Administración de una Base de Datos Replicada.

Implementación de un protocolo de cometido en un problema real.

Trabajo de Grado

Ricardo Javier Lamberti

Universidad Nacional de La Plata

Directores

Bertone, Rodolfo	Pesado, Patricia

INDICE

AGRADECIMIENTOS

Por orden cronológico,

A mis padres que me apoyaron y contuvieron siempre.

A mis abuelos donde quiera que estén.

A mis tías que pusieron el hombro para que pueda venir a estudiar.

A mi grupo de amigos, la barra, porque la peleamos juntos.

A mi hermana, por el tiempo que compartimos el ser estudiantes.

A mi esposa, sin su amor y empuje hubiera bajado los brazos hace tiempo

A mi hija por ser la razón de todo.

Y al que viene en camino…

I. Introducción

I.I. Problemática

Repsol-YPF mantiene en su red de estaciones de servicios un conjunto de sistemas para administrar diversas operaciones. Dichos sistemas se encuentran desconectados entre sí, y solo mantienen un intercambio esporádico de información a través de archivos planos.

Los repositorios de datos locales no son necesariamente procesados por el mismo motor, ni con el mismo formato. (Puede haber bases de datos bajo distintos productos de mercado, como SQL Server, Oracle, Access o, eventualmente como una colección de datos en archivos de textos)

Lo que se desea es que las diferentes aplicaciones existentes dispongan en la base de datos local que permita mantener su autonomía.

I.II. Solución propuesta

Para asegurar la autonomía se buscará que los nodos tengan réplicas de los datos globales, y de esta forma asegurar la velocidad de acceso y la disponibilidad de la información.

Con este fin se aprovechará una capa de software preexistente en todas las aplicaciones (core), redefiniéndola para lograr la administración de las múltiples bases de datos heterogéneas.

I.III. Objetivos

El objetivo de este trabajo, es diseñar, implementar y analizar la solución propuesta.

I.IV. Esquema del trabajo de grado

En el capítulo uno (1), se presentarán conceptos relacionados a base de datos distribuidas. Se estudiarán sus usos, se analizarán sus diferentes variantes, se evaluará las técnicas más comunes de resolución de conflictos y demás aspectos relacionados.

En el capítulo dos (2), se estudiará en particular la replicación de datos, por ser este tema fundamental para la solución propuesta. Se observarán las ventajas y desventajas, los diferentes esquemas de datos, métodos de propagación y regulación. Se hará especial hincapié en el mecanismo denominado "Lazy".

En el capítulo tres (3) y cuatro (4), se revisarán las técnicas de manejo de concurrencia, por ser un punto indispensable para asegurar el correcto funcionamiento del sistema. Se

focalizará en los mecanismos en los cuales los conflictos se resuelvan por retrocesos en lugar de por esperas.

En el capítulo cinco (5), se presentarán en los detalles del problema a resolver, el estado actual, y las herramientas preexistentes a la solución propuesta.

En el capítulo seis (6), presentará la solución implementada, y las razones de las diferentes decisiones tomadas durante el desarrollo.

En el capítulo siete (7), se estudiarán los resultados y las conclusiones arribadas.

CAPÍTULO 1

INTRODUCCIÓN A LAS BASES DE DATOS DISTRIBUIDAS

Un sistema de bases de datos distribuidas tiene la particularidad que los datos se almacenan en varios computadores. Estos sistemas se encuentran interconectados de algún modo, pero no comparten ni memoria principal, ni el reloj.

Se utilizará *el término localidad para hacer hincapié en la distribución física*[Korth y otros,80] de los computadores.

Las localidades participan en transacciones que pueden involucrar datos de otras localidades. Esta particularidad es lo que diferencia un sistema distribuido de uno centralizado ya que en este los datos residen en una sola localidad.

Korth y Silberchatz definen que: *Un sistema distribuido de base de datos consiste en un conjunto de localidades, cada una de las cuales mantiene un sistema de base de datos local. Cada localidad puede procesar transacciones locales. Y Además, puede participar en la ejecución de transacciones globales.* [Korth y otros,80]

En otra visión, Tanenbaum, define a *Un sistema distribuido como una colección de computadoras independientes que aparecen antes los usuarios del sistema como una única computadora.* [Tanenbaum et al., 1996]

1.1. Características de una base distribuida

Las características que conforman una base de datos distribuida ideal, fueron definidas por C.J.Date: [Burleson 1994]

1. Autonomía local
2. Independencia de un sitio central
3. Operación continua
4. Independencia de localidad
5. Independencia de fragmentación
6. Independencia de replicación
7. Procesamiento distribuido de consultas
8. Manejo de transacciones distribuidas
9. Independencia de hardware
10. Independencia de sistemas operativos
11. Independencia de redes
12. Independencia de motor de base de datos.

1.1.1. Autonomía Local

La autonomía local determina que todos los datos son propiedad y manejados por la localidad.

Los sitios de un sistema distribuido deben ser autónomos. La autonomía local significa que todas las operaciones en un sitio dado se controlan en ese sitio; ningún sitio *X* deberá depender de algún otro sitio *Y* para su buen funcionamiento (pues de otra manera el sitio *X* podría ser incapaz de trabajar, aunque no tenga en sí problema alguno, si cae el sitio *Y*, situación a todas luces indeseable). La autonomía local implica también un propietario y una administración local de los datos, con responsabilidad local: todos los datos pertenecen "en realidad" a una base de datos local, aunque sean accesibles desde algún sitio remoto. Por tanto, las cuestiones de seguridad, integridad y representación en almacenamiento de los datos locales permanecen bajo el control de la instalación local.

Si un dato es cambiado por otra localidad, se lo comunica al nodo local y este realiza el cambio en sus dominios. Esto garantiza el control y permite la correcta sincronización de las transacciones.

1.1.2. Independencia de un sitio central.

Idealmente todos los sitios son igualmente “remotos”, no habiendo un sitio destacado con respecto a otro.

La independencia local implica que todos los sitios deben tratarse igual; no debe haber dependencia de un sitio central "maestro" para obtener un servicio central, como por ejemplo un procesamiento centralizado de las consultas o una administración centralizada de las transacciones, de modo que todo el sistema dependa de ese sitio central. Este segundo objetivo es por tanto un corolario del primero (si se logra el primero, se logrará por fuerza el segundo) . Pero la "no independencia de un sitio central" es deseable por sí misma, aun si no se logra la autonomía local completa. Por ello vale la pena expresarlo como un objetivo separado. La dependencia de un sitio central sería indeseable al menos por las siguientes razones: en primer lugar, ese sitio central podría ser un cuello de botella; en segundo lugar, el sistema sería vulnerable; si el sitio central sufriera un desperfecto, todo el sistema dejaría de funcionar. [Batini et al, 1994]

1.1.3. Operación continua

Las localidades funcionan como parte de una red unificada, donde unos sitios pueden acceder a los datos de otros sitios remotos. La disponibilidad de los nodos facilita la implementación de políticas de contingencia en caso de caídas una localidad.

Además, en un sistema distribuido, lo mismo que en uno no distribuido, idealmente nunca debería haber necesidad de apagar a propósito el sistema Es decir, el sistema nunca debería necesitar apagarse para que se pueda realizar alguna función, como añadirse un nuevo sitio o instalar una versión mejorada del DBMS en un sitio ya existente.

1.1.4. Independencia de localidad

Esta propiedad define que el usuario final no conoce, ni se preocupa por la ubicación física de la información.

La idea básica de la independencia con respecto a la localización (también conocida como transparencia de localización) es simple: no debe ser necesario que los usuarios sepan dónde están almacenados físicamente los datos, sino que más bien deben poder comportarse -al menos desde un punto de vista lógico- como si todos los datos estuvieran almacenados en su propio sitio local. La independencia con respecto a la localización es deseable porque simplifica los programas de los usuarios y sus actividades en la terminal. En particular, hace posible la migración de datos de un sitio a otro sin anular la validez de ninguno de esos programas o actividades. Esta posibilidad de migración es deseable pues permite modificar la distribución de los datos dentro de la red en respuesta a cambios en los requerimientos de desempeño.

1.1.5. Independencia de fragmentación

Un sistema maneja fragmentación de los datos si es posible dividir una relación en partes o "fragmentos" para propósitos de almacenamiento físico. La fragmentación es deseable por razones de desempeño: los datos pueden almacenarse en la localidad donde se utilizan con mayor frecuencia, de manera que la mayor parte de las operaciones sean sólo locales y se reduzca al tráfico en la red. Por ejemplo, la relación empleados EMP podría fragmentarse de manera que los registros de los empleados de Buenos Aires se almacenen en el sitio de Buenos Aires, en tanto que los registros de los empleados de Londres se almacenan en el sitio de Londres. Existen en esencia dos clases de fragmentación, horizontal y vertical, correspondientes a las operaciones relacionales de restricción y proyección; respectivamente. En términos más generales, un fragmento puede ser cualquier subrelación arbitraria que pueda derivarse de la relación original mediante operaciones de restricción y proyección (excepto que, en el caso de la proyección es obvio que las proyecciones deben conservar la clave primaria de la relación original). La reconstrucción de la relación original a partir de los fragmentos se hace mediante operaciones de reunión y unión apropiadas (reunión en el caso de fragmentación vertical, y la unión en casos de fragmentación horizontal). Ahora llegamos a un punto principal: un sistema que maneja la fragmentación de los datos deberá ofrecer también una independencia con respecto a la fragmentación (llamada también transparencia de fragmentación). La independencia con respecto a la fragmentación (al igual que la independencia con respecto a la localización) es deseable porque simplifica los programas de los usuarios y sus actividades en la terminal.

1.1.6. Independencia de replicación

Se refiere a la habilidad de generar copias de una base maestra en sitios remotos.

Un sistema maneja réplica de datos si una relación dada (ó en términos más generales, un fragmento dado en una relación) se puede representar en el nivel físico mediante varias copias réplicas en muchos sitios distintos. La réplica es deseable al menos por dos razones: en primer lugar, puede producir un mejor desempeño (las aplicaciones pueden operar sobre copias locales en vez de tener que comunicarse con sitios remotos); en segundo lugar, también puede significar una mejor disponibilidad (un objeto estará disponible para su procesamiento en tanto esté accesible alguna una copia, al menos para propósitos de recuperación). La desventaja principal de las réplicas es, desde luego, que cuando se pone al día un cierto objeto copiado, deben ponerse al día todas las réplicas de ese objeto: el problema de la propagación de actualizaciones.

La réplica como la fragmentación debe ser "transparente para el usuario". En otras palabras, un sistema que maneja la réplica de los datos deberá ofrecer también una independencia de réplica (conocida también como transparencia de réplica); es decir, los usuarios deberán poder comportarse como si sólo existiera una copia de los datos. La independencia de réplica es buena porque simplifica los programas de los usuarios y sus actividades en la terminal. En particular, permite la creación y eliminación dinámica de las réplicas en cualquier momento en respuesta a cambios en los requerimientos, sin anular la validez de esos programas o actividades de los usuarios.

La replicación es un mecanismo poderoso para asegurar la disponibilidad de los datos, más allá de la caída de algún nodo. Aunque la replicación presenta el desafío de mantener las diferentes copias actualizadas.

1.1.7. Procesamiento de consultas distribuidas

Es el poder de ejecutar una consulta que involucra diferentes localidades. Una consulta global se divide en diferentes sub-consultas que son ejecutadas en diferentes localidades.

En este aspecto debemos mencionar dos puntos amplios.

Primero consideremos la consulta "obtener los proveedores de partes rojas en Rosario". Supongamos que el usuario está en la instalación de Buenos Aires y los datos están en el sitio de Rosario. Supongamos también que son *n*/ *n* registros de Rosario a Buenos Aires. Si, por otro lado, el sistema no es relacional, sino de un registro a la vez, la consulta implicará en esencia *2n* mensajes: *n* de Buenos Aires a Rosario solicitando el siguiente registro, y *n* de Rosario a Buenos Aires para devolver eses siguiente registro. Así, el ejemplo ilustra el punto en el cual un sistema relacional tendrá con toda probabilidad un mejor desempeño que uno no relacional (para cualquier consulta que solicite varios registros), quizá en varios órdenes de magnitud.

En segundo lugar, la optimización es todavía más importante en un sistema distribuido que en uno centralizado. Lo esencial es que, en una consulta como la anterior, donde están implicados varios sitios, habrá muchas maneras de trasladar los datos en la red para satisfacer la solicitud, y es crucial encontrar una estrategia suficiente. Por ejemplo, una solicitud de unión de una relación *Rx* almacenada en el sitio *X* y una relación *Ry* almacenada en el sitio *Y* podría llevarse a cabo trasladando *Rx* a *Y* o trasladando *Ry* a *X*, o trasladando las dos a un tercer sitio *Z*.

1.1.8. Manejo de transacciones distribuidas

El manejo de transacciones distribuidas se refiere a las posibilidades del sistema de manejar transacciones de actualización, inserción y borrado en múltiples localidades manteniendo la integridad global de los datos.

El manejo de transacciones tiene dos aspectos principales, el control de recuperación y el control de concurrencia, cada uno de los cuales requiere un tratamiento más amplio en el ambiente distribuido. Para explicar ese tratamiento más amplio es preciso introducir primero un término nuevo, "agente". En un sistema distribuido, una sola transacción puede implicar la ejecución de código en varios sitios (en particular puede implicar actualizaciones en varios sitios). Por tanto, se dice que cada transacción está compuesta de varios agentes, donde un agente es el proceso ejecutado en nombre de una transacción dada en determinado sitio. Y el sistema necesita saber cuándo dos agentes son parte de la misma transacción; por ejemplo, es obvio que no puede permitirse un bloqueo mutuo entre dos agentes que sean parte de la misma transacción. Sobre la cuestión específica del control de recuperación, se debe garantizar que una transacción dada sea atómica (todo o nada) en el ambiente distribuido: el sistema debe asegurarse que todos los agentes correspondientes a esa transacción se comprometan al unísono o bien que retrocedan al unísono. Este efecto puede lograrse mediante el protocolo de compromiso en dos fases.

En cuanto al control de concurrencia, esta función en un ambiente distribuido estará basada con toda seguridad en el bloqueo, como sucede en los sistemas no distribuidos.

1.1.9. Independencia de Hardware

El almacenamiento no depende del hardware instalado en las localidades.

1.1.10. Independencia del sistema operativo

Una consulta no debería ser afectada por el sistema operativo instalado.

1.1.11. Independencia de red

Los protocolos de red deberían ser transparentes para el intercambio de datos.

1.1.12. Independencia de base de datos

La independencia de base de datos posibilita el intercambio de información desde diferentes localidades sin importar el motor ni la arquitectura de las bases de datos locales.

1.2 Ventajas y desventajas

1.2.1. Ventajas

Los sistemas distribuidos basan su fortaleza en la capacidad de permitir el acceso a la información en una forma fiable y eficaz.

1.2.1.1. Utilización compartida de los datos y distribución del control

Cuando varias localidades se encuentran interconectadas, los usuarios pueden compartir información. Aunque esta propiedad también está presente en los sistemas centralizados, los administradores locales tienen la posibilidad, según el diseño de la base, de reservarse cierta autonomía local.

Esta autonomía constituye una ventaja importante.

1.2.1.2. Fiabilidad y disponibilidad

La fiabilidad está dada por la posibilidad que el fallo de una localidad no implique necesariamente la caída del sistema. Esta permanencia del sistema, más allá de los fallos locales, provoca que los datos estén disponibles por más tiempo.

Si se produce un fallo en una localidad en un sistema distribuido, es posible que las demás localidades puedan seguir trabajando. En particular, si los datos se repiten en varias localidades, una transacción o aplicación que requiere un dato específico puede encontrarlo en más de una localidad. Así, el fallo de una localidad no implica necesariamente la desactivación del sistema.

1.2.1.3. Agilización del procesamiento de consultas

La multiplicidad de localidad abre las puertas a la distribución del procesamiento de consultas.

Si una consulta comprende datos de varias localidades, puede ser posible dividir la consulta en varias subconsultas que se ejecuten en paralelo en distintas localidades. En los casos en que hay repetición de los datos, el sistema puede pasar la consulta a las localidades más ligeras de carga.

1.2.2. Desventajas

Las desventajas principales de un sistema distribuido están dadas por la mayor complejidad para lograr la coordinación adecuada entre localidades.

1.2.2.1. Coste del desarrollo del software

Los sistemas distribuidos son más difíciles de estructurar y, por lo general, su costo es mayor. Su mayor complejidad, a menudo se traduce en altos gastos de construcción y mantenimiento. Ya que existen más componentes de hardware, hay más cantidad de cosas por aprender y más interfaces susceptibles de fallar. El control de concurrencia y recuperación de fallas puede convertirse en algo complicado y difícil de implementar, puede empujar a una mayor carga sobre programadores y personal de operaciones y quizá se requiera de personal más experimentado y más costoso.

El procesamiento de bases de datos distribuido es difícil de controlar. Una computadora centralizada reside en un entorno controlado, con personal de operaciones que supervisa muy de cerca, y las actividades de procesamiento pueden ser vigiladas, aunque a veces con dificultad. En un sistema distribuido, las computadoras de proceso, residen muchas veces en las áreas de trabajo de los usuarios. En ocasiones el acceso físico no está controlado, y los procedimientos operativos son demasiado suaves y efectuados por personas que tienen escasa apreciación o comprensión sobre su importancia. En sistemas centralizados, en caso de un desastre o catástrofe, la recuperación puede ser más difícil de sincronizar.

1.2.2.2. Mayor posibilidad de errores

La concurrencia de las localidades hace más difícil su correcto funcionamiento. El procesamiento de base de datos distribuida puede resultar menos confiable que el procesamiento centralizado. Depende de la confiabilidad de las computadoras de procesamiento, de la red, del DDBMS, de las transacciones y de las tasas de error en la carga de trabajo. Un sistema distribuido puede estar menos disponible que uno centralizado. Estas

dos desventajas indican que un procesamiento distribuido no es ninguna panacea. A pesar de que tiene la promesa de un mejor rendimiento y de una mayor confiabilidad, tal promesa no está garantizada.

1.2.2.3. Mayor tiempo extra de procesamiento

Intercambios de mensajes y los cálculos adicionales que se requieren para lograr la sincronía generan tiempos extras que no existen en los sistemas centralizados.

El rendimiento puede ser peor para el procesamiento distribuido que para el procesamiento centralizado. De nuevo, depende de la naturaleza de la carga de trabajo, la red, el DDBMS y las estrategias utilizadas de concurrencia y de falla, así como las ventajas del acceso local a los datos y de los procesadores múltiples, ya que éstos pueden ser abrumados por las tareas de coordinación y de control requeridas. Tal situación es probable cuando la carga de trabajo necesita un gran número de actualizaciones concurrentes sobre datos duplicados, y que deben estar muy distribuidos.

El problema más grande es que las redes de comunicación (las de larga distancia en especial) son lentas. El objetivo es reducir al mínimo el tráfico en la red y esto implica que el proceso mismo de optimización de consultas debe ser distribuido, además del proceso de ejecución de las consultas. Es decir, un proceso representativo consistirá en un paso de optimización global, seguido de pasos de optimización local en cada unos de los sitios afectados.

El diseñador de bases de datos debe balancear estas ventajas y desventajas, existen muchos enfoques a la hora de distribuir los datos, dando lugar a modelos absolutamente distribuidos, hasta diseños que incluyen un alto grado de centralización.

1.3. Propiedades de las transacciones

Las transacciones tienen cuatro propiedades fundamentales. Estas son:

Atómicas: Para el mundo exterior, la transacción ocurre de manera indivisible
Consistentes: La transacción no viola los invariantes del sistema
Aisladas: Las transacciones concurrentes no interfieren entre sí.
Durables: Una vez comprometida la transacción, los cambios son permanentes

Se utiliza la sigla *ACID* (iniciales en Ingles) para referirse a estas propiedades. [Tanenbaum, 1996]

La primera propiedad fundamental de todas las transacciones es que son atómicas. Esta propiedad garantiza que cada transacción no ocurre o bien, se realiza en su totalidad. Mientras se desarrolla una transacción, otros procesos no pueden ver los estados intermedios.

La segunda propiedad dice que son consistentes. Lo que significa que si el sistema tiene ciertos invariantes que deban conservarse siempre, si éstos se conservan antes de la transacción, entonces deben conservarse después de ella.

La tercera propiedad dice que las transacciones son aisladas o serializables. Lo que esto significa es que si dos o más transacciones se ejecutan al mismo tiempo, para cada una de ellas y para todos los demás procesos, el resultado final aparece como si todas las transacciones se ejecutasen de manera secuencial en cierto orden.

La cuarta propiedad dice que las transacciones son durables. Se refiere al hecho que una vez comprometida la transacción, no importa lo que ocurra, la transacción sigue adelante y los resultados se vuelven permanentes. Ninguna falla después del compromiso puede hacer perder los resultados o provocar pérdidas de los mismos.

1.4. Distribución de los datos

Los datos de una transacción se pueden almacenar en una base de datos distribuida de diversas maneras. Pueden estar replicados, el sistema mantiene varias copias idénticas de la relación en diferentes localidades. Fragmentados, la relación se divide en varios fragmentos que se almacenan en localidades diferentes. O también, ambos modos a la vez.

1.4.1. Fragmentación

En la fragmentación una relación se encuentra esparcida en muchas localidades. Cada partición debe contener la información suficiente para poder rearmar la relación original.

