Sobre el Costo de los Protocolos de Commitment

Marcelo Zanconi* Jorge Ardenghi†
Departamento de Ciencias de la Computación
Universidad Nacional del Sur
Av. Alem 1253
Bahía Blanca
Argentina
cczanc@criba.edu.ar

Resumen

Se presenta en este trabajo, una evaluación sobre los costos de los protocolos de terminación en dos fases, [1], relacionados con la correctitud y la serializabilidad, considerando los mecanismos de apropiación de recursos basados en locking, time-stamping y control optimista.

Palabras clave: bases de datos distribuidas, control de concurrencia, manejo de transacciones.

*Becario de Perfeccionamiento del CONICET
†Prof. Titular Sistemas Operativos
[1]Two Phase Commmitment Protocols

Sobre el Costo de los Protocolos de Commitment

1 Introducción

En todo sistema, se ofrece un servicio y se debe pagar por él. La pregunta que surge siempre es ¿vale la pena ofrecer el servicio, si el costo es muy alto? Si trasladamos esta cuestión al ámbito de los Sistemas Distribuidos, SD, o más concretamente de las Bases de Datos Distribuidas, BDD, la respuesta es obviamente que *sí*; existen cuestiones contrapuestas y dicotómicas de independencia, de integración, de autonomía y colaboración que imponen su existencia. Por otro lado, no es cuestión de decidir sobre la existencia o no de estos servicios, pues su necesidad es innegable, sino de no perder conciencia de los costos que se imponen para poder mantenerlo.

Se busca en este trabajo, resumir los resultados existentes sobre evaluación de un SMBDD.

1.1 Modelo de Transacciones

Describiremos brevemente un modelo de transacciones, aceptado en general en la literatura. Una *transacción* T es un programa en ejecución en un determinado sitio o nodo del SD, que debe cumplir con las propiedades de **Atomicidad, Consistencia, Insularidad y Durabilidad**, conocidos como principios ACID, [1], abstraída como una secuencia de operaciones de lectura y escritura.

Toda *transacción distribuida* puede pensarse como un proceso que, nacido en un nodo N, se ha dispersado en la red, internándose en los nodos que administran datos accedidos por la transacción, originándose así varias *subtransacciones*, cada una compitiendo por el acceso a los datos locales. Podemos distinguir entre transacciones *locales* y *globales*; las primeras están bajo control del DBMS local, fuera del ámbito distribuido mientras que las segundas son controladas por el sistema distribuido, y se consideran constituidas por varias transacciones locales.

Las transacciones pueden verse como tipos abstractos de datos con las siguientes operaciones:

- **Begin Transaction**: el sistema distribuido requiere la iniciación de una nueva transacción local. El DBMS local responde con un identificador de transacción.
- **End Transaction**: la transacción local ha terminado su cómputo y espera el mensaje de *commit*.
- **Read, Write**: operaciones de acceso a los datos, que implicita o explicitamente imponen control de concurrencia mediante uno de los mecanismos conocidos, tales como locks, [3], estampillas de tiempo, [5] o control optimista, [4].

- **Abort**: abortar una transacción.
- **Commit**: exportar todos los resultados computados por una transacción como cambios permanentes a la BD.
- **Prepare to Commit**: la transacción ha alcanzado su punto de terminación, **End** y espera el mensaje de **Commit** o **Abort** del DDBMS.

Del modelo anterior, se desprende que una transacción puede encontrarse en uno de los tres estados siguientes: *ejecutándose*, *terminandc* o *abortando*. Veremos que ocurre en cada estado, para analizar el costo del rendimiento del sistema:

