Ambiente de Simulación de Bases de Datos Distribuidas

Di Paolo Leiva, Mónica E.

Junio de 1999

TES
99/19
DIF-02408
SALA

UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMATICA
Biblioteca
50 y 120 La Plata
catalogo.info.unlp.edu.ar
biblioteca@info.unlp.edu.ar

DIF-02408

FACULTAD DE INFORMATICA
15 Jun 1999
ENTRADA

DONACION.....
$.....
Fecha 01 03 - 06
Inv. E.....Inv. B. 2408

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Introducción

En la actualidad un área de creciente importancia dentro de las ciencias de la computación es el relacionado con el procesamiento distribuido sobre arquitecturas heterogéneas.

Dentro del área de Bases de Datos Distribuidas (DDB) es interesante el estudio de la eficiencia en la ejecución de transacciones en bases de datos con diferentes modelos de distribución, con o sin replicación.

Una base de datos es la representación de una parte del mundo real en términos de objetos computables y las relaciones existentes entre ellos. [HSU 93].

Una DDB es un sistema que consiste de una colección de localidades, cada una de las cuales consiste en un sistema de base de datos centralizado. Las localidades de la base de datos son conectadas mediante un soporte de red y la comunicación entre ellas es por medio de mensajes. [MAKK 93].

Un gestor de base de datos (DBM) es el software que permite la administración de los datos de la base. Los gestores de base de datos distribuidos (DDBM) proveen la misma funcionalidad que los DBM excepto que deben proveer la transparencia de ubicación, replicación y fragmentación al usuario. [ÖZSU 96]

El trabajo realizado consiste en la implementación de una herramienta que permite generar un ambiente de prueba sobre el cual simular la ejecución de transacciones de diversa complejidad, parametrizando las características del modelo de distribución y los algoritmos globales a utilizar para la ejecución y el control de concurrencia.

El presente escrito se encuentra organizado en cuatro partes. La primera parte detalla los conceptos teóricos que se utilizaron como base en la implementación del simulador. La parte siguiente hace referencia a las principales características de las herramientas utilizadas durante el desarrollo. La tercera parte incluye la especificación de las características de diseño del trabajo y las experiencias realizadas y resultados obtenidos se encuentran la última parte.

AMBIENTE DE SIMULACION DE BASES DE DATOS DISTRIBUIDAS


BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

PARTE I:

Arquitecturas Cliente – Servidor
Bases de Datos Distribuidas

Arquitectura Cliente / Servidor y Procesamiento Cooperativo

Conceptualmente la arquitectura cliente servidor (C/S) puede ser definida como un caso especial de procesamiento cooperativo, donde una aplicación entera es dividida en un sistema cliente (generalmente una workstation inteligente o programable) y un sistema servidor [BERS 92].

Los componentes de hardware cliente y servidor son especializados para facilitar la cooperación de los componentes de software.

Definición extendida:

El término C/S tiene dos significados: uno es la cooperación entre los componentes de software C/S y otro es una relación entre los componentes de hardware: sistema servidor y una workstation cliente.

Cuando los componentes de hardware no se consideran, los procesos cooperativos de software son llamados "Requerimiento de Servicios" (Server request), y a diferencia del modelo C/S la especialización del hardware juega un papel importante.

El modelo "Server request" se caracteriza por:

- ✓ Los programas request y server pueden corren en la misma máquina.
- ✓ La relación "Server request" existe sólo en el momento en el que el servidor recibe el request y envía el reply.
- ✓ La interacción "Server request" no tiene que ser necesariamente sincrónica.

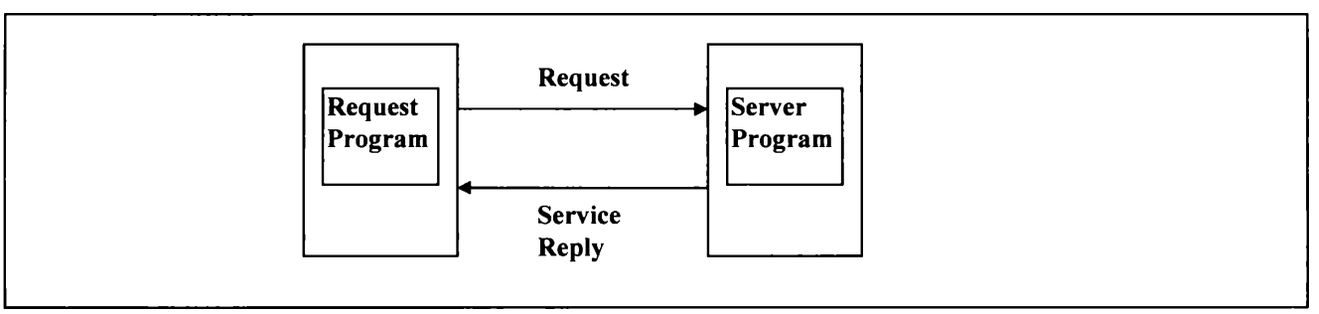


Figura 1

Estructura de los procesos cooperativos

Los procesos cooperativos son un caso especial de los distribuidos. Se caracterizan por

- ✓ la distribución de las funciones de aplicación y componentes en dos o más sistemas de computación

- ✓ Alto grado de interacción entre estos componentes.

Las partes en las que se puede dividir una aplicación son:

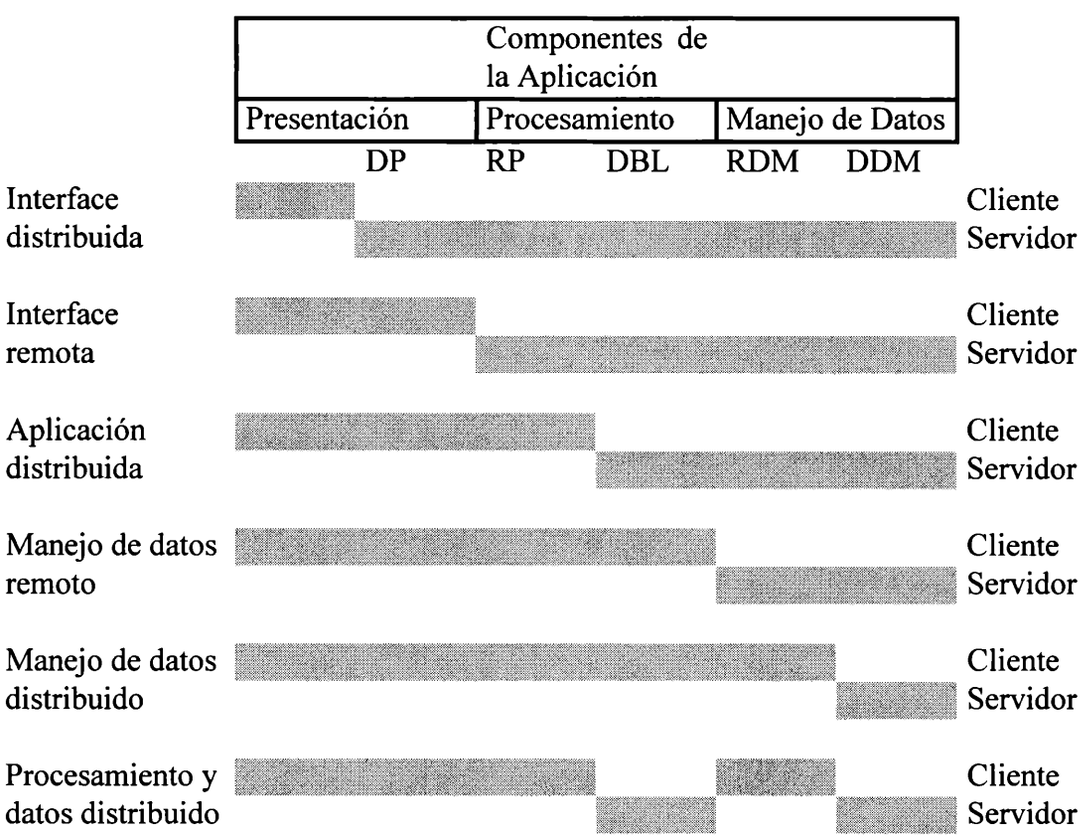
- ✓ Lógica de Presentación (código que implementa la interface de usuario y garantiza seguridad de acceso).
- ✓ Lógica de Procesamiento (código que se encarga de I/O de datos del usuario o de la base de datos y del procesamiento de los algoritmos de la aplicación -a alto orden).
- ✓ Lógica de Manejo de Datos (código referente a la administración de los datos a nivel del Sistema de Administración de Base (DBMS), y al buffering, log y lock, -a nivel del Lenguaje de Manipulación de Datos (DML)[1]).

En el procesamiento basado en host los componentes residen en la misma máquina y son en general linkeados a un único programa. La ventaja del procesamiento en redes es que las aplicaciones pueden compartir recursos.

La arquitectura C/S utiliza el procesamiento cooperativo distribuido para:

- ✓ Distribuir el procesamiento de los componentes de la aplicación.
- ✓ Soportar la interacción entre clientes y servidores en forma cooperativa.

La estructura de la aplicación C/S depende de la parte de la aplicación que se distribuya:



[1] DML: Data Manipulation Languaje. Ejemplo: SQL.

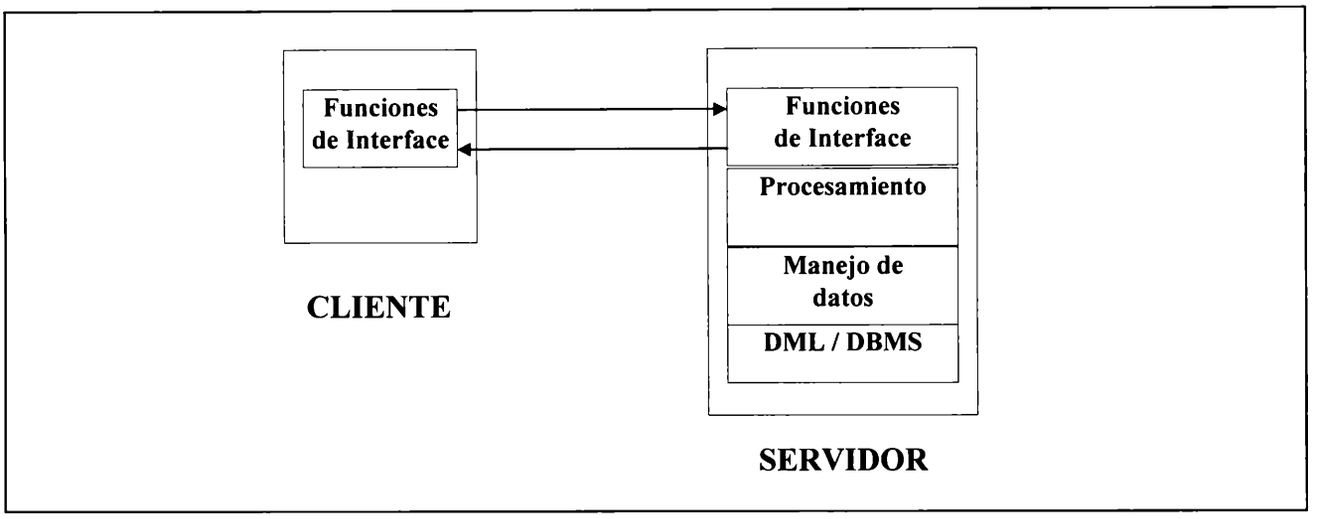


Figura 2

Interface remota

Es el caso en el cual la interface se encuentra completa en un nodo de la red y el resto de la aplicación en otro.

Este tipo de interface puede ser soportado por mecanismos de comunicación punto a punto y el "Remote Procedure Call" (RPC).

Ejemplo:

- ✓ Interface de DEC (sistema de procesamiento de transacciones que corre sobre VAX): se encarga de la validación de las entradas y la generación de ventanas.

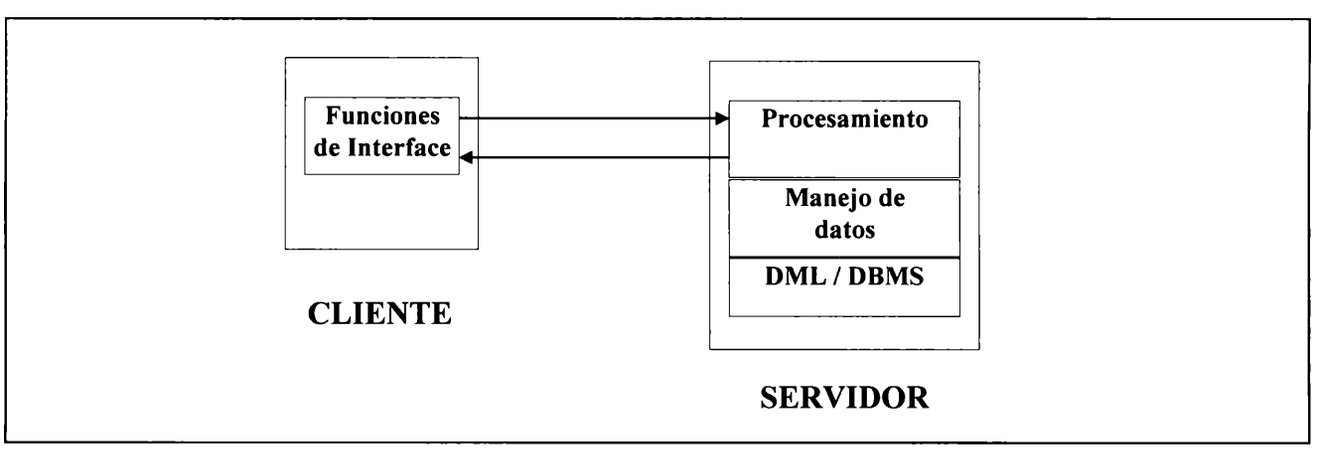


Figura 3

Aplicación

Es conveniente que las funciones de procesamiento se distribuyan en varios nodos para evitar los siguientes problemas:

- ✓ Sistemas menos seguros al existir posibles puntos únicos de fallo.
- ✓ Cuello de botella.
- ✓ Sistemas más lentos frente a la distribución de recursos y la constante actualización de la carga de la aplicación desde el host (donde se está utilizando).

Aplicación distribuida

El procesamiento de la aplicación es dividido en varios sistemas (y cada módulo ubicado en un nodo distinto), así una transacción es ejecutada con la colaboración de todos los componentes de procesamiento y las funciones de manejo de datos.

Una división típica es en dos partes: front-end (componentes de inicialización de las interacciones) y back-end (componentes de reacción). Los componentes front-end se ubican en nodos cliente y los back-end en nodos servidores.

Otra buena distribución que favorece en cuanto a tiempo a aplicaciones muy interactivas es ubicar las funciones relacionadas con el manejo de interface en el nodo de usuario y las relacionadas con el manejo de datos en el nodo que contenga el DBMS (el sistema gestor de la base de datos).

La tecnología subyacente para soportar la distribución del procesamiento incluye RPC o algún mecanismo de comunicación punto a punto.

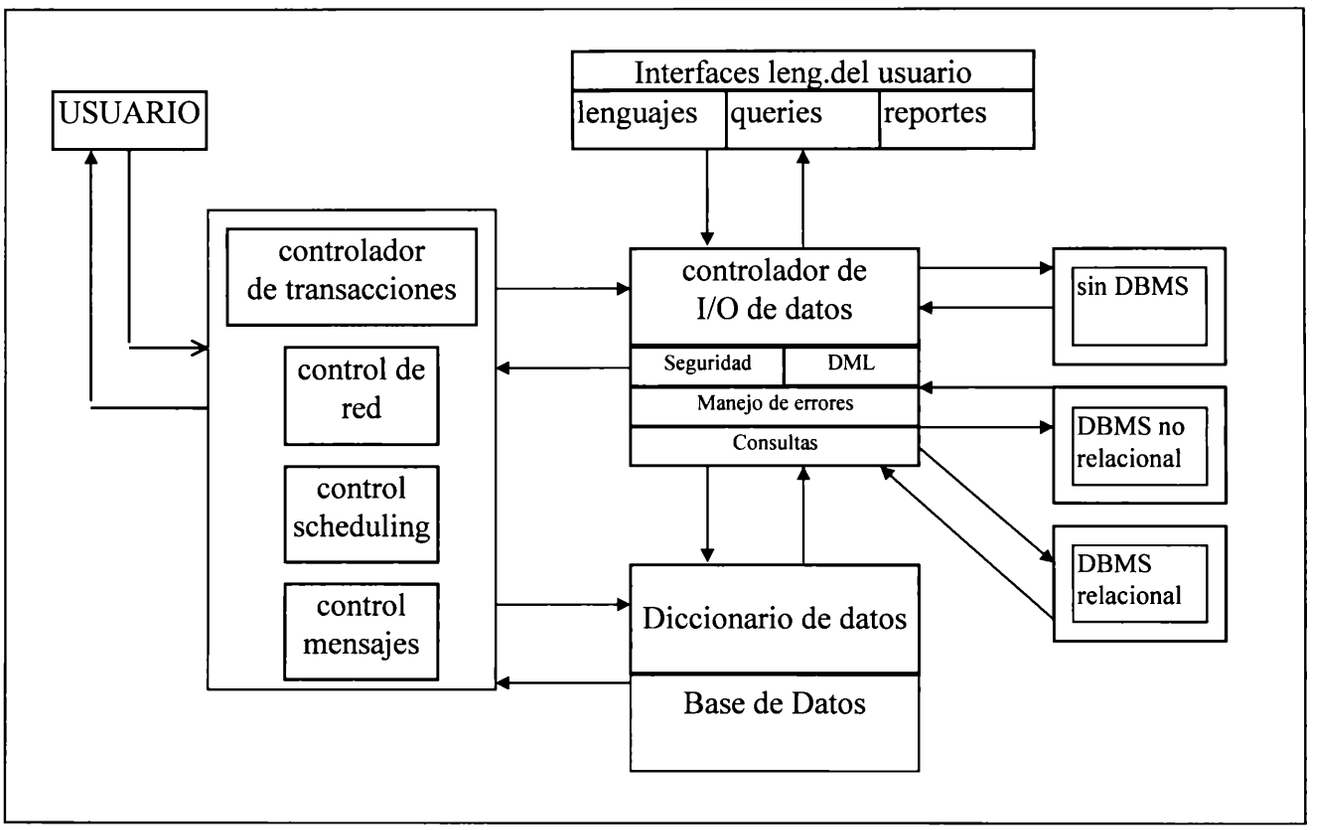
Estas funciones de la aplicación son las más complicadas de implementar, ya que consiste en dos programas compilados por separado que cooperan y por la flexibilidad de la cooperación, estos programas pueden residir en máquinas con distintos sistemas operativos, distintos gestores de bases de datos, diferentes tipos de procesamiento.

Características de una arquitectura de ambiente distribuido:

Un ambiente distribuido se caracteriza por:

- ✓ Los datos son distribuidos en varios workstation.
- ✓ Los procesos son distribuidos en algunos nodos y manejados por un manejador de transacciones distribuido (control de red, coordinación de procesos, sincronización, y schedulling).
- ✓ El acceso a los datos distribuidos es provisto por funciones multinivel como aplicación e interfaces del lenguaje del usuario, controladores de entrada/salida de datos, diccionarios de datos, directorios y catálogos.

Modelo conceptual de ANSI:



Los componentes del modelo ANSI pueden ser encontrados también en ambientes C/S, donde los datos pueden ser distribuidos en varios servidores y los datos locales pueden residir en la workstation del cliente.

La distribución de datos es un paso importante en el diseño de sistemas distribuidos no sólo por la distribución de los datos sino también por el mantenimiento de múltiples copias de los datos críticos en distintas localidades. Existen dos tipos de manejo de datos: distribuido o remoto.

Fundamentos de Bases de Datos

Transacciones

El usuario se comunica con la base de datos a través de transacciones.

Una transacción desde el punto de vista del usuario (operador del sistema, administrador del sistema, etc.) puede ser definida como una unidad "requerimiento / respuesta". [HSU 93].

Desde el punto de vista del sistema, una transacción es una colección de operaciones que realiza una única función lógica. Cada transacción es una unidad atómica que debe ocurrir completa o no ocurrir. [SILB 98].

La ejecución de una transacción conduce al sistema de base de datos de un estado consistente a otro.

Resumiendo lo anterior, las propiedades de una transacción son las siguientes:

- ✓ Atomicidad (no puede ocurrir parcialmente).
- ✓ Consistencia (conduce al sistema de un estado de consistencia a otro estado consistente).

Ejecución de Transacciones

Las transacciones pueden ser ejecutadas en forma serial, esto es, mientras el commit de una transacción no se ejecuta no se puede ejecutar el begin de la siguiente. En una ejecución serial, los recursos se encuentran disponibles para la transacción en ejecución y no existe interferencia. La principal desventaja de la ejecución serial es que el aprovechamiento de los recursos en bajo.

La ejecución concurrente de transacciones aprovecha al máximo los recursos del sistema pero necesita un algoritmo que controle el acceso concurrente a los mismos. Su desventaja es la complejidad de implementación del algoritmo de control de concurrencia.

Para que la ejecución de una transacción mantenga la consistencia del sistema se utilizan algoritmos que establecen:

a) Las acciones tomadas durante el procesamiento normal de las transacciones para asegurarla existencia de información suficiente para asegurar la recuperación de fallos.

b) Las acciones tomadas a continuación de un fallo, para asegurar la consistencia de la base de datos y la atomicidad de las transacciones.

Estados de las transacciones

A continuación se detallan los estados de las transacciones y en el gráfico siguiente se muestra la transición de las mismas de un estado a otro.

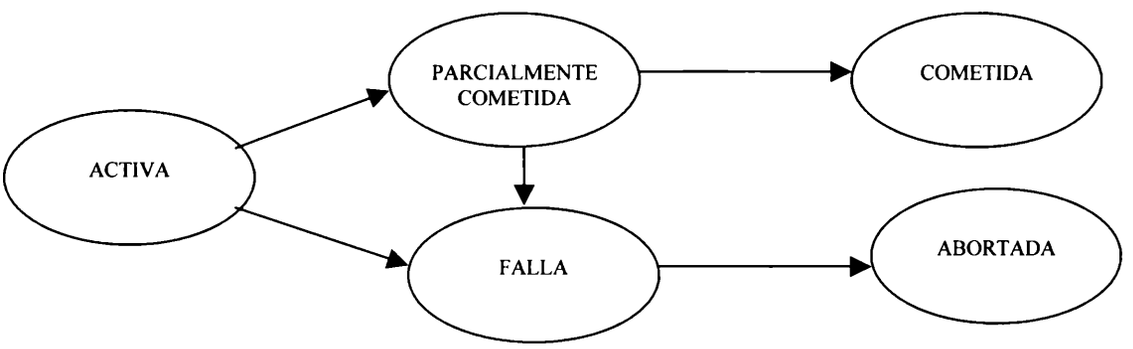
a) **Activa:** Es el estado inicial.

b) **Parcialmente cometida:** tiene lugar luego de la ejecución de la última sentencia.

c) **En falla:** tiene lugar al descubrir que la ejecución normal no puede proseguir.

d) **Abortada:** tiene lugar luego de que la transacción ha retrocedido y se ha restaurado el anterior estado consistente de la base de datos.

e) **Cometida:** tiene lugar luego de la terminación con éxito de la transacción.



Tipos de Almacenamiento

Los tipos de almacenamiento son:

a) **Almacenamiento volátil:** no sobrevive a fallos (Memoria Principal y Cache).

b) **Almacenamiento no volátil:** consiste en los discos, no resiste a fallos.

c) **Almacenamiento estable:** nunca se pierde su información. Se repite la información en varios medios de almacenamiento no volátil (disco).

Tipos de Fallo

Dado un fallo en el sistema de base de datos, lo podemos ubicar dentro de alguno de los siguientes grupos:

a) **Errores lógicos:** entradas inválidas, overflow o exceso del límite de recursos.

b) **Errores del sistema:** El sistema cae en un estado indeseable (ej.: un bloqueo). La transacción puede ejecutarse más tarde.

c) **Caída del sistema:** El hardware funciona mal. Se pierde la memoria volátil.

d) **Fallo de disco:** Un bloque del disco pierde su contenido por rotura de la cabeza o fallos durante una operación de una transacción.

e) **Caída de una línea de comunicación** (en el caso de las Base de Datos Distribuidas).

Jerarquía de Almacenamiento

La base de datos físicamente se divide en unidades de longitud física (bloques), que pueden ser:

a) Bloques físicos (de disco).

b) Bloques de registros intermedios (buffering).

Los movimientos entre bloques físicos y los de Memoria Principal se realizan a través de primitivas INPUT(X) de disco a memoria principal y OUTPUT(X) de memoria principal a disco.

La interacción entre las transacciones y la base de datos se realiza a través de primitivas READ(X,X_i) que asigna el valor X a la variable local X_i; y WRITE(X, X_i), que asigna el valor X_i al elemento de información X. En ambos casos si X no está en memoria principal, se ejecuta un INPUT(X).

Se ejecuta WRITE(X, Xi), cuando la base de datos quiere actualizar X en el disco o por que el manejador del buffer necesita espacio.

Cuando WRITE(X, Xi) actualiza un bloque no necesariamente luego de un WRITE se ejecuta un OUTPUT, por que se puede estar accediendo a parte del bloque.