La fragmentación se puede dar en sentido horizontal o vertical.

La primera envía cada tupla de la relación a una o más localidades, de tal modo que la tabla queda partida en tuplas, representando una selección de tuplas en cada nodo. La relación puede reconstruirse uniendo los diferentes fragmentos.

La segunda, la división se realiza descomponiendo la relación, de tal modo que algunas columnas de la tupla quedan en una localidad y otras en otra. La relación se puede componer utilizando producto natural.

Obviamente ambos métodos no son excluyentes y pueden combinarse en una fragmentación "mixta"

1.4.2. Replicación

La replicación realiza una copia de una relación en una o más localidades.

Podría existir un caso donde todas las tuplas se encuentran en todas las localidades, esto se denomina "Replicación total".

La replicación tiene la ventaja de aumentar la disponibilidad de los datos, ya que el sistema podría localizar la relación r aunque la localidad "originaria" este desconectada.

Otra ventaja es el mayor paralelismo para las operaciones de lectura, cuanto más copias haya, las consultas tienen más probabilidad de hallar una copia local, lo cual reduce el intercambio de información entre localidades.

La desventaja está dada por la complejidad de mantener sincronizadas las copias al momento de las actualizaciones. En adelante nos dedicaremos a buscar una solución a este problema.

CAPÍTULO 2

REPLICACIÓN

Los datos son replicados en múltiples nodos de una red para aumentar la performance y la disponibilidad de los datos.

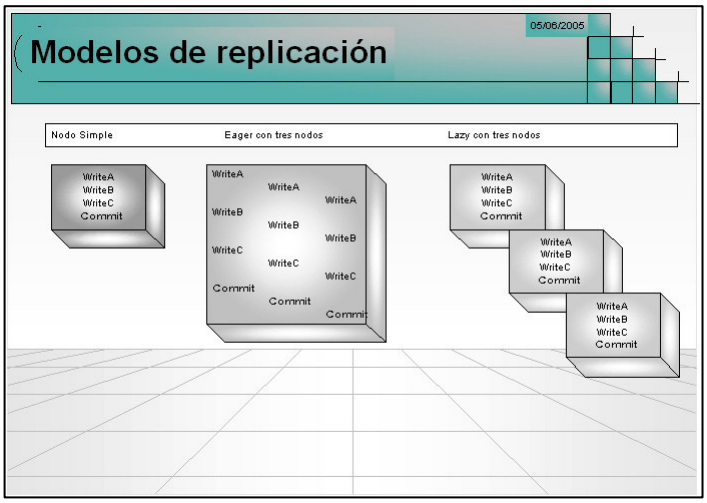
La replicación de datos es una herramienta muy poderosa a la hora de facilitar las consultas, pues, es muy probable que los datos se encuentren localmente, sin necesitar mensajería de intercambio de datos. En contrapartida, se complica extremadamente la coordinación de la actualización de las copias. [Buretta 1997]

Una primera aproximación la llamaremos enfoque Eager que fue descrito por Eager en 1986 [Eager et al., 1986] y mejorado por Gray [Gray et al., 1996], este mecanismo se basa en bloquear los datos durante la actualización. Otro enfoque, es Lazy [Gray et al., 1996], donde se realizan los cambios sin bloquear los datos, y se produce una reconciliación de datos para solucionar los conflictos. [Kemme et al., 1998]

En lo que resta del capítulo analizaremos ambos mecanismo: Eager y Lazy.

2.1. Modelo Eager

El modelo Eager de replicación mantiene a todas las réplicas exactamente sincronizadas en todos los nodos. De tal modo, que todas las localidades se actualizan en forma atómica. Pero este método reduce la performance e incrementa los tiempos de respuesta de las transacciones. [Kemme et al., 2000]



El mecanismo espera o aborta una transacción no finalizada, y así evita el peligro de caer en una violación a la serialización.

La versión más sencilla del método Eager simplemente prohíbe actualizar si una localidad se encuentra desconectada. Otras versiones más flexibles aumentan la disponibilidad permitiendo la actualización cuando un conjunto de nodos las dio como

aprobadas, métodos de quórum o cluster [García Molina 1982], donde los nodos desconectados son informados de los cambios al momento de reconectarse.

La propagación Eager puede utilizar para la sincronización una localidad coordinadora, Eager-Master. O un modelo, donde la coordinación se realiza con nodos equivalentes, Eager-Group. Se profundizará en los diferentes algoritmos en el capítulo III.

Propagación / Propiedad	Lazy	Eager
Group	N Transacciones N Objetos dueños	Una Transacción N Objetos dueños
Master	N Transacciones Un objeto dueño	Una Transacción Un objeto dueños

Aunque todos los nodos estén conectados a la vez, una actualización puede fallar por *"deadlock"*.

El siguiente análisis calcula las tasas de espera y *"deadlock"* para un sistema de un nodo único. [Gray et al., 1996]

Sea,
T = Número de concurrentes transacciones en un nodo.
A = Número de actualizaciones en una transacción
AT = tiempo en que se realiza una acción
DBs= tamaño de la base de datos
TPS = Número de transacciones por segundo originadas en el nodo
N = Número Nodos

En una sistema con nodo único, supongamos una carga donde hay (T x A)/ 2 recursos tomados. Suponiendo que los objetos de la base de datos son usados uniformemente. Las posibilidades de requerir un recurso tomado por otra transacción es (TxA)/(2xDBs).

Una transacción realiza acciones en cada requisición, las posibilidades de que tenga que esperar para usar un objeto durante su tiempo de vida es [Gray et.al, 1996][Gray & Reuter pp.428]:

$$PW = 1-(1-(TxA)/(2xDBs)) \quad (1)$$
$$Acciones = (TxA^2)/(2xDBs) \quad (2)$$

Un *"deadlock"* consiste en un ciclo de transacciones esperando por otra. La probabilidad que una transacción forme un ciclo de tamaño dos es PW^2 dividido el número de transacciones. Ciclos de largo j son proporcionales a PW^j . Aplicando la ecuación, la probabilidad que una transacción entre en *"deadlock"* es:

$$PD = (TPS \times AT \times A5)/(4xDBs2) \quad (3)$$

La ecuación (3) da la posibilidad de *"deadlock"* para una transacción. La tasa de *"deadlock"* para una transacción es la probabilidad de caer en *"deadlock"* en el siguiente segundo. Esto es PD dividido por el tiempo de vida de la transacción.

Tasa_deadlock_por_transaccion = (TPS x Actions4)/(4xDBs2) *(4)*

Un nodo corre muchas transacciones concurrentemente, esto es:

Tasa_deadlock_nodo = (TPS2 x AT x Actions4)/(4xDBs2) *(5)*

Supongamos ahora que muchos sistemas son replicados usando el método Eager. Cada nodo podrá iniciar TPS transacciones por segundo. El tamaño, duración y la tasa de aceptación para el método será como sigue:

Tamaño transacción = A x N
Duración_transacción = A x N x AT
Total_TPS = TPS x N *(6)*

Cada nodo está haciendo ahora el trabajo de aplicar lo que generan otros nodos. Ahora cada transacción requiere de más acciones (N x A) y tienen un mayor tiempo de vida. Como resultado el número de transacciones en el sistema se cuadruplica con el número de nodos.

Total_transacciones = TPS x A X AT x N2 *(7)*

Esta alza en las transacciones activas en el modelo de Eager toma N veces más tiempo que con el sistema Lazy, el cual genera N transacciones. La tasa de acciones es como sigue:

Tasa_acciones = Total_TPS x A x N2 *(8)*

La tasa de acciones y el número de transacciones activas son iguales para ambos métodos. El Eager tiene pocas transacciones pero largas, el Lazy muchas transacciones cortas.

Ignorando el manejo de mensajes, la probabilidad que una transacción espere puede ser calculada como sigue:

PW_eager = Total_transacciones x A4/(4xDBs))=(TPS x AT x A3 x N2)/(2xDBs) *(9)*

Esto es aproximadamente lo que una transacción espera, la tasa esta dada por:

Tasa_eager_espera = (PW_eager / Duración_transacción) x Total_transacciones = ((TPS2 x AT x (A x N) 3) / (2 x DBs) *(10)*

Por lo tanto, la probabilidad de que una transacción entre en *"deadlock"* es:

PD_eager = (Total_transaccion xA 4)x(4 x DBs2)=(TPSxATxA5xN2)/(4xDBs2) *(11)*

Tasa_total_eager_"deadlock" = (TPS2 x AT x A5 x N3) / (4xDBs2) *(12)*

El riesgo de *"deadlock"* se potencia en tres sobre la cantidad de nodos y en cinco con respecto al tamaño de la transacción. Esto es el agregar un nodo a la red aumenta en mil el riesgo de *"deadlock"* y el incremento del tamaño de una transacción en cien mil.

En conclusión, el método Eager tiene dos grandes desventajas.

- Los nodos móviles no pueden usar este esquema mientras están desconectados
- La posibilidad de falla de una transacción aumenta rápidamente a medida que crece el tamaño de una transacción y el número de nodos.

2.2. Modelo Lazy

El modelo Lazy de replicación actualiza la copia local, y luego asincrónicamente comienza a replicar a otras localidades. Este método usado compulsivamente puede generar inconsistencias entre las réplicas, debido a las dificultades para mantener la serialización entre nodos. [Breitbart et al., 1999]

Existen dos esquemas básicos de propagación Master y Group.

2.2.1. Esquema Lazy-Master

La replicación por maestros asigna un dueño a cada objeto, por lo tanto, diferentes objetos podrían tener diferentes propietarios. Estos se encargan de actualizar los cambios, determinar el valor del objeto, y propagar a otras réplicas. [Pacitti et al., 1998]

Cuando una localidad quiere cambiar un objeto de la cual no es dueña, envía un RPC (llamado a procedimiento remoto) al nodo propietario del objeto.

Para simplificar el análisis supondremos que los nodos originadores de la transacción envían , usando *"broadcast",* las réplicas a todos los nodos esclavos después que la master aplicó los cambios.

Las actualizaciones para los esclavos son marcadas con la fecha de generación para asegurar la aplicación en el mismo orden. [Gray et al., 1996]

Este mecanismo no es apropiado para aplicaciones móviles, es decir, aplicaciones donde su operatoria se desarrolla principalmente desconectada de la red..

Este método no requiere de mecanismos de reconciliación. Los conflictos son resueltos por esperas o *"deadlock"*. Ignorando los tiempos de transmisión de mensajes, la tasa de deadlocks es similar a la de un nodo único, donde las tasas son muy eficientes.

La transacción Lazy Master opera solo sobre la copia Master de los objetos. Pero como hay en cada nodo muchos usuarios, y concurrencia de transacciones podría existir un conflicto.

La ecuación sería similar a la (5):

Tasa_lazy_master_deadlock = ((TPS x N)2 x AT x A5) / (4 x DBs2)

Este resultado como se verá es mejor que el de replicación lazy-group. Lazy Master envía pocos mensajes durante la transacción base y los completa rápidamente. Lazy Master tiene una tasa de deadlock muy inferior al método Eager, principalmente porque las transacciones son mucho mas cortas.

2.2.2. Esquema Lazy-Group

Este tipo de actualización permite a cualquier nodo actualizar los datos locales. Cuando la transacción aplica, esta es enviada a otros nodos. Esto puede provocar que varios nodos actualicen un mismo objeto al mismo tiempo. El mecanismo detecta este inconveniente y propone utilizar mecanismos de reconciliación.

Marcas de tiempo son utilizadas para detectar y reconciliar transacciones. Cada objeto lleva en sí el momento de su más reciente actualización. De este modo, las localidades al recibir una actualización foránea pueden chequear si dicha marca coincide con la marca almacenada localmente. De coincidir, se actualiza la marca local, en cambio, si no hay coincidencia se detecta un conflicto y se requiere "reconciliación". [Gray et al., 1996]

Las transacciones que deben esperar en un módelo Eager son análogas en el tiempo de reconciliación al Lazy-group. Aunque, las esperas siempre son más frecuentes que los "*deadlock*", pues se necesitan dos esperas para que se produzca un "*deadlock*". De este modo si las esperas son raros eventos, los "*deadlock*" son aun mas raros. En la replicación Eager las esperas causan demoras, en cambio, los "*deadlock*" producen fallas de aplicación. Con Lazy es mucho más frecuentes que las esperas determinen la frecuencia de reconciliación. La ecuación que calcula la tasa de espera es la siguiente:

Tasa_lazy_group_reconciliation = (TPS2 x AT x (A x N)3)/ (2 x DBs)

El tiempo de propagación de mensajes en este caso es importante, pues cuanto mas tiempo tarde en determinarse el estado global de la red, mayor es el peligro de reconciliación.

El peor caso se da con un nodo móvil, el cual permanece la mayor parte del tiempo desconectado del sistema. Supongamos que el nodo acepta y aplica durante todo un día transacciones. A la noche se conecta al resto de la red y comienza a aceptar las propagaciones pendientes para el y a enviar las propias. El tiempo de propagación del mensaje fue 24 horas.

Si dos transacciones de dos diferentes nodos actualizan el mismo dato durante un período de desconexión, se necesitará reconciliar. Se planteará cual es la posibilidad de que dos transacciones colisionen durante un período de desconexión TD.

Si cada nodo actualiza una pequeña porción de la base de datos cada día, entonces, el número de objetos pendientes al reconectar sería (OU):

OU = TD x TPS x A

Cada una de estas actualizaciones aplica a todas las réplicas de un objeto. Las actualizaciones pendientes (IU) de este nodo hacia el resto de la red son (N-1)

IU = (N-1) x TD x TPS x A

Si los objetos pendientes de recibir y los por enviar se superponen, entonces, se necesita reconciliar. La probabilidad de reconciliar es:

P(colisión) = (IU x OU) / DBs = (N x (DT x TPS x A)2) / DBs

Esta es la probabilidad de un nodo, la global sería

Tasa_lazy_group_reconciliacion=P(colisión)x(N /TD) =(TDx(TPS x A x N)2)/DBs

La naturaleza cuadrática de la fórmula hace pensar que funciona para pocos nodos y transacciones simples, pero se volvería inestable si el sistema es escalado.

2.3. Ventajas y desventajas

El modelo Eager asegura la actualización correcta y sincrónica de los datos. Para lograrlo introduce bloqueos y peligros de *"deadlocks"*. Estos bloqueos pueden ser inaceptables para algunos tipos de sistemas, especialmente si los nodos están muy distantes o las conexiones no son permanentes. [Pacitti et al., 1999]

El modelo Lazy actualiza la información localmente, y luego propaga los cambios. Esto elimina los problemas de bloqueo pero genera el peligro potencial que dos localidades actualicen el mismo dato, produciendo conflictos.

Para solucionar estos problemas se han enunciado dos esquemas. El primero, master, elimina la posibilidad de que dos nodos actualicen los mismos datos, dándole a uno el poder exclusivo de decidir sobre estos. Este método requiere que el nodo maestro esté conectado, esto no es, evidentemente, útil con nodos cuya operatoria se realice principalmente desconectado.

El esquema group, no requiere la conexión pero necesita de un mecanismo de reconciliación en caso de actualizaciones simultáneas de datos. Cabe aclarar, que en ciertos sistemas las reconciliaciones en algunos casos pueden ser inaceptables.

La replicación de datos todo el tiempo en todos los nodos no es una tarea sencilla. No existe una solución que abarque todos los casos. Se pretendería que todos los nodos tengan su réplica con absoluta disponibilidad, sin bloqueos y sin reconciliaciones, más esto no es posible. Por lo tanto, nos queda, buscar la mejor solución para el problema puntual que se presente.

2.4. Comunicación en grupo

Dado que la solución que se implementará utilizará política de grupos, se dará una explicación mas detallada de la misma.

Los procesos simples de comunicación se realizan entre dos entidades, un emisor y un receptor. Existen modelos más complejos donde varios interlocutores se comunican. De tal modo, un emisor le envía un mensaje a un conjunto de receptores a la vez.

Un grupo es una colección de procesos que actúan juntos en cierto sistema o de alguna forma determinada por el usuario [Tanenbaum et al., 96]. La propiedad fundamental de todos los grupos es que cuando un mensaje se envía al propio grupo todos lo reciben, es una forma de comunicación uno a muchos.

Los grupos son dinámicos. Se pueden crear nuevos grupos y destruir grupos anteriores. Un proceso puede unirse a un grupo o puede abandonarlo. Un proceso puede ser miembro de varios grupos a la vez. En consecuencia, se necesitan mecanismos para el manejo de grupos y la membresía de los mismos.

La implementación de la comunicación por grupos depende en gran parte del hardware. En ciertas redes, es posible crear una dirección especial de red a la que pueden escuchar varias máquinas, esta técnica se llama multitransmisión (multicast). [Agrawal et al., 1997]

Las redes que no tienen multitransmisión a menudo tienen transmisión global (broadcast), lo que significa que los paquetes que contienen cierta dirección se entregan a todas las máquinas. Este mecanismo es menos eficiente pues como se reciben todos los paquetes por software se debe descartar los que no pertenecen al grupo de la máquina.

Por último, si no se tienen ninguno de los dos mecanismos se debe optar por la implementación manual, donde paquetes individuales son trasmitidos a cada miembro de un grupo.

Al envío uno a mucho de paquetes de aquí en adelante se lo nombrará con su sigla en inglés "*broadcast*" o "*multicast*" según corresponda.

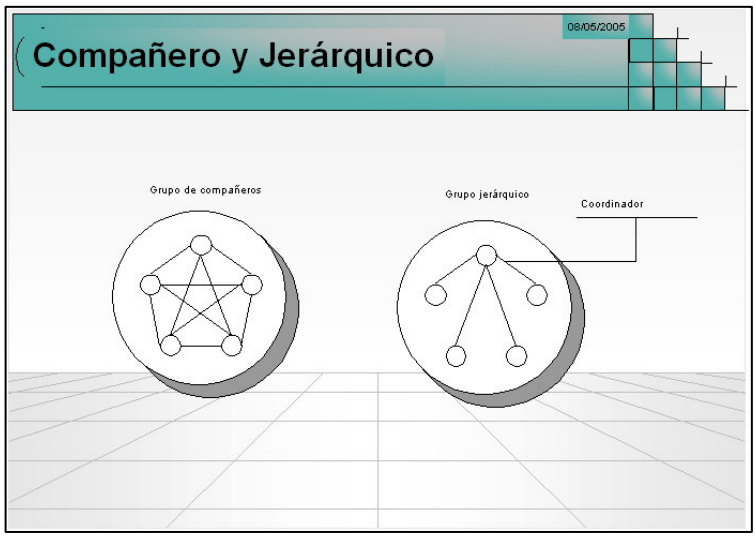
2.4.1. Grupos cerrados y abiertos.

Los sistemas que soportan la comunicación por grupo se pueden dividir en dos categorías según quien pueda enviar a quien. Algunos sistemas soportan grupos cerrados, donde solo los miembros del grupo pueden enviar hacia el grupo. Los extraños no pueden enviar mensajes al grupo como un todo, aunque pueden enviar mensajes a los miembros en particular. En contraste, otros sistemas soportan los grupos abiertos, que no tienen esta propiedad.

2.4.2. Grupos compañeros y jerárquicos.

Otro método de clasificación se basa en como están organizados. Si todos lo procesos son iguales, nadie es el jefe y las decisiones se toman en forma colectiva, se dice, que el grupo es compañero. En cambio, si existe una cierta jerarquía dentro del grupo, de tal manera que una o más entidades coordinen la forma y carga de los trabajos a realizar, se está ante el modelo jerárquico.

Las ventajas del grupo de compañeros es la ausencia de puntos de fallas. Aunque se caigan localidades y el sistema se reduzca, sigue respondiendo. La desventaja está dada por la complejidad que resulta a la hora de la toma de decisiones.



En el jerárquico, la pérdida del líder puede traer un agobiante alto, pero en contraposición los mecanismos de decisión son muy sencillos.

2.4.3. Membresía al grupo.

Se utiliza la comunicación en grupo, se requiere cierto método para la creación y eliminación de grupos, así como permitir que los procesos se unan o dejen grupos.

Un posible método es tener un servidor de grupos, al cual se le envían todas las solicitudes. Este método es directo, simple y fácil de implementar, pero si el servidor de grupos deja de funcionar todo debe reconstruirse de cero.

El método opuesto es manejarlo de manera distribuida. Un miembro envía un mensaje a todos los participantes si quiere ingresar y se despide de todos si quiere salir.

Existen dos aspectos asociados a la membresía que requieren ser analizados con más profundidad. En primer lugar, si un miembro falla, sale del grupo. El problema es que no existe un anuncio apropiado a los demás miembros, estos deben notarlo por la falta de respuesta del miembro perdido.

Otro punto importante es que la entrada y la salida deben estar sincronizadas con el envío de mensajes. No se deben enviar más mensajes a un miembro que ya se fue y enviar los mensajes a los miembros nuevos. Generalmente se logra integrando la mensajería de grupos a la mensajería.

Otro punto se da si fallan tantas máquinas que el grupo ya no puede funcionar. Se necesita cierto protocolo de reconstrucción. Algún miembro deberá tomar la iniciativa.

2.4.4. Propiedades de la mensajería.

2.4.4.1. Atomicidad

La mayoría de los sistemas de comunicación por grupos están basados de modo que un mensaje llegue a todos sus miembros o no llegue a ninguno. Esta propiedad se conoce como atomicidad. [Lampson 1981]

La atomicidad es deseable pues facilita la programación de los sistemas distribuidos. Si un proceso envía un mensaje al grupo no tiene que preocuparse por verificar que haya llegado a todos los miembros.