- **Ejecutándose**: es decir, la transacción prosigue con la ejecución normal. Se destaca dentro de esta etapa, el hecho que la transacción deberá coordinar, junto con sus compañeras de ejecución, la adquisición de datos, de modo de garantizar la serializabilidad.
- **Terminando** : es decir, una vez que una transacción, o mejor dicho una subtransacción, ha alcanzado su punto de terminación, no puede terminar definitivamente, sino hasta que sus compañeras en la distribución estén todas dispuestas a hacerlo, de modo de garantizar la atomicidad. Básicamente, las subtransacciones usarán algún protocolo distribuido de terminación y si todas están en condiciones de terminar, cada subtransacción interviniente hará efectivo los cambios producidos.
- **Abortando** : caso que una transacción se vea impedida de continuar, entrará en un estado de aborto, para terminar abruptamente la tarea. Los cambios producidos no se efectivizarán e informará a la transacción madre de este hecho, quien a su vez, lo reportará a todas las subtransacciones intervinientes, de modo que todas aborten.

Estos pasos quedan resumidos en las figuras 1 y 2.

Para mayores detalles de esta implementación puede consultarse el texto [6]. Veremos a continuación para los diferentes estados de una transacción, los posibles costos que se involucran.

2 Costo de Acceso a los Datos

Cada transacción que se ejecuta localmente, no importando si es local propiamente dicha o parte de una transacción global, debe acceder a los datos allí almacenados. Para ello, y de acuerdo al principio de insularidad, debe proceder previamente a una política de asignación de datos, que garantice una ejecución correcta. No analizaremos aqui los conceptos de correctitud, serializabilidad y sus variantes; para una buena referencia, consultar [3], [8] y [7].

Un método muy común para el acceso a los datos es el de apropiación o *lock*. En este método a cada dato d se le asocia uno de dos estados *libre* u *ocupado*. Cada vez que una transacción T desea acceder a un dato d, consulta su variable de estado; si está *libre* se otorga el permiso a T para acceso, y se cambia el estado a *ocupado*. Caso contrario, se niega el permiso o se puede ubicar a T en una cola de espera, hasta que le toque su "turno".

En principio, podemos considerar varios aspectos relativos al costo de este servicio:

1. **Control de Deadlock**: el método anterior corre el riesgo de ubicar a dos o más transacciones en un estado de *deadlock*. Basta pensar que si T_1 y T_2 son transacciones y ambas acceden a los datos A y B, una ejecución como la siguiente causa deadlock:

$$\alpha_1(a)\alpha_2(b)\alpha_1(b)\alpha_2(a)$$

donde simbolizamos con α una operación de lectura o escritura y usamos los subíndices para indicar la transacción que origina la operación.

Para realizar este control se dispone de varios mecanismos:

- *Evasivos*: que impiden que cada transacción entre en estado de deadlock, anulando las situaciones que potencialmente lo producirían. Entre los mecanismos podemos

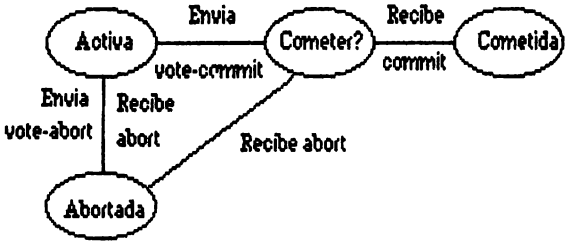

Figura 1: Transacción Participante

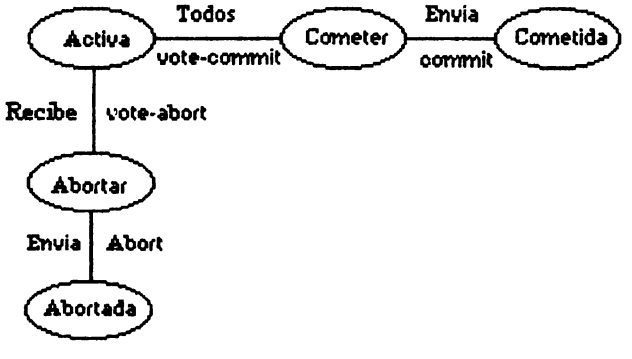

Figura 2: Transacción Coordinadora

mencionar: imponer un orden parcial a los datos y exigir que cada transacción puede bloquear un dato solo en el orden especificado por el orden parcial, [13].