Recuperación de fallos basados en bitácora

La bitácora (o transaction log) es una estructura que se utiliza para guardar las modificaciones a la base de datos.

Cada registro de la bitácora describe una escritura en la base de datos y puede ser un registro de modificación de datos de la base o un registro de estado de una transacción.

Un registro que describe una ***actualización*** de datos en la base posee los siguientes campos:

a) Nombre de la transacción.

b) Nombre del dato modificado.

c) Valor antiguo del dato.

d) Valor nuevo del dato.

Un registro de ***estado*** de alguna transacción, posee los campos:

a) Nombre de la transacción.

b) Estado de la transacción. Donde los estados dependen de los algoritmos utilizados para la ejecución de transacciones, pero podrían ser: START, EXEC, COMMIT, ABORT.

Los registros de la bitácora deben residir en memoria estable para sobrevivir a fallos.

Técnicas para asegurar la atomicidad de las transacciones

Tenemos tres técnicas para asegurar la atomicidad de las transacciones ejecutadas en la base de datos. Las dos primeras de ellas corresponden a las basadas en bitácora: la modificación diferida de la base de datos, la modificación inmediata de la base de datos y paginación doble.

Modificación Diferida

La modificación diferida de la base de datos no se realiza hasta que la transacción termina. Sus registros van guardándose en memoria estable por lo que una transacción T, interrumpida, si no guardo <COMMIT T>, no modificó la base de datos.

Los registros de actualización en este tipo de modificación tienen la siguiente forma:

a) Nombre de la transacción.

b) Nombre del dato modificado.

c) Valor antiguo del dato.

d) Valor nuevo del dato.

La recuperación de fallos para este modo de modificación se realiza ejecutando REDO(T). La operación REDO, es idempotente, por lo que permite la recuperación aún en el aborto de una recuperación, pues en la bitácora se encuentran los valores nuevos de T.

Si durante la recuperación de una caída ocurre otra, puede que se hayan hecho algunas restauraciones pero no todas. El sistema ejecuta REDO(Ti) para cada registro <COMMIT Ti> que encuentre en la bitácora.

Modificación Inmediata

Las modificaciones de la base de datos para este tipo de actualización se graban mientras las transacciones están activas.

Cada vez que se inicia una transacción Ti, se guarda <START Ti> en la bitácora, al terminar se guarda <COMMIT Ti>, pero antes se actualizan los datos en la bitácora.

Empleando la bitácora el sistema puede enfrentarse a cualquier fallo que no resulta de la pérdida de información en memoria no volátil. El esquema de recuperación utiliza dos procedimientos: UNDO() para restaurar los valores anteriores de los datos modificados por cada transacción y REDO() para asignar los nuevos valores.

Los procedimientos UNDO() y REDO() deben ser idempotentes, para poder recuperar una caída que ocurra inclusive en medio de una recuperación.

La recuperación consiste en recorrer la bitácora y por cada transacción Ti:

a) Si existe en la bitácora <START Ti> y no existe <COMMIT Ti>, ejecutar UNDO(Ti)

b) Si existe en la bitácora <START Ti> y <COMMIT Ti>, ejecutar REDO(Ti).

Paginación Doble

Los datos se encuentran en páginas. Para acceder a las páginas existe una tabla de páginas. En esta técnica existen dos tablas: una "actual" y una "doble".

Cada WRITE modifica la tabla "actual" y al comenzar una nueva transacción la "actual" pasa a ser la "doble". En caso de caídas, el estado se recupera fácilmente pues la tabla "doble" se encuentra en memoria estable.

Para cometer una transacción se debe:

a) Grabar en disco todas las páginas de memoria principal modificadas por la transacción Ti.

b) Grabar en disco la tabla de paginas "actual". (No sobre la dirección de la "doble").

c) Guardar la dirección fija de la tabla "doble". (La dirección de la "doble" se sobreescribe con la dirección de la "actual").

Ventajas y desventajas de la "Paginación Doble" sobre los métodos basados en bitácora

a) Consume menos tiempo extra al grabar registros.

b) La recuperación es más rápida (no necesita ejecutar los procedimientos UNDO y REDO.

c) Necesita técnicas de gestión de almacenamiento, lo que lo convierte en más lento y complejo a causa de la fragmentación de datos.

d) Se necesita de la ejecución temporaria de un "garbage collector"[4], ya que al inciar una nueva transacción, las páginas con versiones anteriores no son útiles.

Gestión de buffering

Los registros de la bitácora permanecen en memoria principal un tiempo hasta que son grabados en memoria estable. Como estos pueden perderse si el sistema cae, se imponen más condiciones para la recuperación:

a) Una transacción Ti entra en el estado "cometido", luego de ser grabado el registro <COMMIT Ti> en memoria estable.
b) Antes de grabar el registro <COMMIT Ti> en memoria estable deben ser grabados los registros anteriores de Ti.

Antes de grabar un bloque de memoria principal de la base de datos en memoria estable, deben haberse grabado todos los registros de la bitácora de los datos de ese bloque.

Buffering de la Base de Datos

La base de datos se almacena en disco y los bloques se cargan en memoria principal según se necesita (según el concepto de memoria virtual de los sistemas operativos). Cuando la memoria principal está completa, se sobreescriben bloques y se ejecuta la siguiente secuencia:

a) Se graban en memoria estable los registros de la bitácora pertenecientes al bloque a sobreescribir en memoria principal.

b) Se graba el bloque B a sobreescribir en el disco ejecutando un OUTPUT(B).

c) Se trae el nuevo bloque B2 del disco a memoria principal ejecutando un INPUT(B2).

Checkpoints

Los checkpoints o puntos de verificación comprenden la ejecución periódica por parte del sistema de la bajada a memoria estable del contenido de la bitácora.

Esta bajada de registros comprende:

[4] Garbage Collector: rutina que recupera las direcciones perdidas de memoria.

a) Grabar los registros la bitácora que se encuentran en memoria principal en memoria estable.

b) Grabar en disco los bloques modificados de los buffers.

c) Grabar un registro de bitácora <CHECKPOINT> en la bitácora en memoria estable.

Al caer el sistema se ejecutarán los procedimientos REDO y UNDO según corresponda, a las transacciones cuyo commit no se encuentre antes del último checkpoint.

Para la técnica de actualización inmediata: se ejecutará REDO(Ti) para toda Ti cuyo registro <COMMIT Ti> aparezca en la bitácora después del último checkpoint y se ejecutará UNDO(Ti) para toda transacción Ti para la cual no aparezca su registro <COMMIT Ti>, luego del último checkpoint.

Para la técnica de actualización diferida: se ejecutará REDO(Ti) para toda Ti cuyo registros <COMMIT Ti> aparezca en la bitácora después del último checkpoint.

Fallos con pérdidas en memoria no volátil

Periódicamente se vuelca el contenido de la memoria no volátil (disco) en memoria estable (ej: cinta magnética). Si ocurre una caída, se utiliza el volcado más reciente para restaurar la base de datos y la bitácora para restaurar el sistema de base de datos a un estado consistente anterior.

El procedimiento que se utiliza es el siguiente:

a) Grabar en memoria estable los registros de la bitácora de memoria principal.

b) Grabar los bloques de buffers en disco.

c) Copiar la base de datos a memoria estable (este debe realizarse cuando no se encuentre ninguna transacción activa).

d) Copiar un registro <DUMP> en la bitácora.

La desventaja de esta recuperación es el costo por transferencia de datos, además del desperdicio de ciclos de CPU.

Implementación de Memoria Estable

La memoria estable se implementa copiando cada bloque más de una vez. Si dadas dos copias de un bloque, estas difieren, se ejecuta un procedimiento de recuperación.

Acceso a Datos Distribuidos

Estructura de una Base de Datos Distribuida

Un sistema de base de datos distribuido consiste en un conjunto de localidades, cada una de las cuales mantiene un sistema de base de datos local.

Cada localidad puede ejecutar transacciones locales (las que acceden a datos locales) o distribuidas (las que acceden a datos de varias localidades).

La estructura física del sistema (forma en la cual se encuentran conectadas las computadoras del sistema distribuido), tiene características propias que influyen en la ejecución de las transacciones.

Desventajas de la distribución de datos

La principal desventaja de los sistemas distribuidos de datos consiste en la mayor complejidad de los algoritmos de administración de datos por que deben coordinar la ejecución de transacciones entre localidades.

- ✓ El aumento de la complejidad se refleja en:
- ✓ Mayor costo de desarrollo de software para procesamiento distribuido de los datos.
- ✓ Mayor probabilidad de errores, ya que las localidades ejecutan en forma paralela por lo que los algoritmos pueden fallar con mas frecuencia.
- ✓ Mayor tiempo de comunicación por el intercambio de mensajes entre las localidades para coordinar la ejecución de las transacciones. Este costo extra no existe en los sistemas centralizados.

Métodos de acceso distribuido a datos

Requerimiento Remoto

En el modelo relacional, un "Requerimiento Remoto" es una instrucción SQL[5] que referencia a datos que residen en un sitio remoto. Se utiliza en el método de extracción manual (de distribución de datos).

[5] Véase "El Lenguaje SQL" (ANEXO I).

Transacción Remota

Desde el punto de vista de arquitecturas C/S, las transacciones remotas implican datos remotos ubicados en un servidor y accedidos por una workstation cliente. En el modelo relacional, una transacción remota consiste de varias instrucciones SQL cada una de las cuales referencia a datos que residen en un mismo sitio remoto (servidor).

Ejemplo:

```
BEGIN WORK
SELECT *    FROM SERVER1.BANKDB.CUSTOMER
            WHERE SERVER1.BANKDB.CUSTOMER.CITY="New York"
UPDATE      SERVER1.BANKDB.BRANCH
            SET POSTED_IND="Yes"
COMMIT WORK
```

Las transacciones remotas pueden usarse para el método de extracción manual (de distribución de datos).

Transacción Distribuida

Las transacciones remotas contienen múltiples request de datos, referenciando cada uno a una localidad, no necesariamente la misma que las de los otros request.

En una arquitectura C/S, una transacción distribuida implica datos distribuidos en varios servidores que pueden ser accedidos desde una workstation cliente en una unidad lógica de trabajo. En el modelo relacional, puede consistir de varias instrucciones SQL, donde la ubicación de los datos puede ser en múltiples localidades.

Una transacción distribuida tiene éxito si cada una de las instrucciones SQL (ejecutadas en distintas localidades) tienen éxito.

Requerimiento Distribuido

Un requerimiento distribuido permite a una transacción formada por múltiples requests ser procesada por un servidor de DDB. Una transacción que consiste en varios requerimientos puede ser procesada en múltiples sitios y cada request puede referenciar datos residentes en múltiples sitios.

En una arquitectura C/S, los requerimientos distribuidos pueden ser distribuidos en varios servidores cada uno por fragmentación o replicación, y pueden ser accedidos transparentemente desde una workstation cliente en una transacción (la cual puede consistir de varias instrucciones SQL, cada una referenciando a datos locales en múltiples localidades).

Los cuatro tipos de distribución de datos son requeridos para soportar un DDBMS. Pero sólo el procesamiento del request distribuido soporta los conceptos de un verdadero DDBMS.

Los requerimientos remotos y las transacciones remotas y distribuidas soportan un modelo de computación C/S, pero imponen restricciones de ***cómo los datos son accedidos*** y ***qué puede hacer la aplicación*** (todos requieren que la aplicación conozca la localización física de los datos).

Diseño de Bases de Datos Distribuidos

La base de datos es físicamente distribuida ubicando los datos en diversos nodos de la red utilizando replicación o fragmentación. En un esquema de bases de datos relacionales, la fragmentación divide cada relación horizontalmente (utilizando una operación "selección") y la replicación las divide verticalmente (utilizando una operación de "proyección") [ÖZSU 96].

Replicación

Las ventajas de la replicación son [GRAY 96]:

- ✓ **Mayor disponibilidad de los datos**, pues si falla una localidad con un dato r, el sistema puede continuar procesando con el mismo dato en otra localidad (en la que se encuentre una réplica de r).

- ✓ **Mayor paralelismo** en el sentido en el cual cuanto mayor cantidad de réplicas del dato r existan, en el caso en el que se desee sólo leer el dato, mayor cantidad de localidades podrán ejecutar consultas en paralelo.

La desventaja:

- ✓ Se consume mayor tiempo extra para actualizar datos que se encuentran replicados, ya que se debe mantener la consistencia de los mismos en todas las localidades.

Fragmentación

Existen dos formas de fragmentar una relación: en forma horizontal (en esta lo que se distribuyen son las tuplas de la relación), vertical (en este caso se descompone la relación) o fragmentación mixta que involucra una combinación de los métodos anteriores.

Si tenemos una fragmentación horizontal, se obtiene la relación inicial mediante una operación unión. Para obtener la relación original luego de una fragmentación vertical se debe aplicar la operación (producto natural o join natural).

Diseño de Sistemas de Gestión de Bases de Datos Distribuidas

Características de un Sistema de Gestión de Bases de Datos (DBMS)

Un DBMS es responsable del mantenimiento de la base de datos de un estado consistente a otro. Por definición, la conclusión parcial de las transacciones no debe permitirse.

Un DBMS debe reunir las siguientes características:

- ✓ Implementa un modelo de datos particular (relacional, jerárquico, etc.)
- ✓ Soporta lenguajes de manipulación de datos (por ejemplo: SQL).
- ✓ Provee independencia de datos, consistencia, integridad, seguridad y consulta.
- ✓ Provee interfaces entre la aplicación y el software de red.
- ✓ Soporta varias plataformas de hardware.

Cuando los datos son distribuidos en varias localidades, la parte de la aplicación de manejo de datos debe ser distribuida en parte o en su totalidad. Independientemente del método de distribución de datos utilizado, el acceso a los datos provisto por el DBMS distribuido (DDBM), debe ser ejecutado en forma transparente al usuario y a las aplicaciones.

Diccionarios de datos distribuidos:

Los diccionarios constituyen un catálogo global de todos las localidades de la base de datos distribuida. El diccionario de la base de datos local se actualiza por un proceso que mantiene la información concerniente a su localidad. La replicación de los diccionarios resulta en un diccionario global. [BERN 96].

Los diccionarios de datos distribuidos son utilizados para almacenar información a cerca de la base de datos como: elementos de datos, atributos, entidades, reglas índices, estadísticas, datos remotos, información a cerca de la red y características de los nodos, etc.

Los diccionarios de DDB son almacenados y mantenidos por el servidor de DBMS local.

El diccionario global en sí mismo es una DDB.

Reglas de Date para el diseño de gestores de datos distribuidos

Los diseñadores de sistemas distribuidos deben tener en cuenta las siguientes reglas:

1) **Autonomía de localidad:** los sitios deben ser autónomos unos de otros, o sea reunir las siguientes características:

   - ✓ una base de datos local es procesada por su propio DBMS
   - ✓ el DBMS maneja la seguridad, integridad, consistencia u consulta de su propia DB
   - ✓ Cada sitio es independiente en operaciones locales y colaboran en accesos a datos distribuidos.

2) **Inexistencia de un sitio central.** La centralización de procesamiento en un sitio central provoca el efecto de un cuello de botella, por lo que cada sitio debe manejar su control de concurrencia, su DB local, el diccionario de datos y la replicación. Un DBMS en cualquier localidad puede actuar como un manejador de DB distribuidas (ej: un coordinador de dos fases)

3) **Operación continua:** un ambiente distribuido debe ser implementado teniendo en cuenta que:

   - ✓ Cada servidor de DB debe ser capaz de hacer su backup de la DB on line, mientras procesa otras transacciones.
   - ✓ Debe proveer soporte para recuperación.
   - ✓ Debe proveer soporte para la tolerancia de fallas.

4) **Transparencia de localización de datos:** Si se logra la transparencia de localización se asegura que:

   - ✓ El diccionario de datos distribuidos debe mantener una tabla de elementos de datos, sus alias y localizaciones (pues los usuarios y aplicaciones hacen referencia a los datos por medio de alias).
   - ✓ El DBMS distribuido debe usar y mantener automáticamente el diccionario de datos.

5) **Transparencia de la fragmentación de datos:** la fragmentación debe ser independiente de los usuarios y de las aplicaciones. Una tabla fragmentada se debe ver como una tabla simple.

   Los DBMS que soportan transparencia de fragmentación son INGRES e INFORMIX.

6) **Independencia de replicación:** los datos replicados son copias de datos en otro servidor (por lo tanto, son distribuidos), entonces se requiere transparencia de localización. Además se debe tener en cuenta que:

- ✓ Para sincronizar las actualizaciones se necesita el commit de dos fases. Las actualizaciones (locales o remotas) y los commit o las transacciones deben poder ser retrocedidos (ROLL BACK), en caso de no terminar la transacción.

- ✓ La sincronización entre clientes y aplicaciones es transparente a ellos.

- ✓ La sincronización por medio del LOCK permite el bloqueo del registro que desea modificar.

- ✓ Se debe resolver el problema del DEADLOCK (por ejemplo con TIME-OUT para la espera de recursos y terminación de la transacción). Los DBMS para esto deben usar algoritmos de detección de DEADLOCK y terminación de tareas (por ejemplo terminando la mas corta o la que hizo menos actualizaciones).

  Muchos DBMS distribuidos actuales no soportan transparencia de replica.

7) **Procesamiento de querys distribuidos:** a diferencia de los DBMS relacionares navegacionales (donde el usuario dirige la búsqueda de datos), los RDBMS no navegacionales sólo permiten que el usuario detalle los datos que quiere, no como buscarlos. Generalmente un RDBMS usa un algoritmo de optimización del camino de acceso (transparente al usuario). Estas técnicas de optimización pueden basarse en el costo (camino con menor cantidad de I/O o ciclos de CPU) o en reglas de álgebra relacional (que agilizan los joins por ejemplo).

   En una arquitectura C/S el DBMS debe mantener la lista de los servidores y sus características de procesamiento (para luego buscar caminos óptimos).

   Se necesita acceder a diccionarios de datos distribuidos durante las consultas (y actualizarlo). Los DDBMS deben coordinar la optimización (teniendo en cuenta factores locales y globales, por ej: características de los nodos y de la red) y la sincronización.

8) **Manejo de transacciones distribuidas:** Cualquier algoritmo de control de concurrencia soporta atomicidad, consistencia y durabilidad de la transacción de la base de datos, esto requiere además el esfuerzo de todos los participantes de la ejecución de la transacción[6].
   Los problemas que debe tratar el manejo de transacciones son: locking distribuido, detección de deadlock, restauración de un backup local, logging, administración y seguridad.

   Es conveniente que el manejo de las transacciones este separado del DBMS.

9) **Independencia del hardware:** los sistemas de DDB deberían correr sobre cualquier plataforma de hardware.

10) **Independencia del sistema operativo:** los sistemas de DDB deberían corren sobre cualquier sistema operativo.

11) **Independencia de la red:** los sistemas de DDB deberían corren sobre cualquier red, sin interesar su implementación ni la de sus protocolos.

---

[6] Véase "Protocolos COMMIT de Ejecución de Transacciones" (Parte I).

12) **Independencia del manejador de bases de datos:** los sistemas de DDB deberían poder manejar datos administrados por diferentes DBMS, aun cuando sean de distintos vendedores.

La implementación de ambientes distribuidos heterogéneos no es simple, el problema es que cada DBMS habla su propio lenguaje de acceso a datos.

El acceso a datos heterogéneos complica el manejo de la integridad, locking, administración y seguridad.

Recuperación de Fallas en Sistemas de Bases de Datos Distribuidas

Estructura del sistema de recuperación

El sistema de recuperación de datos de una DDB comprende varios procesos que se encargan de la ejecución de transacciones que se originan en diversas localidades del sistema.

Gestor de Transacciones:

Este proceso se encarga de asegurar la ejecución de transacciones en el sistema conserve la consistencia de los datos.

Cada localidad posee su propio gestor de transacciones y entre ellos cooperan para ejecutar las transacciones distribuidas. Cada uno ejecuta la transacción o una subtransacción de la originada en la localidad en forma concurrente con las demás localidades que poseen replicas de los datos que intervienen en la transacción.

El gestor se encarga de mantener la bitácora (o transaction log) para poder recuperar el sistema ante una falla.

Coordinador de Transacciones:

Coordina la ejecución de varias transacciones tanto locales como distribuidas que se inician en la localidad.

Para cada transacción, el coordinador debe iniciar la ejecución de la transacción, dividir la transacción en varias subtransacciones que se distribuirán en las localidades apropiadas para su ejecución y coordinar la terminación de la transacción, ya sea para que quede ejecutada o abortada en todas las localidades.

En un sistema centralizado no se necesita subsistema coordinador.

Fallas:

Un sistema distribuido puede sufrir cualquiera de las siguientes fallas:

- ✓ de disco, de memoria, típicas de una base de datos no distribuida
- ✓ fallo total de una localidad
- ✓ interrupción de alguna línea de comunicación

- ✓ pérdida de mensajes
- ✓ fragmentación de la red

El sistema debe detectar cualquiera de estos fallos, reconfigurarse, recuperar la falla y proceder su ejecución.

Por lo general no es posible distinguir entre la interrupción de una línea de comunicación, la falla total de una localidad o la pérdida de mensajes de la red, pero se debe poder detectar que existe un fallo.

En el momento en el cual la localidad detecta que existe un fallo, debe iniciar un procedimiento de reconfiguración del sistema que permita continuar normalmente.

El procedimiento consiste en los siguientes pasos:

- ✓ Si en la localidad que está fuera de servicio se almacena información replicada, debe actualizarse el diccionario de la base de datos de manera que las consultan no hagan referencia a la copia que se encuentra en dicha localidad.
- ✓ Si en el momento de la falla existían transacciones activas en la localidad fuera de servicio, se deben abortar.
- ✓ Si la localidad que quedó fuera de servicio es un coordinador central de algún subsistema, se debe elegir otro coordinador central.

Al recuperarse de una caída, la localidad debe actualizar todas sus tablas por si los datos sufrieron modificaciones mientras se encontraba inactiva. Si la localidad posee réplicas de datos deberá actualizarlos antes de comenzar a funcionar nuevamente.

Una alternativa a la recuperación es suspender las operaciones mientras esta localidad se recupera pero el coste es mayor.

Protocolos Control de Ejecución de Transacciones

Para garantizar la atomicidad de las transacciones se debe ejecutar uno de los siguientes protocolos que asegura la ejecución consisten de las transacciones organizando la ejecución de los procesos gestores y coordinadores de las localidades.

A continuación analizaremos dos algoritmos de control de ejecución de transacciones.

Commit de dos fases

Sea la transacción T una transacción que se inicia en la localidad L, y sea C el coordinador de dicha localidad.

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

1. C añade el registro <START T> en el transaction log y la graba en memoria estable. Luego envía un START (T) a todas las localidades en las que se ejecutó T. Al recibir el mensaje, cada gestor de transacciones determina si está dispuesto a ejecutar la parte de T que le correspondió. Si no esta dispuesto, añade un registro <ABORT T> en el transaction log local y luego envía un ABORT(T) a C. Si esta dispuesto, agrega un <START > en el transaction log y graba todos los registros en memoria estable. Luego envía un START(T) a C.

2. Cuando todas las localidades respondieron al START(T) de C, C puede determinar si ejecuta o aborta la transacción. Si alguno no contesta luego de un tiempo aborta la transacción y almacena en el transaction log un <ABORT T>, en caso contrario guarda un <EXEC T>. En este momento C envía a todas las localidades un EXEC(T) o un ABORT(T), dependiendo del caso. Al recibir este mensaje cada localidad lo registra en su transaction log.

Algunas implementaciones del protocolo de dos fases finalizan en el momento en que los gestores envían al coordinador un <COMMIT T> al terminar la ejecución de la transacción, y este al recibir los reconocimientos de todas las localidades participantes guarda en el transaction log un COMMIT(t).

Recuperación de fallos del protocolo de dos fases

Casos de fallas:

1. **Falla de una localidad participante:**
   Para determinar el destino de aquellas transacciones que se estaban ejecutando en el momento de la falla, se debe examinar el transaction log. Dada una transacción T, las posibilidades son:

a) El transaction log contiene un registro <EXEC T>. En este caso la localidad ejecuta un REHACER(T).

b) El transaction log contiene un registro <ABORT T>, la localidad ejecuta un DESHACER(T).

c) El transaction log contiene un registro <START T>, en este caso la localidad debe consultar con el coordinador de la transacción para determinar el destino de T. Si el coordinador de T esta activo, notificará a la localidad de la ejecución o el aborto de T y la localidad deberá ejecutar REHACER(T) o DESHACER(T). Si el coordinador de T está inactivo, debe interrogar a las demás localidades participantes para intentar determinar el destino de T. Esto lo hace enviando un mensaje de consulta de estado de T a las demás localidades del sistema. Si una localidad del sistema, recibe un mensaje de consulta de estado, deberá verificar en su transaction log si T se ejecutó allí y cual fue su resultado (si se ejecutó o se abortó) y luego contestar a la consulta. Si ninguna de las localidades contesta sobre el estado de T (puede ocurrir que también las demás localidades participantes de la transacción hayan caído), la localidad que se está recuperando de la falla no podrá ejecutar ni abortar T hasta que no posea información a cerca de su destino. En este caso deberá enviar periódicamente consultar a cerca del estado de T, hasta que alguna localidad que posea información a cerca de ella se recupere. Se debe tener en cuenta que al menos el coordinador de T posee información a cerca de su destino.

d) El transaction log no posee registros sobre la transacción T: esto significa que la localidad falló antes de responder al <START T>, por esto T se deberá abortar y la localidad que falló ejecutar DESHACER(T).

