción local. Los protocolos 1PC eliminan la primera fase del ACP, evitando que los participantes entren en un estado *Preparado*. En consecuencia, la Atomicidad de las transacciones se apoya ahora en la información registrada en la bitácora del coordinador. En los protocolos de la *Bitácora Coordinadora* y del *IYV*, la bitácora del

[1] Autonomía de sitio significa, entre otras cosas, que la información interna relacionada con la ejecución local de transacciones (por ejemplo, LSN, registros de bitácora, etc.) sigue siendo privada de un sitio y no puede exportarse ni ser explotada por el sistema transaccional distribuido. La preservación de la autonomía del sitio es crucial en la práctica ya que los protocolos commit a menudo deben usarse con sistemas transaccionales comerciales existentes y no deben requerir cambios a estos sistemas.

coordinador contiene registros de rehacer físicos, haciendo el algoritmo de recuperación bastante directo. Los registros de rehacer se reinstalan en el participante que sufrió la falla durante la recuperación de una rama de transacción local, produciendo así el mismo estado local que el que se produjo durante la ejecución inicial. Sin embargo, si el coordinador aprovecha el registro de bitácora lógico en vez del físico, el protocolo de recuperación debe enfrentar dos nuevos problemas:

- ✓ *las operaciones pueden ser no idempotentes:* se dice que una operación op es *no idempotente* si (op(op(x)) ≠ op(x)). Las operaciones no idempotentes deben ejecutarse exactamente una vez en cualquier situación de "crash".

- ✓ *las operaciones pueden ser no conmutativas:* se dice que dos operaciones op1 y op2 son *no conmutativas* si (op1(op2(x)) ≠ op2(op1(x))). Las operaciones no conmutativas deben ejecutarse durante en el momento de recuperación en el mismo orden que durante la ejecución inicial.

Por éstas y otras razones no incluimos el protocolo de una fase (1PC) en el conjunto de protocolos implementados que pueden ser utilizados en el ambiente para realizar una simulación.

CAPITULO 3

PRESENTACIÓN DE LA PROBLEMÁTICA

3.1 Estudios anteriores en PVM

Se contó con una red de procesadores heterogéneos que estaba formado por un gran número de computadoras o nodos linkeados por una red de interconexión. En ellos se pueden observar diferentes tipos de heterogeneidad con respecto a:

- ✓ arquitectura
- ✓ formato de datos
- ✓ velocidad de procesamiento
- ✓ capacidad de almacenamiento

Cada procesador ejecuta su propio programa el cual puede acceder a memoria local o enviar y recibir mensajes de la red. Dichos mensajes son utilizados para comunicarse con otras computadoras o, equivalentemente, para leer y escribir en memorias remotas.

PVM (Parallel Virtual Machine) es un paquete de software que provee protocolos para la comunicación entre computadoras heterogéneas conectadas a través de una red, haciendo que ésta se vea como una única máquina virtual paralela. Su

objetivo principal es permitir que un conjunto de computadoras sean usadas cooperativamente para procesamiento concurrente o paralelo. [MORS94]

Utiliza el modelo de pasaje de mensajes. Maneja transparentemente todos los mensajes de ruteo, conversión de datos y manipulación de tareas que se puedan hallar en una red compuesta por computadoras de arquitecturas heterogéneas, produciendo un software altamente portable. [GEIS94]

Se realizaron simulaciones, bajo una red heterogénea con soporte PVM, de fallos que podían producirse durante la ejecución de una transacción global utilizando el protocolo de commit de dos fases (2PC). Para cada una de las simulaciones, se analizó la recuperación del entorno ante caídas de alguna de las localidades participantes (que se implementaban mediante tareas que se disparaban a distintos sitios de la red) en la ejecución de una transacción distribuida.

Se crearon distintos procesos, aprovechando el potencial de PVM, con el objetivo de representar al sistema distribuido. Existía un proceso maestro que simulaba al *coordinador* y una serie de procesos esclavos que representaban a las *localidades participantes* de la transacción en cuestión.

3.2 Arquitectura utilizada

La implementación del ambiente se realizó en Java. Se utilizó el JDK 1.3, conjuntamente con el Forte For Java 1.0 para construir la interface con el usuario (GUI). De esta forma se logra contar con interoperabilidad relacionada al hardware, permitiendo que cualquier computadora que posea JVM (Java Virtual Machine) pueda ejecutar el ambiente sin ningún inconveniente.

A continuación se la describirá brevemente la herramienta junto a una explicación de la arquitectura de sockets, que conforman los conceptos más importantes en cuanto a la implementación realizada.

3.2.1 Java

Java es un lenguaje de programación y además una plataforma. Como lenguaje de programación destacamos las siguientes características:

- ✓ Simple
- ✓ Orientado a Objetos
- ✓ Distribuido
- ✓ Interpretado
- ✓ Robusto
- ✓ Seguro
- ✓ Portable
- ✓ Multitarea
- ✓ Gran performance
- ✓ Dinámico

Las computadoras en general ejecutan sobre varias plataformas, como pueden ser Microsoft Windows, Macintosh, OS/2, UNIX y NetWare. El software generado debe ser compilado separadamente para que pueda ejecutarse en cada plataforma. El archivo binario de una aplicación que se puede ejecutar sobre una plataforma, no puede hacerlo sobre otra.

La plataforma Java es un software, que permite ejecutar de forma segura, dinámica e interactiva applets y aplicaciones sobre sistemas de redes de computadoras. Esta plataforma reside sobre otras, compila cada programa a *bytecodes*, los cuales no son específicos a ninguna máquina física, pero son instrucciones para una *máquina virtual*. Un programa escrito en el lenguaje Java compilado, genera un archivo de *bytecodes* que puede ser ejecutado con la plataforma Java, pero bajo cualquier sistema operativo. En otras palabras, el mismo archivo puede ejecutarse bajo cualquier sistema operativo que esté corriendo la plataforma Java. Esta *portabilidad* es posible ya que el corazón de la plataforma Java es la JVM (Java Virtual Machine). A pesar que cada plataforma posee la propia implementación de la JVM, existe solo una única especificación de la máquina virtual. Por lo tanto, la plataforma Java provee una

standard y uniforme interface de programación de applets y aplicaciones sobre cualquier tipo de hardware. [KRAM96]

Los programas Java son conocidos por la particularidad de escribirse una sola vez pero ejecutarse en cualquier plataforma. Esta característica es denominada "*write once, run anywhere*", y permite compilar el programa Java en *bytecodes* sobre cualquier plataforma que posea un compilador Java. Luego, los *bytecodes* pueden ser ejecutados sobre cualquier implementación de la JVM (Java Virtual Machine). Por ejemplo, un mismo programa Java puede ejecutarse sobre Windows NT, Solaris y Macintosh. (Figura 3.1)

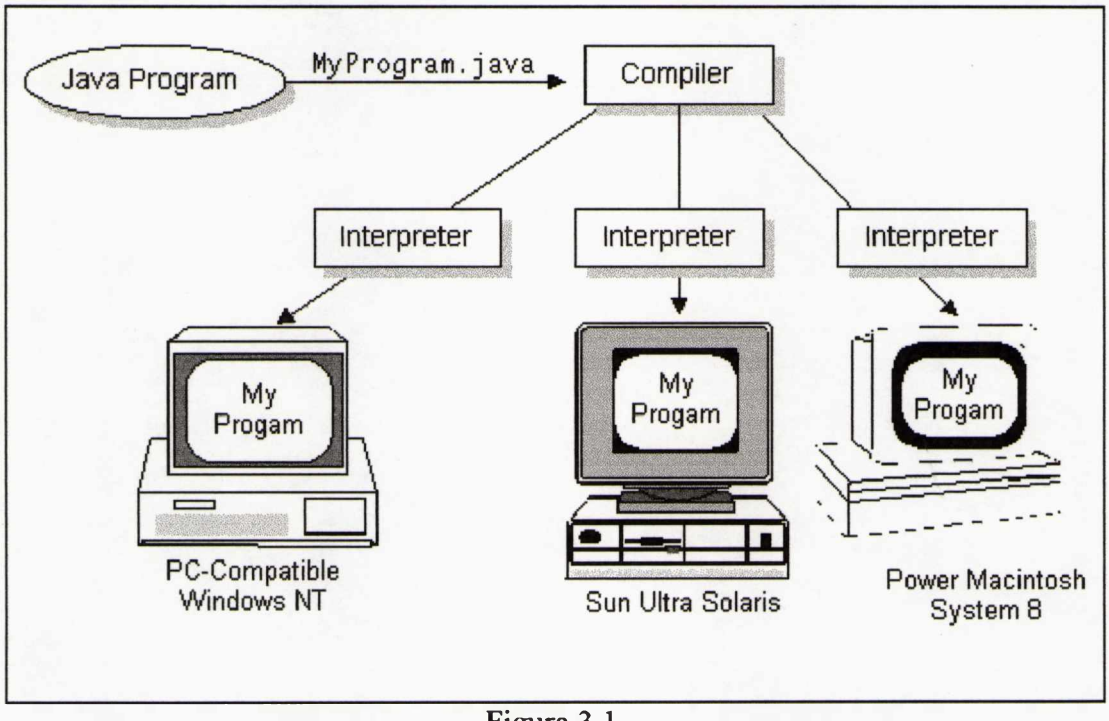


Figura 3.1

Una plataforma es un ambiente de hardware o software en el cual un programa puede ejecutarse. La plataforma Java difiere con la mayoría en que es una plataforma solo de software que se ejecuta sobre otra basada en hardware. A la mayoría se las puede describir como una combinación de hardware y sistema operativo.