La implementación de la transmisión atómica no es sencilla, pues es preciso garantizar los envíos aún en presencia de fallas en los miembros.

2.4.4.2. Ordenamiento

Es importante que los mensajes dentro de un grupo lleguen en el mismo orden a todos los participantes.

La complejidad de estos algoritmos estaá dada por el siguiente problema: Aún sin fallas en los miembros es posible que los mensajes lleguen en distintos órdenes a diferentes miembros. Supóngase que hay tres procesos, el proceso a genera el mensaje 0 y lo retransmite a b y c, en el mismo momento, b genera el mensaje 1 y lo transmite a c y a, c por lo tanto, podría recibir el mensaje 1 de b y luego, el 0 de a, y en a se aplicó el 0 generado por el mismo y luego, el 1 de b. A su vez, b generó el 1 y luego recibió el 0.

La solución a este problema está dada por supeditar la aplicación de los mensajes a un orden global. Este orden no es fácil de obtener por lo que algunos sistemas ofrecen aproximaciones a esta solución, donde dado dos mensajes cercanos en tiempo el sistema elige cual es el "primero" y el "segundo" y lo transmite en ese orden a toda la red, aunque este no sea el orden "real" de aparición.

2.4.4.3. Escalabilidad

Muchos sistemas funcionan bien mientras los miembros sean un número limitado. Es importante que las propiedades antes expuestas se mantengan aún cuando se escale la cantidad de miembros.

Es muy probable que el sistema este compuesto por varias redes unidas por compuertas, y estas no permitan las propagaciones múltiples.

CAPÍTULO 3

SINCRONIZACIÓN

El manejo de la sincronía en los sistemas distribuidos, como se ha mencionado, es mucho más complejo que en sistemas centralizados. Asegurar el orden, la atomicidad y la exclusión mutua no tienen un solución única apta para todas las circunstancias, sino que presentan ventajas y desventajas que el desarrollador debe sopesar. El hecho simple de determinar si un evento precede o antecede a otro requiere una reflexión cuidadosa.

3.1. Sincronización de relojes

Es fundamental en un sistema distribuido establecer el orden de las cosas, para saber si algo sucedió antes o después se debe contar con un reloj que indique el tiempo del sistema, por ser un sistema distribuido no existe un reloj único y global. En lo subsiguiente se analizarán varios mecanismos para sincronizar los relojes de los miembros.

3.1.1. Relojes lógicos

Todas las computadoras tiene un "reloj", más bien un cronómetro, por lo general un cristal de cuarzo. Cuando el cristal de cuarzo se lo mantiene sujeto a tensión oscila a una frecuencia bien definida que depende del tipo de cristal, la forma en que se corte y la magnitud de la tensión. A cada cristal se le asocian dos registros, un contador y un registro base. Cada oscilación del cristal disminuye el contador en uno, cuando el contador toma valor 0, se genera una interrupción y el contador se vuelve a cargar mediante el registro mantenedor. De esta forma es posible programar un cronómetro de modo que genere una interrupción 60 veces por segundo. Cada interrupción recibe el nombre de marca de reloj.

Cuando el sistema arranca por primera vez, el operador ingresa la fecha y la hora, las cuales se convierten al número de marcas después de cierta fecha conocida y se guarda en la memoria. En cada marca de reloj, el procedimiento del servicio de interrupciones añade uno al tiempo guardado en memoria, de esta manera el reloj (de software) se mantiene actualizado.

En el caso de una computadora y un reloj no importa si este se desfasa un poco, puesto que todos los procesos utilizan el mismo reloj. Pero, tan pronto se comienza a trabajar con varios equipos, cada uno con su reloj, la situación es distinta.

Lamport señalo que la sincronización de relojes no tiene que ser absoluta. [Lamport et al., 1990] Si dos procesos no interactúan, no es necesario que sus relojes estén sincronizados, puesto que la carencia de sincronización no sería observable y por lo tanto no provocaría

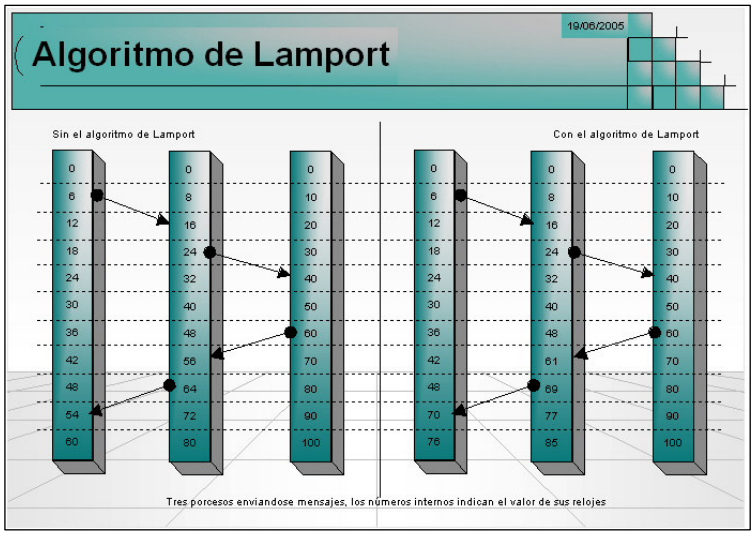
problemas. Además, señaló que lo que importa, por lo general, no es que todos los procesos concuerden en manera exacta en la hora, sino que coincidan en el orden en que ocurren los eventos.

Para la mayoría de los fines, basta que todas las máquinas coincidan en la misma hora. No es esencial que esa hora coincida con la hora real. Lo que importa es la consistencia interna de los relojes, no en su particular cercanía a la hora real. Por esto, en estos casos se hablará de relojes lógicos [Tanenbaum et al., 96], si se requiere que estén sincronizados con la hora real se los denominará relojes físicos.

Para sincronizar los relojes lógicos, Lamport definió una relación llamada "ocurre antes de". La expresión a → b, indica que todos los procesos coinciden en que primero ocurre el evento a y luego el b.

"Ocurre antes de" es una relación transitiva, de modo que si a → b y b → c, entonces a → c. Si dos procesos x e y, están en procesos diferentes que no intercambian mensajes (ni siquiera por medio de un tercero) entonces x → y, ni y → x son verdaderos. Se dice que estos eventos son concurrentes.

Lo que se necesita es un forma de medir tiempo tal que a cada evento a le podamos asignar un valor de tiempo C(a) en el que todos los procesos estén de acuerdo. Estos valores de tiempo deben cumplir la propiedad que si a → b entonces C(a) → C(b) .



Considérense n procesos que se ejecutan bajo diferentes relojes intercambiando mensajería. Todos los mensajes traen consigo la hora de generación en el proceso emisor. Cuando este mensaje llega a un proceso y este nota que la hora de salida es mayor a su hora actual, actualiza su hora actual a un instante posterior a la hora de salida del mensaje.

Este algoritmo presenta problemas cuando dos procesos se generan con la misma marca de inicio. Para esto, se puede adicionar a la marca un decimal indicando el número de proceso que permitirá desempatar en dichos casos.

3.2. Exclusión mutua

La utilización de las regiones críticas es un medio poderoso de obtener sincronía entre procesos distribuidos.

3.2.1. Algoritmo centralizado

La forma mas sencilla de lograr la exclusión mutua es usar el mismo mecanismo que se usa con una sola localidad, se elige un proceso coordinador. Siempre que se desee entrar en sección crítica se envía un mensaje al coordinador, donde se indica la sección a la que se desea entrar y se pide permiso. Cuando el coordinador decide darle el permiso envía una respuesta otorgándolo.

Es sencillo ver que un algoritmo así garantiza la exclusión mutua. Pero el coordinador se convierte en un punto de fallas y un cuello de botella.

3.2.2. Algoritmo distribuido

Con frecuencia el punto de falla del sistema centralizado es inaceptable. Ricart y Agrawala [Ricart, 1981] presentaron el siguiente método:

Cuando un algoritmo decide entrar en sesión crítica construye un mensaje con el nombre de esta, su número de proceso y la hora actual. Se supone que los mensajes son confiables, es decir, cada mensaje tiene su reconocimiento. Al recibir el mensaje los demás procesos realizan acciones de acuerdo a su estado con respecto a la sesión crítica solicitada.

- Si el receptor no está en la región crítica y no desea entrar en ella, envía un mensaje de aceptación.
- Si el receptor está en la región crítica, no responde, sino que encola el mensaje.
- Si el receptor desea entrar en sesión crítica, pero no lo ha logrado todavía, compara las marcas de tiempo del mensaje con las de su solicitud y si es menor el del mensaje responde aceptando. En cambio, si la propia es menor, encola el mensaje y aguarda.

Cuando el proceso solicitante recibe todas las aceptaciones entra en región crítica.

En comparación al algoritmo centralizado ahora tenemos n puntos de falla en vez de uno. Una mejora es que siempre se responda con una confirmación de llegada de mensaje. De este modo, luego de un par de reintentos se puede suponer que la localidad está muerta.

Este mecanismo funciona mejor en sistemas donde se pueden enviar multimensajes "*broadcast*", pues de no existir esta facilidad se debe mantener una lista actualizada de los miembros del grupo.

Otro problema que se produce cuando analizamos algoritmos centralizados, es que al atender a todos los nodos, el centralizador, podría transformarse en un cuello de botella.

Pero en contraposición el administrador decide por si solo cuando bloquear. En el entorno distribuido se obliga a todos a decidir sobre las regiones críticas.

Una mejora posible es suponer que no se necesita que todos otorguen su permiso. Alcanza con que la mitad más uno lo haga, pues esta cantidad solo puede conseguirla una localidad a la vez.

3.2.3. Algoritmo de anillo

Otro método para lograr exclusión mutua es el basado en un anillo. Se establece un orden lógico entre las localidades. No importa el mecanismo con que se establece este orden, lo importante es que cada entidad sabe cual es la siguiente.

Al iniciar se le otorga al primer proceso un "*token*" el cual circula por todo el anillo pasando de la entidad k a la k+1. Cuando una localidad recibe el token verifica si desea entrar en región crítica, si lo desea realiza el trabajo. Luego, haya o no entrado a sesión crítica pasa el token al siguiente.

El anillo es a las clara correcto. El problema se da en caso de pérdida del token, el cual debe se regenerado. La pérdida del token es difícil de detectar pues no está previsto, a priori, cuanto tiempo demorará cada proceso en el uso del token.

3.3. Algoritmo de elección

Muchos de los algoritmos distribuidos se basan en la idea de un coordinador o que desempeñe algún papel especial. Puede suceder que todas las localidades este en posibilidad de brindar dicho servicio y que en caso de fallar se pueda elegir otra y seguir operando. Se analizarán a continuación los algoritmos clásicos.

3.3.1. Algoritmo de supremacía

En este algoritmo [Molina 1982] cuando un proceso observa que un coordinador no realiza su trabajo inicia su proceso de elección de la siguiente manera (Todos las entidades participantes tienen un orden jerárquico asociado)

- envía un mensaje a los demás procesos con su número de orden
- Si nadie responde, gana la elección y se convierte en coordinador
- Si un proceso tiene un número mayor responde y toma el control.

Si un proceso entra en el grupo y no conoce al coordinador envía un mensaje de elección y si es el mayor se transformará en el nuevo coordinador, sino se pondrá a las ordenes del ganador.

3.3.2. Algoritmo de anillo

Cuando algún proceso observa que el coordinador no funciona, genera un token formador, donde guarda su identificador de localidad y lo pasa al siguiente, si el siguiente no responde, lo envía al siguiente del siguiente, y así hasta obtener un respuesta o llegar a si mismo.

Al recibir un token formador, un anillo agrega su identificador a la lista y lo pasa al siguiente.

En un cierto momento el mensaje regresa a su creador, este lo nota por encontrarse él mismo en la lista. Cuando esto sucede nombra coordinador al miembro con el número mas alto, y reenvía un nuevo token informando los resultado y los nuevos componentes del anillo.

Que varios procesos detenten la falta de coordinador e inicien sendos token formadores no afecta al algoritmo, pues ambos deberían arrojar el mismo resultado.

CAPÍTULO 4

MANEJO DE REPLICAS COORDINADAS

Los algoritmos vistos hasta aquí y sus variantes forman las piedras fundamentales en las que se construye un manejo distribuido de réplicas.

Las bases de datos se pueden ver como objetos los cuales son alterados, borrados o creados por transacciones (existen modelos que basan las réplicas en modelos no transaccionales estos exceden el interés de este trabajo). Como se vió, existen dos modos fundamentales de encarar la replicación de objetos: Eager y Lazy.

4.1. Modelo Eager

El mecanismo de implementación mas usado es el protocolo de compromiso en dos fases.

4.1.1. Protocolo de compromiso de dos fases

Se intenta realizar una transacción atómica en múltiples localidades. [Zimran 1992] [Al-Houmaily et al., 1995]

Uno de los procesos funciona como coordinador, por lo general el que ejecuta la transacción. El protocolo de compromiso comienza cuando el coordinador escribe una entrada en la bitácora para indicar que inicia dicho protocolo, seguido del envío de un mensaje a cada uno de los procesos implicados para que estén listos para el compromiso.

Cuando un proceso subordinado recibe la intención de compromiso, verifica si puede comprometerse y de poder escribe una entrada en su bitácora y envía de regreso su decisión.

Cuando el coordinador recibe todas las propuestas sabe si establece el compromiso o aborta. De cualquier modo, el coordinador escribe una entrada en la bitácora y envía a todos los subordinados la decisión final.

4.2. Modelo Lazy

Como se dijo, este mecanismo realiza los cambios en una localidad y luego centra sus esfuerzos en que las transacciones se apliquen en el mismo orden en todas las entidades.

Los mecanismos más utilizados son: Copia primaria, algoritmos basados en la hora de entrada y de anillo.

4.2.1. Copia primaria

Este algoritmo se basa en la existencia de una copia primaria de un objeto O, la cual reside en una localidad, la cual llamaremos localidad primaria para O. Cuando algún nodo quiere modificar el objeto O, envía una solicitud a la localidad primaria de O, esta solicitud se posterga hasta que la localidad pueda atenderla. [Zaslavsky et al., 1996]

Una vez que el cambio es autorizado, la localidad puede replicarlo al resto de los nodos.

Este método encaja en el modelo Lazy-Master

4.2.2. Algoritmo basado en hora de entrada

El algoritmo basado en la hora de entrada consiste en asignar en una hora única a cada transacción, y de esta manera poder determinar desde cualquier localidad el correcto orden de aplicación de los datos. [Reed, 1983] [Bernstein et al., 1980]

Este algoritmo encaja en el modelo Lazy-Group

Existen dos mecanismos clásicos de generación de horas únicas.

4.2.2.1. Algoritmo centralizado

Se basa en una localidad encargada de asignar las horas de entrada. Este servidor de horas puede utilizar un contador o su propio reloj local. El contador puede se visto como el número de versión de la base de datos. De este modo, una localidad no solo puede saber dado una transacción y la hora de la última transacción aplicada cual va primero sino también si faltan transacciones intermedias.

Para determinar el tipo de "hora única" es importante determinar qué se requiere para mantener la sincronía de la base: si alcanza con la última transacción, o se necesita la sucesión completa de transacciones.

4.2.2.2. Algoritmo distribuido

En este algoritmo las localidades son simétricas, cuando desean a una nueva transacción dar una "hora única" esta se toma de un contador o del reloj local a la cual se le concatena el nombre de la localidad. [Lamport et al., 1990]

El orden de aplicación está dado por el orden lexicográfico de las "horas únicas".

Este algoritmo puede generar ventajas entre localidades si se utiliza un contador local y el nodo genera más transacciones que otros. O si se usa reloj local y no está debidamente sincronizados.

4.2.3. Algoritmo basado en anillo

Este algoritmo se basa en generar un anillo lógico entre las localidades, sobre este anillo circula un token.

La localidad que tiene el token puede trasmitir las transacciones generadas a los otros nodos. Cuando termina, pasa el token al nodo siguiente del anillo.

De este modo, todas las transacciones tienen un orden de aplicación.

Las desventajas están dadas por la complejidad de mantener el anillo estable, una localidad que quede desconectada del círculo no recibirá, ni enviará transacciones.

Gran parte de este trabajo se abocará a realizar un algoritmo basado en un anillo que funcione aun en redes particionadas.

4.2.4. Two tier Replicación

Asume que hay dos tipos de nodos:

- Móviles: los cuales permanecen desconectados la mayor parte del tiempo de la red principal
- Base: que permanecen conectados a la red principal

Los datos tienen dos posibles estados en los nodos móviles:

- Tentativos: los cuales aun no fueron confirmados por los nodos base
- Maestros: los cuales fueron confirmados por los nodos base

Análogamente hay dos tipos de transacciones:

- Tentativa: la transacción que trabaja con datos tentativos.
- Base: La que trabaja con datos maestros

Las transacciones tentativas para ser aplicadas en los nodos maestros deben superar un conjunto de reglas específicas, llamada criterio de aceptación.

Se marcan las transacciones locales como tentativas [Gray et al., 1996], cuando son replicadas a un nodo base se la aplica y se chequea que cumpla con los criterios de aceptación, si supera la prueba se acepta la transacción tentativa, y sino se la rechaza. El modelo es similar al "clearing bancario" donde se acepta el depósito de cheques, los cuales se someten a una validación, la cuenta emisora no puede quedar en negativo, por ejemplo.

4.2.5. Algoritmos de reconciliación

Un punto crucial en el modelo lazy son los mecanismos de reconciliación. Estos algoritmos son muy dependientes de la solución propuesta y de las herramientas con que se cuente. De todos modos se pueden rescatar los siguientes:

4.2.5.1. Base de datos Delta

Se basa en la utilización de una base de datos 'limpia' y una base de datos Delta, con datos "sucios". En la segunda se aplican todas las transacciones que llegan. En la primera solo las que se tiene seguridad de que su aplicación fue aceptada por todos los nodos Cuando se necesita reconciliar se realiza una nueva base "sucia" aplicando las transacciones en el nuevo orden correcto desde la base "limpia". El programador puede optar según la problemática propia a solucionar si realizar las consultas en la base "sucia" o esperar los datos seguros de la "limpia" [Yamir97]

4.2.5.2. Restricciones sobre el uso de transacciones

Muchas soluciones se basan en acotar los tipos de transacciones replicables a un grupo específico:

- Transacciones conmutables, es decir que no importa el orden en que se apliquen no producen inconsistencias;
- Transacciones reversibles, este algoritmo almacena la información necesaria para reversar las transacciones aplicadas. Cuando una base de datos requiere reordenación, se deshacen las transacciones y se reaplican en el orden correcto.

4.2.5.3. Con intervención del usuario

En este caso el sistema se desentiende de la reconciliación. Simplemente informa al usuario el problema y este realiza manualmente los ajustes necesarios.

CAPÍTULO 5

PRESENTACIÓN DE UN PROBLEMA REAL

Como se definió en la introducción, este trabajo pretende evaluar e implementar una solución para la integración de diversos sistemas prácticamente aislados. Se procederá ha realizar una presentación formal de este problema.

5.1. Topología

La topología de la red se presenta como un conjunto de redes locales interconectadas con una red principal.

Una red local esta compuesto por N máquinas, y M bases de datos, entre las cuales hay hasta M +1 de PC destacadas, una por tener la posibilidad de salida a la red global, y M por ser las responsables de la actualización de las bases de datos locales. Cada base de datos solo puede ser controlada por una máquina.

Un sistema local mínimo esta compuesto por una única PC que contiene la base de datos, y tiene salida a la red global.

La red central, esta compuesta por servidores, los cuales se pueden interconectar con las redes locales.