2. **Fallo del coordinador:**

Si falla el coordinador en medio de la ejecución del protocolo de dos fases, las demás localidades participantes de T, deberá decidir el destino de la transacción. Las posibilidades son:

a) Si alguna localidad activa contiene un registro <EXEC T> en su transaction log, entonces T se debe ejecutar.

b) Si alguna localidad activa contiene un registro <ABORT T> en su transaction log, T se deberá abortar.

c) Si alguna localidad activa no contiene un registro <START T>, en su transaction log, entonces el coordinador que falló no puedo haber decidido ejecutar T, esto se debe a que una localidad que no posee un registro <START T>, no envió un mensaje START(T), por lo que es preferible abortar T antes de esperar que el coordinador que falló se recupere.

d) Si todas las localidades participantes poseen un registro <START T> y ningún otro registro referente a T, no se puede establecer el destino de T y se debe esperar a que el coordinador que falló se recupere. Esto implica que los datos que T utiliza se encontrarán bloqueados hasta que C se recupere, y no sólo estos sino otros datos necesarios para la ejecución de otras

transacciones que se encuentran a la espera de que T termine. A esto se le llama PROBLEMA DEL BLOQUEO.

3. **Fallo de una línea de comunicación:**

   Cuando falla una línea de comunicación, los mensajes que estaban siendo enviados a través de la línea no llega a sus destinos. Esto puede verse como la falla de una localidad, ya que las localidades que queden aisladas por motivo de la falla de la línea de comunicación no responderán, por esto se puede aplicar el caso del protocolo en el cual falla una localidad, y si el que queda aislado es el coordinador de la transacción, se aplica el caso de falla del coordinador.

4. **Fragmentación de la red:**

   Cuando se fragmenta la red, pueden ocurrir dos cosas:

   a) que el coordinador y todos los participantes de T queden del mismo fragmento de la red, por lo que la falla no tendrá efecto

   b) El coordinador y sus participantes se encuentren en diferentes fragmentos de la red, por lo que los mensajes se perderán y el problema equivale al del fallo de una línea de comunicación.

Commit de tres fases

El protocolo de commit de tres fases está diseñado para impedir la posibilidad de bloqueo en un caso restringido de los fallos posibles. El protocolo requiere que:

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

Para evitar el bloqueo, este protocolo añade una fase más, en la cual se decide el destino de T. Gracias a esto, esta disponible en todas las localidades participantes la información a cerca de T, por lo que se puede tomar una decisión aunque falle el coordinador.

Fases del protocolo

1. C añade el registro <START T> en el transaction log y la graba en memoria estable. Luego envía un START (T) a todas las localidades en las que se ejecutó T. Al recibir el mensaje, cada gestor de transacciones determina si está dispuesto a ejecutar la parte de T que le correspondió.

Si no esta dispuesto, añade un registro <ABORT T> en el transaction log local y luego envía un ABORT(T) a C. Si esta dispuesto, agrega un <START > en el transaction log y graba todos los registros en memoria estable. Luego envía un START(T) a C.

2. Si C recibe un mensaje ABORT (T) de una localidad participante o si se agota el tiempo de espera de respuesta por parte del coordinador, se aborta T, por lo que T no se ejecuta y se guarda ene l transaction log un registro <ABORT T>. Si C recibe un START(T) de todos los participantes, C añade al transaction log un registro <PRE-EXEC T> y lo graba en almacenamiento estable. Luego envía a todas las localidades participantes un PRE-EXEC(T). Cuando una localidad recibe un ABORT(T) o un PRE-EXEC(T), lo graba en el transaction log y envía un reconocimiento a C.

3. Sólo se ejecuta esta fase si la decisión tomada en la fase dos fue de PRE-EXEC(T). Luego de enviar a todas las localidades un mensaje PRE-EXEC(T), C debe espera la respuesta de al menos K participantes reconociendo T. De acuerdo a esto, C envía un mensaje EXEC(T) a todas la participantes y guarda un EXEC(T) en el transaction log.

En algunas implementaciones del protocolo commit de tres fases, cada localidad al recibir un EXEC(T), responde un COMMIT(T) al coordinador. Cuando el coordinador recibe un COMMIT(T) de todas las localidades, guarda en su transaction log un <COMMIT T>.

Recuperación de fallos del protocolo de tres fases

Casos de fallas:

1. **Falla de una localidad participante:**

   Para determinar el destino de aquellas transacciones que se estaban ejecutando en el momento de la falla, se debe examinar el transaction log. Dada una transacción T puede ocurrir:

   a) El transaction log contiene un registro <EXEC T>. En este caso la localidad ejecuta un REHACER(T).

   b) El transaction log contiene un registro <ABORT T>, la localidad ejecuta un DESHACER(T).

   c) El transaction log contiene un registro <START T>, en este caso la localidad debe consultar con el coordinador de la transacción para determinar el destino de T. Si el coordinador responde un ABORT(T), la localidad DESHACER(T), si C responde con un PRE-EXEC(T), la localidad guarda en el transaction log un <PRE-EXEC T> y envía un mensaje de reconocimiento al coordinador, si el coordinador contesta que T se ha ejecutado, se ejecutará REHACER(T). Si C no falla en un período de tiempo establecido, la localidad ejecutará un protocolo de fallo del coordinador.

   d) Si el transaction log contiene un registro <PRE-EXEC T> y ningún registro <ABORT T> o <EXEC T >, la localidad consultará con el coordinador de T, si contesta que ha sido

ejecutada o abortada, la localidad ejecutará REHACER(T) o DESHACER(T). Si C contesta que T está en pre- ejecución, la localidad continuará el protocolo en esta punto. Si C falla al responder dentro del intervalo de tiempo establecido, la localidad ejecutará el protocolo de fallo del coordinador.

2. **Fallo del Coordinador:**

   Cuando el coordinador falla, la localidad que no recibió respuesta ejecuta el "PROTOCOLO DE FALLO DEL COORDINADOR" que consiste en la selección de un nuevo coordinador y cuando el que fallo se recupera asume el papel de una localidad participante.

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 de la red, la única causa posible para esta situación es que fallo el coordinador.

1. Las localidades activas seleccionan un nuevo coordinador utilizando un PROTOCOLO DE SELECCIÓN.

2. El nuevo coordinador CN, envía un mensaje a todas las localidades participantes pidiendo el estado local de T.

3. Cada localidad participante, incluyendo al nuevo coordinador, determinan el estado local de T:

   a) Ejecutada si el transaction log contiene un registro <EXEC T>.

   b) Abortada si el transaction log contiene un registro <ABORT T>.

   c) Lista si el transaction log contiene un registro <START T> y ningún registro <ABORT T> ni <EXEC T>.

   d) Pre-ejecutada si el transaction log contiene un registro <PRE-EXEC T> y ningún registro <ABORT T> ni <EXEC T>.

   e) No lista si el transaction log no contiene ningún registro <START T> ni ningún registro <ABORT T>.

   Cada localidad envía su estado local al nuevo coordinador.

4. Dependiendo de la respuesta recibida, el nuevo coordinador decide si debe ejecutar o abortar T o reinicia el protocolo commit de tres fases.

   a) Si al menos una localidad responde que T fue ejecutada, entonces el nuevo coordinador decide ejecutar T.

b) Si al menos una localidad responde que T fue abortada, entonces se aborta T.

c) Si ninguna localidad responde abortada, y ninguna localidad responde que T fue ejecutada, pero al menos una localidad responde que T fue pre-ejecutada, entonces el nuevo coordinador reinicia el protocolo commit de tres fases, enviando nuevamente mensajes PRE-EXEC(T) a todas las localidades participantes.

d) En cualquier otro caso se aborta T.

El protocolo de fallo del coordinador permite al nuevo coordinador tener conocimientos del estado del coordinador que falló. Si alguna localidad tiene une estado <EXEC T> en su bitácora, el coordinador que falló debe haber decidido ejecutar. Si alguna localidad posee un estado <PRE-EXEC T> entonces el coordinador que falló debe haber decidido pre-ejecutar T, lo que significa que todas las localidades deben haber alcanzado el estado <START T>. Por esto es seguro ejecutar T, pero se puede producir el problema del bloqueo si el nuevo coordinador también falla, por lo que el coordinador reanuda la fase tres.

Si ninguna localidad recibió un PRE-EXEC(T), es probable que el coordinador antes de fallar haya decidido abortar T antes de causar un bloqueo.

Es importante considerar que la falla no se debe a fragmentación de la red, por que ene este caso, se tendrían dos coordinadores que tomaran diferentes decisiones.

Es importante considerar que no todas las localidades pueden fallar a la vez, ya que si fuera así, sólo la última en caer la que podría tomar una decisión. Esto conduciría a un problema de bloqueo, ya que las demás localidades tienen que esperar a que esta última se recupere, además es difícil determinar cuál fue la última en fallar.

Además es importante la elección del parámetro K, ya que estuvieran activos menos de K localidades participantes se produciría un bloqueo.

Comparación del protocolo COMMIT DE DOS FASES y el COMMIT DE TRES FASES

Al ser en general bastante baja la probabilidad de bloqueo no se justifica utilizar el protocolo de commit de tres fases con el costo que este conlleva.

Protocolos de Gestión de Bloqueos

Introducción

En una base de datos las transacciones se ejecutan concurrentemente para utilizar eficientemente los recursos. Para preservar la consistencia de la base de datos, se utilizan mecanismos de control de concurrencia. Los mecanismos de control de concurrencia utilizan el concepto de "serialización" de la ejecución de las transacciones concurrentes. Estos mecanismos planifican una ejecución de las transacciones de forma tal que sea equivalente a una ejecución serial de las mismas. [KUMA 93].

Planificación

Una planificación es una secuencia de ejecución de un conjunto de transacciones.

Planificación en Serie

Una planificación en serie es una secuencia de instrucciones de varias transacciones, donde las instrucciones de una misma transacción aparecen juntas.

Si existen N transacciones, existen N! planificaciones en serie diferentes que concluyen en un estado consistente de la base de datos.

Planificación en Paralelo

Es una secuencia de instrucciones, donde las instrucciones pertenecientes a una transacción no se ejecutan necesariamente juntas.

Una planificación debe tener el mismo efecto que una planificación en serie y debe hacer pasar al sistema de un estado consistente a otro estado consistente.

Conflictos en planificaciones serializables

Una planificación S "equivale en cuanto a conflictos" a otra planificación S', si se puede a partir de S, obtener una planificación S' mediante una serie de intercambios.

Dado un dato Q y operaciones READ y WRITE sobre el mismo, los intercambios posibles dadas dos instrucciones I1 e I2, se detallan a continuación:

a) I1: READ(Q), I2: READ(Q) → si es posible

b) I1: READ(Q), I2: WRITE(Q) → no es posible

c) I1: WRITE(Q), I2: READ(Q) → no es posible

d) I1: WRITE(Q), I2: WRITE(Q) → no es posible

Una planificación S "es serializable en cuanto a conflicto" si es equivalente a una planificación en serie.

Pruebas de seriabilidad en conflictos

Grafos de precedencia

Para determinar si una planificación S es serializable en conflictos, se construye un grafo de precedencias de la misma, donde los vértices son las transacciones y las aristas quedan determinadas por la relación de precedencia entre los vértices. Ej. Si T_0 se ejecuta antes de T_1, entonces una arista del grafo sería $T_0 \rightarrow T_1$.

Si el grafo posee ciclos, la planificación no es serializable en cuanto a conflictos.

Protocolos basados en bloqueo

Existen varios modos de bloqueo de datos:

- ✓ **Compartido**: es el tipo de bloqueo que permite a la transacción que los obtiene, únicamente leer el dato bloqueado.
- ✓ **Exclusivo**: es el tipo de bloqueo que permite a la transacción leer y escribir el dato bloqueado.

Las transacciones piden un bloqueo en el momento adecuado en el dato dependiendo del tipo de operaciones que van a realizar.

Se puede definir una función de compatibilidad con respecto a los bloqueos:

	Compartido	**Exclusivo**
Compartido	SI	NO
Exclusivo	NO	NO

De esto se desprende que si un dato se encuentra bloqueado en modo compartido sólo puede ser bloqueado en modo compartido por otra transacción. Por otro lado, si un dato se encuentra bloqueado en modo exclusivo, no podrá ser bloqueado por otra transacción.

Si una transacción necesita bloquear un dato que se encuentra con un bloqueo exclusivo, debe esperar a que el dato se libere del mismo.

Protocolos de Bloqueo

Protocolo de un único coordinador de bloqueos

El sistema mantiene un único gestor de bloqueos que se encuentra en una localidad L y recibe todas las solicitudes de bloqueo y desbloqueo de datos de la base.

Si una transacción necesita bloquear un dato, enviará a L la solicitud de bloqueo. El gestor de bloqueos determinará si es posible bloquear el dato, y en caso de ser posible se contestará afirmativamente a la localidad que realizó la solicitud, en caso contrario se postergará hasta que pueda atenderse.

Cuando la localidad que solicitó el bloqueo recibe la respuesta afirmativa por parte del gestor de bloqueos puede leer el dato de cualquier localidad donde se encuentre una réplica del dato. Si la operación a realizarse es un escritura, debe escribirse en todas las réplicas.

Ventajas

- ✓ Este protocolo es sencillo de implementar
- ✓ Los bloqueos se manejan sencillamente por que las solicitudes de bloqueo y desbloqueo se realizan en una única localidad.
- ✓ Se produce un cuello de botella.

Desventajas

- ✓ El sistema se torna vulnerable ya que si la localidad donde se encuentra el gestor de bloqueos falla, este se perderá.

Protocolo de la Mayoría

Este protocolo es un enfoque de coordinadores múltiples. Cada localidad posee su propio gestor de bloqueos que administra el acceso a los datos y replicas de datos que se encuentran en su localidad.

Cuando una transacción necesita bloquear un dato que posee N replicas, debe enviar una solicitud de bloqueo a las localidades con réplicas del dato. En cada localidad con replica del dato, el gestor de bloqueos verificará si es posible bloquear el dato inmediatamente, en caso afirmativo responderá que sí al solicitante, en caso contrario demorará la petición hasta que sea posible satisfacerla.

La transacción no operará sobre el dato hasta que no logre que se bloqueen la mayoría de las réplicas (deben contestar afirmativamente la mitad más una de las localidades).

Ventajas

- ✓ La descentralización de la administración de los bloqueos hace que este protocolo carezca de las dificultades que presentaba el protocolo de "único gestor de bloqueos".

Desventajas

- ✓ Es más complicado de implementar que el protocolo anterior. Son necesarios 2 * (n / 2 + 1) mensajes para manejar solicitudes de bloqueo, e igual cantidad de mensajes para atender solicitudes de desbloqueos.

- ✓ Existe posibilidad de deadlock.

Protocolo Preferencial

Este protocolo es similar al protocolo de la mayoría pero diferencia los bloqueos compartidos de los exclusivos.

El sistema mantiene un gestor de bloqueos para cada localidad.

Cuando una transacción necesita bloquear un dato que posee N replicas, debe enviar una solicitud de bloqueo a las localidades con réplicas del dato. En cada localidad con replica del dato, el gestor de bloqueos verificará si es posible bloquear el dato inmediatamente, en caso afirmativo responderá que sí al solicitante, en caso contrario demorará la petición hasta que sea posible satisfacerla.

El gestor de bloqueos determinará que es posible bloquear el dato en forma compartida, si no existe ningún bloqueo exclusivo del dato.

El gestor de bloqueos determinará que es posible bloquear el dato en forma exclusiva, si no existe ningún bloqueo exclusivo ni compartido del dato.

Si el gestor de bloqueos recibe una solicitud que no puede satisfacer, la posterga hasta que sea posible bloquear el dato.

Ventajas

- ✓ Las operaciones de lectura consumen menos tiempo extra.

Desventajas

- ✓ El manejo de bloqueos exclusivos es mucho más complejo de implementar.

PARTE II:

PVM (Parallel Virtual Machine)
Ada

PVM: Conceptos Generales

Parallel Virtual Machine (PVM) es un software que provee un framework unificado con el cual se pueden desarrollar programas de forma eficiente usando el hardware existente [GEIS 94].

PVM permite trabajar con varios sistemas de hardware heterogéneos.

PVM maneja en forma transparentemente el ruteo de mensajes, transferencia de datos y scheduling de tareas a través de arquitecturas de redes diferentes.

Su modelo computacional es simple y general y se adapta para una gran variedad de estructuras de programa.

El usuario escribe su aplicación como un conjunto de tareas cooperativas. Las tareas acceden a los recursos de PVM a través de las librerías de rutinas estándares. Estas rutinas permiten la iniciación y terminación de tareas en la red así como también la sincronización y comunicación entre ellas.

Las primitivas de pasaje de mensajes de PVM son muy variadas y posee constructores fuertemente tipados para la transmisión y el buffering. Entre los constructores de comunicación están los que permiten enviar y recibir estructuras de datos como primitivas de alto nivel (como broadcast, barreras de sincronización y sumas globales).

Las tareas PVM poseen un hilo de control y estructuras dependientes. Desde el punto de vista de la aplicación concurrente, cualquier tarea existente puede dar origen y detener a otras tareas o agregar o borrar computadoras de la máquina virtual. Cualquier proceso puede comunicarse o sincronizarse con otro. El control y las estructuras dependientes pueden ser implementas con constructores de PVM e instrucciones de un lenguaje de control de ejecución.

Se han desarrollado paquetes de software para asistir la programación en PVM en sistemas distribuidos como por ejemplo: P4, MPI y LINDA.

PVM es un conjunto integrado de herramientas y librerías.

Principios de PVM

Los principios fundamentales de PVM son los siguientes:

- ✓ Pool de host: Las tareas de una aplicación se ejecutan en un conjunto de máquinas que son seleccionadas por el usuario para la corrida de un programa PVM. El pool del host puede ser alterado agregando o extrayendo máquinas durante la ejecución (esto es importante con respecto a la tolerancia de fallas).

- ✓ Acceso transparente al hardware: Los programas de una aplicación ven al hardware como una colección sin atributos de elementos de un proceso virtual o pueden probar la ejecución en otras máquinas del pool para ver si es más apropiada, según sus características.

- ✓ Computación basada en procesos: la unidad de paralelismo en PVM es la tarea (un hilo de control independiente que alterna entre computación y comunicación). Como muchas tareas pueden ser ejecutadas en un procesador simple, el hecho de no mapear procesos a procesadores en PVM es implícito.

- ✓ Modelo de pasaje de mensajes explícito: las tareas se comunican explícitamente enviando mensajes.

- ✓ Soporte de heterogeneidad: en términos de maquinas, redes y aplicaciones. El pasaje de mensajes en PVM permite que los mensajes contengan mas de un tipo de datos para intercambiar entre maquinas de distintas representaciones de datos.

- ✓ Soporte de multiprocesadores.

El sistema PVM:

PVM está compuesto por dos partes:

- ✓ Un daemon *pvmd3* (o también llamado *pvmd*) que reside en todas las computadoras que forman parte de la máquina virtual. Cuando el usuario corre una aplicación PVM, este primero crea una máquina virtual para comenzar PVM y luego cualquier aplicación PVM puede arrancar desde un prompt UNIX en cualquier host.

- ✓ Una librería de rutinas de interfaces PVM. Esta contiene rutinas que pueden ser llamadas por el usuario para pasaje de mensajes, sparwing, processes, coordinación entre tareas y modificación de la máquina virtual.

Características de PVM:

- ✓ Para PVM una aplicación es un conjunto de tareas, cada una de estas pueden incluir varias funciones y pueden ser parametrizadas.

- ✓ PVM soporta C, C++ y Fortran

- ✓ Las librerías de interfaces de usuario son implementadas como funciones (al igual que en C).

- ✓ Se pueden definir macros como constantes del sistema y variables globales como *errno* y *pvmerrno.*

- ✓ Cada tarea o task posee un identificador (TID) que es único en toda la máquina virtual. Este identificador es suministrado por PVM y no es modificable por el usuario.

- ✓ Las tareas se comunican vía mensajes entre ellas.

- ✓ Es fácil de configurar y usar.

- ✓ No requiere demasiados recursos.

- ✓ Existen dos variables de ambiente: PVM_ROOT y PVM_ARCH la primera determina el directorio donde se encuentra instalado *pvmd3* y la segunda la arquitectura del host.

El programa PVM:

- ✓ El usuario escribe varios programas secuenciales con llamadas a la biblioteca de PVM. Cada programa es el código de una tarea.

- ✓ Cada programa es compilado por cada arquitectura en el host pool y el archivo objeto es ubicado en un lugar accesible por la máquina virtual dentro del host pool.

- ✓ Para ejecutar una aplicación, un usuario corre la tarea correspondiente y este proceso inicia otras tareas propias de PVM.

ADA

El mecanismo de Rendezvous al igual que el de Remote Procedure Call, son ideales para implementar modelos C/S y son una combinación entre "Pasaje de Mensajes Sincrónico" y monitores.

Generalidades del lenguaje:

- ✓ Los mecanismos de concurrencia de Ada son el Rendevous y las Tasks.
- ✓ Un programa Ada esta conformado por subprogramas, package (TADs[7]) y tasks (PROCESOS).
- ✓ Cada subprograma, package o task posee dos partes: declaración y cuerpo (o implementación). Las cuales pueden compilarse por separado. La parte de declaraciones es visible externamente, pero en cambio la de implementación no.
- ✓ Los subprogramas y package pueden ser genéricos y pueden definirse tipos TASK.

TASKs

La declaración de una Task es como sigue:

```
Task <nombre> is
        declaraciones de entrys
end <nombre>
```

El cuerpo de una Task tiene la siguiente forma:

```
Task body <nombre>
        declaraciones internas e
        implementación de Entrys
end <nombre>
```

El cuerpo puede contener manejadores de excepción.
Tanto el cuerpo como la especificación pueden compilarse por separado.

La declaración de una task crea una instancia de ella. Pueden crearse arreglos de task utilizando tipos TASK, como por ejemplo en el caso de los lectores y escritores:

[7] TADS: Tipos Abstractos de Datos.

```
Task type lector is
 entrys y declaraciones
end lector
 .....
Task type escritor is
 entrys y declaraciones
end escritor
 .....

lectores: array(1..m) of lector

# declaración de arreglo de instancias del tipo tarea: lector #

 escritor: array(1..n) of escritor

# declaración de arreglo de instancias del tipo tarea: escritor #
```

Pueden definirse task en tiempo de ejecución declarando punteros a tipos task (creación dinámica).

Ada no provee un mecanismo para asignación de task a procesadores.
Los parámetros en Ada son pasados por copia: in (valor) o out (valor - resultado).

Entrys

Las entrys son exportadas por la Task que las declara (cada una representa un servicio prestado por el proceso).
Ada también soporta arreglos de entrys (también llamados familias de entrys). Un ejemplo es el caso de los filósofos en el cual tenemos servidores de filósofos y cinco filósofos. Cuando un filósofo está en condiciones de "comer", el servidor debe informarle la situación. Para comunicarse con cada uno de ellos, debe identificarlos (no le sirve la misma entry, pues no es lo mismo enviar un OK al filósofo1 que al filósofo2) por lo que necesita definir un arreglo de ENTRYs (un canal de comunicación entre el servidor y cada filósofo). Veamos:

```
Task server_filosofos is
  Entry puedo_comer (id)();
# se define una entry para cada filósofo #
  ...
end server_filosofos
```

Sentencias de sincronización:

Cuando una task declara una Entry (servidor), esta declarando un servicio que esta presta y otras task (clientes) en el alcance de su especificación pueden invocarla por medio de la sentencia CALL:

```
Task filosofo is
.....
begin
.....
Call server_filosofo.puedo_comer (yo) ()
.....
end filosofo
#El efecto de esta invocación es el pedido por parte del filosofo
#identificador "yo" de permiso para comer al server_filosofo.
```

La task que declara la entry sirve llamados hasta que la entry termine, aborte o alcance una excepción. La task sirve los llamados invocando la sentencia ACCEPT.

```
accept Entry_i (parámetros formales)
sentencias
end accept;
```

El accept demora a la tarea servidora hasta que se invoque a la entry, copia los argumentos de entrada en los formales de entrada, ejecuta las sentencias, copia los parámetros formales de salida en los de salida. Luego el llamador y el servidor continúan ejecutando.

Ejemplo:

```
Task server_filosofo is
.....
begin
.....
accept puedo_comer (i) ()
.....
end accept;
.....
end server_ filosofo
```

El "No determinismo" en Ada:

Para controlar el no determinismo, Ada provee:

- ✓ la sentencia SELECT en su forma más general:

```
select
accept entry_1 () ...end;
or
.....
or
accept entry_n () ...end;
end select;
```

Esta sentencia permite que la task sirva no determinísticamente a alguna de las entrys. En el caso de que la entry sobre la que se invoca accept no haya sido citada por ninguna tarea, la task servidora es demorada hasta que así sea.