- *Preventivos*: que impiden que cada transacción entre en estado de deadlock, detectándolo justo antes que se produzca. Entre los mecanismos preventivos podemos mencionar aquel de expropiación y retroceso de transacciones basado en la asignación de estampillas de tiempo que registran la entrada al sistema, (mecanismos esperar-morir o herir-esperar).
- *Detectivos*: en este caso no se hace nada con respecto al deadlock, sino que se analiza el estado del sistema, manteniendo alguna estructura apropiada, y cada cierto tiempo se analiza dicha estructura para deducir el estado del sistema. El método más famoso es el del *grafo de espera*, [9], debido a R.C. Holt.

En cualquiera de estos casos, el DDBMS debe conocer y controlar varios hechos. En los protocolos preventivos, para controlar un orden parcial, debe, en primer lugar, definirse un orden de los datos, *anterior* a cualquier operación sobre los mismos. Este orden debe ser fácilmente interpretable y extensible, de modo que pueda ampliarse a medida que la BD se amplia. Para controlar que cada transacción mantenga un buen comportamiento debe mantenerse información sobre los datos a los que ha accedido, (por ejemplo, "el mayor" dato accedido) o mantenerse una estructura de datos capaz de almacenar toda la información sobre las estampillas de tiempo. Veremos algunos de estos puntos más adelante.

En el caso de detección, el costo es el de mantener una estructura compleja como un grafo, conocidos los costos de evaluación de la misma, orden de n^2, donde n es el número de nodos[2]. Además en un SD es complejo tener una fotografía instantánea del estado global del sistema, algo similar al principio de incertidumbre, usado en física. A modo ilustrativo de esta situación, mucho mecanismo detectivos caen en el error del *deadlock fantasma*, es decir detectando un deadlock que finalmente no existe.

2. **Control de Serializabilidad** El principio de *insularidad* indica que toda transacción debe ejecutarse correctamente en el sistema, sin importar sus competidores. Esto es, si la transacción en cuestión es correcta, su ejecución *debe ser correcta*, despreciando su situación dentro del sistema, (por ejemplo, puede ser la única transacción o puede estar compitiendo con otras 100 transacciones). Este control es el corazón de todo SMBDD. Tomaremos tres paradigmas de serializabilidad, a saber:

   - **Manejo de Locks**
   - **Manejo de Estampillas de Tiempo**
   - **Control Optimista**

   Manejo de Locks. El mecanismo de apropiación de datos basado en locks requiere un control sobre la concurrencia, no ya relativo al acceso sino a la correctitud de la transacción. Uno de los protocolos más efectivos es el de locks en dos fases[2] que impone

[2]Two Phase Locking Protocol

a cada transacción liberar recursos solo cuando no va a hacer más apropiaciones, o mejor expresado todos los *locks* preceden a los *unlocks*, [3].

El protocolo anterior hace que cada transacción tome datos en forma correcta, siguiendo el criterio de "ejecución serial equivalente". Por ejemplo

$$r_1(a)w_1(a)r_2(b)w_2(b)r_1(b)r_2(a)$$

es una ejecución concurrente de las transacciones T_1 y T_2, pero esta ejecución no es equivalente a T_1, T_2 o T_2, T_1. Para evitar esto, el SMBD debe transformar toda operación de la forma $\alpha_i(d) \in T_i$ en *lock(d)*, $\alpha_i(d)$.

Por otro lado, se debe evitar la disponibilidad de acceso de datos a una transacción T_j, modificados por una transacción T_i que aún no ha terminado. Para ilustrar esto, consideremos el ejemplo a anterior y supongamos que T_2 aborta, antes de hacer $r_2(a)$; está claro que también debería hacerlo T_1, pero tal vez en este punto T_1 ni siquiera esté ejecutándose, pues puede haber terminado. Por lo tanto, las transacciones deben liberar sus datos cuando han alcanzado el estado de *Committed*. Esta restricción evita los abortos en cascada, aunque disminuye terriblemente la disponibilidad de los datos y por lo tanto, baja la concurrencia. En [11] se propone un mecanismo de Terminación Condicional para aumentar la disponibilidad, basado en el mecanismo de preleased, [10].