La plataforma Java posee dos componentes:

- ✓ La máquina virtual (JVM)
- ✓ La interface de programación (Java API)

La API es una gran colección de componentes de software que proveen una gran capacidad, como ser elementos para la realización de interfaces de usuario gráficas (GUI). Está agrupada en librerías (*packages*) de componentes relacionados.

En al figura 3.2 se describe un programa Java, el cual puede ser una aplicación o un applet, que se ejecuta sobre una plataforma Java.

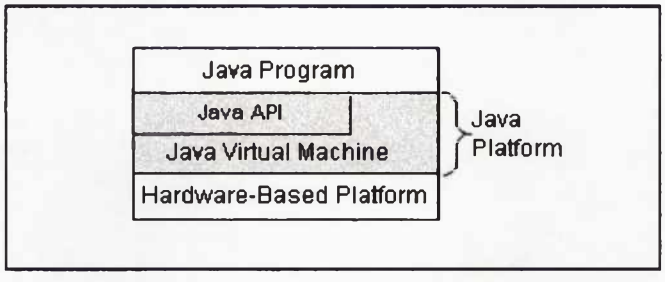


Figura 3.2

A través de las librerías de componentes de software se provee una gran funcionalidad. Principalmente la API de Java ofrece lo siguiente:

- ✓ *Esencial:* Objetos, strings, threads, números, entrada y salida, estructuras de datos, fechas, etc.
- ✓ *Applets*: Un conjunto de convenciones usadas por las applets de Java.
- ✓ *Red*: URLs, TCP y UDP sockets (que se explicarán más adelante del capítulo), y direcciones IP.
- ✓ *Seguridad*: Tanto para bajo o alto nivel, incluyendo firmas digitales, control de acceso a través del manejo de llave pública y privada, etc.

- ✓ *Componentes de software*: Conocidos como JavaBeans, que pueden residir dentro de componentes ya existentes como la arquitectura OLE/COM/Active-X de Microsoft, OpenDoc, etc.
- ✓ *Serialización de objetos*: Permite el manejo de persistencia y comunicación mediante la invocación remota de métodos (RMI).
- ✓ *Conectividad con bases de datos (JDBC):* Provee acceso uniforme a una gran cantidad de bases de datos relacionales.

Como se describió anteriormente, la plataforma Java permite a los desarrolladores crear dos tipos de programas:

- ✓ *Applets*: son programas que necesitan un browser para su ejecución. El tag <applet> es embebido en la página web e indica el nombre del programa a ejecutarse. Cuando la página es accedida por el usuario, el applet automáticamente se dirige desde el servidor a la máquina cliente y se ejecuta en ella.
- ✓ *Aplicaciones*: son programas que no requieren un browser para su ejecución. Cuando una aplicación es invocada, se ejecuta. Por lo tanto podemos ver a las aplicaciones como cualquier otro programa escrito en algún lenguaje. Como los applets, requieren la plataforma Java para poder ejecutarse. Sin embargo, la plataforma puede estar disponible como un programa separado, puede estar embebido dentro del sistema operativo o en la misma aplicación a ser ejecutada.

La utilización de Java presentó ventajas y mejoras respecto a otro tipo de herramientas. Estas mejoras están principalmente relacionadas con el ambiente de simulación, el cual presenta una interface amena. Además, Java permite manipular conexiones vía JDBC (Java DataBase Connectivity) con la base de datos, la cual representa una ventaja para el desarrollador del ambiente en comparación con otras herramientas que actualmente se hallan en el mercado.

3.2.2 Sockets

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. [WEB1]

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 conversar, cada una utiliza 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. [WEB2]

Cada computadora en una red TCP/IP posee una dirección IP única. Los puertos representan conexiones individuales dentro de esa dirección. Cada puerto dentro de una computadora comparte la misma dirección, pero los datos son ruteados dentro de la misma a través de los puertos. Cuando un socket es creado, debe ser asociado a un número de puerto específico. Este proceso es conocido como "ligado a un puerto".

Se pueden clasificar a los sockets según el modo de operar: *orientados a la conexión* o *sin conexión*. Los sockets *orientados a la conexión* operan como un teléfono: deben establecer la conexión primero y luego sí puede comenzar la comunicación. Todo lo que fluye entre los dos puntos de comunicación llega en el mismo orden en que fue enviado. En cambio, los sockets *sin conexión* operan como un mail: la entrega o llegada de la información no está garantizada, y varias partes de un mensaje pueden llegar a destino en diferente orden del que fueron enviados.

El modo que se elija para usar, dependerá de las necesidades de la aplicación que se esté implementando. Si la fiabilidad es importante, los sockets *orientados a la conexión* pueden llegar a ser los más adecuados.

Aquí se utiliza principalmente el package java.net. Cuando los dos sockets están comunicados, el intercambio de datos se realiza mediante InputStreams y OutputStreams. Las operaciones del cliente y del servidor son similares. La principal diferencia se encuentra en que el servidor se puede llegar a comunicar con múltiples clientes, pero el cliente se comunica solo con un servidor, y éste debe generar un número de puerto para esperar posibles conexiones.

La ventaja de este modelo sobre otros tipos de comunicación se ve reflejada en que el servidor 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. [WEB3]

3.3 Presentación del trabajo

El desarrollo consiste en un ambiente de simulación para el estudio de comportamiento de integridad de datos en una base de datos distribuida. A partir de la definición de una transacción local o global, y la elección de un protocolo de commit atómico (ACP), se estudia la respuesta del prototipo ante distintos casos de fallo que pudiesen ocurrir, permitiendo, además, definir el fallo producido y la frecuencia del mismo.

El objetivo perseguido es definir, modelar e implementar un ambiente de simulación para el mantenimiento y recuperación de datos, en un sistema de BDD, donde se pueda estudiar, monitorear y posteriormente comparar resultados a partir de los distintos protocolos de commit elegidos para una misma transacción.

El ambiente desarrollado está centrado en una red LAN interconectada con redes remotas, bajo el soporte de Java.

El desarrollo se basó en cuatro de los protocolos presentados en el Capítulo 2: protocolo de dos fases (2PC), el protocolo presumed commit, el protocolo presumed abort y el protocolo de tres fases (3PC). Para el desarrollo experimental se tuvo en cuenta la importancia o rasgos característicos de los mismos y las diferencias de procesamiento que presentan; lo que permitió realizar un estudio de perfomance entre ellos.

CAPITULO 4

SOLUCIÓN A LA PROBLEMÁTICA PLANTEADA

4.1 Desarrollo de la Solución

Se realizaron simulaciones sobre la ejecución de transacciones en un entorno de datos distribuidos, indicando para cada una de ellas el protocolo de commit atómico (ACP) a utilizar para la recuperación ante distintos casos de fallo, los cuales serán detallados a continuación en el capítulo, manteniendo en todo momento la consistencia de la base de datos. Dicho ambiente se implementó en Java.

Las distintas *situaciones de fallo* que pueden ocurrir durante la ejecución de una transacción distribuida son:

- ✓ *Fallo de una localidad participante.* Para recuperarse, debe examinar su bitácora y a partir de allí, decidir el destino de la transacción.
- ✓ *Fallo del coordinador.* En este caso, las localidades participantes son quienes deben decidir el destino de la transacción. Si no poseen suficiente información, tendrán que esperar a que se recupere el coordinador.

En el ambiente desarrollado solo se simulan fallos correspondientes a localidades participantes de la transacción. Se definieron una serie de instantes en donde un sitio puede llegar a fallar, que se explicarán en detalle más adelante.

Consideremos un sistema distribuido compuesto por un conjunto finito de sitios completamente conectados a través de un conjunto de canales de comunicación. Cada sitio posee memoria local y ejecuta uno o varios procesos. Por simplicidad, se asume

solo un proceso por sitio. Los procesos se comunican entre sí intercambiando mensajes de forma asincrónica.