✓ Otra forma de select es el **WAIT selectivo**, el cual introduce la comunicación guardada a traves de la sentencia WHEN (más una condición):

```
select
when cond_1=>
accept entry_1 () ...end;
or
.....
or
when cond_n=>
accept entry_n () ...end;
end select;
```

La primera condición de sincronización que se vuelva verdadera será la que determine que entry se servirá primero. En caso que varias condiciones sean verdaderas, se elige no determinísticamente una de ellas.

✓ WAIT selectivo con la alternativa ELSE

```
select
when cond_1=>
accept entry_1 () ...end;
or
.....
or
when cond_n=>
accept entry_n () ...end;
else
.....
end select;
```

- ✓ WAIT selectivo con la alternativa DELAY: esto provee un mecanismo de timeout, pues si dentro del tiempo determinado por el delay, ninguna guarda se vuelve verdadera, el select se termina.

```
select
when cond_1=>
accept entry_1 () ...end;
or
.....
or
when cond_n=>
accept entry_n () ...end;
or
delay (x)
end select;
```

- ✓ WAIT selectivo con la alternativa TERMINATE: útil en el caso en que todas las tareas en Rendezvous con esta terminaron o están esperando una alternativa TERMINATE.

```
select
when cond_1=>
accept entry_1 () ...end;
or
.....
or
when cond_n=>
accept entry_n () ...end;
or
TERMINATE
end select;
```

- ✓ En el caso que una task desee hacer pool de otra entonces tiene un ENTRY CALL condicional.

```
select
call entry_i
else
.....
end select;
```

Si en el momento que la task solicita un servicio el servidor se encuentra disponible, se demora en el entry call, en caso contrario se ejecutan las sentencias del else.

Puede utilizarse para un único CALL.

✓ En el caso que una task desee hacer pool de otra entonces tiene un ENTRY CALL con timeout: se utiliza cuando la task llamadora esta dispuesta a esperar por servicio sólo un cierto tiempo.

```
select
call entry_i
or
delay (x);
.....
end select;
```

Si luego de transcurrido el tiempo x, el llamado no ha sido servido, se ejecutan las sentencias a continuación de la sentencia delay. Puede utilizarse para un único CALL.

Atributo COUNT:

El atributo count (de una entry), devuelve el número de procesos que están esperando ser servidos en esta entry. Es muy útil ya que pueden construirse guardas para evitar demoras del servidor en entrys que no han sido invocadas.

Ejemplo:

```
select
when entry_1'count > 0=>
accept entry_1 () ...end;
or
.....
end select;
```

EJEMPLO 1:

Un proceso sirve pedidos de materiales (mat_1 y mat_2) de dos tipos de procesos A, B y C. los procesos A utilizan mat_1, los procesos B utilizan mat_2 y los procesos C utilizan materiales mat_1 y mat_2. Los procesos del tipo C tienen mayor prioridad que los demás tipos. Se supone los siempre existen materiales disponibles.

```
#especificación de la task correspondiente al proceso server#
TASK SERVER IS

        ENTRY NECESITO(tipo: char) (IN cant: int)              #arreglos de entrys#

        ENTRY NECESITO_C (IN mat: char, IN cant: int)

        ENTRY IDENTIFICADOR (tipo: char) (OUT id: int)         #arreglos de entrys#

END SERVER;

#especificación de la task type (tendrá varias instancias)  correspondiente a los procesos tipo A #
TASK TYPE PROCESO_A IS

END PROCESO_A ;

#especificación de la task type correspondiente a los procesos tipo B #
TASK TYPE PROCESO_B IS

END PROCESO_B;

#especificación de la task type correspondiente a los procesos tipo C #
TASK TYPE PROCESO_C IS

END PROCESO_C;


TASK BODY PROCESO_A IS      # implementación de la task type #
        YO: char
        CANT: int
                CALL SERVER.IDENTIFICADOR(A) (YO);
                LOOP
                        CALL NECESITO(A)(CANT);
                        <consume material>
                END LOOP;
END PROCESO_A ;

TASK BODY PROCESO_B IS      # implementación de la task type #

        YO: char
        CANT: int
```

```
            CALL SERVER.IDENTIFICADOR(B) (YO);
            LOOP
                  CALL NECESITO(B)(CANT);
                  <consume material>
            END LOOP;
END PROCESO_B ;

TASK BODY PROCESO_C IS      # implementación de la task type #
      YO: char
      CANT: int
            CALL SERVER.IDENTIFICADOR(C) (YO);
            LOOP
                  CALL NECESITO_C (A, CANT);
                  <consume material>
                  CALL NECESITO_C (B, CANT);
                  <consume material>
            END LOOP;
END PROCESO_C ;


TASK SERVER IS          # implementación de la task server #
      PA: int:= 0;
      PB: int:= 0;
      PC: int:= 0;
      Y:i nt            #inicializada con el total de procesos A, B y C#.

#asignación de identificadores a los distintos tipos de procesos por orden de llegada#
FOR  Y=1 TO CANT_TOTAL DO

SELECT
      WHEN IDENTIFICADOR_A 'COUNT > 0      #ejemplo del uso del atributo count#
                  ACCEPT IDENTIFICADOR (A) (ID)
                        ID:=PA+1;         # asignación de los parámetros formales
                                          a los de salida #
                  END ACCEPT;             #llamador y servidor continúan ejecutando #
OR
      WHEN IDENTIFICADOR_B 'COUNT > 0
                  ACCEPT IDENTIFICADOR (B) (ID)
                        ID:=PB+1;
                  END ACCEPT;
OR
      WHEN IDENTIFICADOR_C 'COUNT > 0
                  ACCEPT IDENTIFICADOR (C) (ID)
                        ID:=PC+1;
                  END ACCEPT;
END SELLECT;

END LOOP;
```

```
LOOP
SELECT
        WHEN NECESITO_C (T,CANT) 'COUNT > 0    # se atiende a los procesos C sin
importar                                        que procesos A o B estén
esperando #
                        ACCEPT NECESITO_C (T,CANT)
                                <asigno materiales>
                        END ACCEPT;
OR
        WHEN (NECESITO(B) 'COUNT > 0) AND (NECESITO_C 'COUNT = 0)
                        ACCEPT NECESITO(B) (CANT)
                                <asigno materiales>
                        END ACCEPT;
OR
        WHEN (NECESITO(A) 'COUNT > 0) AND (NECESITO_C 'COUNT = 0)
                        ACCEPT NECESITO(A) (CANT)
                                <asigno materiales>
                        END ACCEPT;
END SERVER;

END LOOP;

END SERVER;
```

PARTE III:

Especificación del Trabajo Realizado

Introducción

Se trata de investigar el problema del procesamiento distribuido en redes, utilizando un soporte PVM (Parallel Virtual Machine).

El objetivo del trabajo realizado es la implementación de un Ambiente de Prueba para la Simulación de un sistema de "Recuperación y Mantenimiento de Datos" en un sistema de Base de Datos Distribuidos (DDB), en el cual se pueden estudiar y monitorear resultados de variada índole a partir de la ejecución de transacciones.

Las características físicas (con respecto a la arquitectura) y las lógicas (con respecto a los algoritmos de recuperación y actualización) del sistema, pueden ser parametrizados, al igual que particularidades de las transacciones de prueba a ejecutar.

El trabajo consta de tres herramientas importantes a saber, un "Generador de Trazas de Prueba" que permite parametrizar los puntos de interés, el "Simulador" propiamente dicho y un "Generador de Resultados" que permite calcular los resultados de la ultima simulación aun cuando la ejecución de la misma haya sido suspendida.

Dada una traza de simulación, en cuanto a la arquitectura del sistema, podemos identificar a las localidades que intervienen en la misma, el modelo de distribución de datos que posee la base (centralizado o distribuido) y el porcentaje de replicación de los datos (esto es para el modelo distribuido), entre otros. Con respecto a los algoritmos utilizados por el manejador para recuperar y mantener la consistencia de los datos se pueden parametrizar tanto el "algoritmo de control de bloqueos" como el "algoritmo de control de ejecución de transacciones" a utilizar, además de decidir si la actualización de la base de datos se realiza en forma inmediata o diferida durante la ejecución de las transacciones. Por ultimo, al referirnos a las transacciones ejecutadas, se puede definir el porcentaje de las transacciones de la traza de prueba que se ejecutan en forma local o global, el porcentaje de transacciones que solo involucran operaciones de lectura sobre las tablas de la base de datos y la probabilidad de fallo en el algoritmo de control de ejecución de transacciones durante la ejecución de una transaccion en particular (en forma aleatoria).

El sistema se encuentra implementado en C [MCGR 96], con un soporte PVM [GEIS 94] y sobre una plataforma UNIX [KERN 87].

Características de Implementación del Simulador

BIBLIOTECA
FAC. DE INFORMÁTICA

Características del trabajo

Se mencionan a continuación, decisiones de modelización a tener en cuenta como características del trabajo.

1. Mecanismos de almacenamiento de los datos.

Recordemos la clasificación de los mecanismos de almacenamiento de datos[8]:

- ✓ ***Replicación***: El sistema mantiene varias copias idénticas de los datos. Cada copia se almacena en una localidad diferente.
- ✓ ***Fragmentación***: las tablas se dividen en varios fragmentos, cada uno en una localidad diferente. Puede ser vertical u horizontal.
- ✓ ***Replicación y Fragmentación***: se combinan los anteriores, las tablas se dividen en fragmentos y a su vez se pueden mantener copias de los mismos en diferentes localidades.

El estudio de la influencia del porcentaje de replicación de datos en la performance de los sistemas distribuidos es uno puntos fundamentales de este trabajo[9].

2. Modelos de distribución de la Base de Datos:

En el sistema de prueba es posible simular ambientes con las siguientes características:

- ✓ ***Centralizados:*** En estos ambientes, los datos se encuentran en una localidad central, y cualquiera de las otras localidades del sistema puede operar sobre ellos. (No existe replicación). En este tipo de sistemas, la ejecución de las transacciones es local o remota según donde se originan (local si se origina en la localidad central, remota en caso contrario).
- ✓ ***Totalmente Distribuidos:*** En estos sistemas, los datos se encuentran dispersos en distintas localidades pero no existe replicación de los mismos. A diferencia del modelo centralizado, las transacciones se ejecutan en forma local, si operan sólo sobre datos de la localidad, o en forma global, si operan sobre datos que no se encuentran en la localidad.

[8] Véase "Arquitectura C/S y Bases de Datos Distribuidas" (Parte I.)

- ✓ ***Parcialmente Distribuidos:*** En estos sistemas, los datos se encuentran dispersos en distintas localidades, algunos de ellos replicados (el porcentaje de replicación puede variar entre cero y 100). La ejecución de transacciones en estos sistemas, al igual que en los "totalmente distribuidos" puede ser local o global.

3. Operaciones continuas

Cualquier sistema de bases de datos, debe ser capaz de sobrevivir a fallas.

Existen dos tipos de fallas cuya simulación es posible en el ambiente de prueba implementado:

- ✓ **Fallas lógicas:** Este tipo de falla corresponde (en la simulación) a:
  - ✓ Al fallo del proceso coordinador o del proceso gestor de transacciones que intervienen en el protocolo de control de ejecución de transacciones.
  - ✓ Al fallo por bloqueo del protocolo de gestión de bloqueos.
- ✓ **Fallas físicas o del sistema:** Podemos clasificarlas en
  - ✓ fallo de la localidad
  - ✓ interrupción de una línea de comunicación

Las fallas lógicas se pueden simular e inclusive para el caso de fallo del algoritmo de control de ejecución de transacciones, se encuentra parametrizada la probabilidad de falla.[10].

La simulación de fallas físicas se ha diseñado, aunque no se encuentra implementada.[11]

[10] Véase "Generador de Trazas de Prueba" (Parte III)
[11] Véase "Simulación de Fallas Físicas" (Parte III)

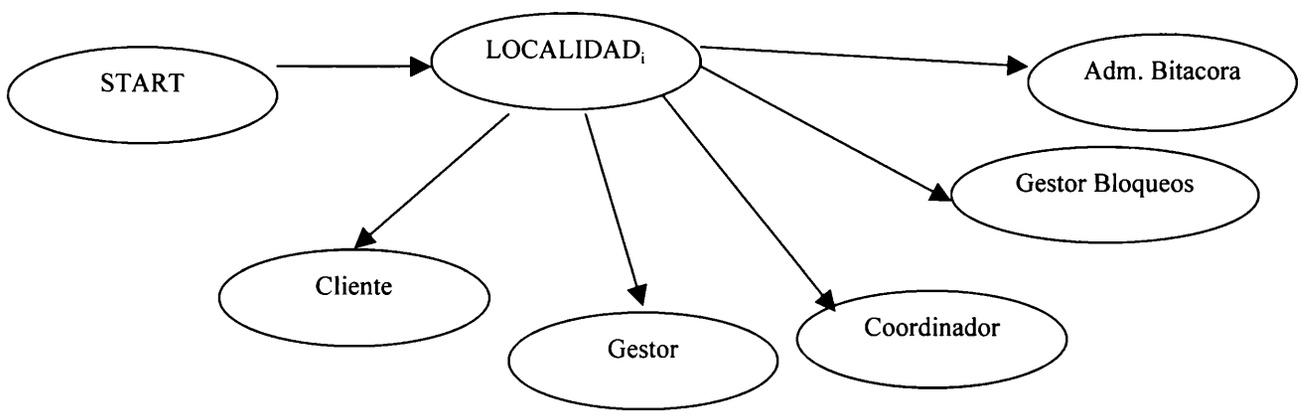
Arquitectura del Sistema de Simulación

Procesos del sistema

En PVM pueden ejecutarse concurrentemente varios procesos que pueden dispararse a su vez en diferentes hosts de la máquina virtual[12].

En cada localidad o (host PVM) residen los siguientes procesos:

- ✓ ***Localidad***: simula una localidad de la DDB.
- ✓ ***Cliente***: simula el ingreso y la ejecución de transacciones por parte del usuario.
- ✓ ***Gestor de transacciones***: es el encargado de la ejecución local de las transacciones locales o de la parte de las transacciones globales que corresponde a su localidad.
- ✓ ***Coordinador de transacciones***: se encarga de controlar la ejecución en paralelo de las transacciones, esto es, que se ejecuten al mismo tiempo en cada una de las localidades participantes para mantener la consistencia de los datos.
- ✓ ***Gestor de Bloqueos:*** es el proceso que se encarga de la administración de los bloqueos en el acceso a los datos de la base, inhibiendo la posibilidad de que el sistema caiga en un estado de deadlock.
- ✓ ***Administrador de la Bitácora:*** se encarga de garantizar la exclusión mutua en el acceso a la bitácora (o transaction Log)



En la figura anterior se muestran los procesos de la simulación. El proceso START es el que dispara cada proceso localidad, en su host de la máquina virtual. El proceso localidad, dispara a su vez a los procesos ***Cliente***, ***Gestor***, ***Coordinador***, ***Gestor de Bloqueos*** y ***Administrador de Bitácora***.

[12] Vease "Parallel Virtual Machine" (Parte II).

Estructuras de datos del sistema

En todo sistema de DDB, es necesaria una estructura de datos para mantener información a cerca de la ejecución de las transacciones, de la red y de las localidades activas.

El sistema posee por cada una de las localidades una "***bitácora***" o transaction log (siempre que el modelo de distribución de datos no sea centralizado, caso en el cual el sistema posee una única bitácora que es mantenida por el administrador de la bitácora de la localidad central). Además de esta estructura de datos, se mantiene un ***"Diccionario de Datos"***. [GARR 97]

- ***Bitácora*** (o Transaction Log):

Es la estructura de datos donde se almacenan registros que denotan el estado de toda transaccion activa o ejecutada.[13]

Por cada operación de escritura en la base de datos, se mantiene una tupla con los datos modificados por la transacción, para poder restablecer el estado anterior de la base de datos en caso de tener que recuperar el sistema de una falla.

- ***Diccionario de Datos de la Base de Datos Distribuida:***

El diccionario de datos de una DDB, es una Base de Datos Distribuida en sí misma, ya que es accedido por todas las localidades para recuperar información general a cerca del sistema distribuido, tal como

- ✓ Localidades del sistema que se encuentran activas.
- ✓ Alias de los datos.
- ✓ Replicas de cada una de las tablas de la base de datos.
- ✓ Procesos Activos en cada localidad.
- ✓ Bloqueos de datos en cada localidad.

El diccionario de datos se implementa con la DDB de PVM[14], cuyos registros son tres-uplas con los campos <CLAVE, INDICE, DATO> y son accedidos por todos los procesos de la máquina virtual.

Localidades activas del sistema

Las localidades activas del sistema se almacenan en el diccionario de datos con la forma:

<LOCALIDAD, ID, TID-localidad[15]>.

---

[13] Véase "Fundamentos de las Bases de Datos" (Parte I).
[14] Véase Parallel Virtual Machine (Parte II).
[15] TID: es el identificador de una tarea en PVM

Donde ID es la identificación de la localidad en la configuración y TID es la identificación de la localidad en el ambiente de prueba.

Procesos activos

Una de las finalidades del diccionario es mantener información a cerca de todos los procesos activos. En la implementación del sistema, la BDD de PVM se utiliza para mantener los identificadores de todos los procesos en cada localidad para que cada uno de ellos tenga una forma de comunicarse con los demás procesos de la simulación.

La información a cerca de los procesos activos del sistema de base de datos y en cada localidad se mantiene con registros de la forma:

<COORDINADOR, TID_LOCALIDAD, TID_COORDINADOR>

<GESTOR, TID_LOCALIDAD, TID_GESTOR>

<CLIENTE, TID_LOCALIDAD, TID_CLIENTE>

<GESTOR_BLOQUEOS, TID_LOCALIDAD, TID_GESTOR_BLOQUEOS>

<ADM_BITACORA, TID_LOCALIDAD, TID_ADM_BITACORA>

Cada proceso almacena su propio TID (o identificador) en la base de datos de PVM al iniciarse.

Replicas de datos

La información a cerca de las réplicas de los datos, se almacena en el diccionario de datos con registros de la forma:

<TABLA, INDICE_REPLICA, LOCALIDAD_CON_REPLICA>

Bloqueos de datos

Los bloqueos de datos en cada localidad, se mantienen en registros de la forma:

<TABLA, TRANSACCION, COORDINADOR> o

<TABLA, TRANSACCION, GESTOR>

Modelización de las Transacciones:

Las transacciones se modelizan como un conjunto de operaciones READ y WRITE, una operación de READ es una operación "select" (u operación de consulta) de una o varias tablas, un WRITE es una operación "insert", una operación "update" o una operación "delete" de una tabla.

Para cada transacción se determinará aleatoriamente por el "Generador de Trazas de Prueba"[16]:

- ✓ La cantidad de operaciones que incluye.
- ✓ La probabilidad de falla en su ejecución del algoritmo de control de ejecución de transacciones[17].
- ✓ La localidad participante que coordina su ejecución (no necesariamente tiene que ser una localidad en la que se encuentren replicas de los datos).

Las actualizaciones de la base de datos pueden ser inmediatas o diferidas. Si la actualización es diferida y la transaccion comprende un INSERT en la tabla T1, entonces se guardara en la bitácora el registro: <WRITE, T1, DATO_VIEJO>. Si la actualización es inmediata se guardará el registro <WRITE, T1, DATO_VIEJO, DATO_NUEVO>

Ejecución de transacciones:

Las transacciones se inician en una determinada localidad. El proceso cliente es el encargado de disparar la transacción[18] cuyo identificador lee desde la entrada standard (el archivo de consultas configurado en el archivo locals.sys para su localidad[19]) y cuyo detalle se encuentra en el archivo de trazas (config.sys)[20].

El cliente envía un "pedido de servicio de atención de la transacción" al proceso coordinador de su localidad (esto lo hace para cada transacción que lee de la entrada standard). El coordinador determina, según el diccionario de datos de la BDD que localidades deberán participar de la ejecución de la transacción. Una vez identificadas las localidades participantes, debe enviar un mensaje a sus gestores para comenzar la ejecución.

Cada gestor al recibir el mensaje de ejecución de la transacción deberá contestar al coordinador si puede ejecutar la transacción o no.

Si un gestor falla en cualquier fase del algoritmo de control de ejecución de transacciones, deberá avisar al coordinador (de ser posible) para que aborte la ejecución. En caso de que un gestor no

[16] Lo determina el "Generador de Trazas de Prueba" y lo registra en el archivo de configuración que genera.
[17] Recordar que sólo se implementaron los fallos lógicos de las transacciones.
[18] Véase "Fundamentos de Bases de Datos" (Parte I)
[19] Véase "Configuración del Ambiente de Prueba" (Parte III).
[20] Véase "Generador de Trazas" (Parte III).

conteste durante un tiempo de timeout, el coordinador abortará la transacción e informará a los gestores que aborten también.

Se implementaron dos algoritmos de Control de Ejecución de Transacciones: el "Commit de dos fases" y el "Commit de tres fases".[21]

Una vez que todos los gestores se encuentran en la fase de ejecución de la transaccion, el coordinador ejecuta una llamada del sistema para "ejecutar la transaccion". Esta llamada invoca al gestor de bloqueos de la localidad que se contactará entonces con los gestores de bloqueos de las demás localidades para coordinar la actualización de las réplicas de los datos involucrados en la transacción.

Esta coordinación, se realiza según el protocolo de bloqueos que corresponda al configurado para la simulación.

Bloqueos

La unidad de bloqueo que se utiliza en el sistema es la "tabla". Si una transacción necesita realizar una operación sobre una tabla, el gestor de bloqueos intentará bloquearla (de acuerdo al protocolo seleccionado[22]) en modo exclusivo o compartido[23].

Especificación general sobre la ejecución del Simulador

Para activar el simulador se debe ejecutar desde la línea de comandos el comando start[24]. Este proceso lee los registros del archivo config.sys para configurar el ambiente de prueba y luego genera los archivos donde se almacenaran los registros de bitácora y los archivos de consultas que leerán los procesos clientes para ejecutar las transacciones.

El proceso start, una vez establecida la configuración ejecuta los procesos "localidad", cada uno en su host de PVM correspondiente[25].

Cada proceso localidad dispara a su vez los siguientes procesos:

- ✓ cliente local
- ✓ gestor de transacciones local
- ✓ coordinador de transacciones local
- ✓ gestor de bloqueos local

[21] Véase "Generador de Trazas" (Parte III).
[22] Se puede optar entre el "Protocolo de la Mayoría" y el "Protocolo Preferencial". Véase "Generador de Trazas de Prueba" (Parte III).
[23] Véase "Algoritmos de Gestión de Bloqueos" (Parte I).
[24] Estando activo PVM.
[25] Según lo configurado en el archivo "config.sys".

- ✓ administrador de la bitácora local

El proceso cliente lee las transacciones del archivo de consultas de su localidad. Para cada transacción invoca al coordinador de su localidad para que coordine la ejecución distribuida de la misma. Según el modelo de distribución de datos configurado, será el protocolo utilizado por el coordinador. Con respecto a la ejecución de las transacciones, si el modelo es centralizado, se utilizará un protocolo centralizado, si es distribuido se utilizará (según e configure) el protocolo COMMIT de dos fases o el COMMIT de tres fases. En estos protocolos intervienen los procesos gestores de transacciones de aquellas localidades donde se encuentran réplicas de los datos involucrados en la transacción. Cuando el modelo es centralizado, el coordinador es el proceso que requiere atención del gestor para la ejecución de su transacción.

Tanto el coordinador de la transacción como los gestores de las localidades involucradas en la misma deben ejecutar localmente la transacción. Para mantener la consistencia de la base de datos, los datos deben ser protegidos en su actualización (se debe garantizar que todas las réplicas de una tabla son actualizadas conjuntamente). Por este motivo, los procesos delegan la ejecución de las transacciones al ***"gestor de bloqueos"*** de su localidad. Este utilizando el ***"protocolo de la mayoría"*** o el ***"protocolo preferencial"***[26] administra los bloqueos de datos.

[26] Véase "Protocolos de Gestión de Bloqueos" (Parte I).

Especificación de los Procesos de la Simulación

Como fue mencionado anteriormente en cada localidad se ejecuta un proceso cliente, un gestor de transacciones, un coordinador, un gestor de bloqueos y un administrador de la bitácora local. A continuación se detalla la especificación de los algoritmos utilizados durante la ejecución de cada uno de ellos.

Algoritmos utilizados por los procesos Gestor y Coordinador de transacciones

El Gestor de transacciones y el Coordinador utilizan uno de los siguientes protocolos de ejecución de transacciones[27]:

- ✓ *COMMIT DE DOS FASES:*
- ✓ *COMMIT DE TRES FASES:*

Algoritmo utilizado por el proceso Gestor de Bloqueos

El Gestor de Bloqueos utiliza uno de los siguientes protocolos de bloqueo[28]:

- ✓ *PROTOCOLO DE LA MAYORIA*
- ✓ *PROTOCOLO PREFERENCIAL*

Ambos protocolos mantienen un gestor de bloqueos en cada localidad (cada uno de ellos administra los bloqueos de los datos o réplicas almacenados en su localidad). La transacción no puede operar sobre el dato hasta que se coloquen bloqueos en mas de la mitad de las copias del mismo[29].