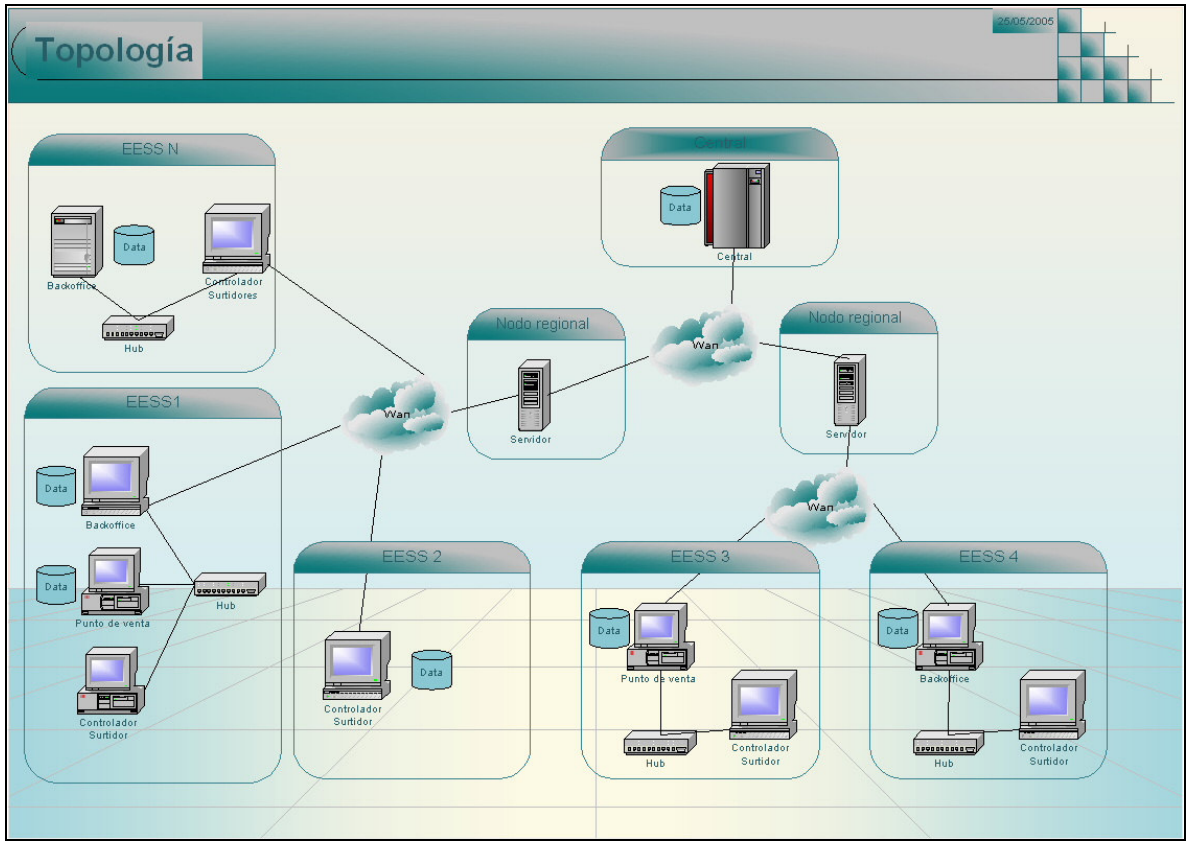
Esta red puede presentar una estructura de árbol, donde servidores zonales alivian el trabajo del servidor central.

5.2. Sistemas

Los sistemas que conviven en la red local pueden ser los siguientes:

- Controlador de surtidores
- Back office
- Punto de ventas
- Sistema de fidelidad
- Sistema de Cuentas corrientes
- Sistema de descuentos

Estos sistemas se encuentran débilmente integrados, y utilizan diferentes formas de almacenamiento bajo diferentes DBMS: SQL Server, Oracle, Access y archivos planos.



5.3. Distribución de los datos

La distribución de los datos varía para cada sistema.

Por ejemplo, en la administración de puntos de venta los datos de los productos y las ventas son decididamente locales, y es de interés global la recepción de las transacciones para decisiones estratégicas y estadísticas.

En sistemas de cuentas corrientes las transacciones están controladas por un saldo y diversos limites globales, el saldo, por ejemplo, no pueden ser menor que cero, esto genera que no se pueden hacer transacciones sin un control central.

Un gran volumen de datos está determinado por datos de configuración, los cuales son generalmente controlados desde central.

Es interés de este trabajo dar solución a todos estos modos de distribución maximizando la disponibilidad de los datos

5.4. Objetos Transaccionales

Se identifican los siguientes tipos de objetos:

5.4.1. Objetos conmutables

Están compuestos por objetos cuyas transacciones son conmutables, es decir, el orden con que se aplican no altera la consistencia de la base.

5.4.2. Objetos no conmutables en una localidad fija

Las transacciones sobre estos objetos deben aplicarse en el mismo orden en toda la base. Pero las localidades donde se encuentran tienen una alta disponibilidad de conexión con los demás nodos, de modo que pueden utilizar una localidad maestra para lograr la sincronía del objeto.

5.4.3. Objetos no conmutables en una localidad móvil

Al igual que en el punto anterior, las transacciones deben aplicarse en el mismo orden en toda la base. Pero las transacciones se realizan mayoritariamente sin conexión al sistema, lo que forzará a un mecanismo de conciliación.

5.4.4. Objetos convergentes en una localidad móvil

Las transacciones están dadas en forma semántica, se transfiere el proceso, ejemplo restar un importe n al saldo. Por lo que no es tan fuerte la necesidad de que se apliquen en el mismo orden, es suficiente con que se apliquen todas y el resultado final sea el mismo. Si tiene asociado un criterio de aprobación de la transacción, el saldo debe ser mayor a cero (por ejemplo), pueden suceder conflictos los que requiere un mecanismo de conciliación.

5.5. Propagación

5.5.1. Sentido de la propagación

El sentido de propagación de los datos se deberá realizar según las necesidades de los objetos. Definir el sentido de la propagación permite crear flujos de información. No todos los objetos de un sistema requieren que los datos se propaguen a todas las entidades. Sino que se establece para cada objeto pasible de propagación una lista de nodos a las que hay que enviar las transacciones de replicación.

El sistema permite definir un orden jerárquico entre los diferentes nodos, dicha jerarquía se organiza en forma de árbol con un padre y n hijos. Basado en dicho orden se definen las posibles reglas de propagación.

5.5.1.1. Del padre a los hijos

Los datos se propagan en sentido descendente, esto es, cuando un nodo se le aplica una transacción este la propaga a todos sus hijos.

Un ejemplo donde utilizar esta propiedad puede estar dado por el siguiente caso: Supóngase que los nodos corresponden a estaciones de trabajo dentro de una tienda, todas las estaciones son hijas de un concentrador de la tienda, los concentradores de tiendas a su vez depende de un centro concentrador regional, y todos los centros regionales de un control central. Cada nodo tiene cierta autonomía de determinar reglas de negocios para los nodos que dependen de él. Se podrían configurar las tablas que contienen las reglas de negocios para que propaguen de padres a hijos y de esta manera modificando las reglas en el nodo padre, propagar las reglas a los nodos hijos sin afectar a l resto de la organización.

5.5.1.2. De los hijos al padre

Propagación ascendente, esto indica que los datos que se aplican en los hijos son enviados al padre.

Este modelo es útil para armar un concentrador de datos. Dada la estructura de tiendas definidas en el punto anterior. Podría definirse a las tablas que llevan las operaciones diarias que propaguen en forma ascendente. De esta manera, los nodos superiores tendrían todas las transacciones de sus entidades dependientes. Las operaciones no se transmitirían a entidades no interesadas.

5.5.1.3. A todos los miembros

Propagación a todos los nodos. Esto indica que un dato es enviado a todos los miembros.

Esta forma de propagación, es la propagación típica, sin usar jerarquías. Los datos que se modifiquen en una entidad son enviados a todos los nodos sin restricciones.

5.5.1.4. No propaga

Los datos no se propagan.

Este modo representa la no propagación de datos. Los objetos marcados como no propagables son aplicados en forma local y no se envían a ninguna otra entidad.

CAPÍTULO 6

IMPLEMENTACIÓN DE UNA SOLUCIÓN

Por todo lo expuesto hasta aquí, se ha buscado una solución que pueda ser escalada con facilidad, que pueda adaptarse a los diferentes requerimientos de las diferentes aplicaciones.

El problema se sintetizará en buscar una solución eficiente y robusta a la replicación de datos, que asegure la consistencia de los datos y que a su vez soporte las diferentes configuraciones requeridas.

Los sistemas se encuentran implementados sobre un framework. Este framework permite la manipulación de la base de datos. Esta manipulación se realiza a través de un objeto denominado JBD. Un JBD sirve tanto para manipular una tabla, una vista o una consulta sobre un motor de base de datos cualquiera.

Cada objeto manipulables se representará en el framework como derivaciones de la clase JBD. Estas subclases especializarán la forma de acceder a los datos deseados.

Las acciones pasibles de realizar sobre estos objetos se corresponderán con métodos de los objetos. Estos métodos serán ejecutados por un objeto JExec que se especializa en procesar los métodos de los JBD. El JExec contiene un método Do() el cual será redefinido con la acción que se desea realizar sobre el JBD en el momento de la instanciación.

La ventaja de usar la clase JExec es poder agregar una meta-operatoria rodeando la operación a realizar. Esta meta-operatoria se encarga de controlar las transacciones de las base de datos (BeginTrans, commit y rollback), como así también el manejo de la propagación.

A cada JBD se le puede asociar una lógica de propagación. Esta lógica es implementada por el JExec. En la sección 6.3.2 Se realizará una descripción más detallada del funcionamiento de estas clases.

Los JBD se los puede configurar según las posibles necesidades de las aplicaciones, los cuales se combinarán para lograr las diferentes necesidades expuestas en él capitulo anterior:

- Conmutable global: El orden de aplicación de las acciones no altera el resultado final.
- No conmutable consigo mismas: El orden de aplicación de las acciones debe conservarse solo con respecto al mismo objeto.
- No Conmutable: Las transacciones deben aplicarse en el mismo orden en toda la red.

También se puede configurar el sentido de la propagación, esto es:

- Del padre a los hijos: Los datos se replican de los padres a los hijos
- De los hijos al padre: Los datos se replican de los hijos a los padres
- A todos los nodos: El dato debe ser replicado a todos los nodos
- No se propaga: Los datos no se propagan

6.1. Conmutabilidad

6.1.1. Replicaciones conmutables globales

Las replicaciones conmutables son aquellas donde el orden de aplicación de las acciones no altera el resultado final. Esto es:

Sea $a_i(o)$ acciones sobre el objeto o en la base de datos, y e un estado de la base de datos. Y App(e,a) una función que devuelve el estado de una base luego de aplicar la acción a al estado e, X,Y objetos conmutables:

$$\forall X,Y\ App(App(e,ai(X)),ai+1(Y)) = App(App(e,ai+1(Y)),ai(X))$$

Ejemplos de este tipo de transacciones es el caso de un concentrador de transacciones. La única acción permitida sobre el objeto transacción es 'alta'. Al no haber dependencias entre las diferentes transacciones éstas se aplican como llegan.

6.1.2. Replicaciones no conmutable consigo mismas

Las replicaciones no conmutables consigo mismas son aquellas donde debe preservarse el orden solo con respecto al mismo objeto. Sea X, Y objetos no conmutables consigo mismo:

$$\forall X,Y\ App(App(e,ai(X)),ai+1(Y)) = App(App(e,ai+1(Y)),ai(X)),\ X \neq Y$$
$$\forall X,Y\ App(App(e,ai(X)),ai+1(Y)) \neq App(App(e,ai+1(Y)),ai(X)),\ X = Y$$

Ejemplos de este tipo de transacciones es el caso de un ABM de un objeto que no tenga relaciones de integridad con otras estructuras. De esta forma, sólo es importante que se halla aplicado el alta, antes de la baja o la modificación, o que no se intente modificar un dato borrado, y así.

6.1.3. Replicaciones no conmutables

Estas replicaciones requieren aplicarse en el mismo orden en toda la red. Sea X, Y objetos no conmutables:

$$\forall X,Y\ App(App(e,ai(X)),ai+1(Y)) \neq App(App(e,ai+1(Y)),ai(X))$$

Ejemplos de este tipo de replicas son aquellas que tienen una alta dependencia entre diferentes objetos. Ordenes de compra que aplican sobre un stock, donde debe existir el producto y el stock debe ser siempre positivo.

6.2. Sentidos de Propagación

Como se vio en el capítulo anterior es necesario que la solución permita definir las reglas del sentido de la replicación para cada objeto.

Para dar solución a esto el framework permite definir a cada nodo un padre y asociarles un conjunto de reglas de replicación.

La definición de un nodo padre permite crear la jerarquía previamente mencionada.

Las reglas de replicación permiten definir a cada objeto pasible de replicación, las características de propagación que lo gobiernan.

Las posibles reglas de replicación que se pueden asociar a cada objeto son las antes vistas:

6.2.1. Sentido del padre a los hijos.

Esta replicación se refiere a la propagación que solo puede realizarse de los padres a los hijos. Esto es, una acción realizada en el padre es enviada a todos los hijos. Ejemplos de este tipo de operaciones suelen ser las configuraciones del sistema que se refieren a las reglas de negocio establecidas por el servidor.

6.2.2. Sentido del hijo a los padres

Esta replicación se refiere a la propagación que solo puede realizarse de los hijos a los padres. Esto es, una acción realizada en los hijos es enviada al padre. Ejemplos de este tipo de operaciones suelen ser los concentradores de información.

6.2.3. A todos los nodos

Esta replicación se refiere a la propagación que debe ser conocida por todos los nodos. Ejemplos de estas acciones son datos que son de interés en todos los nodos.

6.2.4. No se propaga

Esta aplicación se refiere a los objetos que no se propagan Un dato no propagable podría estar dado por configuraciones privadas del nodo.

6.3. Modelos simples

Las replicaciones conmutables representan un modelo mucho más sencillo.

Luego de realizar la transacción, se crea un mensaje, el cual se construye serializando un objeto, el método y los parámetros utilizados. Este mensaje se graba en la tabla de mensajes a enviar, y se realiza el commit.

Paralelamente un servicio realiza el 'delivery' de los mensajes.

Un nodo que recibe un mensaje de propagación, lo rearma, y ejecuta el método. Si este falla, se informa a la localidad emisora del resultado fallido.

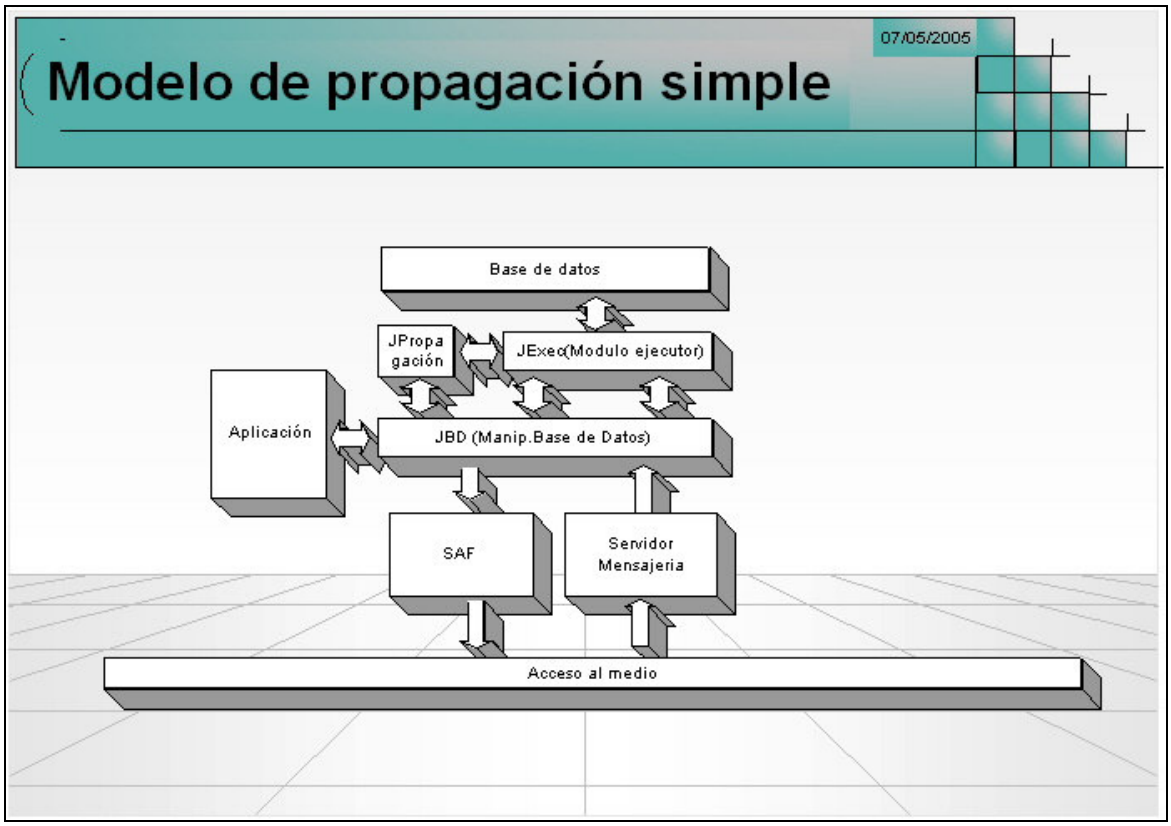
6.3.1. Arquitectura

La arquitectura se basa en localidades simétricas, es decir, todos realizan las mismas operaciones.

En cada nodo se encuentra un servicio de 'delivery', denominado SAF (Store and Forward). Se graban en él, tantas copias del mensaje como nodos a los que debe transferirse. Cuando el nodo pierde comunicación los mensajes aguardan hasta la recuperación del vínculo.

También existe un servicio de mensajería que escucha los mensajes que llegan a la localidad. Y se encarga de interpretar los objetos rearmarlos y ejecutar el método solicitado.

JBD es la capa que administra el acceso a la base de datos. Esta utiliza el módulo JExec para realizar sus transacciones, JPropagación gobierna a JExec e implementa las políticas replicación.



.

6.3.2. Funcionamiento del algoritmo

La idea del algoritmo se basa en repetir una operación realizada en un nodo en todos los nodos. El módulo JBD es invocado para realizar una transacción en la base.

Dicha transacción al momento de ser ejecutada consulta por la configuración asociada al objeto modificado. Y genera la mensajería para los nodos que debe replicar dicha operación.

Los mensajes quedan almacenados en la base de datos, donde un proceso denominado "*store and forward*" los envía en "*background*".

Si un mensaje falla al momento de aplicar se retorna el error y se llena una bitácora de errores, donde un usuario tiene la posibilidad de forzar el reenvío del mensaje.

Las estructura del mensaje de propagación esta en XML y es la siguiente:

```
MENSAJE::= <root><clase>NAME</clase><metodo>NAME</metodo><data>BASE</data></root>
BASE::=<table name=NAME no_exc_select=SN> FILTROS ORDERBY GROUPBY FILTROSFIJOS TABLE
</table>
FILTROS::=<filters> FILTRO <filters/> | <filters> FILTRO FILTROS </filters>
FILTRO::=<filter table=NAME field=NAME operator=NAME value=NAME type=NAME
relation=NAME group=NAME/>
ORDERBY::=<order_by> NAME </order_by>
GROUPBY::=<group_by> NAME </group _by>
FILTROSFIJOS::=< fixed_filters > NAME </fixed_filters>
TABLE::=ROW TABLE| ROW
ROW::=<row CAMPO=VALOR/>
CAMPO::=NAME
VALOR::=NAME
NAME::=a|b|c|…z|A|…|Z|0|..|9|NAME
SN::=S|N
```

Para la serialización del mensaje se está utilizando este formato propietario, fácilmente se puede adaptar a algún estándar, como SOAP, CORBA o RMI.

6.3.3. Implementación

En este apartado se analizarán los detalles más relevantes de la implementación:

6.3.3.1. JBD

Esta clase representa un objeto transaccional en la base de datos. Cualquier acceso a los datos de la base se realiza a través de ella y sus herederas.

Cada objeto transaccional debe definir una clase que herede de JBD. Estas deben implementar tres métodos: addVarProperties, addFixedProperties y GetTable.

En el primero relacionan cada campo de la base con un miembro de la clase. En el segundo se extiende la información de cada campo para que el objeto pueda ser manipulado correctamente (se señalan campos claves, se fijan reglas de integridad, y otras).

Finalmente el tercero, relaciona el objeto con una tabla o una consulta a la base. Si es una consulta, este método devuelve vacío y se implementan otros métodos para fijar los parámetros de la consulta a realizar.

Ejemplo de estos métodos son:

```
public void addVarProperties() throws Exception {
  this.AddItem( "id_propagate", pIdpropagate );
  this.AddItem( "description", pDescription );
  this.AddItem( "propagate", pPropagate );
  this.AddItem( "fromChildren", pFromChildren );
  this.AddItem( "toChildren", pToChildren );
  this.AddItem( "toMaster", pToMaster );
}
/**
 * Adds the fixed object properties
 */
public void AddFixedProperties() throws Exception {
  this.AddItemFijo( "Clave", "id_propagate", "Id propagate", true, true, 50 );
  this.AddItemFijo( "Campo", "description", "Description", true, true, 100 );
  this.AddItemFijo( "Campo", "propagate", "Propagate", true, true, 1 );
  this.AddItemFijo( "Campo", "fromChildren", "Acepta mensaje de los hijos", true, true,
  1 );
  this.AddItemFijo( "Campo", "toChildren", "Envia mensaje de los hijos", true, true, 1
  );
  this.AddItemFijo( "Campo", "toMaster", "Envia mensajes al master", true, true, 1 );
}
/**
 * Returns the table name
 */
public String GetTable() { return "PRO_PROPAGATE"; }
```

JBD tiene definidas las operaciones básicas que pueden realizarse con la objeto, aunque una clase heredada puede establecer las suyas propias, estos métodos típicos son ExecProcesarAlta, ExecProcesarBaja y, ExecProcesarModif.

```
 public final void ExecProcesarAlta() throws Exception {
  JExec oExec = new JExec(this, "ProcesarAlta") {public void Do() throws Exception
  {ProcesarAlta();}};
  oExec.Procesar();
}
public void ProcesarAlta() throws Exception {
  InsertarRegistro();
}
```

Como se observa el método ProcesarAlta es redefinible, no así el ExecProcesarAlta el cual utiliza JExec para llamar a un método propio.

6.3.3.2. JExec

Esta clase es invocada cada vez que JBD quiere ejecutar un método propio en una sesión crítica. Todas las operaciones sobre la base de datos las realizara pasaran por esta clase, aquí se engancha el modulo de propagación y decide si es necesario propagar los operación que se esta realizando.