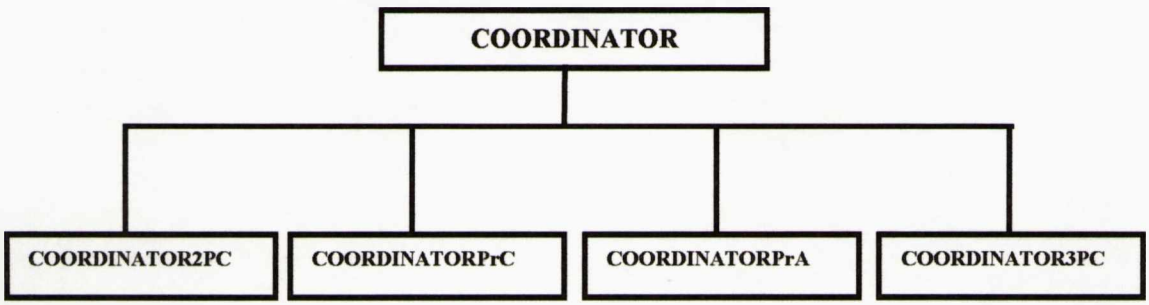
Cada uno de los procesos posee acceso al almacenamiento estable (bitácora), en donde se mantiene la información necesaria para el protocolo de recuperación. Esto significa que, durante la recuperación, el proceso restablece su estado normal usando la información de la bitácora. Además, ante una instrucción de modificación de datos se opera con la técnica de modificación inmediata de los datos, debiendo almacenar el valor viejo y el valor nuevo.

Otra de las suposiciones que se tienen en cuenta es que si un mensaje es enviado desde el proceso P_i a P_k , es eventualmente recibido por P_k , siempre y cuando P_i y P_k no se encuentran "en fallo".

Se dispone de cuatro clases diferentes de tareas:

- ✓ TransactionManager
- ✓ TransactionServer
- ✓ Coordinator
- ✓ Agent

Los procesos que simulan al coordinador y a los agentes participantes de la transacción serán "Threads" que son creados por los procesos "TransactionServer" que residen en cada localidad. En realidad, tanto la clase "Coordinator" como "Agent" son clases abstractas pertenecientes a una jerarquía que representa los distintos protocolos de commit que pueden ser seleccionados para la simulación. O sea que el proceso "TransactionServer" creará la instancia correspondiente al ACP seleccionado, de acuerdo a las siguientes jerarquías:



En un ambiente de base de datos distribuida, una transacción puede acceder a datos que están almacenados en más de un sitio de la red. Cada transacción es dividida en una serie de subtransacciones, una por cada sitio en donde existan datos que la transacción original necesita procesar. Estas subtransacciones son implementadas por los procesos "Agent" que representan cada localidad de la BDD.

Inicialmente, se ejecuta el proceso "TransactionManager", el cual provee la interface para que el usuario pueda realizar cualquier tipo de operación sobre la base de datos. Esto significa la realización de un "Query", el cual puede ser tanto una consulta como una operación de inserción, borrado o actualización.

La idea es que a través de este proceso se pueda configurar todo el ambiente distribuido que vamos a utilizar para realizar las distintas simulaciones. Para ello, se debe ingresar:

- ✓ *Ubicación del Coordinador de la transacción*: esto significa determinar en que sitio de la red se desea iniciar la ejecución de la transacción, y por tanto el lugar en donde va a residir el "Coordinador" de la misma. Para conseguir la mayor abstracción posible, aquí se debe ingresar el número de IP del sitio correspondiente. Esta elección permite al usuario de la simulación que una misma operación sobre la BDD se inicie en diferentes lugares, y así poder, luego, obtener y comparar resultados, teniendo en cuenta la distribución inicial de los datos en cada localidad. El sitio que origina una transacción actúa como el coordinador de la misma, y como tal, es el encargado de tomar todas las decisiones respecto de su ejecución.

- ✓ *Configuración de cada una de las Localidades del ambiente distribuido*: aquí se indica la ubicación de cada sitio del ambiente, también mediante el número IP correspondiente; las tablas de la BDD que va a tener acceso ese sitio y por último se debe indicar si esa Localidad puede llegar a fallar durante la ejecución de la transacción que se va a simular. Esto dependerá principalmente del ACP y del tipo de fallo elegido que se explicará más adelante. También se tiene que definir si la Localidad que se está configurando va a albergar al Coordinador de la transacción, ya que en caso afirmativo, no se la incluye en la lista de localidades que intervendrán en la simulación, teniendo en cuenta las tablas de la BDD que manipularán y las que utilizará la transacción.

- ✓ *Protocolo de Commit Atómico*: este ACP es el que se va a utilizar para mantener la integridad de la BDD mientras se produce la ejecución de la transacción que se va a simular. Entre los protocolos que se pueden elegir se encuentran:

    - Protocolo de dos fases (2PC)
    - Protocolo presumed commit (PrC)
    - Protocolo presumed abort (PrA)
    - Protocolo de tres fases (3PC)

  Esto permitirá que para una misma traza de ejecución, se pueda observar como responde el prototipo ante distintos protocolos para poder realizar una comparación de los resultados obtenidos.

- ✓ *Tipo de error a simular*: esto dependerá del ACP elegido, ya que se definieron distintos momentos en la simulación en donde la localidad puede llegar a fallar, que dependen del protocolo de commit. Para el caso de 2PC, PrC y PrA se definieron cuatro lugares claves, durante el procesamiento de los mismos, en donde se puede producir un fallo:

- *Antes de recibir el "Prepare"*

- *Luego de recibir el "Prepare" y antes de enviar el "Ready/Abort"*

- *Luego de haber enviado el "Ready/Abort" y antes de recibir alguna respuesta del Coordinador.*

- *Luego de hacer "Commit" o "Abort"*

En cambio, para el protocolo de tres fases (3PC), existen dos lugares más, ya que posee una etapa nueva en comparación con los protocolos anteriores:

- *Antes de recibir el "Prepare"*

- *Luego de recibir el "Prepare" y antes de enviar el "Ready/Abort"*

- *Luego de haber enviado el "Ready/Abort" y antes del "Precommit/Abort"*

- *Luego del "Precommit/Abort" y antes del "Receive"*

- *Luego del "Receive" y antes del "Commit/Abort"*

- *Luego de hacer "Commit" o "Abort"*

✓ *Información de la Transacción a ejecutar*: aquí se genera la operación SQL que va a ser partícipe de la transacción. Para esto se debe indicar primeramente las tablas que intervendrán en la ejecución (ésta información es utilizada por el ambiente para luego determinar que localidades deben participar de la simulación), y luego la operación SQL. Si llegase a ser

parametrizada, también se debe definir el tipo y valor de cada uno de los parámetros de la operación.

A partir de toda esta información para la configuración del ambiente distribuido, se puede iniciar la simulación. También es imprescindible que en cada sitio de la red se esté ejecutando un proceso "TransactionServer".

Las experiencias realizadas con el entorno de simulación definido utilizaron una red local; pero tranquilamente se puede llevar esta simulación hacia una red distribuida geográficamente, debiendo contar solamente con estaciones de trabajo para lograr este objetivo sin necesidad de hacer cambios en el entorno definido. Esto se produce ya que para la creación de los sockets sólo es necesario indicar la dirección física de cada localidad y posteriormente ejecutar en cada una de ellas procesos "TransactionServer".

4.2 Modelo de Simulación

En el caso de querer ejecutar transacciones globales, se debe definir para dicha transacción el número de subtransacciones que la componen, y que involucran, a priori, diferentes localidades donde se ejecutarán.

El proceso "TransactionManager", a partir de las especificaciones de las transacciones, determina qué proceso "TransactionServer" residente en alguna localidad del entorno de simulación, será el Coordinador de la transacción. A medida que se van configurando cada una de las localidades del ambiente distribuido, se va llenando una base de datos con información de configuración. Esto incluye, principalmente, la generación automática y posterior almacenado en la BD de los números de puerto que utilizará cada una de las localidades. Se debe recordar que la comunicación en el ambiente se produce mediante "sockets", y para la creación, tanto de un "Socket" como de un "ServerSocket", es necesario el número de puerto por el que se va a producir la comunicación.

Los procesos "TransactionServer" se inician en algún sitio de la red recibiendo inicialmente el número de IP de la máquina en donde están corriendo, y se quedan esperando por algún cliente, el cual puede ser el "TransactionManager" o algún tipo de "Coordinador". En el caso que sea el "TransactionManager" significa que deberá albergar al coordinador de la transacción que se está por simular, y por lo tanto creará una instancia de alguna de las subclases de "Coordinator", dependiendo del protocolo de commit elegido. También recibe del mismo toda la información necesaria para continuar con la simulación, como el ACP seleccionado, el tipo de error, etc. Por el contrario, si el cliente es algún tipo de "Coordinador", significa que deberá crear una instancia de alguna de las subclases de "Agent" dependiendo también del ACP elegido.

A partir de este instante, tendremos en el ambiente distribuido ejecutando un thread para el coordinador y uno para cada una de las localidades participantes de la transacción, cada uno de ellos ubicado en el sitio de la red correspondiente al "TransactionServer" que lo creó. Para ello, el coordinador debe verificar en una BD local (en donde se encuentra la información de configuración) cuales son los sitios pertenecientes al ambiente distribuido que participarán en la ejecución de las subtransacciones que se generen a partir de la transacción a simular. Dichas subtransacciones se crean a partir de la BD local, que posee la información de las tablas que manipula cada una de las localidades. Es muy importante destacar, que ésto se realiza independientemente de la operación SQL que se va a ejecutar, ya que aunque sea del tipo "Select", se deberá obtener la información de todas las localidades posibles ya que en nuestra BDD no manejamos ningún porcentaje de *replicación*.