La diferencia entre el "Protocolo de la Mayoría" y el "Protocolo Preferencial" radica en que el segundo da prioridad a las operaciones de lectura, permitiendo dos tipos de bloqueos (exclusivos y compartidos).

[27] Véase "Protocolos de Control de Ejecución de Transacciones" (Parte I)
[28] Véase "Protocolos de Gestión de Bloqueos" (Parte I)
[29] Véase "Protocolos de Gestión de Bloqueos" (Parte I)

Proceso Localidad

```
TASK LOCALIDAD
        Entry Init (in id: integer);
END LOCALIDAD

TASK BODY LOCALIDAD IS
        # precondición: falla=false
        BITACORA

        ACCEPT INIT(id)
                Mio:=id
        END ACCEPT

        IF FALLA →
                GESTOR[mio].Recuperar_caída()

        [] NOT FALLA
                FALLA := TRUE
                                #previniendo fallas
                <iniciar gestores, coordinadores y gestores de bloqueo>
                <inicializar la bitácora>
        END IF
        <consultar>
        <enviar señal de fin a todos los procesos> {mantener una tabla de procesos activos}
        FALLA := FALSE;

END LOCALIDAD
```

Proceso Cliente

```
TASK CLIENTE is
        Entry Init (loc: integer)
        Entry Resultado ()
end CLIENTE;


TASK BODY CLIENTE IS separate

        ACCEPT Init (loc)
                donde:=loc;
        END ACCEPT

        LOOP
                <diseñar transaccion t>
                GESTOR[donde].Transacción(t)
        END LOOP;

END CLIENTE
```

Proceso Gestor

```
TASK GESTOR IS
        Entry Init (loc: integer)
        Entry Transaccion(t: transaction)
        Entry Preparar (t: transaction, coord)
        Entry Ejecutar ()
        Entry Recuperar_caída ()
        Entry Kill()
        Entry Estado? (t: transaction, id: integer)
        Entry Estado_Rehacer (t: transaction)
        Entry Estado_Abortar (t: transaction)
END GESTOR;

TASK BODY GESTOR IS separate
                donde: integer;
                coord: integer;
                trans: transcaction;
                bitácora
                kill: boolean := false

        ACCEPT Init (loc)
                donde:=loc;
        END ACCEPT;
        WHILE NOT kill
        SELECT
                ACCEPT transacción (t)
                        Trans := t;
                END ACCEPT
                COORDINADOR[donde].Iniciar(trans);
        OR
                ACCEPT Preparar (t, c)
                        Trans := t;
                        Coord := c;
                END ACCEPT;
                IF (cree poder ejecutar trans)
                        Lista (bitácora , trans)
                        COORDINADOR [coord].Listo (trans);
                        ACCEPT Ejecutar ()  END ACCEPT;
                        Modificar_BD (trans);   → VER
                        Completa (bitácora, trans)
                        COORDINADOR [coord].Fin ();
                ELSE
                        Abortar( bitácora, trans)
                        COORDINADOR [coord].Abortar ();
                        Deshacer(Bitacora, T) =>deshace la transaccion
```

```
        END IF;
OR
        ACCEPT Recuperar_caída ()
                If(Existe_Registro(Bitacora, (Ejecutar,T)) →
                        Rehacer(Bitacora, T)
                [](Existe_Registro(Bitacora, (Abortar,T)) →
                        Deshacer(Bitacora, T)
                []Not (Existe_Registro(Bitacora, (Abortar,T)) And Not
                (Existe_Registro(Bitacora, (Ejecutar,T))) And Not
                (Existe_Registro(Bitacora, (Lista,T))) →
                        Deshacer(Bitacora, T)
                [](Existe_Registro(Bitacora, (Lista,T)) And Activo(Coordinador(T)))→
                        COORDINADOR[coordinador(t)].estado?(t, id) → VER
                [](existe_registro(bitacora, (lista,T)) and not(activo(coordinador(t)))→
                                                {según tabla de procesos activos}
                        FOR ALL participantes(t) i
                                GESTOR[i].estado? (T, donde)
                        ## espera la primera respuesta acerca del estado de la transacción
                        SELECT
                                ACCEPT Estado_Rehacer (T)
                                        Rehacer (bitácora, T)
                                END ACCEPT
                        OR
                                ACCEPT Estado_Abortar(T)
                                        Deshacer (bitácora, T)
                                END ACCEPT;
                        END SELECT;
                END IF;
        END ACCEPT;             # recuperar caída

OR
        ACCEPT Estado? (T, id)
                IF (Estado (bitácora, T) = "rehacer")
                        ## el que consulta el estado espera la primera respuesta
                        SELECT
                                GESTOR [id]. Estado_Rehacer(T)
                        OR
                                DELAY (timeot)
                        END SELECT
                [] (Estado (bitácora, t) = "abortar")
                        SELECT
                                GESTOR [id]. Estado_Abortar(T)
                        OR
                                DELAY (timeout)
                        END SELECT;
        END ACCEPT;
OR
        ACCEPT KILL()
```

```
                <Termina con su ejecución>
                kill:=true;
            END ACCEPT
    END SELECT
    END DO;
    END GESTOR;
```

Proceso Coordinador

```
TASK COORDINADOR IS
        Entry Init (loc: integer)
        Entry Iniciar (t: transaction)
        Entry Listo ()
        Entry Abortar (id: integer)
        Entry Fin ()
        Entry Kill()
        Entry Estado? (t: transaction, id: integer)
END COORDINADOR;



TASK BODY COORDINADOR IS separate
                trans: transaction;
                yo: integer;
                abortaron: boolean := false;
                terminaron:integer:=0;
                participantes: set of integer;

        ACCEPT Init (loc)
                yo := loc;
        END ACCEPT;

        WHILE NOT (kill)

        SELECT
                ACCEPT Iniciar (T)
                        trans:=T;
                END ACCEPT;
                <Indicar coordinador[donde] ACTIVO>
                FORK
                Preparar (bitácora, trans);
                participantes:= Participantes (trans);
                participantes:= participantes – {yo};

                FOR ALL participantes → j
                        GESTOR [j].Preparar (trans, yo)
                        ## avisa a todos los participantes de la transacción un aviso para que se
preparen
                SELECT
                        ACCEPT Listo ()
                                terminador:=terminadon + 1;
                                ## un participante está listo para comenzar la ejecución
                        END ACCEPT;
```

```
OR      ACCEPT Abortar (id)
            abortaron:=true;
            ## un participante abortó la ejecución
        END ACCEPT;
OR
        DELAY (timeout);

END SELECT

IF abortaron OR timeout
        Abortar (bitácora, trans)
        ## avisar a los demás participantes que deben abortar trans
        FOR ALL participantes → j =/= id
                    GESTOR [j]. Abortar ()
ELSE
        Ejecutar (bitácora, trans)
        ## avisar a los demás participantes que deben ejecutar trans
        FOR ALL participantes (trans) j
            GESTOR [j]. Ejecutar ();

        WHILE NOT (abortaron) AND NOT(contestaron todos los participantes)
            SELECT
                ACCEPT Fin ()
                END ACCEPT;
                ## espera que los participantes terminen la ejecución
            OR
                DELAY (timeout);
            END SELECT
        END LOOP;
        IF NOT timeout
            Modificar_BD (trans, yo); → ver
            Completa (bitácora, trans)
        ELSE
            Abortar (bitácora, trans)
            ## avisa a los demás participantes que aborten trans
            FOR ALL participantes (j) =/= id
                        GESTOR [j]. Abortar ()
            Deshacer (bitácora, trans)
        ENDIF
END IF;
END FORK
<Indicar coordinador[yo] NO ACTIVO> {en la tabla de procesos activos}
OR
        ACCEPT Estado? (t, id)
            ## consulta el estado de t en su botácora
            trans = t;
            IF (Estado (bitácora, trans) = "rehacer")
```

```
                        GESTOR [id]. Estado_Rehacer(trans)
                [] (Estado (bitácora, trans) = "abortar")
                        GESTOR [id]. Estado_Abortar(trans)
            END ACCEPT;
        OR
            ACCEPT KILL ()
                ## termina su ejecución
                kill:=TRUE;
            END ACCEPT

END SELECT;
END LOOP;
END COORDINADOR;



FUNCTION PARTICIPANTES (t: transaction): set
    ##  retorna en un conjunto los identificadores de los participantes de la transacción t
    part= {};
        FOR cada dato d en T
            part:= pat U {localidad (d)}
    return (part).
END PARTICIPANTES


MODIFICAR_BD (TRANS, yo)
    ## modifica la base de datos
    GESTOR_BLOQUEOS[yo].Bloquear(dato (trans,donde), modo (trans), donde)
    <escribir o leer en la DB local según corresponda>
    GESTOR_BLOQUEOS[yo].Desbloquear(dato (trans,donde), modo (trans), donde)
END MODIFICAR_BD
```

Proceso Gestor de Bloqueos

```
TASK GESTOR_BLOQUEOS
        ENTRY Init (in loc: integer)
        ENTRY Bloquear (dato, modo, id)
        ENTRY Desbloquear (dato, modo, id)
        ENTRY Puedo_bloquear (dato, modo, id)
        ENTRY OK_bloqueo (dato, modo, id)
        ENTRY KILL()
END GESTOR_BLOQUEOS


 TASK BODY GESTOR_BLOQUEOS
                kill: boolean:=false;

                lecturas: list of (dato, id_localidad, cantidad_bloqueos, hora_llegada)
                        {en cola de espera}
                escrituras: list of (dato, id_localidad, cantidad_bloqueos, hora_llegada)
                        {en cola de espera}
                OK_bloqueos: integer
                        {para un dato mantiene la cant. de loc. dispuestas a bloquearlo}
                LOCKS[d: dato] of (modo_bloqueo, id_localidad)
                        {para cada dato guarda sus bloqueos}

ACCEPT Init (loc)
        yo := loc;
END ACCEPT;

WHILE NOT kill

SELECT
        ACCEPT Bloquear (d, m, quien)
                dato:=d;
                modo:=m;
                id:=quien;
        END ACCEPT;
        FORK
        IF (modo = "compartido") AND (se puede bloquear) →
                        LOCKS [dato] add: (modo, id)

        [](modo = "exclusivo") AND (se puede bloquear) →
                        ## consulta a las localidades que poseen copias del dato si se puede bloquear
                        FOR ALL Loc_con_copias (dato) → i
                                GESTOR[i].puedo_bloquear? (dato, modo, yo)

                        ## espera que contesten los gestores de bloqueo correpondientes
                        WHILE not (timeout) and (loc_con_copias(dato)>=i)       (*****)
```

```
                    SELECT
                        ACCEPT OK_bloqueo (dato, modo, id)
                            OK_bloqueos := OK_bloqueos + 1;
                        END ACCEPT
                    OR
                        DELAY (timeout)
                    END SELECT
                END LOOP

                IF (loc_con_copias (dato)/2 +1 <= OK_bloqueos)→
                        ## bloquea el dato porque contestó la mayoría
                        LOCKS [dato] add: (modo, id)
                [] (modo = "compartido") AND not (se puede bloquear)→
                        ## es una solicitud de lectura
                        lecturas add: (dato, id, OK_bloqueos, time)
                [](modo = "exclusivo") AND not (se puede bloquear) →
                        ## es una solicitud de lectura
                        escrituras add: (dato, id, OK_bloqueos, time)
                END IF;
END IF;
END FORK;

OR
        ACCEPT Puedo_bloquear?(dato, modo, id)
                IF (modo es compatible) →
                        GESTOR[id]. OK_bloqueo
                        ## en caso de no ser compatible el interesado llegará a agotar su
                        timeout y queda testeando la posibilidad de bloquear en (****)
        END ACCEPT;
OR
        ACCEPT Desbloquear (dato, modo, id)
                LOCKS [dato] remove: (modo, id)
        END ACCEPT

OR
        ACCEPT OK_bloqueo (dato, modo, id)
                IF ((OK_bloqueos + 1) = (loc_con_copias (dato)/2 +1)) →
                        ## puede bloquear el dato por que contestó la mayoría de las
                        localidades
                        LOCKS [dato] add: (modo, id)
                ELSE
                        <registrar un nuevo OK para el bloqueo(dato, OK_bloqueos + 1)>
                END IF
        END ACCEPT;
OR
        ACCEPT KILL
                <Termina con su ejecución>
                kill:=true;
```

```
            END ACCEPT
        ELSE
            <Tomar el sgte. pedido de L/E  s/ orden de llegada (dato, id, bloqueos,
hora_llegada)>
            IF (puede bloquear (dato, id, OK_bloqueos)) →
                GESTOR_BLOQUEOS [id]. OK_bloqueo (dato, modo, id)
                # testeo periódico

END SELECT;
END LOOP;
END GESTOR_BLOQUEOS        ;
```

Bitácora o Transaction log

El proceso localidad mantiene en su bitácora información a cerca de la ejecución de las transacciones en cuya ejecución participa su coordinador o su gestor de transacciones. La bitácora es actualizada por el proceso "***administrador de la bitácora***" y es consultada por el proceso gestor y el coordinador de transacciones.

Los registros de control de transacciones guardan la identificación de la transacción y el estado de la misma. Una transacción durante su ejecución pasa de un estado a otro: cuando se inicia se encuentra en estado START, luego pasa al estado EXEC o COMMIT según el protocolo de ejecución de transacciones utilizado, si falla estando en estado START, EXEC o COMMIT, pasa al estado ABORT.

Cada cierto período de tiempo, la localidad graba el contenido de su bitácora en memoria estable y un registro CHECKPOINT en la bitácora.

Las operaciones posibles sobre la bitácora se especifican a continuación.

PACKAGE BITACORA

START (BITÁCORA,T)

ABORT (BITÁCORA, T)

EXEC (BITÁCORA, T)

COMMIT (BITÁCORA, T)

EXIST_RECORD (BITÁCORA, R): BOOLEAN

STATE (BITACORA, T): ESTADO

UNDO (BITÁCORA, T)

REDO (BITÁCORA, T)

STRUCTURE
    Estado = (PREPARAR, ABORTAR, EJECUTAR, COMPLETAR)
    Bitácora = (Id_transacción, Alias_dato, Valor_antiguo) U (Estado, Id_transacción)
END STRUCTURE

PROCEDURE REDO (BITÁCORA, T)

```
        ## se debe rehacer T en caso de que su Start y su commit se encuentren en la bitácora
END REDO

PROCEDURE UNDO (BITÁCORA, T)
                                → ver porque la modificación de la base de datos es diferida
        ## borrar todos los registros que tengan que ver con T desde su start
END UNDO

PROCEDURE START (BITÁCORA, T)
        #agrega un registro <lista, T> en la bitácora
        Bitácora add: (START, T)
END START

PROCEDURE COMMIT (BITÁCORA, T)
        #agrega un registro <completa, T> en la bitácora
        Bitácora add: (COMMIT, T)
END COMMIT

PROCEDURE ABORT (BITÁCORA, T)
        #agrega un registro <abortar, T> en la bitácora
        Bitácora add: (ABORT, T)
END ABORT


PROCEDURE EXEC (BITÁCORA, T)
        #agrega un registro <ejecutar, T> en la bitácora
        Bitácora add: (EXEC, T)
END EXEC

FUNCTION EXIST_RECORD (BITÁCORA, R): BOOLEAN
        ## busca en la bitácora el registro R y retorna true o false
END EXIST_RECORD

FUNCTION STATE (BITÁCORA, T): ESTADO
        ## retorna si T debe abortarse o deshacerse
        si no se encuentran registros concernientes a T, entonces se abortó, por lo que se debe
        abortar
                STATE = abortar
        si se encuentra el START y el COMMIT se debe rehacer
                STATE = rehacer
END STATE

END BITACORA
```

Implementación de Fallas

Fallas Lógicas

Se simulan distintos tipos de fallas, utilizando la DDB de PVM.

- ✓ Fallas lógicas: se simulan haciendo fallar a los procesos gestor y coordinador de las localidades. En este caso, existen dos situaciones de falla:

- ✓ El proceso falla y no puede continuar con la ejecución de la transaccion por cuestiones concernientes a la BD, entonces avisa a todos los participantes que aborten, si es el coordinador o avisa al coordinador de la transaccion correspondiente si es el gestor.

- ✓ El proceso falla y no puede continuar con su propia ejecución, por lo que no tiene posibilidad de avisar a los demás proceso retrasados por su respuesta y entonces los demás abortan por TIMOUT.

Como la probabilidad de fallas lógicas (falla del algoritmo de ejecución de transacciones), pueden parametrizarse, en el momento en el que el proceso "start" inicializa el ambiente de prueba, lee las transacciones a ejecutar y almacena en la base de datos de PVM el identificador de la transacción a fallar, así como las características propias de la falla (proceso que falla, fase del algoritmo de ejecución de transacciones, localidad que la cancela, etc).

El gestor en cada una de las etapas de la coordinación consulta la BD de PVM para ver si existe una tupla que le indica que esa transaccion debe fallar. En caso de que exista dicha tupla, el gestor simula que falla y envía al coordinador correspondiente el mensaje indicado para abortar la transacción. Esto se puede configurar desde el archivo ***"config.sys"***, indicando a continuación del identificador de la transacción y de los participantes en su coordinación, una byte seteado a S o N que indica si debe fallar o no, a continuación un byte indicando el código de la etapa en la que fallar :

(S)→FALLA_START: falla en el momento en el que el gestor recibe el ***"start*** "de parte del coordinador de la transacción.

(E) →FALLA_EXEC: falla en el momento en el que el gestor recibe el EXEC de parte del ***"coordinador"*** de la transacción.

(C) →FALLA_COMMIT: falla en el momento en el que el gestor recibe el COMMIT de parte del ***"coordinador"*** de la transacción.

Y al final un identificador de la localidad que falla.

El formato de las tuplas cuya recuperación permite la simulación de este tipo de falla es el siguiente:

<ID_TRANS, CODIGO_FALLA, TID_LOCALIDAD_FALLA>

donde el código de falla puede ser: FALLA_START, FALLA_EXEC Y FALLA_COMMIT.

Se especifica la función de reconocimiento de falla a continuación.

```
falla (id_trans[MAX_INT], falla)  -- retorna TRUE o FALSE
{
 cc = pvm_lookup(id_trans, falla, &tid);
 if (cc < 0)
        {
        switch(cc)
        {
        case :
        return -1;
        break;
        case :
        return -1;
        break;
        }
        }
 if (pvm_mytid() == tid)
        {
        /*fallo en esta etapa*/
        return 1;
        }
 else
        {
        return 0;
        }
}
```

El proceso gestor, en cada etapa del algoritmo de control de ejecución de transacciones debe controlar:

```
if falla(id_trans, FALLA_START)
        {
        send (mi_localidad, "Falla en la TRANSACCION");
        cc = send(coord_tid, id_trans, ABORT_START);
        if (cc < 0)
                {
                return -1;
                }
        }
if falla(id_trans, FALLA_EXEC)
        {
        send (mi_localidad, "Falla en la TRANSACCION");
        cc = send(coord_tid, id_trans, ABORT_EXEC);
        if (cc < 0)
                {
                return -1;
                }
        }
if falla(id_trans, FALLA_COMMIT)
        {
        cc = send(coord_tid, id_trans, ABORT_COMMIT);
        if (cc < 0)
                {
                return -1;
                }
        send (mi_localidad, "Falla en la TRANSACCION");
        }
```

El proceso coordinador en cada etapa del algoritmo de control de ejecución de transacciones debe controlar:

```
if falla(id_trans, FALLA_START)
        {
        send (mi_localidad, "Falla en la TRANSACCION");
        /*ENVIA A todos los participantes el ABORT_START.
          Preguntar por si nadie envio un ABORT o si no
          fallo*/
        }
if falla(id_trans, FALLA_EXEC)
        {
        send (mi_localidad, "Falla en la TRANSACCION");
        /*ENVIA A todos los participantes el ABORT_EXEC.
          Preguntar por si nadie envio un ABORT o si no
          fallo*/
        }
if falla(id_trans, FALLA_COMMIT)
        {
        send (mi_localidad, "Falla en la TRANSACCION");
        /*ENVIA A todos los participantes el ABORT_COMMIT.
          Preguntar por si nadie envio un ABORT o si no
          fallo*/
        }
```

Fallas Físicas

En este trabajo se ha diseñado la recuperación de las localidades frente a una falla física. Este apartado no se ha implementado en su totalidad.

Una falla física se produce por un error en la localidad o la perdida de una línea de comunicación de la red.

Los procesos que intervienen en la recuperación de una caída son:

- ✓ la localidad
- ✓ el administrador de la bitácora local

Este problema puede implementarse de la forma siguiente: al iniciarse una localidad, esta verifica en su bitácora local la existencia de un registro CHECKPOINT. Si existe un registro CHECKPOINT, esto indica que se inicia correctamente. Si este no existe, esto indica que retorna de una falla física y debe proceder a volver el sistema al estado consistente anterior a la caída. En el momento de la falla puede ser que hayan quedado inconclusas varias transacciones o que se hayan terminado y no se hayan almacenado sus registros en memoria estable.

Refinamiento del algoritmo de recuperación

A continuación se detalla la especificación de la simulación de fallas físicas. El llamado a la función "falla probable" por parte de la localidad simula la lectura de la bitácora para verificar que el último registro sea un CHECKPOINT. Si resulta cierta la situación de fallo, la localidad debe recuperar el sistema, para esto indica al proceso "Administrador de la Bitácora Local" que recupere el sistema en función al contenido de la bitácora.

```
if (falla_probable())
{
    recuperar(adm_tid);
}
```

El Administrador de la Bitácora, al recibir un pedido de recuperación, procede a recuperar la falla. A continuación se detalla el algoritmo de recuperación:

```
int recuperar_falla(bitacora, commit, actualizacion)
 {
   //los protocolos de recuperacion de una transaccion en particular
   //dependen de si la localidad que falla es participante o coordinadora.

   <Verificar ultimo registro de la bitacora>
   If (ultimo_registro == CHECKPOINT)
```

```
        <sigue con su funcionamiento normal>
    else
     {
     while (haya transacciones sin commit (trans))
          {
             if ((coordinador(bitacora, trans)) == coordinador_de_transaccion)//la loc coord es mi
padre
               {
                 protocolo_coord(bitacora, trans, commit, actualizacion);
               }
            else
               {
                 protocolo_participante(bitacora, trans, commit, actualizacion);
               }
          }
     }

int protocolo_participante(bitacora, t, esta, commit, actualizacion)
     {

          if (commit == 2)
           {
             if (actualizacion == 0) //inmediata
                {
                 if (state_exec(bitacora, t))
                   rehacer(bitacora, t);
                 else
                   {
                        if (state_abort(bitacora, t))
                          deshacer(bitacora, t);
                        else
                          {
                            if (state_start(bitacora, t))
                                {
                                  local_coord = coordinador(bitacora, t);
                                  if (activo(local_coord))
                                    {
                                         switch(estado(local_coord, bitacora, t))
                                          {
                                           case EXEC:
                                             rehacer(bitacora, t);
                                             break;
                                           case ABORT:
                                             deshacer(bitacora, t);
                                             break;
                                          }
                                    }
                                  else //localidad coordinadora de t no activo
```

```
                        {
                            switch(estado(0, bitacora, t))
                             {
                               // proceso = 0 --> todos los participantes
                              case EXEC:
                               rehacer(bitacora, t);
                               break;
                              case ABORT:
                               deshacer(bitacora, t);
                               break;
                              }
                        }
                    }
                  else //no existe estado de t en la bitacora
                    {
                      deshacer(bitacora, t); //pues se debe abortar su ejecusi¢n
                    }
            }//el estado no es abortar
            }//el estado de t no  es exec
        }
    else //actualizacion == 1 (diferida)
        {
          if (state_commit(bitacora, t))
           rehacer(bitacora, t);
          //else no rehace porque la BD no se modifico
        }
 }
else //commit de 3 fases
 {
   if (actualizacion ==  0) //inmediata
        {
          if (state_exec(bitacora, t))
           rehacer(bitacora, t);
          else
           {
              if (state_abort(bitacora, t))
               deshacer(bitacora, t);
              else
               {
                 if (state_start(bitacora, t))
                    { //consulta el estado al coordinador de t
                      switch (estado(coordinador(bitacora, t), bitacora, t))
                       {
                       case EXEC:
                           rehacer(bitacora,t);
                           //enviar mensaje de reconcocimeinto a coord
                           break;
                       case ABORT:
```

```
                                deshacer(bitacora, t);
                                break;
                            case COMMIT:
                                rehacer(bitacora, t);
                                break;
                            //case NO_RESPONDE:
                                //ejecuta protocolo de fallo del coordinador
                                //break;
                            }
                        }
                    else
                        if (state_exec(bitacora, t))
                        {
                          switch (estado(coordinador(bitacora, t), bitacora, t))
                            {
                            case EXEC:
                                rehacer(bitacora,t);
                                //enviar mensaje de reconcocimeinto a coord
                                break;
                            case ABORT:
                                deshacer(bitacora, t);
                                break;
                            case COMMIT:
                                rehacer(bitacora, t);
                                break;
                            //case NO_RESPONDE:
                                //ejecuta protocolo de fallo del coordinador
                                //break;
                            }

                        }
                    else
                        {
                        }
                }
            }
        else //actualziacion == 1 diferida
            {
              if (state_commit(bitacora, t))
                rehacer(bitacora, t);
              //else no rehace porque la BD no se modifico si no se
              //guardo el commit
            }
    }
}

int protocolo_coord (bitacora, t, esta, commit, actualizacion)
```

```
{
  if (commit == 2)
   {
      if (actualizacion == 0) //inmediata
       {
         if (esta == EXEC) //alguna localidad activa participante
                          //contiene el estado EXEC
            ejecutar(bitacora, t);
         else
            {
              if (esta == ABORT) //alguna localidad activa participante
                              //contiene el estado ABORT
                abortar (bitacora, t, pvm_mytid());
              else
               {
                  if (esta == NO_START)//alguna localidad activa participante
                   {              //contiene no contiene un registro START
                     abortar(bitacora, t, pvm_mytid());
                   }
                  else
                   {
                     if (esta == NINGUN_START)  //Ninguna localidad activa participante
                      {                    //contiene el estado ABORT
                          //esperar_localidad_coordiandora();
                          abortar(bitacora, t, pvm_mytid());
                      }
                   }
               }
            }
       }
      else //actualizacion == 1 diferida
       {
         abortar(bitacora, t, pvm_mytid());
       }
   }
  else //commit 3
   {
      //se define un nuevo coordinador

      switch (estado(0, bitacora, t))
       {
        case  COMMIT: //al menos un commit
         ejecutar(bitacora, t);
         break;
        case ABORT: //al menos un abort
         abortar (bitacora, t, pvm_mytid());
         break;
        case EXEC:
```

```
            //retomar el protocolo de 3 fases enviando a todos EXEC
            break;
          default: //en todos los demas casos
            abortar(bitacora, t, pvm_mytid());
            break;
          }
        if (actualizacion == 0) //inmediata
          {
            abortar(bitacora, t, pvm_mytid());
          }
        else //actualizacion == 1 diferida
          {
            abortar(bitacora, t, pvm_mytid());
          }
    }
}
```

La probabilidad de falla fisica en general es baja en los sitemas distribuidos, pero es un parametro interesante en el estudio de trazas. En caso de ser implementado el algoritmo de recuperacion, seria interesante adicionarlo como un nuevo parámetro para el "Generador de Trazas de Prueba"[30].