Manejo de Estampillas de Tiempo. A cada transacción T_i se le asocia una estampilla, $ts(T_i)$, asignada en el momento que comienza su ejecución. A dos transacciones diferentes, nunca se le asocia una misma estampilla y este ordenamiento de estampillas, es el ordenamiento de acceso. Cada SMBD local debe mantener para cada dato en actividad d dos estampillas de tiempo: *r-ts(d)* y *w-ts(d)*, último tiempo de acceso para lectura y escritura respectivamente.

El scheduler debe realizar un test para decidir si se otorga un dato, basado en el concepto definido por Lamport, [12] sobre ordenación de eventos: toda lectura de un dato d debe proceder de una transacción con estampilla de tiempo *mayor* a aquella última que realizó una modificación de d; recíprocamente, toda escritura de un dato d debe proceder de una transacción con estampilla de tiempo *mayor* a la última que realizó una operación de lectura de d. En caso contrario, la transacción peticionante se aborta y reingresa al sistema como una nueva transacción con una *nueva* estampilla de tiempo.

La principal desventaja de este método, consiste en que las transacciones pueden "ver" datos modificados por otras transacciones aun activas. Se impone un costo extra que es mantener una estructura de *prewrites*, obedeciendo la siguiente política: una vez que una transacción ha realizado un preescritura de un dato con estampilla Ts, el SMBD no debe proceder a realizar una escritura permanente de dicho dato a una transacción con estampilla mayor a Ts. Esto es, a los efectos de no guardar en la BD datos "viejos", que eventualmente sobreescribirían a datos más recientes, [9].

Otro problema es la administración de las estampillas. Dentro de cada sistema, el reloj es único y no existen mayor problemas en asegurar a cada transacción que entra al

sistema una estampilla única, mayor a cualquiera otorgada hasta el momento. Pero ?'que ocurre cuando una transacción proviene de otro nodo? Afortunadamente Lamport ha definido un mecanismo de homogeinización de relojes, basado en una relación *happens_before*, [12], que facilita un ordenamiento parcial de transacciones.

Control Optimista. En este mecanismo no se impone ninguna restricción para *acceder* a los datos, pero todas las modificaciones se hacen en un área de memoria local. Se imponen las siguientes reglas, [4]:

(a) No hay restricciones para las operaciones de lectura, que siempre se toman de la BD.

(b) Toda operación de escritura se hace en 3 fases: lectura, validación, escritura. En la primera fase, fase de computación, todas las escrituras se hacen en copias locales. En la fase de validación se realiza un test, que en caso de dar positivo, exporta los cambios. La fase de escritura procede a realizar todas las modificaciones en forma atómica. El test se basa en la asignación de un número $t(i)$ a cada transacción T, tal que T_i precede a T_j si $t(i) < t(j)$. Para cada transacción T_j y $\forall T_i : t(i) < t(j)$:

   i. T_i termina su fase de escritura antes que T_j comience su fase de lectura.

   ii. El conjunto de datos que T_i escribe no intersecta con el conjunto de datos que T_j lee y T_i termina su fase de escritura antes que T_j comience su fase de escritura.

   iii. El conjunto de datos que T_i escribe no intersecta con el conjunto de datos que T_j lee o escribe y T_i termina su fase de lectura antes que T_j termine su fase de lectura.

   La condición 2(b)i es trivial. La condición 2(b)ii indica que T_i, "más vieja", no sobreescriba a T_j, "más nueva". La condición 2(b)iii impone que T_i no afecte la fase de lectura o escritura de T_j. En cualquiera de las dos condiciones últimas, T_j no puede afectar la fase de lectura de T_i.