A partir del establecimiento de los canales de comunicación entre la localidad generadora de la transacción (la cual actúa como coordinadora) y cada uno de las localidades intervinientes se comienza la ejecución de la transacción distribuida. Cada "Agent" ejecuta su parte de la transacción y al finalizar indica al coordinador dicho estado. Este coordinador implementa, como se dijo anteriormente, el protocolo de commit atómico (ACP) elegido al principio de la simulación, para mantener íntegra la BDD.

4.3 Condiciones a comparar

Desde el punto de vista de la performance, los protocolos de commit pueden ser comparados teniendo en cuenta lo siguiente:

- ✓ *Efecto sobre el procesamiento normal*: Se refiere a como el protocolo afecta a la performance de procesamiento de la transacción distribuida normal (sin fallas). Esto es, cuánto nos cuesta proveer atomicidad usando este protocolo?

- ✓ *Flexibilidad en fallos*: Un protocolo de commit es *no bloqueante* si, en el caso de que un sitio falle, permite que los otros sitios sigan ejecutando sin tener que esperar a que el que falló se recupere. Para permitir esta funcionalidad, se incurre usualmente al pasaje de mensajes adicionales, a los que se utilizan en los protocolos *bloqueantes*. En general, el *protocolo de commit dos fases* es *bloqueante* mientras que el *de tres fases* es *no bloqueante*.

- ✓ *Velocidad de Recuperación*: Se refiere al tiempo requerido por la base de datos para recuperarse ante un fallo de un sitio. Esto es, cuánto tiempo pasó, antes que el procesamiento de la transacción pueda comenzar nuevamente, en el sitio que se acaba de recuperar?

Los primeros dos son los más importantes, ya que afectan directamente al procesamiento de la transacción. En comparación, el último punto, aparece menos crítico por diversas razones, entre otras, la duración de una falla posee usualmente un orden de magnitud mayor al tiempo de recuperación del sitio. Desde este punto de vista, es más importante enfocarse en los mecanismos requeridos durante la operación normal de recuperación, en vez del tiempo de recuperación mismo.

Por lo tanto, se indican a continuación las distintas experiencias realizadas con el ambiente de simulación y los resultados obtenidos, teniendo como principal objetivo la comparación de los distintos ACP implementados bajo la ejecución de una misma transacción.

CAPITULO 5

PRUEBAS Y RESULTADOS OBTENIDOS

5.1 Introducción

Para la generación de las pruebas, se definieron tres transacciones tipo, que involucran las tres operaciones SQL más significativas, teniendo en cuenta la integridad de datos en la BDD. Esto significa utilizar las operaciones de alta, baja y modificación de datos. A partir de cada transacción se generaron diferentes pruebas iterando a través de los distintos protocolos de commit atómicos implementados y de los casos de fallo que pueden simularse a partir de las características propias de cada protocolo. Se simularon distintos casos de fallo en los sitios del ambiente distribuido, y también se ejecutaron transacciones en donde no se producía ninguna caída.

Para cada una de las transacciones se definieron situaciones o momentos que representaron los distintos fallos que se podían llegar a simular. La primera situación consistió en probar cada uno de los protocolos teniendo en cuenta la carencia de fallos en las localidades participantes de la transacción. El resto de las situaciones implicó la simulación de los distintos tipos de errores explicados en el capítulo anterior; esto significa disponer de cuatro a seis casos de fallos diferentes (según el protocolo de commit seleccionado).

A partir de las definiciones anteriores, y teniendo en cuenta las tres operaciones básicas originales, más las situaciones de fallo (4 a 6) aplicadas a cada uno de los cuatro protocolos implementados, se generan alrededor de sesenta experiencias para simular. Debido a la tal envergadura, se orientaron los casos de prueba en las experiencias más significativas, en pos de obtener la mayor cantidad de resultados representativos para

los casos de estudios de interés. Sin embargo, y aunque no fueron presentados en esta tesis, se realizaron experiencias concretas utilizando todas las variantes.

Cabe destacar que también se contempló la situación de un fallo general del ambiente, como por ejemplo un corte de luz en medio de la ejecución de una transacción. Este tipo de situación es tratada como si fuera un fallo más en la ejecución de la transacción de acuerdo a lo que dictamina cada protocolo. El ambiente, al iniciarse, chequea por el estado final de la última simulación (transacción) realizada. En el caso que se detecte una finalización anómala, se inicia el protocolo de recuperación, de acuerdo al que oportunamente se estaba utilizando, con el objetivo de mantener la integridad de la BDD que se podía haber perdido con la ejecución parcial de la transacción.

Es importante tener en cuenta que las operaciones de las tres transacciones definidas no tendrán diferencias significativas en cuanto al tiempo de ejecución que necesiten para evaluarse, ya que para realizar el ambiente de simulación no se contempló, en esta etapa, el manejo y posterior actualización de índices o estructuras auxiliares. Por este motivo operar realizando un alta, una baja o una modificación insumen, a priori, tiempos equivalentes.

Además de los resultados que se obtuvieron, teniendo en cuenta todos los parámetros utilizados, se verificó, para cada una de las simulaciones realizadas, que la integridad de la BDD se mantenga, independientemente del resultado de la transacción. Por ejemplo, en el caso que se quiera realizar un alta, y la transacción culmine abortando, es muy importante verificar que la operación no se haya realizado en ninguno de los sitios en donde pudieron haber hecho las inserciones.

5.2 Experiencias realizadas y resultados

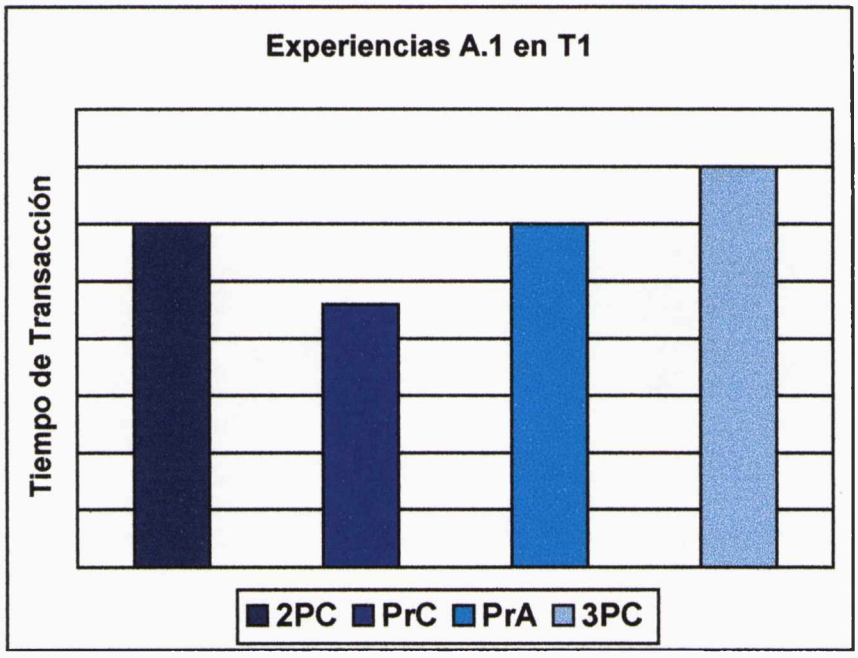
Se presentan a continuación los resultados de las pruebas realizadas. Para cada transacción (*Ti*) se definieron tres situaciones (*Sj*), y cada una de ellas generó una experiencia (*Experiencia j i.k*), donde *k* será alguno de los cuatro protocolos implementados en el ambiente.