Al momento de la instaciación se redefine el método Do() para que ejecute la acción que se desea realizar sobre el JBD. La filosofía usada es una operación transversal a la operatoria, resultando en una implementación basada en aspectos.

```
public void Procesar() throws Exception {
  try {

    if ( JBDatos.ifBasesAbiertas() ) JBDatos.GetBases().BeginTransaction();
    Do(); // ejecuta el método de la clase llamadora

    if ( oBD != null ) {
      try { JbLogTrace.Registrar( sMetodo, oBD ); } catch ( Exception E ) {}
      try { oBD.PropagarEjecucion(sMetodo,null);} catch ( Exception E ) {}
    }

    if ( JBDatos.ifBasesAbiertas() ) JBDatos.GetBases().Commit();

  } catch (Exception e) {

    if( e instanceof java.lang.reflect.InvocationTargetException ) {
      if( e.getCause() instanceof Exception ) { e = (Exception)e.getCause(); }
    }

    if ( !(e instanceof InterruptedException) && !(e instanceof JExcepcion) && !(e
instanceof CanceledByUserException) ) {
      JDebugPrint.logDebug(e, "Error en JExec#Procesar()");
    }

    if ( ! (e instanceof JConnectionBroken) ) {
      if ( JBDatos.ifBasesAbiertas() )
        JBDatos.GetBases().Rollback();
    }


    throw e;
  }
  }
```

6.3.3.3. JbPropagate y JbPropagateDetail

La información de que propagar o no también se almacena en la base de datos, JbPropagate y JbPropagateDetail son las clase que hereda de JBD y manipulan la configuración de propagación activa.

Obsérvese que al heredar de JBD son susceptibles de ser replicadas.

Las reglas admiten varias configuraciones, cada configuración contiene un detalle con el nombre del objeto transaccional y los atributos sobre el sentido y mecanismo de propagación utilizado.

El sistema siempre corre con un nodo activo, visión lógica de la localidad, cada nodo tiene una configuración activa. Los datos de esta tabla se mantienen en un hash de memoria para asegurar eficiencia a la hora de utilizarla.

6.3.3.4. JbNodo

Esta clase almacena la información del nodo, también es configurable en esta tabla la estructura jerárquica del nodo. Esta tabla no solo contiene el nodo actual sino también los nodos remotos.

6.3.3.5. JbBaseMensajes

Esta clase administra los mensajes a enviar, entre los posibles mensajes existen unos marcados para almacenar y enviar en momentos de óseo. Esta clase se encarga de levantar el thread y operar con estos mensajes.

```
public static void  sendSAFMessages() throws Exception {
  int counter = 0;
  while ( true ) {
    if( Thread.currentThread().isInterrupted() )  throw new InterruptedException();
    JBDs oMensajes = new JBDs(JbMensaje.class);
    oMensajes.SetFiltros("saf","S");
    oMensajes.SetFiltros("ruteo","S");
    oMensajes.SetOrderBy("transaccion");
    oMensajes.ReadAll();
    counter = 0;
    while (oMensajes.NextRecord()) {
        JbBaseMensaje oMensaje = (JbBaseMensaje) oMensajes.GetRecord();
        try {oMensaje.execReprocesar(); }
        catch (Exception e) {
           JDebugPrint.logError("Error Processing Message: "+e.getMessage());
        }
        counter++;
        if (counter>100) {
          Thread.sleep(1000);
          counter = 0;
        }
    }
    Thread.sleep(1000);
  }
}
```

JbBaseMensaje también hereda de JBD pues almacena datos propios de configuración, por esto es a su vez a si misma replicable.

6.4. Modelos complejos

6.4.1. Introducción

Los casos donde las replicaciones no son conmutables utilizaremos una replicación activa que mantenga su funcionamiento aún en caso de una red particionada. La idea principal se basa en el trabajo "Robust and efficient replication using group communication" [Yamir97] .

La replicación se basará en la garantía que se aplicarán las mismas acciones en el mismo orden sobre todas las bases. Por esto, es requerimiento que el estado de la base esté determinado por el estado actual y la siguiente acción.

Introducimos la siguiente notación:

Sea S un grupo de servidores,
$a_{s,i}$ es la acción en i realizada por el servidor s
$D_{s,i}$ es el estado de la base luego de i acciones en el servidor s
Estabilidad(s,r) es un predicado que denota la existencia de un conjunto de servidores conteniendo a s y r al mismo tiempo.

Los requerimientos para la replicación están dados por:

$\forall s,r \in S,\ D_{s,o} = D_{r,o}$ (El estado inicial de la base el mismo para todos los servidores)

$\forall s \in S,\ D_{s,i} = App(D_{s,j-1},\ a_{s,j})$ (El siguiente estado de la base esta determinado completamente por el estado de la base y la acción aplicada);

La corrección de la solución está definida como sigue:

Seguridad: Si dos servidores ejecutan la misma acción en ambos el resultado es idéntico

$\exists a_{s,i}, a_{r,i} \rightarrow a_{s,i} = a_{r,I}$

Persistencia: Si el servidor s ejecuta una acción y existe un conjunto de servidores conteniendo también a r, al mismo tiempo, el servidor r ejecutara la acción:

$(\exists a_{s,i} \wedge Estabilidad(s,r)) \rightarrow \exists a_{r,i}$

Acciones Reversibles: Subconjunto de acciones tal que para una acción a existe ~a que vuelve la base al estado anterior

$\forall s \in S,\ D_{s,i} = App(App(D_{s,i},\ a_{s,j+1}), \sim a_{s,i+1}) = D_{s,I}$

Los fundamentos de la solución se basan en las leyes de la comunicación por grupos. Inicialmente todas los nodos pertenecerán a un único grupo, por los nodos del grupo circula un "token". El nodo propietario del token es el único que puede liberar mensajes a sus pares, de esta manera se asegura el orden total de las operaciones.

En caso de detectarse una partición de la red, el sistema genera múltiples anillos que abarquen los nodos de cada partición. Existe un nodo destacado, el anillo que contenga este nodo será considerado el anillo 'Verde'. Los restantes de existir serán anillos 'rojos'

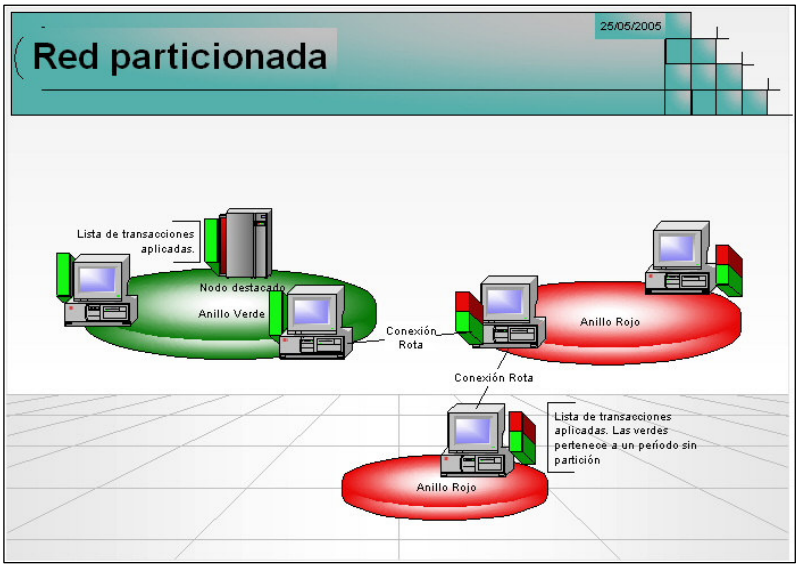
Las acciones que se generen mientras no se tenga el 'token' aguardan para ser aplicadas. Al llegar este, conocen su orden global, se almacenan y se propagan a todos los nodos con el color del anillo.

6.4.2. Arquitectura

Al igual que en el modelo simple las entidades son simétricas, pero se asegura que en todas las bases las acciones (transacciones) se aplicarán en el mismo orden.

Cada localidad contendrá un servicio encargado de mantener las bases de datos replicadas. Dicho servicio es dueño de una copia local de la base de datos, la cual mantiene. La acciones pueden ser verdes, lo cual significa que conocen su orden global, o rojas sin aun es desconocido.

Los consultas que se realizan sobre las bases de datos pueden ser limpias, si solo utilizan acciones verdes, y sucias si se generan a partir de rojas.



Para que la base sea consistente, se debe exigir una respuesta limpia siempre, el algoritmo nunca aplicará a la base acciones rojas, estas aguardarán hasta conocer su orden global.

Una solución más simple puede ser proveer una consistencia suave "*weak*", donde solo las consultas que cambian datos deben esperar.

Algunas aplicaciones pueden requerir hacer consultas 'sucias' aún a riesgo de que al momento de la sincronización alguna transacción sea rechazada. En estos casos se puede usar un base de datos Delta que contenga las acciones 'rojas'.

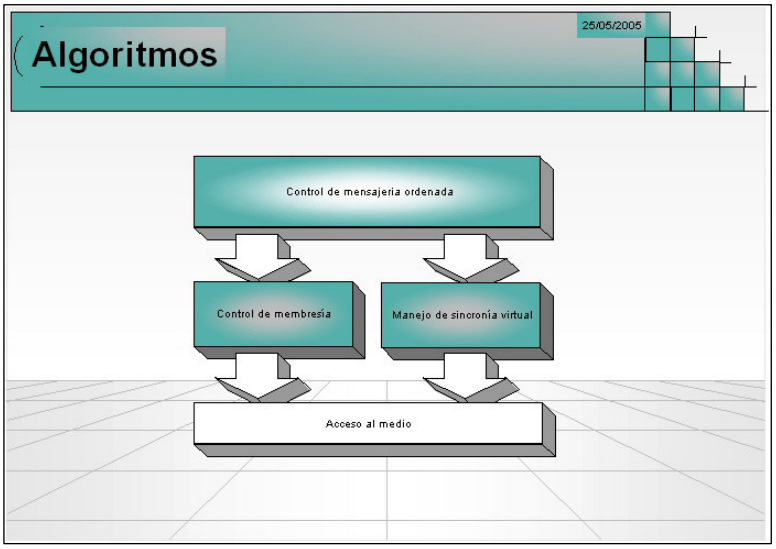
En una operación normal cuando la aplicación requiere una operación del servidor de replicaciones, este genera un mensaje conteniendo la acción. El mensaje es pasado al manejador del anillo, si el color del anillo es verde, el mensaje es enviado a todos los participantes del grupo, con la marca de verde, indicando que se conoce perfectamente su orden global. Si en cambio el anillo es rojo, solo es enviado si el mensaje es también 'rojo', un mensaje 'verde' debe esperar.

La regla simple sería: Una anillo verde, transfiere acciones verdes, y si encuentra rojas las hace verdes. En cambio, un anillo rojo solo replica acciones rojas.

6.4.3. Funcionamiento de los algoritmos

La solución se compone de tres algoritmos, los cuales aseguran la sincronía de la base. La mensajería interviniente en estos tres algoritmos se mueve en derredor de los anillos.

Los algoritmos son:



Mensajería ordenada: Es responsable de mantener el orden total de los mensajes, también maneja las omisiones de mensajes, incorporando lógica de retransmisión. [Melliar91]

Membresía: Maneja las caídas y recuperaciones de la red, particionando y remezclando los anillos. [Yamir97]

Sincronía virtual: Este algoritmo es invocado después que un cambio de membresía es detectado y la nueva configuración fue establecida. Su objetivo es manejar la mensajería que haya quedado pendiente de entrega al momento de la caída. [Yamir97]

6.4.3.1. Mensajería ordenada

En esta sección se describe el algoritmo que mantiene el orden de los mensajes.

Solamente los procesadores en posesión del token podrán enviar mensajes al resto de las localidades. En esta sección asumiremos que no hay pérdidas del token, ni caídas de las localidades.

El orden de los mensajes se establece simplemente numerando los mensajes. A cada mensaje se le adjunta su número de orden antes de ser enviado, de tal modo, al recibir una localidad dicho mensaje sepa el orden total que le corresponde.

Un mensaje de transacción tiene los siguiente campos:

6.4.3.2. Mensaje de transacción

Tip : mensaje tipo BD_TRANS

Id_conf: Identificador que indica la configuración con la cuál el mensaje fue enviado.

Id_loc: Localidad que envía el mensaje

Sec: Número de mensaje, este número indica el orden total.

Color: color del mensaje

Data: Contenido del mensaje

6.4.3.3 Mensaje del token

El mensaje del token contiene lo siguiente:

Tipo: mensaje tipo BD_TOKEN

Id_conf: Identificador que indica la configuración con la cuál el token fue enviado.

Sec: El mayor número de secuencia que ha sido enviado con esta configuración. Al comienzo de cada configuración, Sec se pone en cero.

Color: color del anillo

Tec: Es el número Tec (todo lo enviado correctamente) que indica el número de mensaje que todos los procesadores del anillo han recibido, y por lo tanto no requieren ser retransmitidos. Cuando ya no se requiere retrasmisión se pueden descartar los mensajes menores del control. Al comienzo de cada configuración el valor se pone en cero.

Rtr: Una lista de requerimientos de retrasmisión, cada elemento de la lista contiene (Id_conf, Sec) del mensaje requerido

Ccf: El número de mensajes (contador del control de flujo) actualmente enviados por todos las localidades del anillo en la última rotación del anillo, incluido retrasmisiones.

Los mensajes se manejan con la siguiente lógica:

Cada localidad mantiene una variable local mi_tec conteniendo el número de secuencia mayor de todos los mensajes recibidos. Al comienzo de cada configuración setéa esta variable en cero. Cada vez que recibe un mensaje actualiza m_tec. Cada localidad, además mantiene la lista de mensajes recibidos, los mensajes seguros pueden ser eliminados de la lista.

Al recibir el token, la localidad comienza a enviar mensajes, actualiza el token y lo trasmite al siguiente nodo. Por cada nuevo mensaje incrementa el campo Sec del token y setéa el nuevo mensaje con el número de orden Sec.

Haya o no, enviado mensajes, el procesador compara el campo tec del token con mi_tec y si, mi_tec es menor, setea tec con mi_tec. Si la localidad previamente redujo el valor de tec y el token retorna con el mismo valor, entonces setea tec igual a mi_tec . Si sec y tec son iguales, entonces incrementa tec y mi_tec en concordancia con Sec.

Si el campo Sec del token indica que un mensaje ha sido enviado a la localidad y no fue recibido, la localidad carga él id del mensaje en el campo Rtr. Si la localidad encuentra un mensaje que ha recibido en la lista rtr, retransmite el mensaje, y lo remueve de Rtr.

El uso ccf es para controlar la cantidad de mensajes circulantes y se describirá cuando se trate el manejo de performance

6.4.3.4. Membresía

El algoritmo de membresía se usa en conjunto con el de mensajería ordenada y la sincronía virtual. El algoritmo maneja todo los aspectos referidos a mantener los anillos estables. Incluye fallas en las localidades y recuperaciones, perdida del token, partición de redes y remezclado de la misma. El algoritmo utiliza un solo representante para cada anillo, este representante se encarga de administrar las membresías al anillo. No se debe confundir con un líder o maestro de un nuevo o viejo anillo. Mientras un nuevo anillo es formado se intenta usar al máximo posible los conocimientos de las viejas configuraciones.

6.4.3.5. Estructura del mensaje

El algoritmo de membresía usa dos tipos especiales de mensajes, los cuales son desconocidos por la capa de aplicaciones y no tienen número de orden.

Intento de unión: mensaje enviado por un representante para formar u nuevo anillo de dos o más anillos.

Unión: un mensaje enviado por un representante con una conjunto de localidades para formar un nuevo anillo.

Para formar un nuevo anillo se hace circula un token especial, llamado formador. El token formador tiene la siguiente estructura:

6.4.3.6. Mensaje del token formador

Tipo : Token Formador

Id_form: un identificador único del token, constituido por el nombre del enviador y una marca de tiempo.

List_Union: Una lista de identificadores de representantes

List_Miem: Una lista con todos los identificadores de todos los miembros del nuevo anillo y su posición en el nuevo anillo. Para cada miembro la lista contiene su viejo identificador (id_conf).

List_Conf: Una lista conteniendo las antiguas configuraciones de las que fue miembro cada miembro del nuevo anillo. El campo se usará en el algoritmo de sincronía virtual para recuperar viejos mensajes.

6.4.3.7. Definición de eventos

Hay cinco eventos correspondientes al algoritmo de membresía

Recibir un mensaje foráneo: puede ser un mensaje de tipo:

- Mensaje Regular de orden enviado por una localidad extraña al anillo.
- Intento de unión:
- Unión

Recibir un token formativo: La primera vez que se recibe el token formativo indica una nueva propuesta de unión, recibirlo por segunda vez indica actualizaciones a la configuración del anillo.
Expiración por pérdida del token: Este evento indica que una localidad no recibió el token o mensaje de algún procesador en un período de tiempo.

Expiración de armado: Este evento indica que expiró el período de tiempo para formar un nuevo anillo

Expiración confirmación: Este evento indica que una localidad participante en la formación de un nuevo anillo le expiró el tiempo de espera para confirmar el nuevo anillo.

6.4.3.8. Definición de estados

Hay cinco estado, denominados:

Estado operacional: Este es el estado donde el anillo opera sin cambios de membresía

Estado de armado: Los representantes del nuevo anillo son coleccionados, se envían muchos mensajes de Intento _ unión y unión. Como así también posibles Expiraciones de armado.

Estado confirmación: Los representantes envían propuestas para formar el nuevo anillo

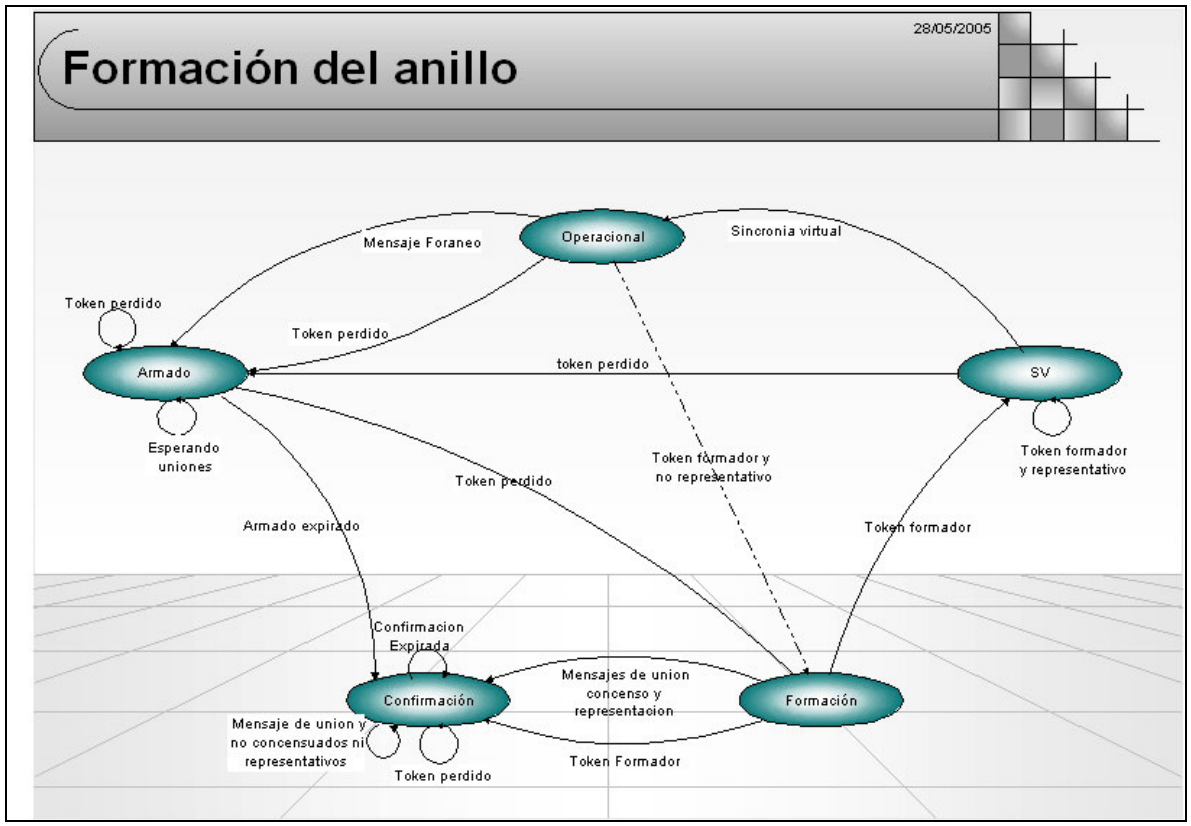
Estado formación: El nuevo camino ya esta determinado y se envía un token formativo para juntar la información de cada miembro.

Estado SV: Se llama a la rutina de sincronía virtual para garantizar el correcto manejo de los mensajes de anteriores configuraciones.

6.4.3.9. Formando un nuevo anillo

Se examina el algoritmo sin considerar fallas de procesador ni perdidas de token, esto se lo deja para más adelante.

El algoritmo es invocado cuando una perdida de token es detectada o cuando un mensaje foráneo es recibido por algún miembro del anillo. Una localidad no-representante ignora los mensajes foráneos.



En el estado operacional, cuando recibe un mensaje foráneo, envía un mensaje de intento de unión para advertir a todos los miembros la intención de ampliar el anillo. Luego queda en estado de Armado.