   El costo de implementación de un sistema como este, es sumamente caro, si se piensa para cualquier tipo de BDD. En particular, los propios autores reconocen que el control optimista es muy bueno para sistemas donde predominan las consultas sobre las actualizaciones. Su uso para una BDD cualquiera requiere que el sistema maneje todo un conjunto de conjuntos de datos de escritura y lectura; se impone necesariamente un ordenamiento, para facilitar las operaciones de intersección y búsquedas; se requieren mecanismos de asignación de estampillas de tiempo, y de números para transacciones; es preciso manejar un control de inanición, sobre todo para aquellas transacciones de larga duración o que acceden a una gran variedad de datos. Por otro lado, el costo de evaluación del test es despreciable, si se piensa en que la disponibilidad es máxima, pero por otro lado, la penalidad también es extrema, toda vez que un fallo, implica deshacer todo lo computado por una transacción.

Todos los costos anteriores pueden ser mejor comprendidos si se tiene en cuenta una relación muy sencilla, definida en [15], e ilustrada en la figura 3.

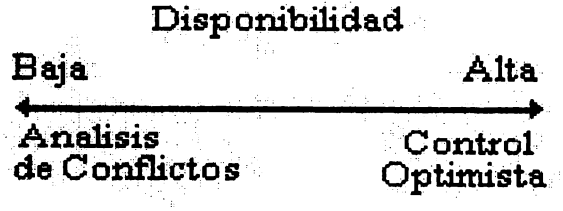

Figura 3: Espectro de Soluciones

Existen algunos otros parámetros que nos pueden dar medidas del costo de un SMBDD. Entre ellas podemos mencionar:

1. **Manejo de Múltiples Copias**: varios sistemas poseen varias copias anexas de un dato, con fines de backup o de disponibilidad. En el primer caso, el SMBDD en algún momento decide, per se, actualizar las copias de respaldo; este sistema tiene como objetivo reducir la posibilidad de pérdida de un dato. En el segundo caso, el SMBDD debe actualizar, simultáneamente todas las copias; este sistema tiene como objetivo aumentar la disponibilidad de acceso a los datos, sobre todos en forma paralela por varias transacciones de lectura.

2. **Mantenimiento de las Tablas**: cuando los datos se hallan distribuidos a lo largo de un red, una transacción T debe saber como distribuirse en subtransacciones. Los SMBDD deben mantener "tablas" de datos, de modo de poder encargarse de esta distribución. Estas tablas no tienen por que ser exactamente estructuras de datos con información sobre $\langle dato, localidad \rangle$, sino que pueden ser mucho más abarcativas, incluyendo fórmulas, basadas en operadores relacionales, de modo de facilitar la fragmentación horizontal.

Se ha probado, [17] que el protocolo de apropiación en dos fases, a pesar de no estar libre de deadlock, no es más caro que el mismo mecanismo incorporando restricciones para que esté libre de deadlock. Para el primer caso, se puede usar un agregado de "timeouts" que hace que una transacción que deba esperar más de un determinado tiempo para el acceso a un dato, se aborte y se reinicie más tarde.

Naturalmente, el costo de evaluación de un mecanismo 2PL junto con timeouts, debe compararse con el costo total de una transacción que es abortada y luego reiniciada. Sin embargo, basándose en que son más las transacciones que terminan con éxito, que aquellas que abortan y que es preferible un mecanismo de detección de deadlock que no sobrecargue al sistema, se puede concluir que este protocolo combinado es mejor.

3 El Costo de la Atomicidad

El principio de atomicidad establece que un proceso, por más complejo que sea debe considerárselo como una *unidad de control y acceso a los datos.* En esta concepción, entendemos que o bien un proceso debe realizarse en forma total, o no debe realizarse absolutamente nada. Es decir, sus efectos se traducen en forma completa a la BD o bien, su "paso" por el sistema resulta imperceptible, si el proceso tiene algún inconveniente que le impide terminar.

Este principio de correctitud, aplicado a una BDD, se interpreta como que una transacción distribuida debe ejecutarse en forma atómica. Esto es, todas las subtransacciones deben ejecutarse en forma atómica en los nodos locales y el resultado final debe alcanzarse por consenso entre las subtransacciones: si todas terminaron con éxito, la transacción distribuida termina con éxito; si alguna ha fallado, la transacción distribuida falla.