Por lo tanto para una transacción *Ti*, y una situación *Sj* vamos a disponer de las siguientes experiencias:

- ✓ **Experiencia j i.1:** Ejecución de T1 utilizando el protocolo de dos fases (2PC)
- ✓ **Experiencia j i.2:** Ejecución de T1 utilizando el *PrC* (Presumed Commit)
- ✓ **Experiencia j i.3:** Ejecución de T1 utilizando el *PrA* (Presumed Abort)
- ✓ **Experiencia j i.4:** Ejecución de T1 utilizando el protocolo de tres fases (3PC)

Transacción 1(T1): "Operación de INSERT sobre una tabla de la BDD"

Situación A: "No se producen fallos en la ejecución de la transacción"



Conclusiones de las Experiencias A.1 en T1:

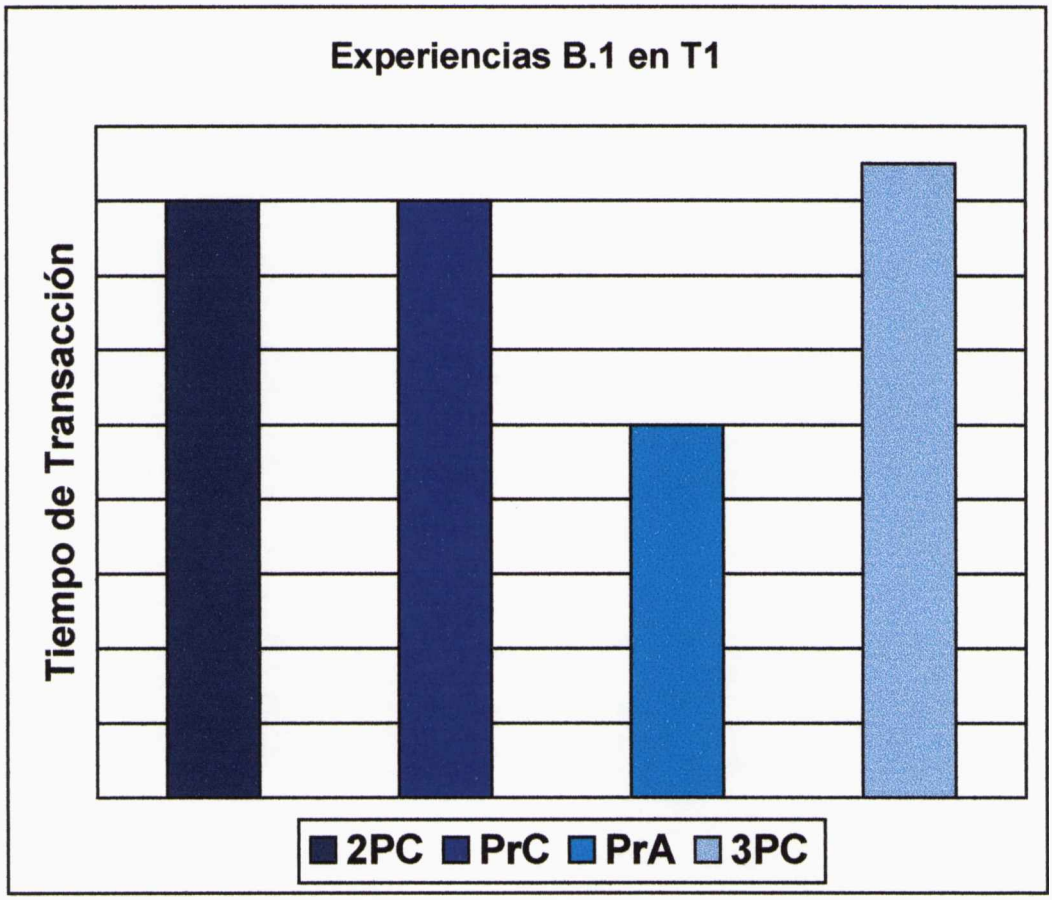
Las pruebas relacionadas con la transacción T1, consistieron en la ejecución de una inserción en una tabla de la BDD, la cual se encontraba fragmentada en varias localidades de la red.

Al no producirse fallos en las simulaciones, el resultado final de las mismas es Commit. Por lo tanto, como se puede observar en el gráfico, se produjo una mejora del 24,6% en el tiempo del *PrC* con respecto al *2PC* y al *PrA*, que obtuvieron tiempos similares. Esto se debe a que el *PrC*, al cometer la transacción, no realiza las últimas etapas del protocolo, porque asume que si no se recibe respuesta del coordinador, la transacción terminó favorablemente. Esto hace que se mejore en la performance del protocolo con respecto al de *2PC* y al *PrA*.

Por último el *3PC* registra un aumento del 16,6% con respecto a 2PC, debido, principalmente, a que su implementación incluye la ejecución de una nueva fase.

Las siguientes situaciones van a simular casos de fallo en algunas de las localidades. Se optó por trabajar con los fallos de Tipo 2 y 3, ya que son los más significativos, teniendo en cuenta que con el de Tipo 2 la transacción termina abortando, mientras que con el de Tipo 3 termina cometiendo.

Situación B: "Se producen el fallo tipo 2 en la ejecución de la transacción"



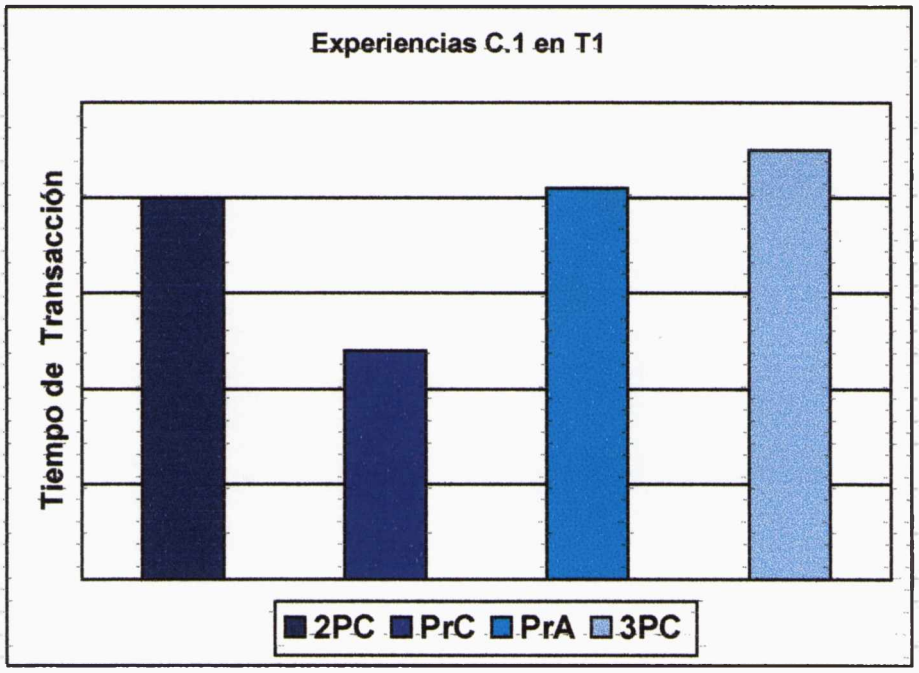
Conclusiones de las Experiencias B.1 en T1:

En las experiencias B.1 se simularon fallos Tipo 2, o sea, se produce la caída de alguna de las localidades luego del mensaje **Prepare** y antes del **Ready/No**. Como consecuencia de este fallo, la transacción culminará abortando.

Además de verificar que el prototipo mantenga la integridad de la BDD ante una transacción que culmina erróneamente, se observó que una mejora del 37,5% en el tiempo del *PrA* con respecto al *2PC* y al *PrC*, que obtuvieron tiempos similares. Esto se debe a que el *PrA*, al abortar la transacción, no realiza las últimas etapas del protocolo, porque asume que si no se recibe respuesta del coordinador, la transacción terminó erróneamente. Como se produjo un fallo, en la etapa de recuperación de la localidad, al no recibir respuesta del coordinador, determina por si sola que la transacción abortó. Esto hace que se mejore en la performance del protocolo con respecto al de *2PC* y al *PrC*.

Por último el *3PC* registra un aumento del 6,2% con respecto a *2PC*. Cabe destacar que el *3PC*, al ser no bloqueante, permitirá que los sitios continúen con su ejecución, a pesar del fallo. Sin embargo, esta ventaja produce que el tiempo del protocolo se incremente, ya que se maneja una fase más en el mismo.

Situación C: "Se producen el fallo tipo 3 en la ejecución de la transacción"



Conclusiones de las Experiencias C.1:

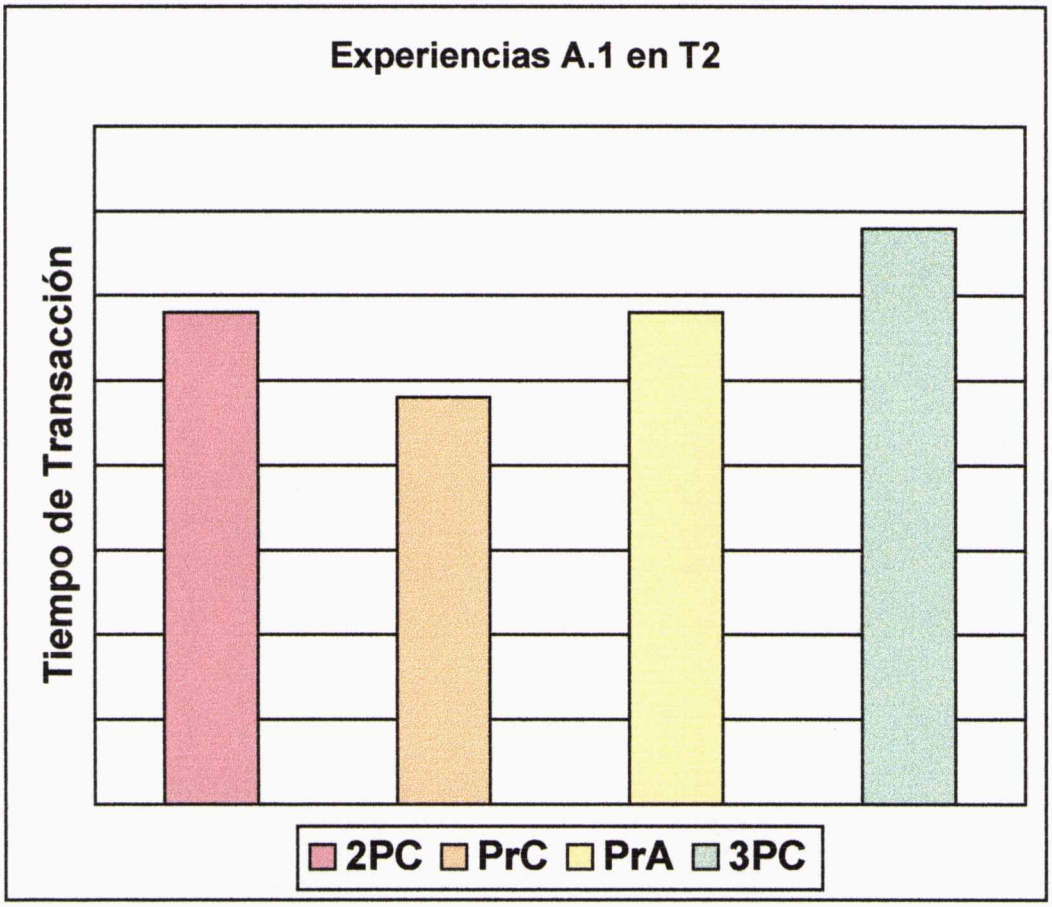
En las experiencias C.1 se simularon fallos Tipo 3, o sea, se produce la caída de alguna de las localidades luego del mensaje **Ready/No** y antes del **Commit/Abort** para el caso de los protocolos *2PC*, *PrA* y *PrC*, y antes del **Precommit/Abort** para el *3PC*. A pesar de este fallo, la transacción culminará cometiendo.

Se observó una mejora del 40% en el tiempo del *PrC* con respecto al *2PC* y al *PrA*, que obtuvieron tiempos similares. Esto se debe a que el *PrC*, al cometer la transacción, no realiza las últimas etapas del protocolo, porque asume que si no se recibe respuesta del coordinador, la transacción terminó satisfactoriamente. Aquí sucede algo parecido a las Experiencias B.1, ya que en la etapa de recuperación, la localidad no recibe respuesta del coordinador, y decide cometer la transacción. Esto hace que se mejore en la performance del protocolo con respecto al de *2PC* y al *PrA*.

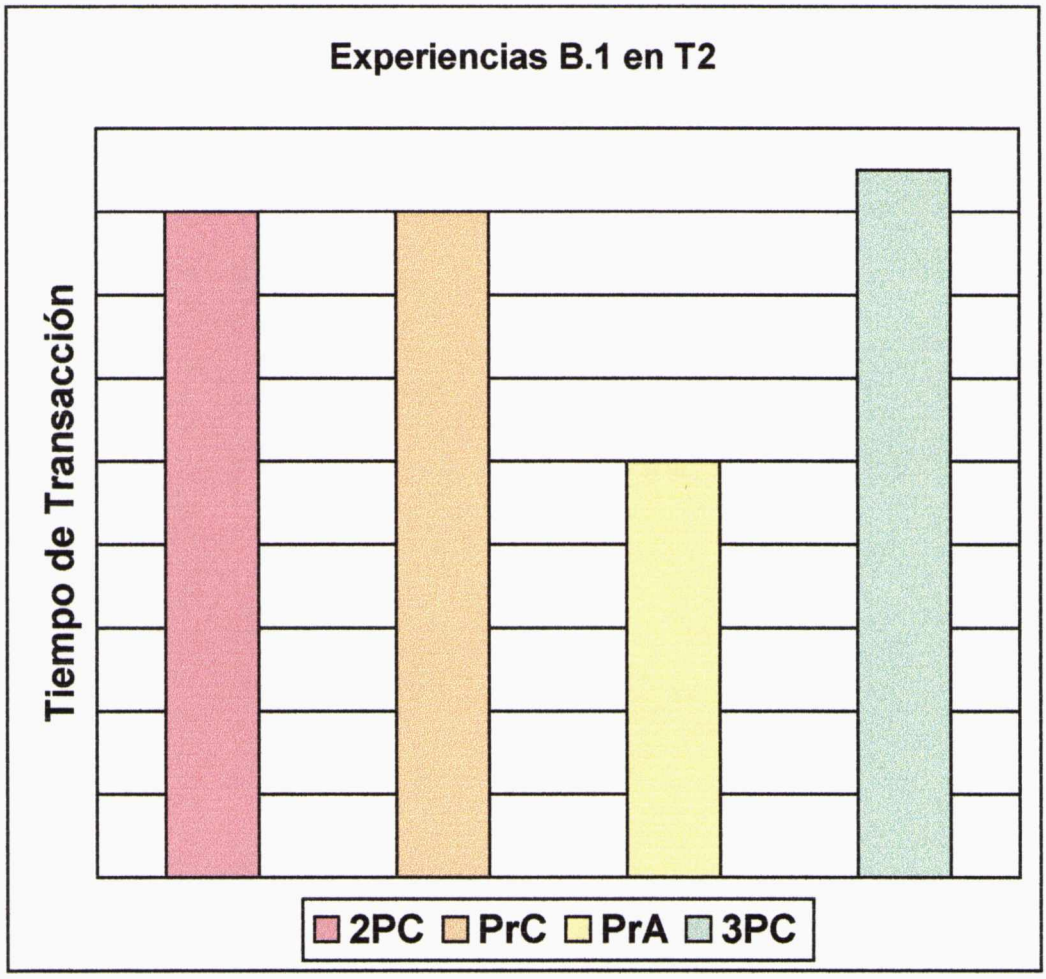
Por último el *3PC* registra un aumento del 12,5% con respecto a *2PC*. Como en el caso anterior, el *3PC*, al ser no bloqueante, permitirá que los sitios continúen con su ejecución, a pesar del fallo. Sin embargo, esta ventaja produce que el tiempo del protocolo se incremente, ya que se maneja una fase más en el mismo.

Transacción 2(T2): "Operación de DELETE sobre una tabla de la BDD"

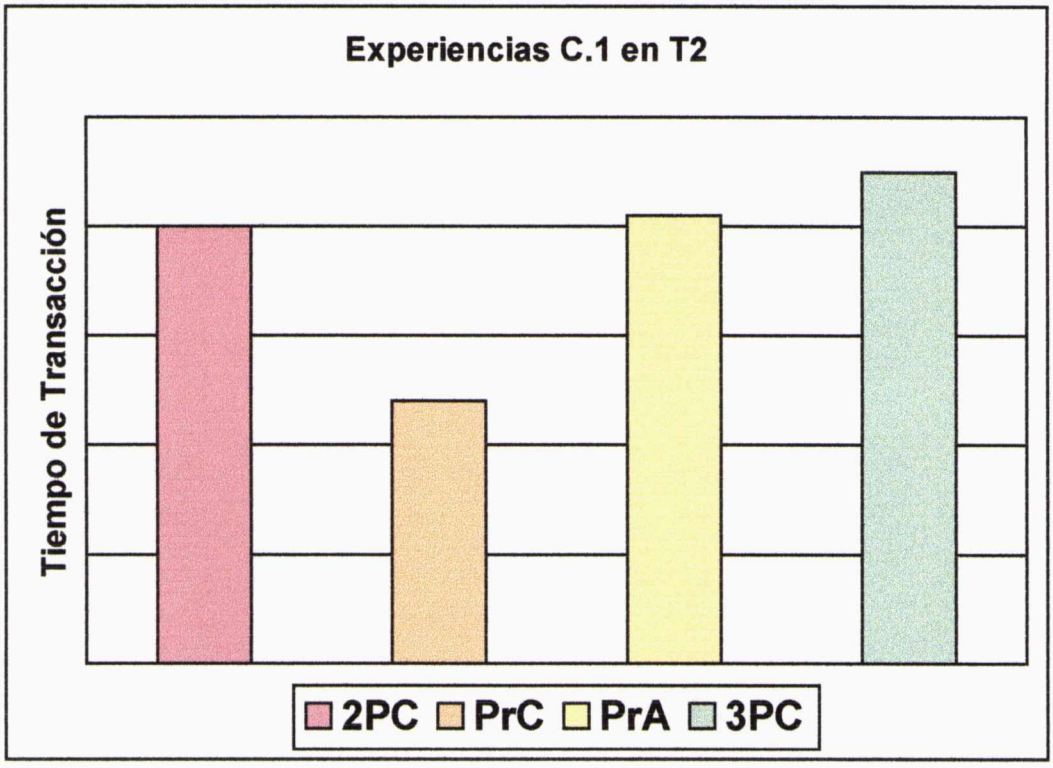
Situación A: "No se producen fallos en la ejecución de la transacción"