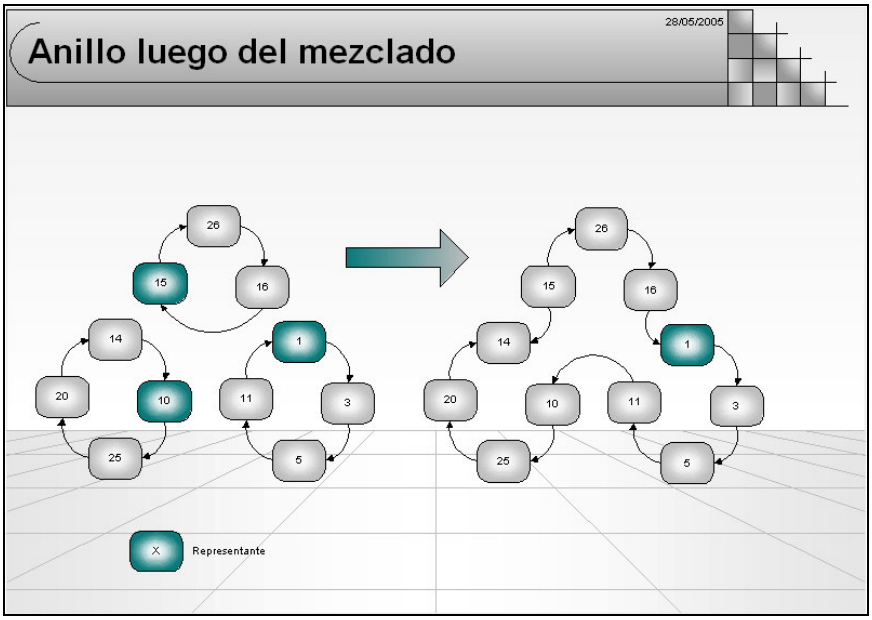
En estado de armado espera un tiempo para que se junten la mayor cantidad de representantes posibles para el nuevo anillo. Cuando el tiempo expira, envía mensajes de Unión a todos los nuevos representantes y pasa al estado de confirmación.

En este estado espera que cada representante confirme su intención de unirse al anillo. Cada representante envía su aceptación con dos conjuntos de representantes uno de exitosos y otro de fallidos. La confirmación es aceptada cuando existe un conjunto de representantes exitosos y fallidos, listados en el mensaje de Unión, tal que cada representante exitoso contiene exactamente estos mismos conjuntos. De no coincidir se debe enviar otro mensaje a todos los representantes conteniendo la unión de los mensajes de Unión.

Si el tiempo de expire de la confirmación llega antes de la confirmación. El representante inserta todos los representantes de los cuales no recibió mensaje de Unión dentro del conjunto de representantes fallidos, reenvía mensajes de Unión, y reinicia la espera tratando de formar un nuevo anillo.

El representante que propone el nuevo anillo, cuando la confirmación se logra, genera un token de Formación. El token de formación circula por todos los miembros del nuevo anillo. Al recibir el token los miembros pasan al estado de formación. Después de una rotación de

este token, los representantes conocen toda la información del estado de los miembros, dicha información se utiliza por el algoritmo de sincronía. Se pasa al estado SV. Se envía una segunda vuelta del token formador, y luego se ejecuta el algoritmo de sincronía.



6.4.3.10. Perdida del token fallas del procesador y partición de red

El algoritmo no diferencia entre fallas del procesador, partición de la red o pérdidas del token. Todos estos errores se detectan por Expiración por perdida de token, es decir, se detecta que un token no llega a destino.

Por lo general este evento ocurre en estado operacional. Una localidad que detecta la perdida del token se declara a sí mismo representante y procede a pasar al estado de armado.

El token se puede perder también cuando sé esta en estado de Armado o Confirmación, en este caso el algoritmo de formación continua, pero seguramente quedará representándose a sí mismo.

La perdida del token de Formación puede ocurrir en el estado de formación. En este caso, el antiguo anillo ya no esta operando y el nuevo anillo aún no esta constituido. El estado vuelve al modo Armado, pero para asegurarse no quedar en un ciclo infinito elimina el miembro de mas alto identificador.

6.4.3.11. Sincronía virtual

La idea básica del algoritmo de sincronía es usar el anillo recientemente creado y mensajes de transacciones para recobrar los mensajes perdidos. Este algoritmo se ejecuta con todos los nodos en estado SV. En este modo no se envían nuevos mensajes, y ahí solamente un nuevo token circulando en el anillo.

En la primera vuelta cada nodo determina sus agujeros de información, y pide las retrasmisiones necesarias.

Al ir recibiendo la mensajería hace rollback de las transacciones fuera de orden y las encola para enviarlas al final de la lista.

6.5. Implementación

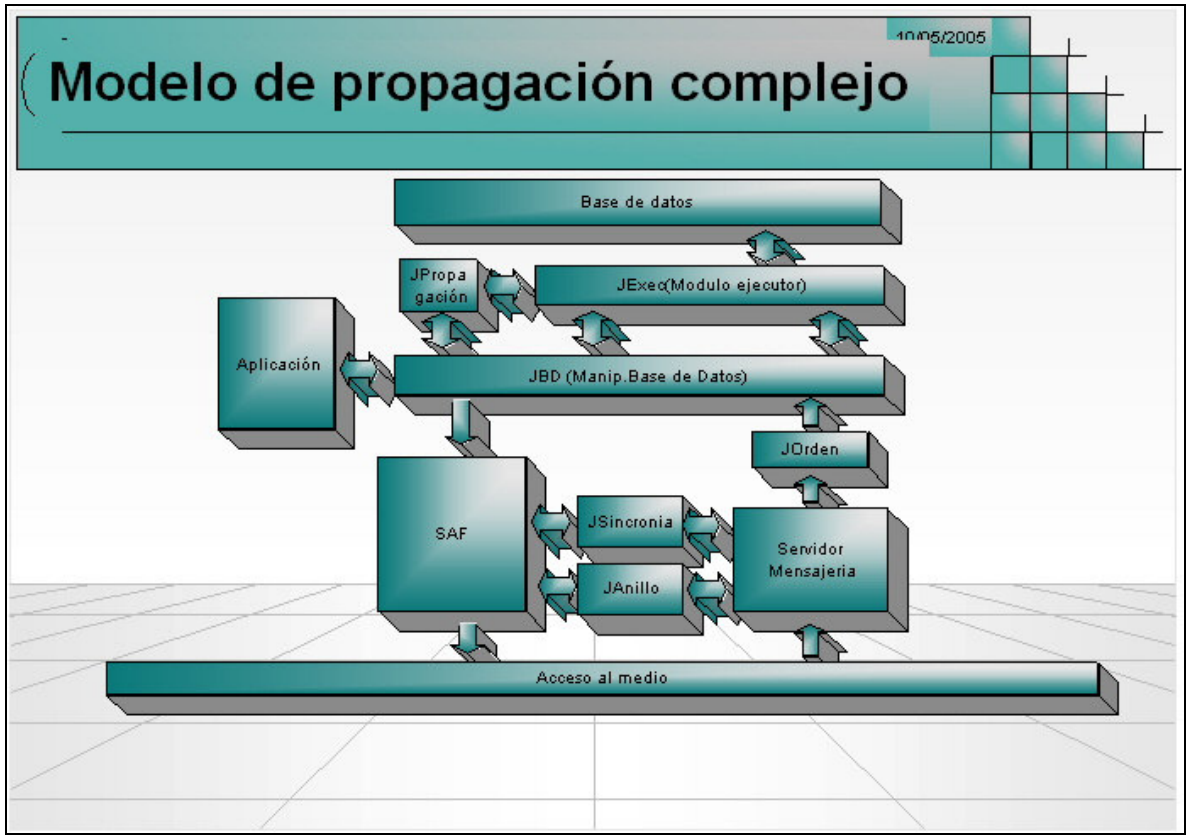
La implementación del modelo complejo se realizará sobre las bases del modelo simple.

Se definirá la clase JbToken la cual hereda de JBD, por lo tanto es propagable. Se redefinirá el método de propagación para que siempre propague al siguiente elemento del anillo.

JbTokens tendrá cuatro especializaciones, JbTokenTrans, JbTokenFormador, JbTokenTest y JbTokenEVS. Cada uno de los cuales implementará los algoritmos de control del anillo.

El anillo será manipulado por JbRing y contendrá una lista de JbRingsMembers. Los datos del anillo se almacenaran en la base de datos y su clave será el identificador de configuración. Este JBD se propaga a todos los miembros.

Se levantará un proceso, que estará atento al tiempo de llegada del token. Este proceso esperará recibir cada cierto tiempo un mensaje de una entidad remota con las transacciones a propagar, si nos las recibiera en un cierto tiempo pasará a intentar formar un nuevo anillo. Como esta clase tiene conocimiento de la estructura del anillo, tendrá un tiempo de espera inferior si es el siguiente nodo a recibir el token.



6.5.1. JbToken

JbToken maneja la lógica de los token. Su fin es ejecutar el método procesarToken() de todos los nodos en el orden establecido por la tabla JbRingsMembers.

Como se observa en el código su implementación no difiere de una clase de propagación simple. Se envía el objeto token serializado, se rearma, ejecuta el método indicado, se vuelve a serializar y se reenvía al siguiente.

La operatoria dependerá de las reimplementaciones del token:

- JbTokenTrans: implementa la operatoria normal en un anillo constituido
- JbTokenFormador: implementa el token formador
- JbTokenTest: testea el anillo antes de considerarlo formado
- JbTokenEVS: implementa el token durante la sincronía virtual.

```
  public void PropagarEjecucion(String zMetodo,JSetupPropagate zSetup) throws Exception {
    // fuerza propagacion al siguiente
    JSetupPropagate setup = new JSetupPropagate();
    setup.SetPropagate = true;
    setup.setPropagateNext();
    JbPropagarMje oPropagarMje = new JbPropagarMje();
    oPropagarMje.SetOrigen(JbNodo.GetNodoLocal().GetNodo());
    oPropagarMje.SetObjBD(this);
    oPropagarMje.SetMetodo(zMetodo);
    oPropagarMje.SetPropagateRule(setup);
    oPropagarMje.ProcesarAlta();
  }

   public void ExecToken() throws Exception {
    JExec oExec = new JExec(this, "ProcesarToken") {public void Do() throws Exception
{ProcesarToken();}};
    oExec.Procesar();
  }

  public void ProcesarToken() throws Exception {
    doToken();
  }

  public void doToken() throws Exception {
  }
```

6.5.1.1. JbTokenTrans

JbTokenTrans maneja la lógica del anillo durante la operatoria "normal", un anillo constituido y operando.

Los miembros de esta clase y los métodos implementados son los descriptos en el apartado 5.4.3.1.1.2..

```
    if (getNodoGenerador().equals(JbNodo.GetNodoLocal().GetNodo()))
      return;
    JbRing  ring= JbRing.getRing();
    if (!getRtr().equals(""))
      sendOldMessage(getRtr());
    if (pSec.GetValor()>ring.getMyTec()) {
      String rtr = getRtr();
         for (long i=ring.getMyTec();i<pSec.GetValor();i++)
           rtr+=""+ring.getIdConf()+","+i+"|";
         setRtr(rtr);
    }
    JbBaseMensaje.sendRingMessages(ring.getIdConf(),pColor.GetValor());
    if (ring.getMyTec()==ring.getMySec())
      pTec.SetValor(ring.getMySec());
    if (ring.getMyOldTec()!=0 && ring.getMyOldTec()==pTec.GetValor())
      pTec.SetValor(ring.getMyTec());
    if (ring.getMyTec()<pTec.GetValor()){
      pTec.SetValor(ring.getMyTec());
      ring.setMyOldTec(pTec.GetValor());
    }
    else
        ring.setMyOldTec(0);
  JbTokenTrans newToken = new JbTokenTrans();
  newToken.pIdConf = pIdConf;
  newToken.pColor = pColor;
  newToken.pTec.SetValor(ring.getMyTec());
  newToken.pSec.SetValor(ring.getMySec());
  newToken.ExecProcesarToken();
```

6.5.1.2. JbTokenFormador

JbTokenFormador maneja la lógica del anillo durante la operatoria de rearmado del anillo.

Los miembros de esta clase y los métodos implementados son los descriptos en el apartado 6.4.3.3..

6.5.1.3. JbTokenEVS

JbTokenEVS maneja la lógica del anillo durante la operatoria posterior a la constitución del anillo.

Los miembros de esta clase y los métodos implementados son los descriptos en el apartado 6.4.3.5..

6.5.2. JbRing y JbRingsMembers

Esta clase maneja la lógica del anillo. La estructura actual del anillo se almacena en la base de datos. El representante que deba formarlo solo lo almacena y la lógica de propagación lo replica al resto del anillo.

JbRing se encarga además de llevar el control del flujo, si se demorara mucho tiempo en recibir mensajes propagados de otros nodos o del token, este clase se encarga de la formación del nuevo anillo. El control se realizará en un método estático, levantado como servicio.

También se encarga de enviar mensajes a los nodos conocidos no pertenecientes al anillo solicitando que se unan al anillo.

```
  public static void controlRing() {
public static void controlRing() {
  try {
    boolean waitEVS = false;
    long timeoutEVS =0;
    long pTimeout = 0;
    boolean first = true;
    JAplicacion.AbrirSesion();
    JAplicacion.GetApp().AbrirApp("Control_Anillo", JAplicacion.AppTipoThread(), true );

    JbRing myRing = getRing();
    while (!Thread.currentThread().isInterrupted()) {
      switch ((int)myRing.getStatus()) {
      case ST_DESCONOCIDO: // si el estado es desconocido intenta ir a un estado conocido
      {
        myRing.formList ="";
        myRing.nodoWinner="";
        waitEVS =false;
```

```
        JbTokenFormador token = new JbTokenFormador();
        myRing.nodoWinner = JbNodo.GetNodoLocal().GetNodo();

        token.setNodoGenerador(JbNodo.GetNodoLocal().GetNodo());
        token.pIdForm.SetValor(myRing.getIdConf());
        token.pListConf.SetValor("");
        token.pListMiem.SetValor("");
        token.pListUnion.SetValor("");
        token.nodoWiner.SetValor(JbNodo.GetNodoLocal().GetNodo());
        token.ExecProcesarToken();
        pTimeout = System.currentTimeMillis();
        myRing.setStatus(ST_ESPERANDO_FORMADOR);
      }
      break;
      case ST_ESPERANDO_FORMADOR: // esperando por formadores
        if (pTimeout+JbRing.TIMEOUT_ESPERA_FORMADOR< System.currentTimeMillis())
          myRing.setStatus(ST_FORMADOR);
        first = false;
        break;
      case ST_FORMADOR: // Paso el periodo de formacion envio el nodo de generacion
        if (JbNodo.GetNodoLocal().GetNodo().equals(myRing.nodoWinner)) {
          if (!waitEVS) {
            JbTokenTest tokenTest = new JbTokenTest();
            tokenTest.setNodoGenerador(JbNodo.GetNodoLocal().GetNodo());
            tokenTest.setRtr(myRing.formList+"|"+JbNodo.GetNodoLocal().GetNodo());
            tokenTest.ExecProcesarToken();
            timeoutEVS = System.currentTimeMillis();
            waitEVS = true;
          }
          if (timeoutEVS+JbRing.TIMEOUT_ESPERA_TEST<System.currentTimeMillis())
            myRing.setStatus(ST_EVS);
        }
        first = false;
        break;
      case ST_EVS: // Paso el periodo de construccion envio el nodo de recuperacion
        if (JbNodo.GetNodoLocal().GetNodo().equals(myRing.nodoWinner)) {
          JbTokenEVS tokenEVS = new JbTokenEVS();
          tokenEVS.setIdConf(myRing.getIdConf()+1);
          tokenEVS.setSec(myRing.mySec+1);
          tokenEVS.setNodoGenerador(JbNodo.GetNodoLocal().GetNodo());
          tokenEVS.ExecProcesarToken();
        }
        first = false;
        break;
      case RING_STATUS_OK:
        if (first) {
          myRing.mySec = 0;
          myRing.myTec = 0;
          first = false;
        }
        break;
      }

      Thread.sleep(1000);
    }
  }
  catch( Exception e) {
    JDebugPrint.logError(e);
  }
}
```

6.5.3. JbOrden

Esta clase se encarga de resolver los conflictos producto del reordenamiento de las transacciones al producirse la unión entre dos anillos. En el modelo que se presenta solo se permite unirse un anillo rojo a uno verde, y es el verde quien decide el orden correcto de las acciones provenientes del anillo rojo.

En las implementaciones se opto por el diseño consistente, donde no se aplican acciones rojas hasta que no son confirmadas. Por lo que esta clase es muy simple.

Se podría implementar en ella políticas más elaboradas, como llevar una base de datos delta con las aplicaciones de las acciones rojas. En este caso, esta clase debería reconstituir la base delta para que sirva de partida a la aplicación de futuras acciones rojas.

O establecer una política de reversado de las transacciones problemáticas. Donde las transacciones fuera de orden sean reversadas.

Ver la sección 34.3 para más detalles sobre algoritmos de reconciliación.

6.6. Rendimiento

Se realizaron diversas métricas a fin de determinar la eficiencia del sistema. Las pruebas se simularon en una sola PC.

6.6.1. Modelo Simple

6.6.1.1. Estudio de la escalabilidad

Se evaluará como responde el sistema aumentando la cantidad de transacciones pendientes de replicación y la cantidad de nodos.

El SAF corre en background y chequea cada 100 milisegundos si tiene transacciones pendientes de replicación. Cuando determina que tiene tarea pendiente la realiza completamente antes de volver a dormirse otros 100 ms. Entre envío y recepción deja el procesador esperando el mensaje de retorno. Se midió un mensaje simple en una máquina con carga normal, conectada a otra en una red LAN sin compuertas a 100 Megabits y cada replicación tarda un promedio de 2 milisegundos en enviar un mensaje de 1 KB y recibir la confirmación de respuestas.

Por lo tanto se supone que el tiempo de respuesta aumentando la cantidad de transacciones y la cantidad de localidades a replicar va ha ser igual a:

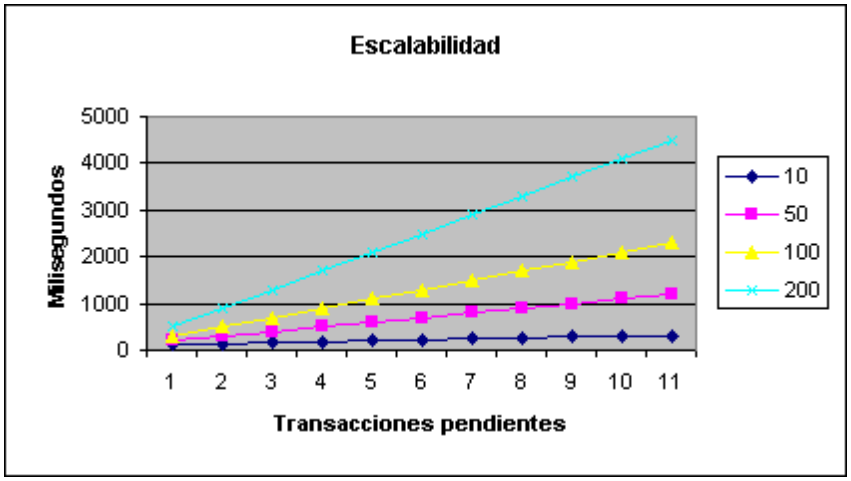
Tiempo Respuesta = 100 ms + ((transacciones pendientes * tiempo mensaje) * cantidad de nodos)

Esto da la siguiente estimación:

Tiempo de propagación en milisegundos

Tiempo mensaje 2

Trans.\Nodos	10	50	100	200
1	120	200	300	500
2	140	300	500	900
3	160	400	700	1300
4	180	500	900	1700
5	200	600	1100	2100
6	220	700	1300	2500
7	240	800	1500	2900
8	260	900	1700	3300
9	280	1000	1900	3700
10	300	1100	2100	4100
11	320	1200	2300	4500



6.6.1.2. Estudio de eficiencia

Se evalúa la eficiencia del sistema, se supone un nodo transmitiendo una cantidad n de transacciones por segundo a una cantidad m de nodos. La eficiencia esta dada por el inverso de la razón de mensajes que se despachan en un segundo. De este modo si el mensaje es menor a 1 el sistema pierde eficiencia ya que no esta logrando trasmitir todos los mensajes generados en el periodo de tiempo acumulando para el siguiente.