[30] Véase "Generador de Trazas de Prueba" (Parte III)

Generador de Trazas de Prueba

Para poder parametrizar características del sistema distribuidos, se provee junto con el "Simulador" una herramienta que genera trazas de transacciones de prueba. Esta herramienta crea el ambiente de prueba del sobre el cual ejecuta transacciones el simulador. Al ejecutarla se obtiene como resultado una traza en el archivo config.sys que tomará como entrada el proceso start para comenzar la simulación.

Antes de ejecutar el generador de trazas, se debe cargar en el archivo locals.sys la información a cerca de las localidades participantes de la simulación.

Los parámetros requeridos para la generación de la traza son los siguientes:

- ✓ Cantidad de tablas de la base de datos
- ✓ Cantidad de transacciones de la traza
- ✓ Porcentaje de replicación de los datos:
  - ✓ 100% → Modelo totalmente replicado.
  - ✓ 0 % → Modelo totalmente distribuído.
  - ✓ 0% < X < 100% → Modelo parcialmente distribuido.
- ✓ Modelo de distribución de los datos a representar
  - ✓ Centralizado → Excluye porcentaje de replicación
  - ✓ Distribuido
- ✓ Porcentaje de Transacciones que se solucionan con accesos Locales.
- ✓ Porcentajes de transacciones que involucran sólo lecturas en la Base de datos
- ✓ Protocolo de Control de Ejecución de Transacciones
  - ✓ Commit de dos fases
  - ✓ Commit de tres fases
- ✓ Protocolo de Gestión de Bloqueos

- ✓ Protocolo de la Mayoría
- ✓ Protocolo Preferencial

✓ Actualización de la base de datos

- ✓ inmediata
- ✓ diferida

✓ Probabilidad de fallas lógicas (0 <= X <= 1).

Configuración del Ambiente de Prueba

Los pasos a seguir para la configuración del sistema son:

1. Cargar el archivo ***locals.sys*** con las localidades participantes. El archivo contiene el siguiente formato:

ID. DE LOCALIDAD	HOST	ARCHIVO DE CONSULTAS	TRANSACTION LOG	RESULTADOS	BLOQUEOS
1	127.0.0.1	/usr/local/consultas.dat	/root/bitacora.dat	/root/RESUL1	bloqu1
2	localhost	/usr/local/pvm3/cons.dat	/bitacora.dat	/resultado_1	../bloqueos4

El identificador de la localidad es un numero entero mayor o igual a uno.

El host es la dirección física o lógica de una máquina que se encuentre dentro de la máquina virtual de PVM.

El camino del archivo de consultas es el nombre del archivo de donde se leen las transacciones a ejecutar sobre la base de datos (este archivo es generado automáticamente por proceso de simulación).

El camino de la bitácora es el nombre del archivo donde se almacenaran los registros de control correspondientes al transaction log (este archivo es generado automáticamente por el Simulador).

El camino de resultado es el nombre del archivo donde se almacenaran los registros de resultado de la transacción (si termino con éxito o no y cuanto fue su duración en tp).

El camino de bloqueos donde se almacenan los bloqueos de cada localidad, indicando transaccion y operación a bloquear o desbloquear.

Los campos deben estar separados por blanco.

2. Una vez cargado este archivo, lo cual es fundamental para la simulación se debe ejecutar el ***"Generador de trazas de Prueba"***. Este toma como entrada las localidades configuradas en ***"locals.sys"*** y otros parámetros requeridos durante su ejecución. El resultado del generador es una traza de prueba en el archivo "config.sys". Este archivo contiene el detalle de la traza a ejecutar (las características del ambiente de prueba, las tablas de la base de datos simulada y sus realizaciones, las transacciones y las operaciones que involucran sobre los datos).

3. Ejecutar PVM.

4. Ejecutar el comando "start".

5. Verificar los resultados[31] de la simulación consultando los archivos de salida.

Resultados de la Simulación

Al terminar la simulación se pueden auditar los resultados, analizando el contenido de varios archivos a continuación mencionados, algunos de ellos son archivos que son utilizados durante la simulación por el sistema, otros son meramente de resultado.

a) Archivo de Configuración de localidades (***locals.sys***).

b) Archivo de Configuración de la traza de prueba (***config.sys***).

c) Archivo de Simulación (***simula.dat***)

d) Archivos de Consultas (existe uno por localidad).

e) Archivos de Resultados (existe uno por localidad).

f) Archivos de Bloqueos (existe uno por localidad).

g) Archivos de Bitácora (existe uno por localidad).

1. Archivo de configuración de las localidades de la simulación.

Este archivo es configurado por el usuario y en el se detallan los parámetros básicos para la simulación y la generación de trazas de prueba.

Estos parámetros fundamentales son los siguientes y se detallan para cada localidad de la simulación:

- ✓ La identificación de la localidad (un número mayor a 1).
- ✓ Identificación física o lógica de la localidad (nombre del host o IP).
- ✓ Camino del archivo de consultas de la localidad (entrada estándar).
- ✓ Camino del archivo de bitácora o transaction log de la localidad.
- ✓ Camino del archivo de resultados de la localidad.
- ✓ Camino del archivo de bloqueos de la localidad.

[31] Véase "Resultados de la Simulación" (Parte III).

2. Archivo de configuración del ambiente de prueba y la traza de Prueba.

Este archivo es creado por el "Generador de trazas de prueba", el cual necesita como base el archivo "***locals.sys***" para saber el número de transacciones del ambiente de prueba y la identificación de las mismas.

En este archivo se encuentra información a cerca de:

- ✓ Modelo de Distribución de datos <D>istribuido o <C>entralizado
- ✓ Protocolo de Control de Ejecución de Transacciones utilizado Commit <2> o <3> fases
- ✓ Protocolo de Control de Bloqueos <M>ayoria o <P>referencial
- ✓ Actualización de la base de Datos <0> Inmediata o <1> Diferida
- ✓ Porcentaje de replicación entre 0 y 100 %
- ✓ Porcentaje de Accesos locales entre 0 y 100 %
- ✓ Porcentaje de Transacciones de solo lectura entre 0 y 100 %
- ✓ Cada una de las tablas de la simulación
- ✓ Localización de las réplicas de cada tabla.
- ✓ Cada transacción de la simulación.
- ✓ Localidad Coordinadora de cada transacción.
- ✓ Las operaciones que involucra cada transacción y sobre que tabla.
- ✓ Si la transacción falla o no, en que etapa del protocolo de Control de Ejecución de Transacciones y que localidad es la que cancela la transacción.

Ejemplo: los siguientes tres registros del archivo ***config.sys*** indican que la tabla ocho se encuentra almacenada en la localidad diez y posee dos replicas mas en las localidades dos y ocho.

```
TABLA 8 LOC 10
TABLA 8 LOC 2
TABLA 8 LOC 8
```

Para ejemplificar como se denota en el archivo "***config.sys*** " la localidad en la que origina cada transacción veamos los registros siguientes donde se muestra como localidad coordinadora de la transacción uno a la seis.

Las operaciones que involucra cada transacción se ejemplifica con los registros siguientes, en los cuales se indica que la transaccion ciento sesenta y siete, iniciada en la localidad 1, comprende cuatro operaciones sobre diferentes tablas (un DELETE sobre la tabla cuarenta y ocho y tres SELECT sobre las tablas cuarenta, treinta y uno).

```
TRANS 167
COORD 1
TABLA 48 OPERACION 73
TABLA 40 OPERACION 70
TABLA 30 OPERACION 70
TABLA 1 OPERACION 70
```

La posibilidad de que falle el algoritmo de Control de Ejecución de Transacciones y en el que caso positivo la fase en la cual fallara (START, EXEC, COMMIT) o (START, COMMIT) según el protocolo utilizado.

```
FALLA N
```

3. Archivo de resultado de la simulación: "Simula.dat"

En este archivo se muestran los tiempos medios de acceso de la simulación, al finalizar su ejecución.

En caso de que se interrumpa la simulación se pueden calcular los resultados parciales (con respecto a las transacciones ya ejecutadas) utilizando una herramienta llamada ***"Resul"***. Esta herramienta toma como entrada el archivo de resultado de cada localidad y su salida es el archivo "***simula.dat***".

a) l*os parámetros de la simulación:*

- ✓ Modelo de Distribución de Datos (DISTRIBUIDO, CENTRALIZADO)
- ✓ Algoritmo de Control de Ejecución de Transacciones (COMMIT de 2 Fases, COMMIT de 3 Fases)
- ✓ Algoritmo de Control de Bloqueos (Protocolo de la MAYORIA, Protocolo PREFERENCIAL)

- ✓ Actualización de la Base de Datos (INMEDIATA, DIFERIDA)
- ✓ Porcentaje de replicación [0%,100%]
- ✓ Porcentaje de Accesos Locales [0%, 50%]
- ✓ Porcentaje de Operaciones de Lectura [0%,60%]
- ✓ Localidades de la Simulación (identificación y nombre host)
- ✓ Tamaño de la traza (número de transacciones)
- ✓ Probabilidad de fallas lógicas (con respecto al algoritmo de Control de ejecución de Transacciones).

b) Los resultados por localidad:

- ✓ Tiempo total en tp en transacciones Exitosas
- ✓ Tiempo total en tp en transacciones Abortas
- ✓ Tiempo total en tp en transacciones Canceladas
- ✓ Cantidad de Transacciones Ejecutadas
  - ✓ con éxito
  - ✓ abortadas
  - ✓ canceladas
- ✓ Tiempo medio en tp de acceso en transacciones
  - ✓ con éxito
  - ✓ abortadas
  - ✓ canceladas

c) Resultados generales diferenciando transacciones locales y globales

- ✓ Tiempos medio de acceso en tp en transacciones Locales
  - ✓ con éxito
  - ✓ abortadas

- ✓ canceladas

- ✓ Tiempos medio de acceso en tp en transacciones Globales
  - ✓ con éxito
  - ✓ abortadas
  - ✓ canceladas

d) Resultados generales teniendo en cuenta todas las localidades:

- ✓ Tiempo total en tp de transacciones
  - ✓ con éxito
  - ✓ abortadas
  - ✓ canceladas
- ✓ Tiempo medio de acceso en tp de transacciones
  - ✓ con éxito
  - ✓ abortadas
  - ✓ canceladas

4. Archivos de Consultas

Estos archivos contienen solo los identificadores de las transacciones que se originan en cada localidad. Este archivo interno es leído por el proceso cliente.

Cada lectura del cliente sobre su entrada standard, simula una transacción que es disparada por un usuario en una localidad.

5. Archivos de Resultados

Estos archivos contienen información a cerca del tiempo de ejecución de cada transaccion y el estado en el cual culminó: si resultó exitosa, si fue abortada por falla planificada del algoritmo de Control de Ejecución de Transacciones o si se cancelo por bloqueo.

Su formato es el siguiente:

```
TRANS    2    182350    CANCEL    GLOBAL
TRANS    7    90        EXITO     LOCAL
```

En estos se muestra el número de transaccion, el tiempo en tp consumido, si resulto exitosa, cancelada o abortada y si se trata de una transacción local o global.

Pueden consultarse mientras se corre la simulación.

6. Archivos de Bloqueos

Estos archivos poseen información a cerca del estado de las transacciones y se puede monitorear durante la ejecución.

Los registros de estos archivos poseen la siguiente forma:

```
TABLA    39    TRANS    20    OP    70    1
TABLA    68    TRANS    4     OP    70    1
TABLA    76    TRANS    61    OP    70    0
TABLA    55    TRANS    48    OP    71    1
TABLA    55    TRANS    48    OP    71    0
```

En estos se muestra el identificador de la tabla bloqueada o desbloqueada, el identificador de la transaccion que opera sobre ella, el tipo de operación que se aplica (70: SELECT, 71: INSERT, 72: UPDATE, 73: DELETE) y por último si se trata de un bloqueo (1) o desbloqueo (0).

7. Archivos de Bitácora

Al terminar la simulación, queda en cada localidad el archivo configurado como bitácora con los registros de control correspondientes a la ejecución de toda la traza de transacciones.

Si el modelo es centralizado sólo contiene registros de control la bitácora de la localidad en la que se encuentre el 100% de los datos.

En el caso distribuido cada localidad tendrá su propia bitácora con los correspondientes registros de control.

Ejecución de una traza de ejemplo

En este apartado se analiza la ejecución del "simulador" para una traza de prueba y la generación de resultados.

Dada una traza de trescientas transacciones en una base de datos parcialmente distribuida de quinientas tablas, donde el porcentaje de replicación es del 30% y el 60% de las transacciones son operaciones que involucran sólo lecturas de tablas en la base de datos y el número de transacciones locales predomina sobre el número de transacciones globales a razón de 80% a 20%.

Consideramos que la actualización de la base de datos es diferida y que los algoritmos utilizados son los siguientes:

- ✓ Commit de dos fases para el Control de Ejecución de Transacciones.
- ✓ Protocolo Preferencial para el control de bloqueos.

Lo primero que se debe hacer es completar el archivo de localidades del ambiente de prueba: "***locals.sys***". Si el sistema esta constituido por ocho localidades, el archivo de configuración contendrá ocho registros como los siguientes:

1 lidi /home/consul1.dat /home/bita1.dat /home/resul1.dat bloq1.dat

2 12.130.0.17 consul2.dat /usr/local/bita2.dat /usr/local/resul2.dat /usr/local/bloq2.dat

3 localhost consul3.dat tesis/bita3.dat /usr/local/pvm3/tesis/resul3.dat bloq3.dat

4 localhost /home/mio/consul4.dat /home/mio/tesis/bita4.dat resul4.dat /home/mio/bloq4.dat

5 localhost consul5.dat bita5.dat /usr/local/pvm3/tesis/resul5.dat bloq5.dat

6 localhost consul6.dat bita6.dat /usr/local/pvm3/tesis/resul6.dat /usr/local/pvm3/tesis/bloq6.dat

7 127.0.0.1 consul7.dat bita7.dat /usr/local/pvm3/tesis/resul7.dat /usr/local/pvm3/tesis/bloq7.dat

8 pepe consul8.dat bita8.dat resul8.dat /usr/local/pvm3/tesis/bloq8.dat

Luego debemos ejecutar el "Generador de Trazas de Prueba". Luego obtenemos una traza como la siguiente en el archivo "***config.sys***"

```
MODELO D
COMMIT 2
BLOQUEOS P
ACTUALIZACION_BD 1
REPLICACION 30
ACCESOS_LOCALES 50.000000
LECTURAS 70
PROBA_FALLA_FISICA 0.000000
TABLA 1 LOC 5
TABLA 1 LOC 2
TABLA 1 LOC 9
TABLA 1 LOC 8
TABLA 1 LOC 6
TABLA 2 LOC 4
TABLA 2 LOC 9
TABLA 2 LOC 2
TABLA 3 LOC 7
TABLA 3 LOC 8
TABLA 3 LOC 6
 TABLA 498 LOC 8
 TABLA 499 LOC 6
 TABLA 500 LOC 5
 TRANS 1
 COORD 2
 TABLA 190 OPERACION 72
 FALLA N
 TRANS 2
 COORD 1
 TABLA 57 OPERACION 70
 TABLA 83 OPERACION 70
 FALLA N
 TRANS 3
COORD 8
 TABLA 470 OPERACION 70
 FALLA N
 TRANS 4
 COORD 4
 TABLA 35 OPERACION 70
 TABLA 39 OPERACION 70
 TABLA 75 OPERACION 70
 TABLA 68 OPERACION 70
 FALLA N
 TRANS 5
 COORD 2
 TABLA 29 OPERACION 70
 TABLA 80 OPERACION 72
 FALLA N
```

```
.
.
TRANS 296
COORD 7
TABLA 197 OPERACION 73
FALLA N
TRANS 297
COORD 2
TABLA 199 OPERACION 70
FALLA N
TRANS 298
COORD 3
TABLA 282 OPERACION 72
FALLA N
TRANS 299
COORD 2
TABLA 388 OPERACION 72
FALLA N
TRANS 300
COORD 5
TABLA 459 OPERACION 70
FALLA N
.
```

Luego de obtener el archivo de configuración del ambiente de simulación, se debe ejecutar PVM y luego el proceso "start" para comenzar la simulación.

Al terminar la simulación por timeout de todos los procesos, el proceso "start" genera el archivo final ***"simula.dat"***. En este archivo se detallan los tiempos promedios de ejecución de las transacciones locales y globales por localidad y en general.

Durante la simulación se pueden monitorear el resto de los archivos para verificar y seguir su funcionamiento.

El contenido de los archivos de bitácora tiene la siguiente apariencia:

```
2 LOC:262147  TRANS: 209  TID:263374
1 START       TRANS: 209  TID:263374
1 COMMIT      TRANS: 209  TID:263374
--- CHECKPOINT ---
1 ABORT       TRANS: 219  TID:263297
2 LOC:262147  TRANS: 212  TID:263382
1 START       TRANS: 212  TID:263382
1 COMMIT      TRANS: 212  TID:263382
2 LOC:262147  TRANS: 225  TID:263378
1 START       TRANS: 225  TID:263378
1 COMMIT      TRANS: 225  TID:263378
2 LOC:262147  TRANS: 201  TID:263398
1 START       TRANS: 201  TID:263398
1 COMMIT      TRANS: 201  TID:263398

2 LOC:262147  TRANS: 224  TID:263404
1 START       TRANS: 224  TID:263404
1 COMMIT      TRANS: 224  TID:263404
2 LOC:262147  TRANS: 222  TID:263423
1 START       TRANS: 222  TID:263423
--- CHECKPOINT ---
1 COMMIT      TRANS: 222  TID:263423
--- CHECKPOINT ---
2 LOC:262147  TRANS: 230  TID:263428
1 START       TRANS: 230  TID:263428
1 COMMIT      TRANS: 230  TID:263428
```

Se observan diversos tipos de registros: los registros identificados con "1" denotan el estado de las transacciones en ejecución. Los registros identificados con "2", son los registros de inicio de transacción, en el cual se identifica la localidad en la que se inicio la transacción y el identificador del coordinador de la transacción. El idenficador del coordinador de una transacción es útil en el caso en el cual una localidad necesite recuperarse de una falla física. En el momento de la recuperación de una falla física, frente a la duda sobre el destino de una transacción se debe consultar al coordinador de la ejecución de la misma.

El contenido de los archivos de resultados, se parece al siguiente conjunto de registros:

```
TRANS    2     18173  CANCEL    GLOBAL
TRANS    7     92     EXITO     LOCAL

TRANS    18    119    EXITO     LOCAL

TRANS    23    115    EXITO     LOCAL
TRANS    25    18053  CANCEL    GLOBAL
TRANS    29    93     EXITO     LOCAL
```

TRANS	130	86	EXITO	LOCAL
TRANS	134	5490	EXITO	GLOBAL
TRANS	147	88	EXITO	LOCAL
TRANS	151	114	EXITO	LOCAL
TRANS	165	109	EXITO	LOCAL
TRANS	183	107	EXITO	LOCAL
TRANS	195	99	EXITO	LOCAL
TRANS	202	18027	CANCEL	GLOBAL
TRANS	213	10206	EXITO	GLOBAL
TRANS	215	229	EXITO	LOCAL

En estos registros se observa para cada transacción, su tiempo de éxito, aborto o cancelación en tps y si se ejecuto localmente o necesito coordinación global.

PARTE IV:

Resultados
Conclusiones
Trabajos Futuros

Estudio de los resultados de las Simulaciones

En esta parte del trabajo se analiza la performance de sistemas de bases de datos con diferentes características de software y de hardware.

El objetivo principal es estudiar la influencia del incremento del porcentaje de accesos locales y del porcentaje de operaciones de sólo lecturas en los sistemas de bases de datos distribuidos y centralizados y la incidencia del factor replicación en sistemas distribuidos.

Métricas de Performance

La performance es un concepto complejo y multifasético. Para medir la performance de un programa concurrente o paralelo se puede considerar el tiempo de ejecución, la escalabilidad del algoritmo computacional, los mecanismos por los cuales se generan los datos, el almacenamiento, la transmisión de mensajes en la red, el acceso a disco y el costo incurrido en las fases del ciclo de vida del software. [FOST 95].

De esta manera, las medidas con las cuales se puede evaluar la performance pueden ser tan diversas como: el tiempo de ejecución, la eficiencia del paralelismo, los requerimientos de memoria, la latencia, la frecuencia de entrada / salida, la utilización de la red, los costos de diseño, los costos de implementación, el potencial de reuso, los requerimientos de hardware, los costos de hardware, los costos de mantenimiento, la portabilidad y la escalabilidad. La importancia de estas métricas es relativa a la naturaleza del problema analizado. En una especificación existen métricas que se adecuan mejor al problema, mientras que algunas se consideran óptimas y otras directamente son ignoradas.

En este trabajo, las métricas de performance elegidas son:

a) El tiempo medio de ejecución de transacciones locales y globales o remotas (según lo que corresponda al modelo de base de datos estudiado).

0

b) El número de transacciones canceladas por bloqueo (sólo para los sistemas distribuidos).

Tiempo de Ejecución de Transacciones

El tiempo de ejecución de una transacción T_j, se calcula sumando el tiempo de computación, de comunicación y el tiempo ocioso que invirtió cada proceso del sistema en la ejecución de T_j. Sea p es el número de procesos del sistema, el tiempo de ejecución de T_j, es:

$$T_{T_j} = \sum_{i=0}^{i=p-1} (T_{computación} + T_{comunicación} + T_{ocioso})$$

Considerando que si un proceso i no interviene en la ejecución de T_j:

$$(T_{computación-i} + T_{comunicación-i} + T_{ocioso-i}) = 0$$

Se consideran como procesos del sistema que intervienen en la ejecución de T_j, al cliente, coordinador de transacciones y gestor de bloqueos de la localidad donde se originó T_j y a los gestores de transacciones y de bloqueos de las localidades donde se encuentra alguna réplica de los datos involucrados en T_j[32].

El tiempo de computación de un algoritmo es el tiempo consumido ejecutando instrucciones del mismo y depende de las características del problema, del número de tareas o procesos que participen en el algoritmo (si es paralelo o concurrente) y de las características de la workstation donde se ejecute en caso que la red este compuesta por workstation heterogéneas.

El tiempo de comunicación de un algoritmo es el tiempo que se consume enviando y recibiendo mensajes. Existen dos tipos de comunicación: interproceso e intraproceso. En la comunicación interproceso, los procesos que se comunican residen en diferentes localidades o workstation. En la comunicación intraproceso, los procesos que se comunican residen en la misma localidad o workstation.

Los tiempos de computación y de comunicación son especificados explícitamente en un algoritmo paralelo o concurrente, el tiempo ocioso del algoritmo es el tiempo excedente que sumado a los dos anteriores determina el tiempo de ejecución.