El mecanismo general que se aplica para alcanzar este resultado es el Protocolo de Terminación de Dos Fases[3], [16].

Este mecanismo impone que luego de terminar la ejecución de las operaciones de una transacción, el Manejador Global de Transacciones, MGT, envía un mensaje de *prepare-to-commit* a cada uno de los n nodos donde opera la transacción. Al recibir el mensaje, cada nodo vota con *ready-to-commit* o *abort*. Si el manejador obtiene todos votos de *ready-to-commit* entonces envía un mensaje de *commit*, caso contrario, envía un mensaje de *abort*.

Como se puede observar, esta política es muy burocrática, pues impone $3 \times n$ mensajes en la red, donde n es el número de subtransacciones. Por otra parte, cada subtransacción enviando un mensaje *ready-to-commit* debe mantener en memoria estable todas las actualizaciones que podría tener que realizar. Este mecanismo asegura que, aun cuando el sistema sobre el que se está ejecutando fallara, al recuperarse e interrogar al MGT, podría resolver la situación.

Este método es censurado por aquellos que sostienen que cada nodo debe mantener su independencia, y que sus decisiones sobre la ejecución deben ser autónomas, [1]; el hecho que las transacciones deban indicar un voto de *ready-to-commit* da pie para otro protocolo distribuido de terminación, donde se obvia este paso.

El mecanismo consiste en considerar a cada servidor de datos como el participante, sin mencionar el DBMS local. El MGT puede colectar los mensajes de *committed*, enviado por los servers y en el caso de verse obligado a abortar una transacción global, envía una operación de *undo* a aquellos que enviaron un *committed*. Este protocolo requiere bastante sobrecarga del servidor local, para tener éxito. En primer lugar, el server debe mantener una bitácora sobre las actualizaciones realizadas. En segundo lugar, antes que el server envíe al MGT el voto *commit* de T_i, cada transacción de la cual leyó datos T_i debe haber cometido; caso contrario, podemos caer en los efectos de aborto en cascada o en inconsistencias de la BD.

Hemos dejado en esta presentación dos aspectos muy importantes que discutiremos a continuación: la bitácora y el envío de mensajes de reconocimiento[4].

La bitácora y los mensajes de reconocimiento El coordinador envía mensajes de *prepare-to-commit* a los participantes. Cada participante escribe en la bitácora su registro *prepared* y envía el mensaje *ready-to-commit* aguardando la decisión consensuada. En caso que el participante decida abortar, no se escribe nada en bitácora, libera sus recursos y se retira del sistema. Cuando el coordinador ha alcanzado todos los votos positivos, escribe un registro de *commit* en bitácora y envía el mensaje de *commit* a todos los participantes. Los participantes, al recibir este mensaje, escriben en bitácora el registro *commit* y envían un mensaje de reconocimiento al coordinador. Luego, escriben sus datos en la BD, exportando los resultados de su cómputo. Finalmente, se pueden retirar del sistema. Por el contrario, cuando los participantes reciben el mensaje de *abort*, escriben un registro de *abort*, envían el

[3] Two Phase Commitment Protocol
[4] acknowledge messages

mensaje de reconocimiento y se retiran del sistema. El coordinador, luego de colectar todos los mensajes de reconocimiento escribe un registro de *end* y se retira del sistema.

Con este protocolo, el coordinador se asegura que todos han alcanzado la misma decisión. Por otro lado, al obligar a cada participante a escribir en bitácora la decisión final, el coordinador se asegura que, si algún sitio falla, al recuperarse sabrá que decisión tomar.

Cada registro en bitácora contiene información autosuficiente como para poder actuar en caso de fallos: tipo de registro, identidad del proceso que lo originó, nombre de la transacción, identidad del coordinador, nombres de los datos apropiados.

4 Conclusión

Se han mostrado diferentes protocolos conocidos para reducir el problema de correctitud y serializabilidad. Cada uno de estos protocolos alcanza su misión a costa de sacrificar los recursos, tanto de datos, -disponibilidad-, como computacionales, -tiempos de acceso, mantenimiento de estructuras, tráfico en la red-.