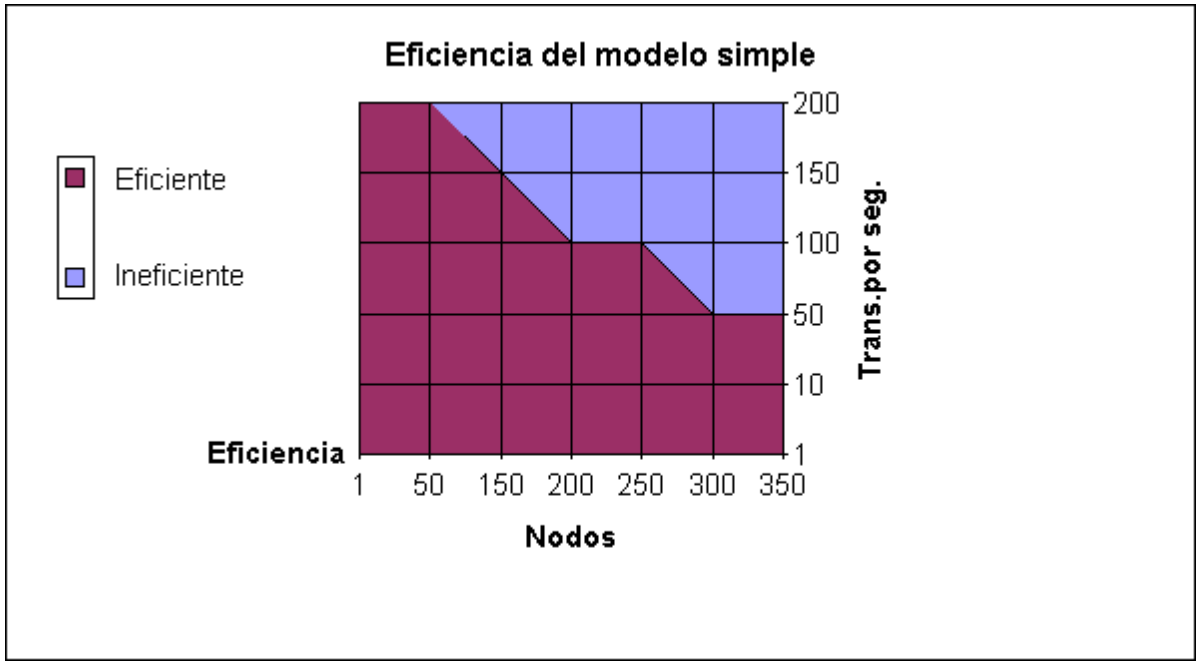
La formula sería:

Eficiencia =1/(100 ms + ((transacciones pendientes * tiempo mensaje) * cantidad de nodos))/60000ms)

Eficiencia del modelo simple

Tiempo mensaje 2

Nodos\Transac.	1	10	50	100	150	200
1	588.235294	500	300	200	150	120
50	300	54.5454545	11.7647059	5.94059406	3.97350993	2.98507463
150	150	19.3548387	3.97350993	1.99335548	1.33037694	0.99833611
200	120	14.6341463	2.98507463	1.49625935	0.99833611	0.74906367
250	100	11.7647059	2.39043825	1.19760479	0.79893475	0.5994006
300	85.7142857	9.83606557	1.99335548	0.99833611	0.66592675	0.49958368
350	75	8.45070423	1.70940171	0.85592011	0.57088487	0.42826552



6.6.2. Modelo Complejo

6.6.2.1. Estudio de la escalabilidad

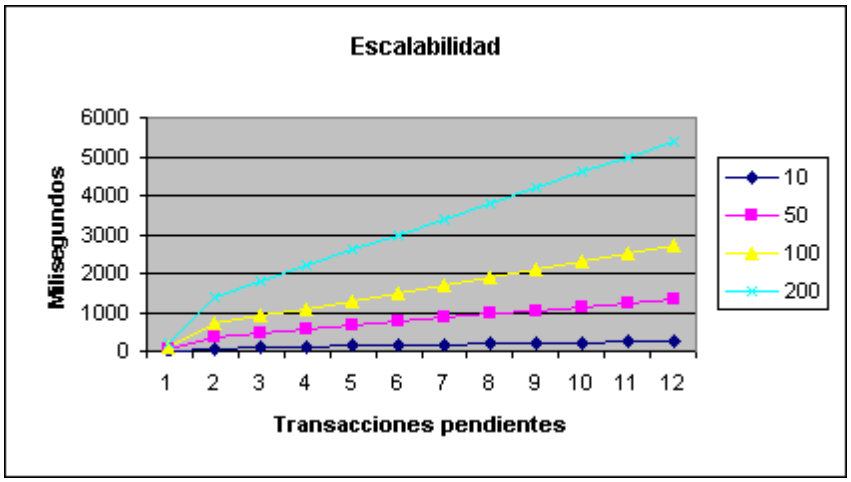
Para estimar la escalabilidad agregando el anillo, el tiempo fijo de 100ms en el modelo simple es cambiado por el peor caso de espera que resulta de esperar una vuelta completa del anillo. Suponiendo que solo un miembro genera información, el retardo del anillo es el tiempo de transferencia del token multiplicado por la cantidad de nodos.

Tiempo respuesta =(cantidad de nodos * Transf..token) + ((transacciones pendientes * tiempo mensaje) * cantidad de nodos)

Tiempo de propagación en milisegundos

Tiempo mensaje 2 Tiempo anillo 5

Trans.\Nodos	10	50	100	200
1	70	350	700	1400
2	90	450	900	1800
3	110	550	1100	2200
4	130	650	1300	2600
5	150	750	1500	3000
6	170	850	1700	3400
7	190	950	1900	3800
8	210	1050	2100	4200
9	230	1150	2300	4600
10	250	1250	2500	5000
11	270	1350	2700	5400



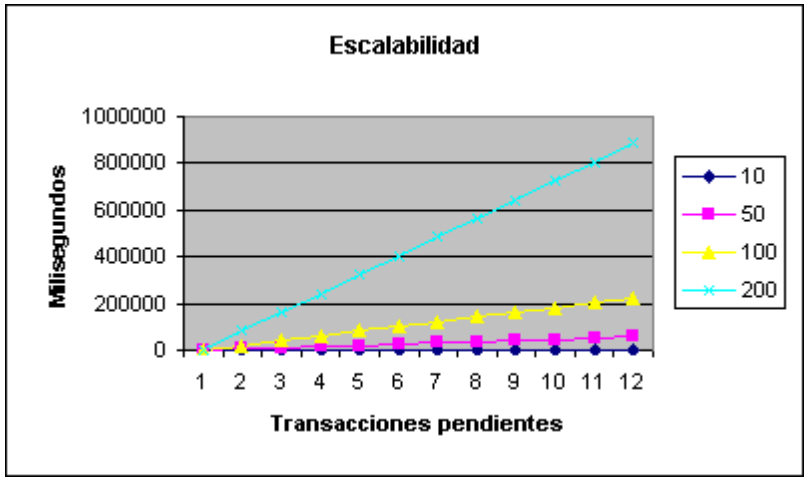
Si se supone que cada nodo tiene la misma carga. Un giro completo al anillo no solo tomará el tiempo de transferencia del token sino que además deberá esperar a que cada nodo trasmita sus transacciones pendientes. El tiempo resultante queda como sigue:

Tiempo respuesta =(cantidad de nodos *(Transf..token+((transacciones pendientes * tiempo mensaje) * cantidad de nodos))) + ((transacciones pendientes * tiempo mensaje) * cantidad de nodos)

Tiempo de propagación en milisegundos

Tiempo mensaje 2 Tiempo anillo 5

Trans.\Nodos	10	50	100	200
1	270	5350	20700	81400
2	490	10450	40900	161800
3	710	15550	61100	242200
4	930	20650	81300	322600
5	1150	25750	101500	403000
6	1370	30850	121700	483400
7	1590	35950	141900	563800
8	1810	41050	162100	644200
9	2030	46150	182300	724600
10	2250	51250	202500	805000
11	2470	56350	222700	885400

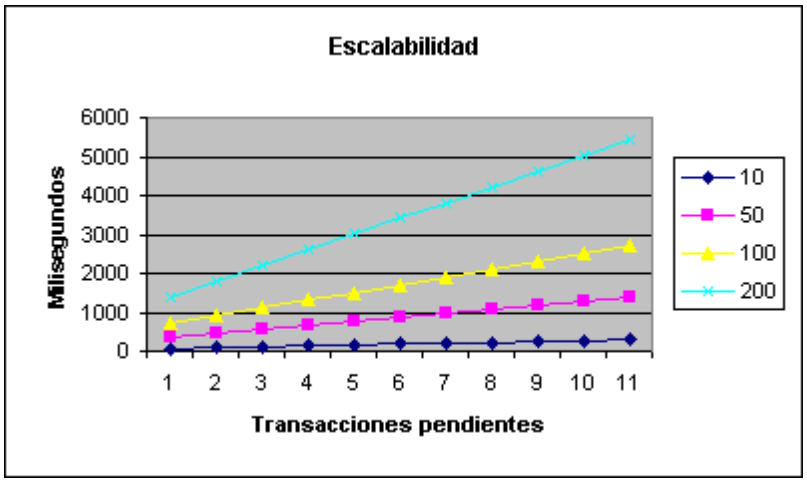


Como se ve en este caso los tiempos se explotan, por esto se incluye un análisis de la posibilidad de usar mensajes "*broadcast*". Se uso la misma fórmula quitando el multiplicador de la cantidad de nodos.

Tiempo de propagación en milisegundos

Tiempo mensaje 2 Tiempo anillo 5

Trans.\Nodos	**10**	**50**	**100**	**200**
1	72	352	702	1402
2	94	454	904	1804
3	116	556	1106	2206
4	138	658	1308	2608
5	160	760	1510	3010
6	182	862	1712	3412
7	204	964	1914	3814
8	226	1066	2116	4216
9	248	1168	2318	4618
10	270	1270	2520	5020
11	292	1372	2722	5422



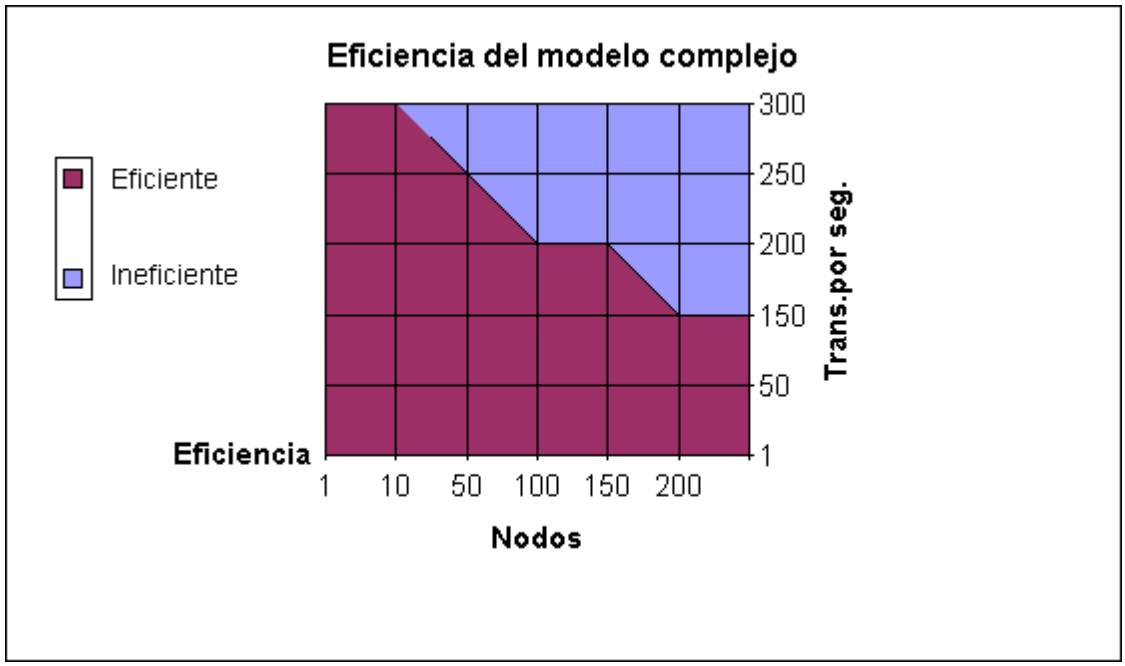
6.6.2.2. Estudio de la eficiencia

Se realizó el mismo análisis que en el modelo simple sobre la eficiencia. Se puso una entidad a generar n transacciones por segundo a m nodos, y se midió la eficiencia del sistema para absorber dicha carga.

Eficiencia del modelo complejo

Tiempo mensaje 2 Transf.Token 5

Nodos\Transac.	1	10	50	100	150	200
1	8571.42857	857.142857	171.428571	85.7142857	57.1428571	42.8571429
50	571.428571	57.1428571	11.4285714	5.71428571	3.80952381	2.85714286
150	196.721311	19.6721311	3.93442623	1.96721311	1.31147541	0.98360656
200	148.148148	14.8148148	2.96296296	1.48148148	0.98765432	0.74074074
250	118.811881	11.8811881	2.37623762	1.18811881	0.79207921	0.59405941
300	99.1735537	9.91735537	1.98347107	0.99173554	0.66115702	0.49586777
350	85.106383	8.5106383	1.70212766	0.85106383	0.56737589	0.42553191



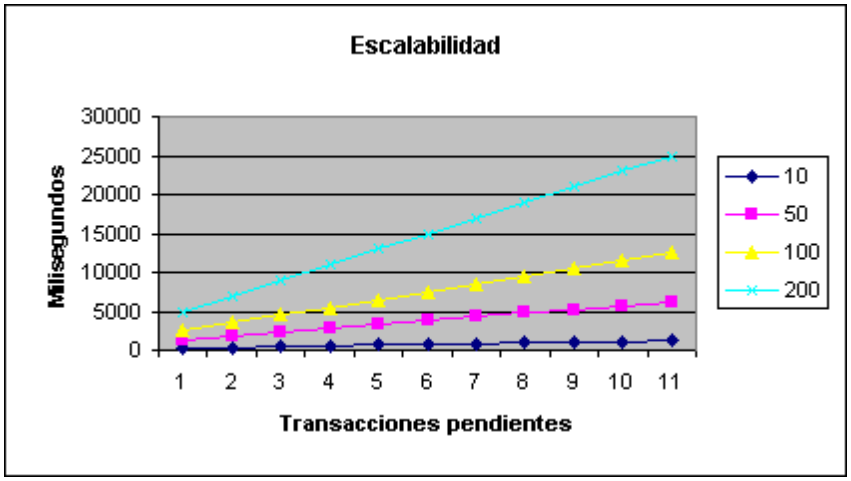
6.6.2.3. Estudio de la recuperación de fallos

Se dividió una red de n nodos en dos mitades iguales, se las puso generar m transacciones por cada localidad, luego se volvió a unir la red. El tiempo para rearmar la red será igual a tres giros del token formador, más la retransmisión de todas las transacciones generadas en cada anillo para cada partición.

Tiempo de propagación en milisegundos

particiones	2		
Tiempo mensaje	2	Tiempo anillo	5

Trans.\Nodos	10	50	100	200
1	250	1250	2500	5000
2	350	1750	3500	7000
3	450	2250	4500	9000
4	550	2750	5500	11000
5	650	3250	6500	13000
6	750	3750	7500	15000
7	850	4250	8500	17000
8	950	4750	9500	19000
9	1050	5250	10500	21000
10	1150	5750	11500	23000
11	1250	6250	12500	25000



CAPÍTULO 7

7.1. Conclusiones

Durante el transcurso de este trabajo se han analizado diferentes opciones para la implementación de soluciones distribuidas sostenibles.

A quedado claro que no existe una solución única, por esta razón se amplio el "*framework*" para que soporte diferentes mecanismos de propagación y dejar así la libertad al desarrollador la posibilidad de optar por la más adecuada para la funcionalidad que desea implementar.

La panacea de una base replicada, sin bloqueos y sin reconciliaciones es lógicamente imposible. En la mayoría de los casos se caen en situaciones donde las decisiones que se deben tomar en el momento de la reconciliación deben ser informadas al usuario y es este quien deberá tomar las decisiones subyacentes.

Esto acota la utilización de la replicación a los planteos donde esta intervención resulte natural o donde las arbitrarias reordenaciones de las transacciones no resulte un inconveniente.

En la práctica el sistema funcionó muy bien para remplazar los maestros periódicos de actualización. También resulto útil como recopilador de información estratégica. En cambio, para manejos de saldos en cuentas corrientes, por ejemplo, se optó por una actualización centralizada de los mismos, ya que no era aceptable la posibilidad de permitir ventas por debajo del saldo.

Lo más interesante del desarrollo es ver como fácilmente permite combinar las diferentes soluciones, y ver que el verdadero poder está en esta mezcla de estrategias para lograr la mayor velocidad y eficiencia.

Los futuros trabajos deberían abocarse a mejorar la rutina de reconciliación, la cual si estuvieran integradas al motor de la base podrían mejorar notoriamente el desempeño de las resincronizaciones.

BIBLIOGRAFÍA

[Agrawal et al., 1997] *Exploiting atomic broadcast in replicated datases*. D. Agrawal, G. Alonso, E. Abbadi, I. Stanoi. In Proceedings of EuroPar, Passau Germany. 1997.

[Al-Houmaily et al., 1995] *Two Phase Commit in Gibabit-networked Distributed Databases*. Y. Al-Houmaily, P. Chrysanthis. Proc. of 8th Intl. Conf. On Parallel and Distributed Computing Systems. September 1995.

[Attiya et al., 1994] *Sequential consistency versus linearisability*. H. Attiya, J. Welch. ACM Transactions on computer Systems, 12(2): 91-122. Mayo 1884.

[Bassiouni 1988] *Single site and Distributed Optimistic protocols for concurrency control*. M. Bassiouni. IEEE Transactions on Software Engineering vol SE-14, num 8 (agosto 1988) pags. 1071-1080

[Batini et al, 1994] *Diseño conceptual de Bases de Datos. Un enfoque de entidades-interrelaciones*. Carlo Batini, Stefano Ceri, Shamkant Navathe. Addison Wesley Iberoamericana S.A. 1994

[Bernstein et al., 1980] *Timestamp based Algorithms for Concurrency Control in Distributed Databases Systems*. P. Bernstein, N. Goodman. Proceedings of the international Conference on Very Large Data Bases 1980. pags. 285-300

[Bernstein et al., 1987] *Concurrency Control and Recovery in Database Systems*. P. Bernstein, V. Hadzilacos, N. Goodman. Addison Wesley 1987.

[Bertone et al., 2001] Distributed processing in replicated image Data Bases. Efficiency analysis. R. Bertone, S. Ruscuni, A. De Giusti, A. Mauriello. Miami, USA 2001

[Bhargava, 1987] *Concurrency and Reliability in Distributed Database Systems*. B. Bhargava (editor). Van Nostrand Reinhold, 1987.

[Bobak 1993] *Distribuyes and multidatabase systems.* A. Bobak. Bantam Books. 1993

[Braginski 1991] *The X/Open DTP Effort*. E. Graginski. Proc. Of the 4th Int'l Workshop on High Performance Transactions Systems, Asilomar. California. Septiembre 1991.

[Breitbart et al., 1984] *ADDS Heterogeneous distributed database system*. Y.J. Breitbart, L.R. Tieman. Proceedings 3rd. Int. Seminar on Distributed Data Sharing Systems, Parma Italia. 1984.

[Breitbart et al., 1999] *Update propagation protocols for replicated databases*. Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, A. Silberschatz. In Proc. Of the ACM SIGMOD Int. Conf. on Management of Data (Philadephia, Pennsylvannia). Pags. 97-108. Junio 1999.

[Buretta 1997] *Data Replication. Tools and techniques for managing distributed information*. Marie Buretta. Editorial: John Wiley & Sons, Inc. 1997.

[Burleson 1994] *Managing Distributed Databases. Building bridges between database islands*. Donald K. Burleson. Wiley QED. 1994. Pag. 56

[Ceri et al., 1983] *Correctness of Query Execution Strategies in Distributed Databases*.S. Ceri, G. Pelagatti. ACM Transactions on DataBase Systems. Vol. 8 num 4 (diciembre 1983) pags. 577-607

[Ceri et al., 1991] *A clasification of update methodos for replicated databases.* Technical report, Computer Science Department, Stanford University. CS-TR-91-1392. 1991.

[Chandy et al., 1983] *Distributed "deadlock" detection* K.M. Chandy, L.M. Haas, J. Misra. ACM transactions on Computer Systems. Vol 1, num 2 (mayo 1983) pags. 144-156

[Chen et al., 1992] *A structural classification of integrated replica control mechanisms. Technical Report.* Department of Computer Science, Columbia University. S.W. Chen, C.Pu, CUCS-006-92.

[Cheung et al., 1990] *The gris protocol: A high performance scheme for maintaining replicated data.* S. Y. Cheung, M. Ahamad, H.H. Ammar. Proc of the Int. Conf. on Data Engineering (IDCE) Los Angeles, California, pag 438-445. Febrero 1990.

[Chrysanthis et al., 1998] *Recovery and Performance of Atomic Commit Processing in Distributed Database Systems.* P.K. Chrysanthis, G. Samaras, Y.J. Al-Houmaily. http://citeseer.nj.com/chrysanthis 98recovery. 1998

[Cooper 1992] *Análisis of distributed Commit Protocols.* E. Cooper. Proc. Of the ACM SIGMO Int'l Conference on Management of Data, pag. 175-183. Junio 1992.

[Cormen et al., 1990] *An introduction to algorithms.* T. Cormen, C. Leiserson, R. Rivest, McGraw Hill, 1990.

[Coulouris et al, 2001] *Sistemas Distribuidos. Conceptos y Diseño.* George Coulouris, Jean Dellimore, Tim Kindberg. Addison Wesley, 2001.

[Cristian et al., 1990] *A low cost Atomic Commit Protocol.* J. Stamos, F Cristian. Proc of 9th Symp. On Reliable Distributed Systems, October 1990

[Date et al, 1998] *Foundation for Object/Relational Databases. The third manifesto.* C.J. Date, Hugh Darwen. Addison Wesley. 1998

[Date 2001] *Introducción a los Sistemas de Bases de Datos.* C.J. Date. Prentice Hall 2001.

[DeWitt et al., 1990] *The Gamma database Machine Project.* D. DeWitt, S. Ghandaharizadeh, D. Schneider, A. Bricker, H. Hsiao, R. Rasmussen. IEEE Transactions on Knowledge and Data Engineering, pag. 44-69. Junio 1990.