Tiempo medio de Ejecución de Transacciones

Se considera como tiempo medio de ejecución al cociente entre la sumatoria del tiempo consumido en la ejecución de cada transacción y la magnitud de la traza de prueba. Sea n el número de transacciones de una traza, el tiempo medio de la misma es:

$$T_{medio} = \frac{\sum_{i=0}^{i=n-1} T_{T_i}}{n}$$

Si en un sistema centralizado o distribuido resultan n_1 las transacciones ejecutadas localmente, el tiempo medio de ejecución para transacciones locales resulta:

$$T_{medio\_locales} = \frac{\sum_{i=0}^{i=n_1-1} T_{T_i}}{n_1}$$

siendo T_0 ,.., T_{n1-1} transacciones locales.

Si el sistema de base de datos es centralizado o distribuido y resulta n_2 el número de transacciones ejecutadas en forma distribuida, el tiempo medio de ejecución para las transacciones globales resulta:

$$T_{medio\_globales} = \frac{\sum_{i=0}^{i=n_2-1} T_{T_i}}{n_2}$$

siendo T_0 ,.., T_{n2-1} transacciones globales.

---

[32] Véase "Especificación del Trabajo Realizado" (Parte III)

Definición de conjuntos de prueba

Conjunto de Prueba I: "Modelo de Datos Distribuido"

Se estudian los tiempos medios de acceso de transacciones locales y globales que se ejecutan en un ambiente de base de datos cuya distribución de datos corresponde al "Modelo Distribuido" constituido por diez localidades, donde los datos se encuentran diseminados en diferentes proporciones.

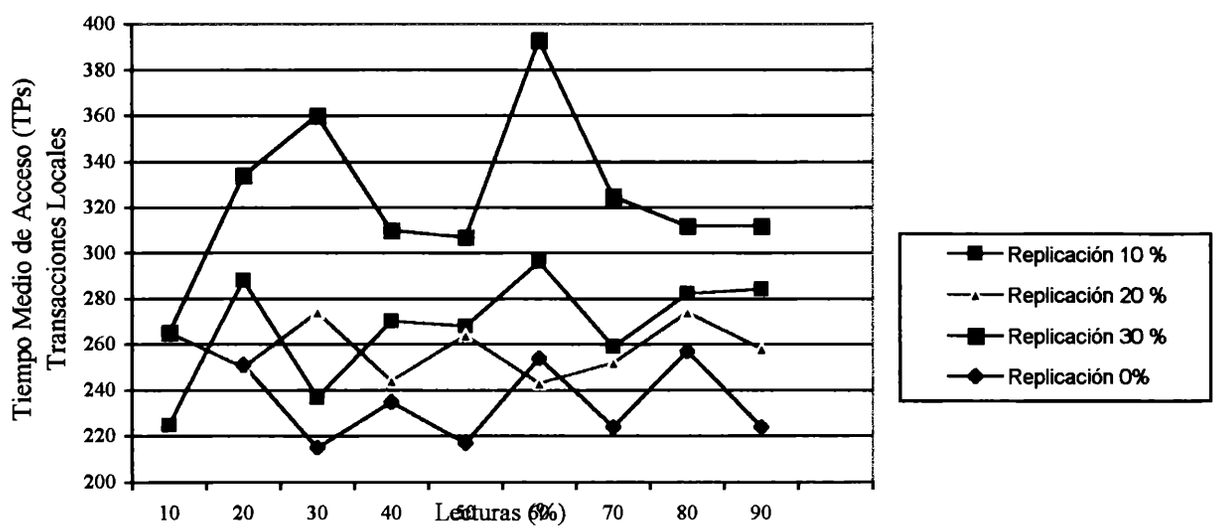
Durante la ejecución de las transacciones, las actualizaciones en la base de datos se realizan en forma "diferida" y el porcentaje de replicación es un parámetro variable.

El "Algoritmo de Control de Ejecución de Transacciones" que utiliza el sistema es el "Protocolo Commit de Dos Fases" y se considera que la probabilidad de que este falle es cero.

El Algoritmo de Gestión de Bloqueos utilizado es el "Protocolo Preferencial" y el porcentaje de fallos del mismo es aleatorio (no puede parametrizarse), por lo que se analiza la cantidad de transacciones canceladas por bloqueo.

Se analiza como influye el porcentaje de replicación en la ejecución de transacciones, estudiando dicho porcentaje para diferente proporción de transacciones locales y diferente proporción de transacciones que implican sólo lecturas de la base de datos.

Las trazas comprenden la ejecución de 300 transacciones en una base de datos de 500 tablas.

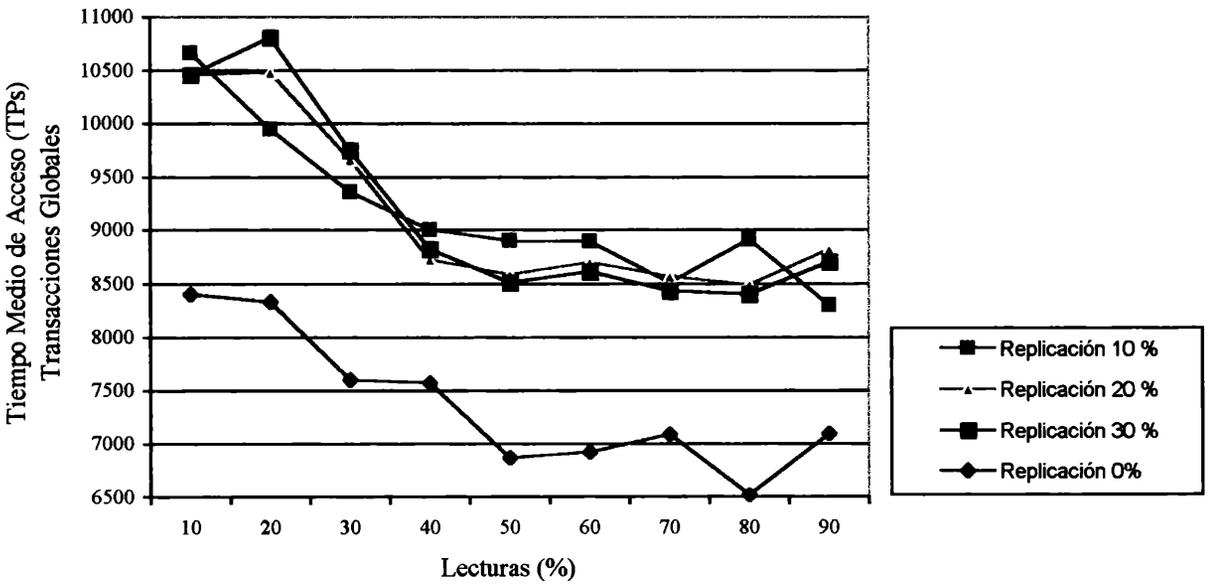


Ejecución de Transacciones Locales (50% Accesos Locales) (1)

(1)

En los modelos replicados, a mayor porcentaje de replicación, los tiempos de acceso son mayores debido al tiempo extra de comunicación que consumen tanto el protocolo de control de bloqueos como el de control de ejecución de transacciones que intervienen en la ejecución de las transacciones globales.

Para el modelo distribuido sin replicación, el tiempo medio de acceso se mantiene con poca variación. Los tiempos de accesos son menores a los observados en modelos replicados, pues la ejecución de las transacciones involucra al menos un conjunto de operaciones remotas, por lo que si bien es necesaria la coordinación de localidades para la ejecución, no es necesario mantener la consistencia de las réplicas..

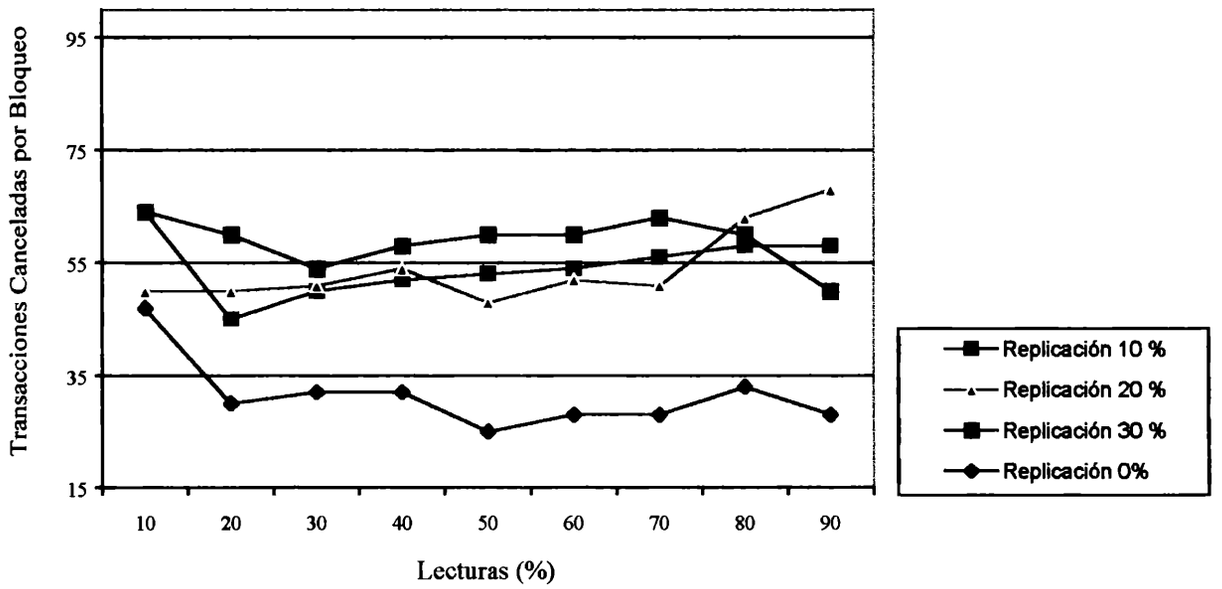


Ejecución de Transacciones Globales (50% Accesos Locales) (2)

(2)

Al aumentar el porcentaje de operaciones de lecturas en la BD, en las trazas de prueba, los tiempos medios disminuyen debido a que las operaciones de lectura consumen menos tiempo de ejecución que las escrituras de la base de datos por que el protocolo de control de bloqueos utilizado ("Protocolo Preferencial") da preferencia a las lecturas.

Las diferencias con respecto a tiempos entre el modelo distribuido con replicación y el modelo distribuido sin replicación se denotan claramente en este gráfico, aunque no difieren considerablemente los modelos correspondientes al 10%, 20% y 30 % de replicación debido a la duración homogénea de las transacciones y la poca diferencia de replicación entre uno y otro modelo.



Transacciones Canceladas por Bloqueo (50% Accesos Locales) (3)

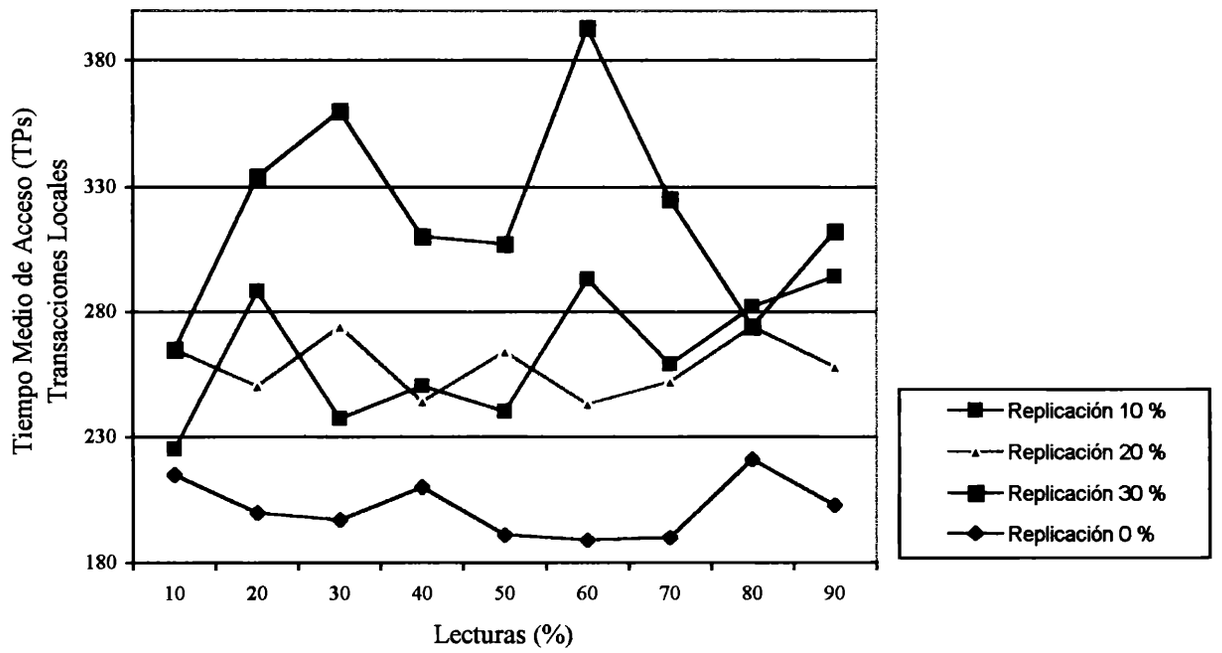
(3)

En este gráfico observamos que a medida que aumenta el porcentaje de replicación, aumenta el número de transacciones que se cancelan por bloqueos. El índice más bajo de transacciones canceladas por bloqueos corresponde al modelo sin replicación.

Para los modelos replicados, la media de cancelaciones por bloqueo es de 59 transacciones, mientras que para el modelo sin replicación, la media es de 35 transacciones.

(1,2,3)

Las ventajas de la replicación no se denotan en estos sistemas replicados (con 50% de accesos locales) debido a que la probabilidad de acceso local o global es coincidente. Al ser demasiado alto el porcentaje de ejecuciones globales, el número de transacciones canceladas por bloqueo aumenta, lo que pesa en los tiempos medios de ejecución en transacciones globales y da mayor fluctuación a las curvas resultantes.

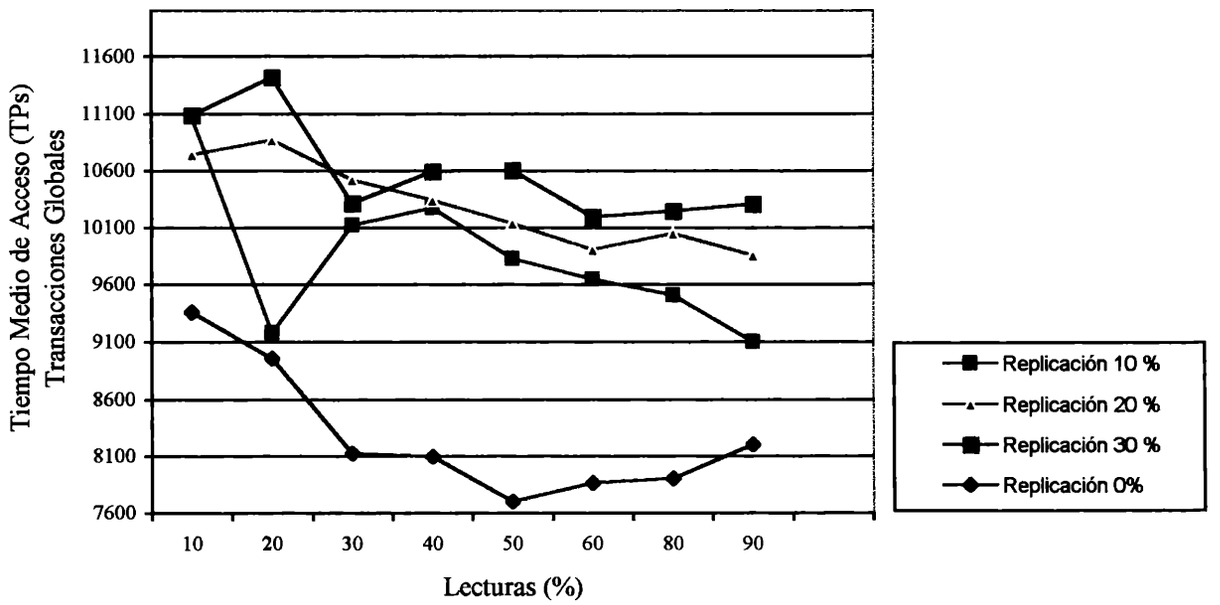


Ejecución de Transacciones Locales (70% Accesos Locales) (4)

(4)

Al aumentar el porcentaje de replicación, la ejecución de las transacciones locales se ve retardada por los tiempos consumidos por los algoritmos empleados en la coordinación de las transacciones globales. Por este motivo es también que se observa que los tiempos no convergen, sino que son inestables y existen puntos atípicos, esta situación es más acentuada cuanto mayor es el porcentaje de replicación.

Con respecto a los sistemas con el 50% de accesos locales, los tiempos medios de ejecución de transacciones locales son menores. Por otro lado, es lógico que la disminución del número de transacciones globales afecte a los tiempos medios, reduciéndolos.

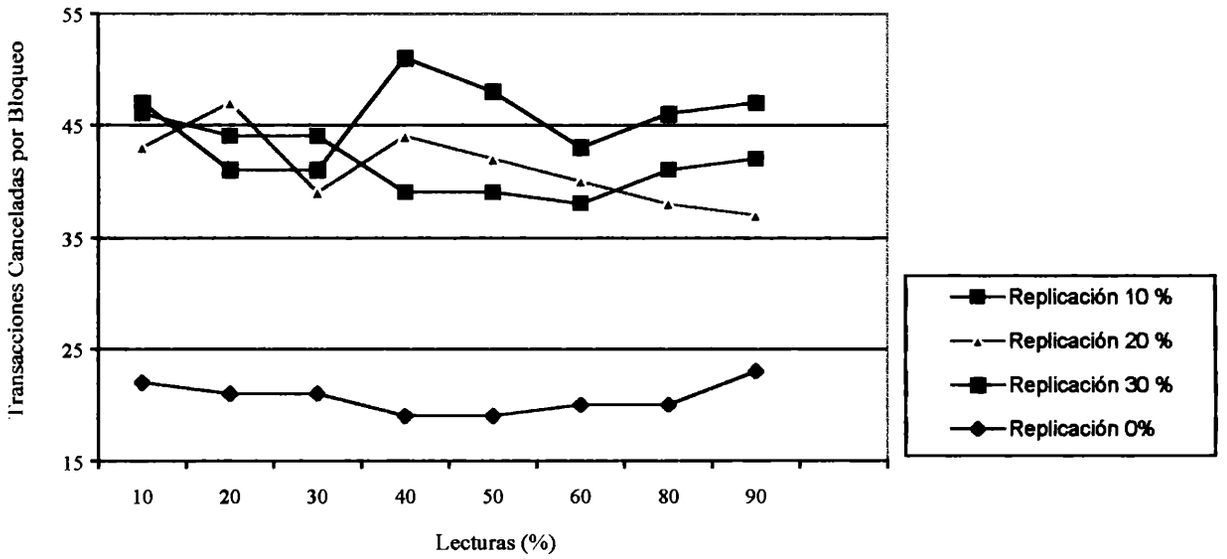


Ejecución de Transacciones Globales (70% Accesos Locales) (5)

(5)

Podemos observar aquí que al aumentar el porcentaje de transacciones que involucran solo lecturas en la base de datos, los tiempos van disminuyendo tanto en el modelo con replicación como en el modelo sin replicación. El 30% de transacciones globales ejecutadas en el modelo sin replicación muestran tiempos mucho menores a los de los modelos con replicación.

Con respecto a los modelos replicados de (2), en estos sistemas con el 70% de transacciones locales, las curvas se diferencian entre ellas y los tiempos medios de ejecución de las transacciones globales son, en general, levemente mayores. En (2) los tiempos oscilan en el intervalo (8250 TPs;11000 TPs), mientras que en (5) en el intervalo (9100 TPs;11600 TPs).



Transacciones Canceladas por Bloqueo (70% Accesos Locales) (6)

(6)

Si comparamos la media de cancelaciones por bloqueo de sistemas replicados como los analizados en (3) y estos con el 70% de transacciones locales, vemos que ha disminuido en un 27%, pues tenemos una media de 38 transacciones canceladas. Con respecto al modelo sin replicación, la media es de 18 transacciones canceladas, frente a 35 de los modelos con 50 % de accesos locales (ha disminuido la cancelación en un 48,5%). Esta importante disminución en el modelo sin replicación (con respecto al estudiado en (3)) se debe a la disminución de transacciones ejecutadas en forma no local.

Los valores para los modelos replicados se mantienen en el intervalo (35, 55), mientras que en los modelos con 50% de accesos locales el intervalo se extiende a (40, 75).

En el modelo sin replicación de (6), las cancelaciones se mantienen inferiores a las 25, mientras que en (3) se mantienen normalmente inferiores a los 35.

(7, 8, 9)

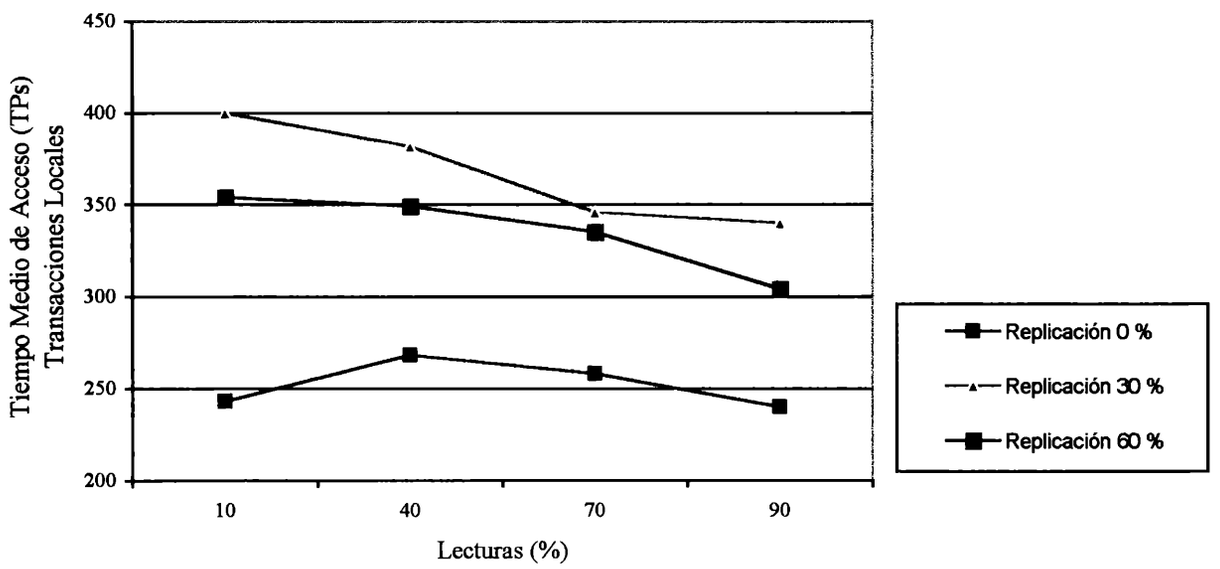
La influencia del incremento del porcentaje de replicación en la recuperación de datos no es notoria en los sistemas anteriores (con 10%, 20% y 30% de replicación). Se estudian en (7), (8) y (9) sistemas replicados en porcentajes con diferencias de mayor magnitud entre ellos y 90% de transacciones locales.

Estamos frente a sistemas de consultas con actualizaciones no frecuentes.

(7)

En estos casos se observa la ventaja de la mayor disponibilidad de datos por parte de las localidades del sistema, notoria en los tiempos promedios de las transacciones locales con una tendencia clara a disminuir a medida que se aumenta el porcentaje de transacciones que involucran únicamente lecturas en la base de datos.

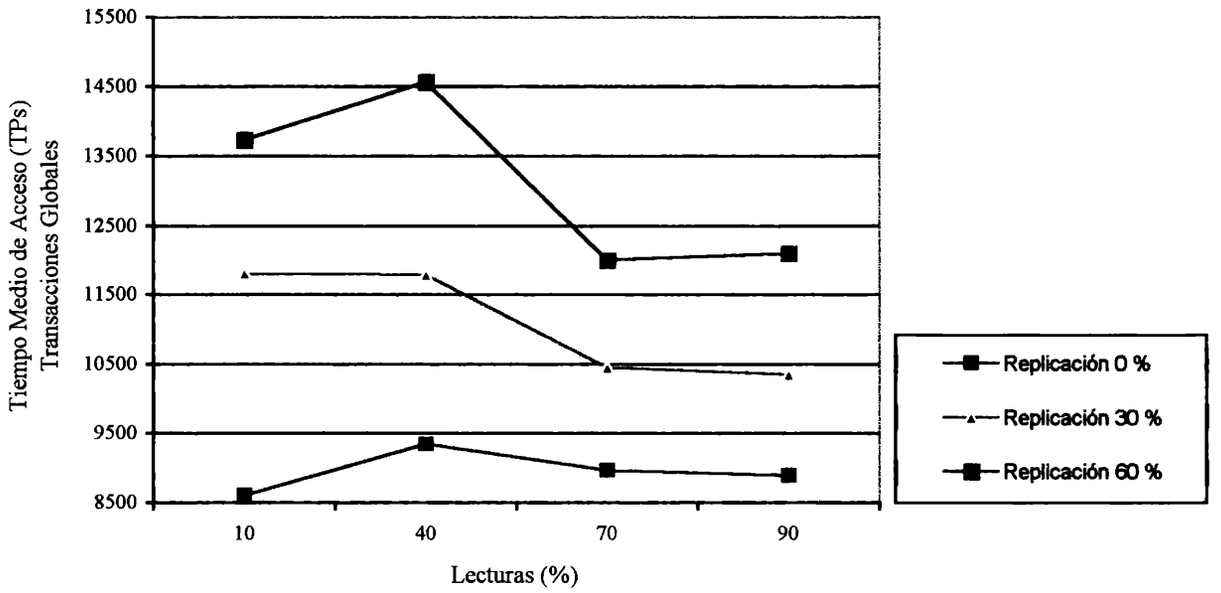
Por otro lado, los sistemas sin replicación muestran los menores tiempos medios debido a que no es necesario el tiempo extra en la coordinación de la ejecución de las transacciones, para mantener la consistencia de réplicas.