Situación B: "Se producen el fallo tipo 2 en la ejecución de la transacción"



Situación C: "Se producen el fallo tipo 3 en la ejecución de la transacción"



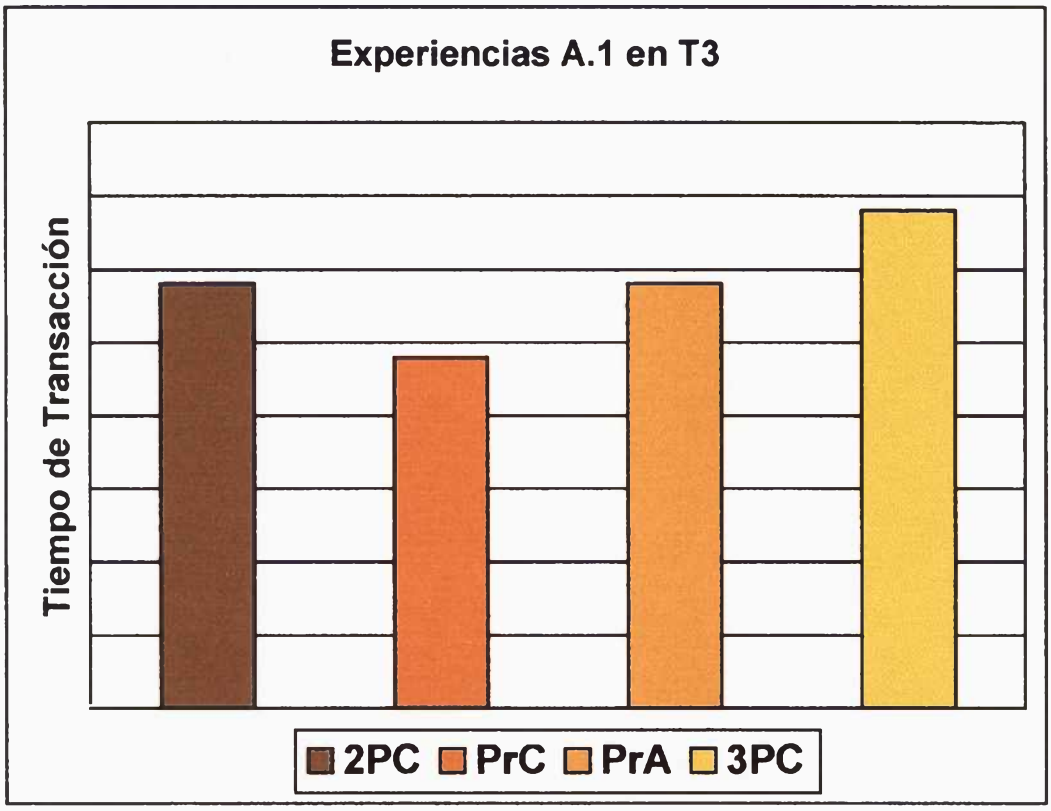
Conclusiones de las Experiencias A.1, B.1, C.1 en T2:

La transacción T2 consistió en una operación de borrado en una tabla de la BDD, que se encontraba fragmentada en varias localidades de la red. Como se puede observar en los tres gráficos anteriores, se verificaron los mismos porcentajes que en las simulaciones con la transacción T1. Esto se debe a que el tiempo que demora cada sitio en ejecutar la transacción es despreciable, en comparación al resto del procesamiento, por causas y motivos explicados anteriormente.

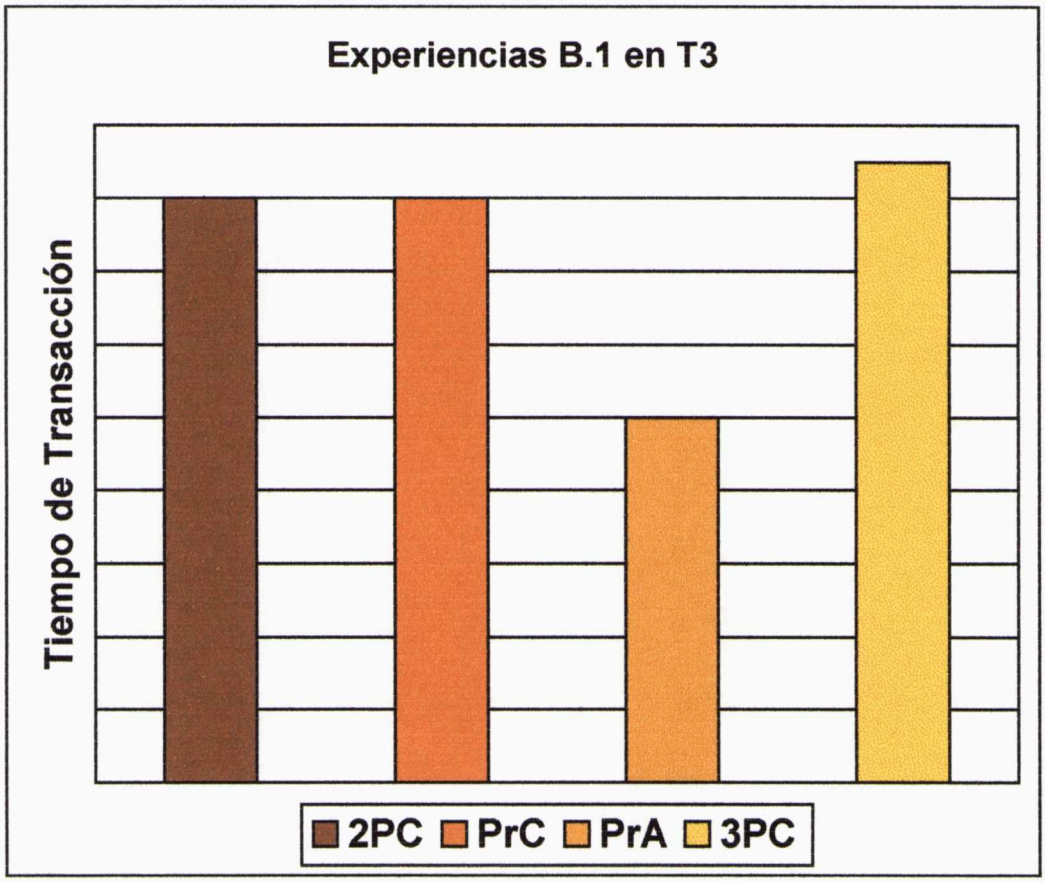
Aquí también se verificó que se mantenga la integridad de la BDD. Esto significa que, ante las pruebas que culminaron cometiendo, se hayan producido las eliminaciones en los sitios correspondientes; mientras que con las simulaciones que abortaron, el estado de la BDD no se haya alterado.

Transacción 3(T3): "Operación de UPDATE sobre una tabla de la BDD"

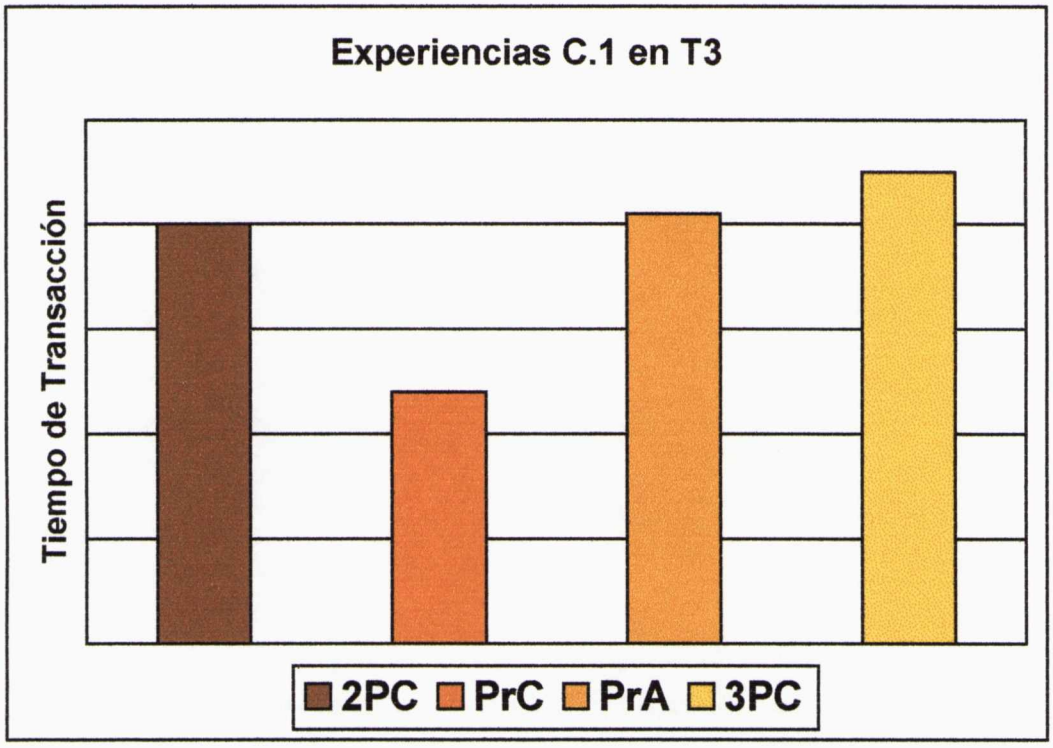
Situación A: "No se producen fallos en la ejecución de la transacción"



Situación B: "Se producen el fallo tipo 2 en la ejecución de la transacción"



Situación C: "Se producen el fallo tipo 3 en la ejecución de la transacción"



Conclusiones de las Experiencias A.1, B.1, C.1 en T3:

La transacción T3 consistió en una operación de actualización de una serie de campos, pertenecientes a una tabla de la BDD, que se encontraba fragmentada en varias localidades de la red. Como se puede observar en los tres gráficos anteriores, se verificaron los mismos porcentajes que en las simulaciones con la transacción T1. Esto se debe a que el tiempo que demora cada sitio en ejecutar la transacción es despreciable, en comparación al resto del procesamiento, por causas y motivos explicados anteriormente.