[Di Paolo et al., 1999] *Ambiente Experimental para Evaluación de Bases de Datos Distribuidas.* Monica Di Paolo, Rodolfo Bertone, Armando De Giusti, Anales ICIE 99. International Congress of Information Engineering. Buenos Aires 18-20/08/99. pp 729-743

[Di Paolo 1999] *BDD. Estudio de consistencia de transacciones con dos modelos de subsistemas de comunicaciones.* Mónica Di Paolo. Tesina de Grado. Facultad de Ciencias Exactas. 1999. Director: Rodolfo Bertone

[Eager et al., 1986] *Adaptive Load Sharing in Homogeneous Distributed systems* A. Eager D.L, Lozawska, E.D. Zahorjan, J.: IEEE Trans. On Software Ingeneering, vol SE-12. pp 662-675, mayo 1986.

[El Abbadi et al., 1989] *Maintaining availability in partitioned replicated databases.* A. El Abbadi, S. Toueg. ACM Transactions on Database systems. 14,2, pag 264-290.

[Elmasri et al., 2000] *Fundamento de sistemas de Bases de Datos*. 3ra Edición. Rames Elmasri, Shamkant Navathe. Addison Wesley. 2002.

[Eswaran et al., 1976] *The notions of Consistency and Predicate Locks in a Database Systems*. K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger. Communications of the ACM, vol. 19, num. 11 (noviembre 1976) pags. 624-633.

[Fernandez et al., 1981] *Database security and Integrity*. E.B. Fernandez, R.C. Summers, C. Wood. Addison Wesley 1981.

[García Molina 1982] *Elections in Distributed Computing Systems*H. Garcia Molina. IEEE Transactions on Computers, vol c-31, num 1 (enero 1982) pags. 48-59.

[Gray 1978] *Notes on Data Base Operating Systems*. IN Bayer R. , R. M. Graham, and G. Seegmuller (Eds), Operating Systems: An Advanced Course, Lecture Notes in Computer Science, Vol 60 pags 393-481. Sprenger Verlag, 1978

[Gray 1981] *The transaction Concept: virtues and limitations*. Jim Gray. VLDB 1981

[Gray et al., 1993] *Transaction Processing: Concepts and Techniques*, Jim Gray, A. Reuter. Morgan Kaufmann 1993.

[Gray et al., 1996] *The dangers of replication and a solution*. Jim Gray, Pat Helland, Patrick O'Neil, Dennis Shasha. SIGMOD '96. 6/96. Montreal, Canada., pp 173-182.

[Gupta et al., 1997] *Revisiting Commit Processing in Distributed Database Systems*. Ramesh Gupta, Jayant Haritsa, Krihi Ramamritham. SIGMOD '97. AZ, USA, pp 486-497. Mayo 1997

[Haase et al., 1995] *Error propagation in Distributed Databases*. O. Haase, A. Henrich, CIKM'95 Baltimore MD USA, 1995. ACM 0-89791-812-6/95/11 pags. 387-402

[Hallsall 1992] *Data communications, Computer Networks and Open Systems*. F. Halsall. Addison Wesley, Reading, MA 1992.

[Hansen et al., 1997] *Diseño y Administración de Bases de Datos*. Gary W. Hansen, James V. Hansen. Prentice Hall, 1997.

[Haristsa et al., 1997] *More Optimism about Real-Time Distributed Commited Procesing*. Jayant Haritsa, Ramesh Gupta, Krithi Ramamritham. IEEE 1997. pp 123-133

[Hoffer et al., 2002] *Modern Database Management*. Jeffrey Hoffer, Mary Prescott, Fred McFadden. Editorial: Prentice Hall. 2002

[Holliday et al., 1998] *Database Replication: If you must be Lazy, be Consistent*. JoAnne Holliday, Divyakant Agrawal, Amr El Abbadi. Departement of Computer Science. University of California at Stan Barbara. 1998.

[Holliday et al., 1999] *The performance of database replication with group multicast*. JoAnne Holliday, Divyakant Agrawal, Amr El Abbadi. In proceedings of IEEE International Symposium on Fault Tolerant Computing, pags 158-165. 1999.

[Hwang et al., 1996] *Data Replication in a Distributed Systems. A performance study*. S. Y. Hwang, K. S. Lee, Y. H. Chin. Lecture Notes in Computer Science, number 1143. Editors: Wanger & Thoma. Springer Verlag, Pags 708-717. Septiembre 1996.

[Imielinski et al., 1994] *Wireless Mobile Computing: Challenges in Data Management*. T. Imielinski, B. Badrinath. Communications of the ACM 37(10):págs 18-28. 1994.

[Kemme et al., 1998] *A suite of database replication protocols based on group communication primitives*. B. Kemme, G. Alonso. Proc. Of 18th International conference on Distributed Computing Systems, pag. 156-163. Mayo 1998

[Kemme et al., 1999] *Processing transactions over optimistic atomic broadcast protocols*. B. Kemme, F. Pedone, G. Alonso. In proceedings of the International Confrerence on Distributed Computing Systems. Texas, Junio 1999.

[Kemme et al., 2000] *A new Approach to Developong and Implementing Eager Database Replication Protocols*. Bettina Kemme, Gustavo Alonso. Preliminary release, accepted by ACM Trnasactions on Database Systems. 2000.

[Korth y otros,80] Fundamentos de las bases de datos. Pag. 507

[Krishna Reddy et al., 1996] Reducing the blocking in two-phase commit protocol employing backup site. P. Krishna Reddy, M. Kitseregawa. Institute of Industrial Science. The University of Tokyo

[Krishnakumar et al., 1991] *Bounded ingnorance in replicated systems*. N. Krishnakumar, A. Bernstein. In Porc. Of the ACM SIGACT-SIGMOD-SGART Symp. On Principles of Database Systems. Denver Colorado. Pag 63-74. Junio 1991.

[Kroenke 1996] *Procesamiento de Bases de Datos. Fundamentos, diseño e Instrumentación*. David M. Kroenke. Editorial: Prentice Hall. 1996.

[Ladin et al., 1992] *Providing High Availability Using Lazy Replication*. Rivka Ladin, Barbara Liskov, Liuba Shrira, Snajay Ghemawat. ACM Transaction on Computer Systems, Vol 10. Nro 4. Pag. 360-391. Noviembre 1992.

[Laing et al., 1991] *Transaction Management Support in the VMS Operating Systems Kernel*. W. Laing, J. Johnson, R. Landau. Digital Technical Journal, Vol3.. Nro 1, Winter 1991.

[Lamport et al., 1990] *Concurrent Reading and Writing of Clocks* A. L. Lamport, ACM Trans on computer System vol 8 pp. 305-310 noviembre, 1990.

[Lampson et al., 1976] *Crash Recovery in a Distributed Data Storage Systems* L. Lampson, H. Sturgis. Technical Report, Computer Science Laboratory, Xerox, Palo Alto Research Center, Palo Alto, CA. 1976

[Lampson 1981] *Atomic Transaction*. B. Lampson, Distributed Systems: Architecture and Implementation – An Advances Course, B. Lampson (Ed.) Lecture Notes in Computer Science. Vol 105, pp 246-265. Springer Verlag, 1981

[Lampson et al., 1993] *A new Presumed Commit Optimization for Two Phase Commit*. L. Lampson, D. Lomet. Proc. Of the 19th conference on Very Large Databses, pags 630-640. Agosto 1993.

[Landin et al., 1992] *Providing High Availability Using Lazy Replication*. Rivka Ladin, Barbara Liskov, Sanjay Ghemawat, Liuba Shrira. ACM Transactions on Computer Systems, Vol 10, No. 4, November 1992

[Lee et al., 1998] *A new replication strategy for unforeseeable disconnection under Agent-Based Mobile Computing Systems.* K. Lee, Y. Chin. Department of Computer Science. National Tsing Hua University. Taiwan. 1994.

[Lee et al., 2002] *A Fast commit Protocol for Dsitributed Main Memory Database Systems.* Inseon Lee, Hean Yeom. Seuol National University. Korea. 2002

[LeLann 1981] *Error Recovery. Distributed Systems: Architecture and Implementation- An Advanced Course.* G. LeLann. Lecture Notes in Computer Science, Vol. 105, pag. 371-376. Springer-Verlag, 1981.

[Len et al., 2001] *Estudio de actualización de Réplicas de datos en entornos de BDD.* Sergio Len. Rodolfo Bertone. Sebastián Ruscuni. Anales: Cacic 2001. Congreso Argentino de Ciencias de la Computación. El Calafate. Octubre 2001. pp 695-706.

[Len 2001] *Actualización de Réplicas de datos en entornos de BDD.* Sergio Len. Tesina de Grado. Facultad de Informática. 2000. Director: Rodolfo Bertone.

[Liu et al., 1994] *The performance of two-phase commit protocols in the presence of site failure.* M. Liu, D. Agrawal,A. El Abbadi. Proc of 24th Intl.Symp. on Fault Tolerant Computing, June 1994.

[Lorie 1977] *Physical Integrity in a Large Segmented DataBase.* R. A. Lorie. ACM transaction on Database Systems, vol. 2, num 1 (marzo 1977) pags. 91-104

[Maekawa 1985] *A √n algorithm for mutual exclusion in descentralized systems.* M. Maekawa. ACM transactions on Computer Systems, 3, 2, Pag 145-159.

[Melliar91] P.m.Melliar-Smith L.E.Moser and V.Agarwal. Ring-based Ordening Protcols. In proceedings of the internacional Conference on Information Engineering. Pages 882-891. December 1991

[McFadden 1994] *Modern Database management.* 4ta. Edición. F. McFadden, Denjamin Cummings Publishing Company.

[Miaton et al., 1998] *Expediencias en el Análisis de Fallas en Bases de Datos Distribuidas.* Ivana Miatón, Sebastián Ruscuni, Rodolfo Bertone, Armando De Giusti. Análes: Cacic 98. Congreso Argentino de Ciencias de la Computación. Neuquen. Octubre 1998. pp 265-276

[Miaton et al., 1999] *Ambiente de simulación para la Recuperación en un Entorno con BDD.* Ivana Miatón, Sebastián Ruscuni, Rodolfo Bertone, Armando De Giusti. Análes: Cacic 99. Congreso Argentino de Ciencias de la Computación. Tandil. Octubre 1998.

[Miaton et al., 2003] *Actualización de RÉplicas utilizando Agentes Móviles en Entornos Distribuidos sin Conexión Permanente: una experiencia.* Ivana Miatón, Rodolfo Bertone. Congreso Brasileiro de Computación. 2003

[Mohan et al., 1983] *Efficient Commit protocols for the tree of processes model of distributed transactions.* C. Mohan, B. Lindsay. Proc of the 2nd ACM SIGACT/SICOPS Symposium on Principles of Distributed Computing. Agosto 1983.

[Mohan et al., 1986] *Transaction Management in the R* Distributed Data Base Management Systems.* C. Mohan, B. Lindsay and R. Obermarck. ACM transactions on Database Systems, 11(4):378-396. Diciembre 1986.

[Oracle 1997] *Oracle 8 TM Server Replication. Concept Manual.* Oracle. 1997.

[Özsu et al., 1991] *Principles of Distributed Database Systems.* Second Edition. M.Tamer Özsu, Patrick Valduriez. Prentice Hall. 1991.

[Özsu et al., 1996] *Distributed and Parallel Database Systems.* M.Tamer Özsu, Patrick Valduriez. ACM Computing Surveys, Vol 28, No1 March 1996.

[Pacitti et al., 1998] *Improving Data Freshness in Lazy master Schemes.* Esther Pacitti, Eric Simon, Rubens Melo. Int. Proc. Of ICDCS'98. Amsterdam Netherlands. Mayo 1998.

[Pacitti et al., 1999] *Fast Algoritms for maintaining replica consistency in lazy master replicated databases.* E. Pacitt, P. Minet, E. Simon. In Proc. Of the Int. Conf. on Very Large Databases (VLDB) Edinburgh, Scotland. Pag. 126-137. Septiembre 1999.

[Papadimitriou 1979] *The serializability of Concurrent Database Updates.* C.H. Papadimitriou. Journal of the ACM, vol 26, num. 4 (octubre 1979), pag 631-653

[Park et al., 1999] *A New Approach for distributed Main Memroy Database Systems: Causal Commit Protocol.* T Park, I Lee, H. Yeom. Computer Networks. Amsterdam Netherlands. 1999.

[Pedone et al., 1997] *Transaction reordering in replicated databases.* F. Pedone, R. Guerraoui, A. Schipper. In proceedings of the 16th Simposium on Reliable Distributed Systems. Durham, North Carolina, USA. Octubre 1997.

[Pedone et al., 1998] *Exploiting atomic broadcast in replicated databases.* F. Pedone, R. Guerraoui, A. Schipper. In proceedings of EuroPar, Septiembre 1998.

[Perez 1998] *Agentes Móviles en Bibliotecas Digitales.* C. V. Pérez Lezama. Tesis Maestría. Ciencias con Especialidad en Ingeniería en Sistemas Computacionales. Departamento de Ingeniería en Sistemas Computacionales, Escuela de Ingeniería, Universidad de las Américas-Puebla. Mayo. Universidad de las Américas-Puebla. 1998.

[Peterson et al., 1994] *Sistemas Operativos. Conceptos Fundamentales.* J. Peterson, A.Silberschatz, P. Galvin. Editorial: Addison Wesley. 1994

[Primatesta 1995] *TUXEDO, An open Approach at OLTP.* F. Primatesta. Prentice Hall, 1995.

[Pu et al., 1991] *Replica control in distributed systems: an asynchronous approach.* C. Pu, A. Leff. In Proc. Of the ACM SIGMOD Int. Conf. on Management of Data (Denver, Colorado) págs. 377-386. Mayo 1991.

[Ramamritham et al., 1997] *Advances in Concurrency Control and Transaction Processing.* D. Ramamritham, P.K. Chrysanthis. IEEE Computer Society Press, 1997.

[Reed 1983] *Implementing Atomic Actions on Decentralized Data.* D. Reed. ACM Transactions on Computer Systems, vol 1, num 1. (febrero 1983) pags 3-23.

[Reuter et al., 1993] *Transaction Processing: concepts and techniques.* J. Gray, A. Reuter. Data Management Systems. Morgan Kaufmann Publishers, Inc. San Mateo (CA) USA. 1993

[Rosenkrantz et al., 1978] *System Level Concurrency Control for Distributed Data Base Systems.* D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis II. ACM Transactions on Data Base Systems, vol. 3, num 2 (marzo 1978) pags. 178-198

[Rothermel et al., 1990] *Open Commit Protocols for the Tree of Processes Model.* K. Rothermel, S. Pappe. Proc ot eh 10th Intl Conference on Distributed Computer Systems. pag. 236-244. 1990.

[Rothermel et al., 1993] *Open Commit Protocols Tolerating Commission Failures* K. Rothermel, S. Pappe. ACM Transactions on Database Systems. 18(2): 236-244. Junio 1993.

[Ruscuni et al., 2000] *Evaluación de Replicación y Consistencia en Bases de Datos Distribuidas*. Sebastián Ruscuni, Rodolfo Bertone. Cacic 2000. Congreso Argentino de Ciencias de la Computación. Ushuaia, Octubre 2000. pp 145-158

[Ruscuni 2000] *Estudio de Recuperación de errores en BDD*. Sebastián Ruscuni. Tesina de Grado. Facultad de Informática. 2000. Director: Rodolfo Bertone.

[Samaras et al., 1993] Two Phase Commit Optimizations and Tradeoffs in the Commercial Enviroment. G. Samaras, K. Britton, A. Citron and C. Mohan. Proc. Of the 9th intl conference on data engineering, pag. 520-529. Febrero 1993.

[Samaras et al., 1995] Two Phase Commit Optimizations in the Commercial Distributed Enviroment. G. Samaras, K. Britton, A. Citron and C. Mohan. Distributed and Parallel Databases, 3(4): 325-360. Octubre 1995.

[Samaras et al., 1993] *Two Phase Commit optimizations and Tradeoffs in a Commercial Distributed Enviroment*. G. Samaras, K Britton, A.Citron, C. Mohan. Proc. Of the 9th Int'l Conference on data Engineering, pag. 520-529. Febrero 1993.

[Samaras et al., 1995] *Two Phase Commit optimizations in a Commercial Distributed Enviroment*. G. Samaras, K Britton, A.Citron, C. Mohan. Journal of Distributed an Parallel Databases. 3(4): pag 325-360. Octubre 1995.

[Sánchez 1997] *A taxonomy of agents*. J. Sanchez. Reporte Técnico ICT-97-l. ICT. Laboraty of Interactive and Cooperative Technologies, Departament of Computer Systems Engineering, Universidad de las Américas-Puebla, México, Enero 1997.

[Sherman 1993] *Architecture of the Encina Distributed Transaction Processing Family*. M. Sherman. Proc of the ACM SIGMOD Int'l Conference on Management of data pags 460-463, Mayo 1993.

[Silberschatz et al, 1998] *Fundamentos de Bases de Datos*. Abraham Silberschatz, Henry F. Korth, S. Sudarshan. Editorial: Mc Graw Hill. 1998. Tercera Edición

[Simon 96] *Distributed Information Systems. From C/S to distributed multimedia.* Errol Simon. Editorial: Mc Graw Hill. 1996.

[Skeen 1981] *Nonblocking commit protocol* D. Skeen. Proceedings of ACM SIGMOD Conference Junio 1981.

[Sommerville 2002] *Ingeniería de Software.* 6ta Edición. Ian Sommerville. Adddison Wesley. 2002.

[Spector 1991] *Distributed Transaction Processing with Encina.* A. Spector. Proc of 4th Int'l Workshop on High Performance Trnsaction Systems, Septiembre 1991.

[Spiro et al., 1991] *Designing an Optimized Transaction Commit Protocol*. P. Spiro, A. Joshi, T. Rengarajan. Digital Technical Journal. 1991.

[Stacey 1994] *Replication: DB2, Oracle, or Sybase*. D. Stacey. Database Programing & Design. 7, 12. 1994

[Stamos et al., 1990] *A low Cost Atomic Commit Protocol*. J.Stamos, F Cristian. Proc of the 9th Symposium on Reliable Distributed Systems, pags. 66-75. 1990.

[Stamos et al., 1993] *Coordinator log transation execution Protocol*. J. Stamos, F. Cristian. Journal of Distributed and Parallel Databases. 1993.

[Stonebraker 1979] *Concurrency control and consistency of multiple of data in distributed INGRES*. M. Stonebraker. IEEE transactions on Software Engineering, 5(3) pag. 188-194. Mayo 1979.

[Tanenbaum et al., 1996] *Computer Networks*. A. S. Tanenbaum. Prentice Hall, Englewood Cliffs, NJ 1996.

[Tanenbaum et al., 96] D*istributed System Operatives*. Prentice Hall, Englewood Cliffs, NJ 1996. Pag. 2

[Traiger et al., 1982] *Transactions and Consistency in Distributed Databse Management systems*. I.L.Traiger, J.N. Gray, C.A. Galtieri, B.G. Lindsay. ACM Transactions on Database Systems, vol 7, num 3 (septimebre 1982) pag. 323-342

[Triantafillou et al., 1995] *The location-Based Paradigm for replication: Achieving Efficiency and Availability in Distributed Systems*. Peter Triantafillou, David Taylor. IEEE Transaction on Software Engieneering, Vol 21, No. 1, January 1995.

[Upton 1991] *OSI Distributed Transaction Processing. An Overview*. F. Upton. Proc of the 4th Int'l Workshop on High Performance Transation System. Septiembre 1991.

[Wiesmann et al., 2000] *Understanding Replication in Database and Distributed Systems*. M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, G. Alonso. Proc of 20th International Conference on Distributed Computing Systems, pags 264-274. Taipei Taiwan, Abril 2000.

[Wolfson et al., 1992] *Distributed Algorithms for dynamic Replication of Data*. Ouri Wolfson, Sushil Jajodia. ACM 0-89791-520-8. 11th Principles of Databse Systems 6-92. San Diego California 1992.

[Wong 1983] *Dynamic Rematerialiation processing Distributed Queries Using Redundant Data*. E. Wong. IEEE transactions on Software Engieneering, vol SE-9, num 3 (mayo 1993). Pags. 228-232.

[Yeom et al., 2002] *A single Phase Distributed Commit Protocol for Main Memory Database System*. I.Lee, H. Yeom. School of Computer Science. Seoul National University. 2002

[Yamir97] Y.Amir d.Dolev P.M.Melliar Smith L.E. Moser Robust an efiicent replication using group communication. Hebrew university of Jerusalem. University of California, Santa Barbara.

[Zaslavsky et al., 1996] *Primary copy method and its modifications for database replication in distriburted mobile computing environment*. A. Zaslavsky, M. Faiz, B Srinivasan, A. Rashedd, S. Lai. Proc. 15th Symposium on Reliable Distributed Systems. Ontario Canada, pags. 178-187. Octubre 1996.

[Zhang et al., 1994] *Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems*. Aidong Zhang, Marian Nodine, Bharat Bhargava, Omran Bukhres. SIGMOD 94 (mayo 1994) pags. 67-78

[Zimran 1992] *The TwoPhase Commit Performance of the DECdtm Service*. E. Zimran. Proc of the 11th symposium on reliable distributed systems, pags 29-38. 1992.