Se concluye que, al elegir un sistema de manejo de bases de datos, deben tenerse en cuenta muchas variables. Los sistemas existentes proveen diferentes mecanismos de apropiación, control de concurrencia y control de atomicidad de transacciones.

En sistemas orientados hacia consultas, tales como bases de datos estadísticas, el control optimista resulta un método apropiado, pues no recarga el sistema con controles, que a la postre resultarán inútiles.

Por el contrario, en sistemas de alto tránsito de datos, resulta preferible un método más conservador. La elección de un sistema con control basado en locking o en estampillas de tiempo, dependerá de las probabilidades de operaciones simultáneas con datos comunes, del tiempo de respuesta que se espera obtener y de la organización de la red. En caso de operaciones con datos comunes, es preferible la elección de locking, buscando una política de asignación que prime la seguridad sobre la rapidez, -debemos tener en cuenta que puede haber retrocesos en cascada usando estampillas de tiempo.

Bibliografía

[1] Y. Breitbart, H. García Molina, A. Silberschatz. *Overview of Multidatabase Transaction Management.* Department of Computer Science. University of Texas at Austin. TR-92-91. Mayo 1992.

[2] A.V. Aho, J. Hopcroft y J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA. 1983.

[3] K. Eswaran, J. Gray, R. Lorie, I. Traiger. *The notion of Consistency and Predicate Locks in a Database System.* Communications of the ACM. Vol 19, N° 11, Nov. 1976, pp 624-633.

[4] H. T. Kung, J. T. Robinson. *Optimistic Concurrency Control.* ACM Transactions on Database Systems. Vol 8, N° 2, June 1981, pp 312-326.

[5] D. Reed. *Implementing Atomic Actions on Decentralized Data.* ACM Transactions on Computer Systems. Vol 1, N° 1, Febrary 1983, pp 3-23.

[6] A. Silberschatz, H. Korth. *Fundamentos de Bases de Datos.* McGraw-Hill, Madrid. 1993.

[7] C.H. Papadimitriou. *The serializability of Concurrent Database Updates.* Journal of the ACM. Vol 26, N° 4, Octubre 1979, pp 631-653.

[8] C.H. Papadimitriou, P.A. Bernstein y J. Rothnie. *Some computational Problems Related to Database Concurrency Control.* Proceedings of the Conference on Theoretical Computer Science. 1977, pp 275-282.

[9] P.A. Bernstein, N. Goodman. *Concurrency control in distributed database systems.* ACM Computing Surveys. Vol 13, N° 2, Junio 1981, pp 185-222.

[10] P. Ancilotti, A. Bertolino y M. Fusani. *An approach to efficient distributed transactions.* Distributed Computing. Febrero 1988, pp 201-212.

[11] M. Zanconi, J. Ardenghi. *Alcance de Consenso en Sistemas Distribuidos.* II Congreso y Exposición Internacional de Informática. Mendoza. Junio 1994.

[12] L. Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of ACM, vol 21, N° 7, Julio 1978, pp 558-565.

[13] J.N. Gray, R. Lorie, G. Putzolu. *Granularity of Locks and Degrees of Consistency in a Shared Database.* Proceedings of the International Conference on Very Large Data Bases. 1975, pp 428-451.

[14] J.N. Gray. *Notes on Data Base Operating Systems.* Operating Systems: An Advanced Course. Springer-Verlag. Berlin. 1978.

[15] H. García Molina, B. Kogan. *Achieving High Availability in Distributed Systems.* IEEE Transactions on Software Engineering. Vol 14, N° 7. Julio 1988, pp 886-896.

[16] P.A. Bernstein, V. Hadzilacos, N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA. 1987.

[17] O. Wolfson. *The Overhead of Locking (and Commit) Protocols.* ACM Transactions on Database Systems. Vol. 12, N° 3. Septiembre 1987, pp 453-471.