Aquí también se verificó que se mantenga la integridad de la BDD. Esto significa que, ante las pruebas que culminaron cometiendo, se hayan producido las actualizaciones en los sitios correspondientes; mientras que con las simulaciones que abortaron, el estado de la BDD no se haya alterado.

5.3 Conclusiones

Las primeras observaciones se pueden relacionar a los resultados obtenidos, en donde se evaluó el tiempo de simulación que deparó cada uno de los protocolos de commit atómicos, a partir de las tres transacciones explicadas anteriormente.

Como lo expresan los gráficos anteriores, para las simulaciones en donde las transacciones culminaban cometiendo, el *PrC* resultó el mejor, teniendo en cuenta que no realiza todas las etapas del protocolo, ya que al no recibir respuesta del coordinador, el ACP termina asumiendo el commit de la transacción. Por el contrario, si el resultado final de la transacción era Abort, el protocolo que obtuvo los mejores tiempos fue el *PrA*; que, debido a las mismas circunstancias que el *PrC*, asume el Abort de la transacción al no recibir la respuesta del coordinador.

El 3*PC* fue el protocolo que realizó los mayores tiempos para todas las simulaciones, ya que su implementación utiliza una fase más en comparación al resto de los protocolos utilizados. Como se simularon fallos de localidades, no se pudo ver reflejado en los tiempos las mejoras del *3PC*, que se hubiesen producido al realizar simulaciones con la caída del coordinador. Por otra parte, se debe destacar que es el único protocolo no bloqueante de todos los simulados. Esto permitirá que, ante un fallo, los sitios de la red continúen trabajando.

El 2PC es considerado un poco ineficiente ya que introduce un "substancial delay" (bloqueo) en el procesamiento de una transacción. Este delay está supeditado al costo de tiempo asociado con la coordinación entre mensajes y las escrituras forzadas en la bitácora. [RUSC00]

Supongamos que n es el número total de participantes (incluyendo al coordinador), *2PC* requiere tres etapas de comunicación (el requerimiento para el "voto", el "voto" y la "decisión") y $2n + 1$ escrituras forzadas en almacenamiento estable, para una transacción con ausencia de fallos. Esto introduce una considerable latencia en el sistema, la cual incrementa el tiempo de ejecución de la transacción.

La probabilidad de que en la práctica ocurra un bloqueo es lo suficientemente baja para que no esté justificado el coste extra que supone el *3PC*.

Ambos protocolos pueden ser perfeccionados para reducir el número de mensajes enviados y para reducir el número de veces que los registros son grabados en almacenamiento estable.

Estas limitaciones han incentivado a varios investigadores a trabajar en versiones optimizadas o alternativas de *2PC*. Una de ellas es el protocolo "Presumed Commit" (*PrC*) y otra el protocolo "Presumed Abort" (*PrA*). Estos protocolos reducen el costo de 2PC en términos de escrituras en bitácora y complejidad de mensajes.

La latencia de un ACP está determinada por la cantidad de escrituras de bitácora forzadas y pasos de comunicación llevados a cabo durante la ejecución del protocolo, y hasta que se llegue a una decisión en cada participante. La Figura 5.1 compara los diferentes protocolos en términos de latencia y complejidad del mensaje necesarios para encomendar/cometer (commit) una transacción. Al comparar con el 2PC, el PrA no reduce el costo de encomendar/cometer (commiting) las transacciones. El PrC requiere menos mensajes y escrituras de bitácora forzadas que el 2PC pero no reduce la cantidad de pasos de comunicación requeridos para encomendar/cometer (commit) una transacción.

Protocolo de Commit Atómico (ACP)	Número de Mensajes	Latencia	Latencia
		Escrituras en Bitácora	Número de etapas de comunicación
2PC	$4(n-1)$	$1+2n$	3
PrA	$4(n-1)$	$1+2n$	3
PrC	$3(n-1)$	$2+n$	3

Figura 5.1

Las trazas de ejecución probadas hasta el momento nos demuestran que las variantes del 2PC (PrC y PrA) reducen el overhead del protocolo y proveen una mejora con respecto a 2PC solo en situaciones muy puntuales. PrC presenta una mejor performance cuando el grado de distribución de los datos es alto, pero en las

aplicaciones actuales, el grado de distribución es usualmente bajo. Por otro lado, PrA ofrece una ventaja con respecto al 2PC cuando la probabilidad de transacciones que aborten es baja.

5.4 Trabajos Futuros

Las líneas de trabajo futuras son varias, con el objetivo final de llegar a perfeccionar el ambiente. Inicialmente se va tratar de contemplar fallos que pueden producirse con el coordinador de la transacción. Esto permitirá disponer de experiencias de prueba en donde el protocolo de tres fases (*3PC*) mejore sus tiempos con respecto al resto de los protocolos, situación que no se pudo simular hasta el momento.

También se va a incluir el protocolo optimista (*OPT*) al conjunto que posee el ambiente de simulación para la realización de pruebas. Este protocolo trata de minimizar la fase bloqueante del *2PC*, permitiendo que una transacción que necesita un dato bloqueado por otra que se encuentra en el estado PREPARE, pueda acceder al mismo.

Contemplar el manejo de transacciones concurrentes es otro de los objetivos futuros. Esto conlleva al uso locks y protocolos de seguridad e integridad de datos que no se habían tenido en cuenta hasta este momento.

Bibliografía

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

[BERS92] *Client Server Architecture.* Berson, Alex. Mc Graw Hill Series. 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

[BURL94] *Managin Distributed Databases. Building Bridges between Database Island.* Burleson, Donal. 1994

[COUL95] *Distributed Systems Concepts and Design.* Coulouris, G., Dollimore, J. , Kindberg, T. 1995

[DATE94] *Introducción a los sistemas de Bases de Datos.* Date, C.J. Addison Wesley 1994.

[GEIS94] *PVM: Paralell Virtual Machine. A user guide and tutorial for networked parallel computer.* Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam, The MIT Press, 1994.

[KROE96] *Procesamiento de Bases de Datos.* Kroenke, David. 1996.

[KRAM96] *The Java Platform. A White Paper*. Kramer, Douglas. Mayo 1996.

[LARS95] *Database Directions. From relational to distributed, multimedia, and OO database Systems.* Larson, James. Prentice Hall. 1995

[MIAT99] *Ambiente de simulación para la recuperación en un entorno con Bases de Datos Distribuidas.* Miaton, Ivana; Ruscuni, Sebastián; Bertone, Rodolfo; De Giusti, Armando. Anales CACIC 99. Neuquén Argentina.

[MORS94] *Practical Parallel Computing*, Stephen Morse, AP Profesional 1994.

[ÖZSU91] *Principles of Distributed Database Systems.* Özsu, M. Tamer; Valduriez, Patric. Prentice Hall 1991.

[RUSC00] *Ambiente para la simulación de distintos ACP y recuperación de errores en Bases de Datos Distribuidas.* Ruscuni, Sebastián; Bertone, Rodolfo. Anales CACIC 2000. Ushuaia Argentina.

[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

[SILB98] *Fundamentos de las Bases de Datos.* Silbershatz; Folk. Mc Graw Hill. 1998.

[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

[UMAR93] *Distributed Computing and Client Server Systems.* Umar, Amjad. Prentice Hall. 1993.

[WEB1] *www.itlibrary.com/library/1575211971/ch45.htm*

[WEB2] *www.itlibrary.com/library/1575211971/ch26.htm*

[WEB6] *java.sun.com/docs/books/tutorial/networking/sockets/definition.html*

[ZANC96] *Análisis de Replicación en Bases de Datos Distribuidas*, Marcelo Zanconi, Tesis de Magister en Ciencias de la Computación, Univ. Nac. Del Sur, Bahía Blanca, 1996