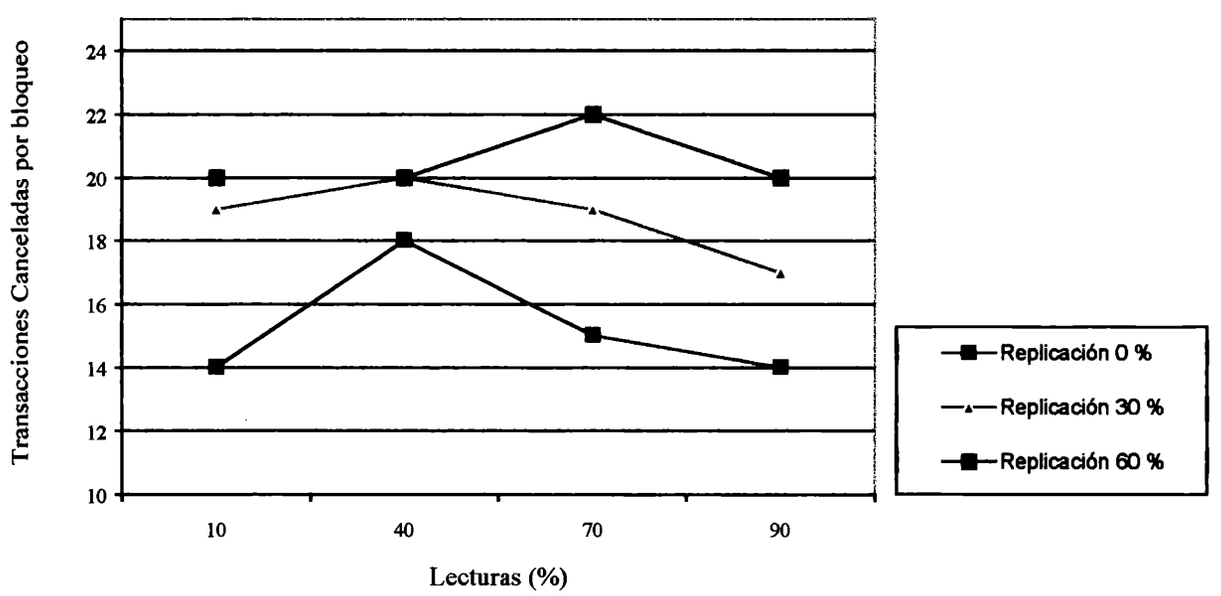
Transacciones Locales (90% Accesos Locales) (7)

(8)

Es considerable la diferencia en los tiempos de ejecución de transacciones globales al aumentar la replicación, incrementados por las funciones de aseguramiento de consistencia de réplicas de los algoritmos globales utilizados.



Transacciones Globales (90% Accesos Locales) (8)



Transacciones Canceladas (90% Accesos Locales) (9)

(9)

Es importante destacar también, la disminución del número de transacciones canceladas por bloqueo con respecto a los sistemas analizados anteriormente con menores porcentajes de accesos locales (50% y 70%).

Resumen

En las siguientes tablas se comparan los sistemas estudiados en función de la media de los tiempos de ejecución observados.

	Replicación	50% Accesos Locales	70% Accesos Locales	90% Accesos Locales
Transacciones locales	**10%**	267.66	263.11	
	20%	258.22	258.22	
	30%	324.22	320.00	367.00
	60%			335.50
	0%	234.62	201.78	252.25

Tabla 1: Tabla comparativa de tiempos medios de ejecución de transacciones locales en sistemas distribuidos con diferente porcentaje de accesos locales y replicación (expresados en TPs).

	Replicación	50% Accesos Locales	70% Accesos Locales	90% Accesos Locales
Transacciones globales	**10%**	9166.00	7635.33	
	20%	9166.11	9839.75	
	30%	9164.88	10302.00	11099.25
	60%			13097.25
	0%	7399.55	8273.50	8944.50

Tabla 2: Tabla comparativa de tiempos medios de ejecución de transacciones globales en sistemas distribuidos con diferente porcentaje de accesos locales y replicación (expresados en TPs).

En la tabla 3 se comparan los sistemas estudiados en función del número de transacciones canceladas por bloqueo.

	Replicación	50% Accesos Locales	70% Accesos Locales	90% Accesos Locales
Transacciones canceladas	**10%**	54	41	
	20%	54	41	
	30%	59	45	19
	60%			20
	0%	31	21	15

Tabla 3: Tabla comparativa de número de transacciones canceladas por bloqueo en sistemas distribuidos con diferente porcentaje de accesos locales y replicación (expresado en unidades).

Al compara el número de transacciones canceladas por el algoritmo de control de bloqueos, se observa que al incrementar el porcentaje de accesos locales o la replicación, la tendencia del número de cancelaciones es a aumentar.

Conjunto de Prueba II: "Modelo de Datos Centralizado"

Se estudian los tiempos medios de acceso de transacciones locales y remotas que se ejecutan en un ambiente cuya distribución de datos corresponde al "Modelo Centralizado" constituido por diez localidades, donde los datos se encuentran en una localidad central.

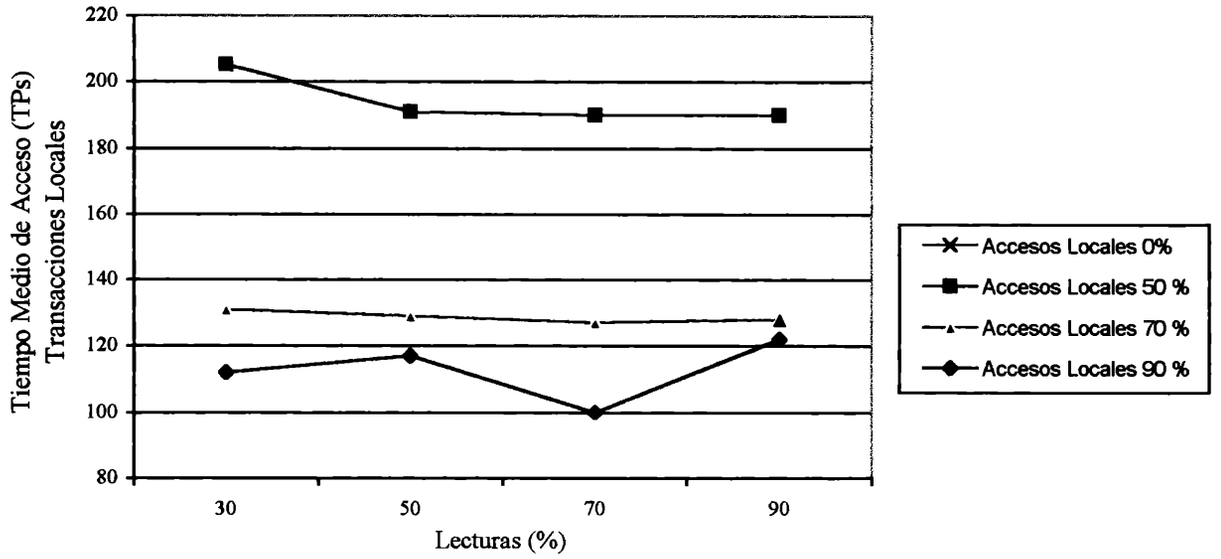
Los casos comparados corresponden a sistemas con datos centralizados de bases de datos con ejecución de transacciones por parte de la localidad central (porcentaje de accesos locales mayores a cero) y sistemas centralizados con servidores dedicados sin acceso local (sin accesos locales).

La política utilizada por la localidad central para la atención de pedidos de ejecución de transacciones es FIFO (primero en llegar, primero en salir).

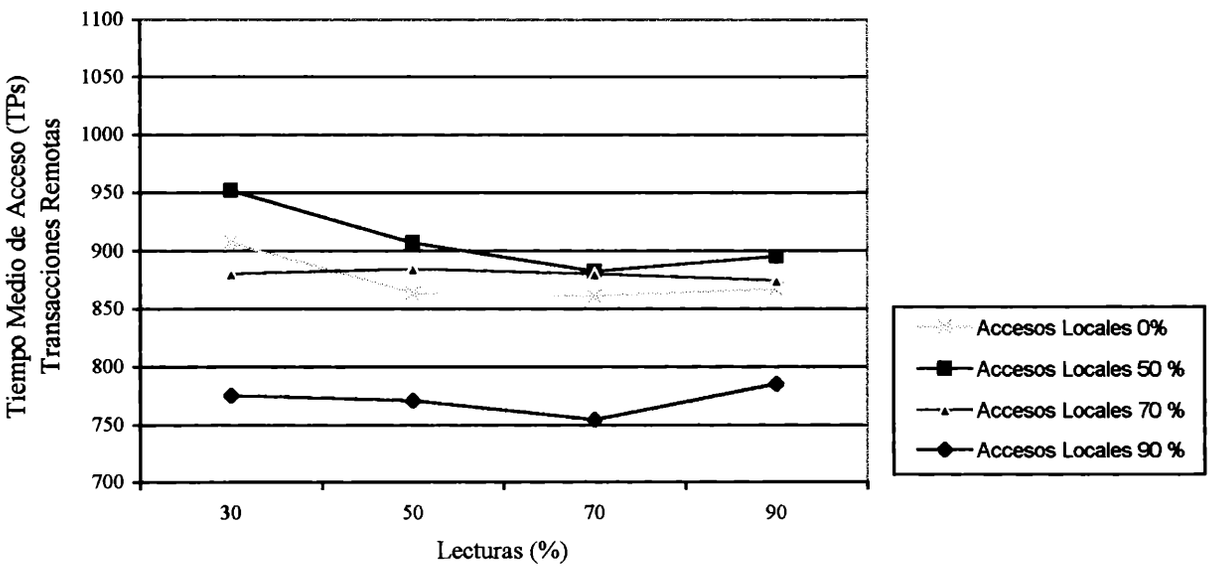
La actualización de la base de datos se realiza en forma "diferida". Las trazas comprenden 100 transacciones ejecutadas sobre una base de datos de 300 tablas.

(10)

En este gráfico se denota la influencia del porcentaje de accesos locales en los tiempos medios de ejecución de las transacciones locales. La atención de las transacciones en orden de llegada genera tiempos homogéneos de ejecución. Para los modelos donde la localidad central es un servidor dedicado de datos, no existe curva de tiempos de ejecución de transacciones locales.



Ejecución de Transacciones Locales (Modelo Centralizado) (10)



Ejecución de Transacciones Remotas (Modelo Centralizado) (11)

(11)

En la ejecución de transacciones remotas, al aumentar el porcentaje de accesos locales, los tiempos medios de acceso, son menores.

Es considerable la diferencia de los tiempos locales y los remotos, en los sistemas con accesos locales, ya que estos últimos requieren un tiempo extra de comunicación interproceso. Mientras que las transacciones locales involucran comunicación intraproceso (de procesos que se encuentran en la localidad central).

En el caso en el que todas las transacciones involucren accesos remotos (0 % accesos locales), el cual representa un sistema en el cual todas las transacciones se inician en una localidad diferente de la central, los tiempos de ejecución son medios, esto se debe a que los únicos procesos en ejecución continua en la localidad central son el gestor de transacciones central y el gestor de bloqueos del sistema. Por este motivo se consume menos tiempo de comunicación y computación local, pero por otro lado aumenta el tiempo de comunicación interproceso de la central con el resto de las localidades que requieren la ejecución de sus transacciones.

(10, 11)

A diferencia de los modelos distribuidos, los modelos centralizados no se ven afectados por la variación del porcentaje de operaciones de lecturas, por el hecho de la existencia de un gestor de transacciones principal y un gestor bloqueos central que administran la ejecución y el acceso a los datos (según el caso) en forma centralizada sin diferenciar operaciones de lectura o escritura.

Conclusiones

La replicación es un factor importante en la implementación de sistemas de DBD, el cual favorece a la disponibilidad de los datos en las localidades.

El número de tablas de datos de la base de datos, el número de localidades, el número de transacciones ejecutadas concurrentemente en una localidad, el número de transacciones originadas por segundo en una localidad, el número de operaciones de escritura que involucra una transacción, el tiempo de ejecución de una actualización, el tiempo entre la actualización de un dato durante la ejecución de una transacción y la actualización del resto de las réplicas del mismo y el tiempo de procesamiento y transmisión de mensajes necesario para la actualización del resto de las réplicas, son sin duda los elementos determinantes de los tiempos de ejecución del sistema de base de datos.

Variando el porcentaje de replicación conjuntamente con el porcentaje de accesos locales, se observa que en sistemas altamente replicados y/o con alto porcentaje de transacciones globales, los tiempos medios de ejecución se incrementan notoriamente, por la complejidad de los algoritmos de ejecución de transacciones y gestión de bloqueos.

El aumento del porcentaje de accesos locales, manteniendo constante el parámetro de replicación, disminuye el número de transacciones globales y en consecuencia se reduce el número de transacciones canceladas por bloqueo. Además, es menor el tiempo ocioso de los algoritmos globales en espera de la actualización del resto de las réplicas (ignoradas por el usuario) de los datos involucrados en cada transacción.

La influencia del porcentaje de operaciones de sólo lectura es relevante en sistemas distribuidos cuyo algoritmo de control de bloqueos da prioridad a las lecturas. En estos casos, al incrementar la proporción de operaciones de lectura (en los sistemas con o sin replicación), se observa una tendencia a disminuir de los tiempos medios de ejecución.

En los sistemas sin replicación, donde la ejecución de las transacciones es local o global con accesos a lo sumo remotos, los tiempos de ejecución de las transacciones locales y globales son menores a los observados en los sistemas replicados y es menor también el número de cancelaciones.

En los sistemas centralizados, los algoritmos de ejecución y de gestión de bloqueos insumen menos costos de implementación en cuanto a software, pero la simplicidad de implementación del modelo no llega a suplir la desventaja de las que carecen los sistemas distribuidos: el fenómeno de "cuello de botella", el cual implica tiempo ocioso de las localidades en la ejecución de transacciones, que se ven demoradas hasta que la localidad central atienda su pedido y produzca sus resultados, independientemente de la política de scheduling que se utilice[33].

[33] Recordar que el trabajo de implementó una política de scheduling FIFO.

Trabajos Futuros

Se puede ampliar la funcionalidad del Sistema de Simulación en cualquiera de los siguientes puntos:

- ✓ Parametrización de la cantidad de localidades con datos de la base de datos.
- ✓ Implementación, en función del diseño existente, de la recuperación de fallas físicas de las localidades del sistema.
- ✓ Implementación de la fragmentación de la red y de los datos.
- ✓ Implementación de diferentes modelos de mantenimiento de réplicas.
- ✓ Utilización del Ambiente de Simulación generado para estudiar el impacto en la variación de otros factores que pueden ser paramezados en el "Generador de Trazas". Por ejemplo:
    - ✓ Estudio de sistemas centralizados y distribuidos, con o sin replicación, variando del número de localidades y de tablas de la base de datos;
    - ✓ Comparación de los algoritmos de ejecución de transacciones: Commit de dos fases y Commit de tres fases, con o sin probabilidad de falla;
    - ✓ Comparación de los algoritmos de gestión de bloqueos: Protocolo Preferencial y Protocolo de la Mayoría;

ANEXO I: El Modelo Relacional y el lenguaje SQL

El modelo relacional de bases de datos revolucionó el mundo de las bases de datos permitiendo que las computadoras personales reemplazaran a las minicomputadoras y mainframes en muchas aplicaciones.

Este modelo se apoya en el cálculo relacional desarrollado por E. F. Codd en los años 70. Su principal aporte radica en que la relación entre los datos deja de expresarse a través de punteros a los datos que deben almacenarse conjuntamente con ellos.

En el modelo relacional, los vínculos entre las entidades de datos se establecen mediante la abstracción de relaciones entre las distintas tablas. Estas tablas se obtienen del análisis de la naturaleza intrínseca de los datos y no de las particularidades físicas de almacenamiento.

El modelo relacional de bases de datos de Codd representa una descripción teórica de cómo deben usarse las bases de datos relacionales. Codd desarrolló en 1970 para IBM un lenguaje denominado SEQUEL (Structured English Query Language), diseñado para una base de datos relacional denominada "System R".

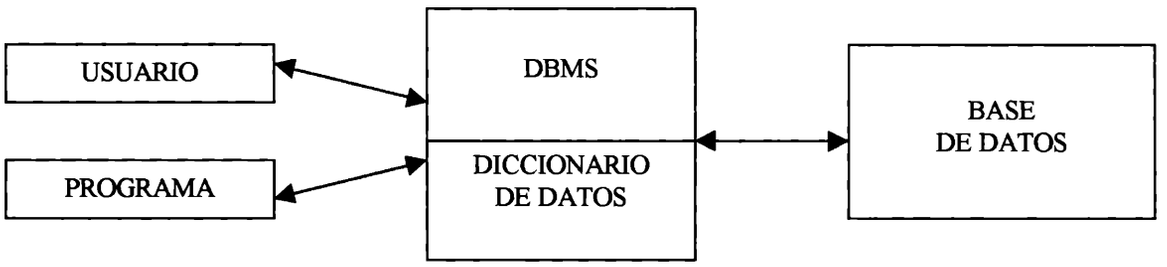
IBM desarrolló posteriores versiones del lenguaje, que pronto pasó a llamarse SQL y era utilizado por a un sistema de la compañía basado en la arquitectura mainframe denominado DB2.

SQL (Structured Query Language) es un lenguaje que permite expresar operaciones diversas sobre datos almacenados en bases de datos relacionales, en los que la información se almacena en tablas, con los datos dispuestos en filas y columnas [GARR 97].

Una de las características más importantes de SQL frente a los lenguajes sobre bases de datos no relacionales radica en que sus sentencias permiten manejar conjuntos de registros, en lugar de un solo registro cada vez. La mayoría de los gestores, tanto los basados o no en la arquitectura C/S, como los entornos de programación más usados hoy en día, tienen al lenguaje SQL como su medio de acceso a los datos.

La competitividad entre las diversas compañías hizo que el lenguaje fuera evolucionando de manera diversa en los distintos productos que lo fueron incorporando.

Se intentó estandarizar el lenguaje en primera instancia por el Comité X3H2 patrocinado por el American Commitee of Standards Institute. Esta primera versión estandarizada de SQL (ANSI SQL) se logró entre 1986 y existe una posterior del año 1992.



ANSI SQL

Las operaciones que pueden realizarse con el lenguaje SQL pueden clasificarse en los siguientes grupos:

Consultas de datos

Consisten en sentencias que permiten obtener datos de las tablas y especificar la forma en la que deseamos que se presenten. La sentencia más relevante de este grupo del lenguaje es SELECT que permite extraer datos de una o más tablas.

Manipulación de datos

En este grupo están incluidas las sentencias que permiten modificar, añadir y borrar filas en las tablas de las bases de datos. Entre las sentencias se incluyen: INSERT, UPDATE y DELETE.

- ✓ INSERT: permite añadir una o varias filas a una tabla.
- ✓ UPDATE: permite modificar una o varias filas de una tabla.
- ✓ DELETE: permite borrar una o más filas de una tabla.

Definición de Datos

Las operaciones de este grupo permiten definir nuevos objetos o destruir objetos existentes mediante las sentencias CREATE y DROP, así como establecer restricciones para los campos de las tablas (NOT NULL, CHECK Y CONSTRAINT) y establecer relaciones entre las tablas (PRIMARY KEY, FOREIGN KEY y REFERENCES).

Control de datos

Permite controlar diversos aspectos, como por ejemplo la confidencialidad de los datos. Las sentencias GRANT Y REVOKE permiten conceder y retirar la autorización para el acceso de un

usuario a una tabla. Un usuario no podrá consultar o actualizar datos de una tabla si previamente no posee permite para ello.

Procesado de transacciones

Estas sentencias permiten controlar que una serie de órdenes se ejecuten de manera cohenrente (se ejecuten todas o en caso contrario no se ejecute ninguna de ellas). Estas sentencias son BEGIN TRANSACTION, COMMIT y ROLLBACK.

DONACION.....

$.....

Fecha 01-03-06

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

Bibliografía

[BARK 97] Linux. Naba Barkakati. Anaya Multimedia, Madrid, España, 1997.

[BARU 97] The Maintenance of Common Data in Distributed System. Baruch Awerbuch and Leonard J. Schulman. Journal of the ACM , Vol 44, Nro. 1, January 1997, pags. 86-103.

[BERN 96] Middleware: A Model for Distributed System Services. Philip A. Bernstein. Comunication of ACM. Vol 39 Nro.2, February 1996.

[BERS 92] Client / Server Architecture, Alex Berson. Mc Graw Hill 1992.

[DATE 94] Introducción a los sistemas de Bases de Datos. Date, C.J. Addison Wesley, 1994.

[DEIT 95] C / C++, H.M. Deitel, P.J. Deitel. Prentice Hall, México 1995.

[FRAN 95] Transactional Clien-Server Cache Consistency: Alternatives and Performance. Michael J. Franklin, Michael J. Carey, Morin Livny. ACM Transaction on Database Systems, Vol. 22 Nro. 3, September 1997, Pages 315-363.

[GARR 97] Microsoft Sql Server, Alberto Delgado Garron. Prentice Hall, España 1997.

[GEIS 94] PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel computing, Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam. 1994.

[FOST 95] Defining Performance. Ian Foster, 1995.

[GOME 95] Ada, Sintaxis y Metodologia, Lucas Gomez.

[GRAY 96] The Dangers of Replication and a Solution. Jim Graay, Pat Helland, Patrick O'Neil, Dennis Shasha. SIFMOD 96, Montreal, Canada.

[HSU 93] Transactions and Database Processing, Vijay Kumar, Meichun Hsu., 1993.

[KERN 87] El entorno de programacion UNIX, Brian W. Kernighan, Rob Pike. Prentice Hall, México, 1987.

[KUMA 93] Concurrency Control Mechanisms and Their Taxonomy. Vijay Kumar.

[MAKK 93] Detection and Resolution of Deadlocks in Distributed Database Systems. Kia Makki, Niki Pissinou.

[MEHR 91] Nonserializable Executions in Heterogeneous Distributed Database Systems. Mehrotra, S; Ratogi, R.; Korth, H.; Silberschatz, A. PDIS, Miami Beath, December 1991.

[MUNS 96] A Concurrency Control Framework for Collaborative Systems. Jonathan Munson and Prasun Dewan. Computer Supported Cooperative Work 96, Cambridge MA USA, 1996.

[ÖZSU 96] Distributed and Parallel Database Systems. M. Tamer Özsu, Patrick Valduriez.. ACM Computing Surveys, Vol. 28, Nro. 1, Marzo 1996.

[PAPP 93] Manual de Borland C++. Chris Pappas. William H. Murray, III. Mc Graw Hill, Madrid España, 1993.

[REUT 93] An Analytic Model of Transaction Interference. Andreas Reuter.1993.

[SILB 98] Fundamentos de Bases de Datos. Silberchatz; Folk. Mc Graw Hill, 1998.

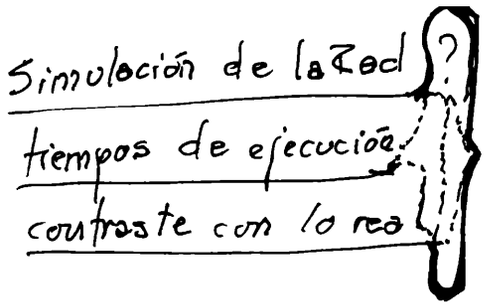
[TANE 93] Sistemas Operativos Distribuidos. Andrew Tanenbaum. Prentice Hall Hispanoamericana. Mexico, 1993.

[UMAR 93] Distributed Computing and Client Server Systems. Umar, Amjad. Prentice Hall, 1993.

[ULLM 82]Principles of Database Systems. Jeffrey D. Ullman. Second ediction. Computer Science Press, 1982.

[WAIM 96] The Model Asisted Global Query System for Multiple Database in Distributed Enterprises. Waiman Cheung.

[WOLF 97] An Adaptative Data Replication Algorithm. Ouri Wolfson. ACM transaction on Database Systems, Vol. 22, Nro.2, Páginas 255-314. Junio 1997.



BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

TES
99/19
DIF-02408
SALA

UNIVERSIDAD NACIONAL DE LA PLATA
FACULTAD DE INFORMÁTICA
Biblioteca
50 y 120 La Plata
catalogo.info.unlp.edu.ar
biblioteca@info.unlp.edu.ar

DIF-02408