Trabajo de Grado

Base de Datos Distribuidas
Estudio de Actualización de Réplicas

Len Segio Javier

Facultad de Informática – UNLP
(1900) La Plata, Buenos Aires, Argentina
slen@lycos.com

Dirigida por

Lic. Bertone Rodolfo

LIDI - Facultad de Informática – UNLP
(1900) La Plata, Buenos Aires, Argentina
pbertone@lidi.info.unlp.edu.ar

Septiembre de 2001

La mayoría de las personas son termómetros
que graban o registran la temperatura de la opinión de la mayoría,
no termostatos que transforman y regulan la temperatura de sociedad.

- Marin Luther King, Jr.

Agradecimientos

Quiero agradecer fundamentalmente a mis padres, a mis hermanos, a toda mi familia en general por el tiempo que no les he dedicado, gracias por haber confiado en mi, por la oportunidad que me han dado de descubrir lo que me gusta, de dedicarme a ello, y sobre todo porque sin ellos hubiera sido imposible realizar mi carrera.

A mis amigos, compañeros de trabajo, a quienes me preguntaban como iba (cuándo terminaba), y de alguna manera han hecho posible el avance de este trabajo. En general, a todos aquellos que me animaron y apoyaron en los momentos más difíciles.

A mi director, Rodolfo Bertone, por el esfuerzo y tiempo dedicado para que este proyecto salga adelante.

Indice

Lista de Figuras

Introducción

Tradicionalmente, se agrupaba toda la información en un solo lugar. La idea original era que todos los accesos a datos podrían ser integrados en un solo lugar mediante un *Sistemas de Bases de Datos Centralizadas*. En éste, todos los componentes del sistema residen en un solo nodo o sitio, pudiendo tener acceso remoto a través de terminales conectadas a él. Sin embargo, mas allá de ser un sistema satisfactorio para un buen número de usuarios, el mayor inconveniente es que hay un simple punto de falla. Si una falla ocurre y los datos no están disponibles, podría resultar que todo el sistema dejara de funcionar si depende totalmente de la base de datos para su funcionamiento. Otra desventaja es la posibilidad de ocasionar un cuello de botella para las aplicaciones cuando la carga excede la máxima productividad del nodo central.

En años mas recientes, la evolución de los sistemas de información y el crecimiento no planeado de la información dentro de las organizaciones, ha traído dispersión de los datos en sitios locales o geográficamente dispersos. La necesidad de integrar y compartir dicha información, junto a la disponibilidad de las bases de datos y de las redes de computadoras ha promovido el desarrollo de un nuevo campo denominado Bases de Datos Distribuidas [ÖV91, BG92].

En contraste a las bases de datos centralizadas, las *Bases de Datos Distribuidas (BDD)* no se almacenan completamente en una localidad o sitio central, sino que se distribuye en una red de localidades que pueden estar geográficamente separadas y conectadas por enlaces de comunicaciones. Cada localidad tendrá su propia base de datos y está capacitada para acceder a los datos de otras localidades.

El propósito de distribuir datos sobre múltiples sitios es para ser más confiable, disponible y eficiente, entre otros. En estas aplicaciones, a menudo, el acceso concurrente de una gran cantidad de usuarios al sistema de base de datos, demanda alta disponibilidad y rápidos tiempos de respuesta. La replicación es el factor clave para una mayor disponibilidad (confiabilidad) y eficiencia. Los datos replicados son almacenados en múltiples sitios para que estos puedan ser accedidos por los usuarios cuando alguna de esas copias no está disponible debido a fallas en los sitios, proporcionando mayor disponibilidad. También la performance de los sistemas de base de datos distribuidas es mejorada ubicando los datos en los lugares próximos a donde se usan y poder de esta manera acceder a copias locales, reduciendo la necesidad de costosos accesos remotos.

Por lo tanto se llega a la conclusión de que los sistemas de base de datos distribuidas que utilizan la replicación de sus datos, (comúnmente llamados *Sistema de Bases de Datos Replicadas*), almacenan copias idénticas y redundantes (*réplicas*) de una base de datos distribuidos en diferentes localidades unidas por una red con fines de lograr una mayor performance y disponibilidad de sus datos.

Motivación

En la actualidad, un área de creciente importancia dentro de las Ciencias de la Computación es el relacionado con el procesamiento distribuido sobre arquitecturas heterogéneas, incrementando la eficiencia en la ejecución de un algoritmo global. En particular las aplicaciones de Bases de Datos Distribuidas representan problemas de gran importancia profesional y académica.

Dentro del tratamiento de Bases de Datos Distribuidas uno de los temas de interés consiste en el estudio de actualizaciones sobre diversas réplicas de datos existentes en el entorno. Las ventajas que nos brinda la replicación de datos en cuanto a rendimiento y disponibilidad de los datos, en ocasiones se ven empañadas por las actualizaciones ya que el sistema tiene que asegurar que las copias del dato sean actualizadas apropiadamente, lo cual suponen una sobrecarga mayor. Por estos motivos se consideró un punto de estudio muy importante para la replicación de datos.

Objetivo General

En este proyecto se trata de investigar las características de replicación y fragmentación de la información, en particular la propagación y regulación de las actualizaciones de datos. El esquema de replicación planteado en este estudio se considera estático y preestablecido, interesando para este estudio el comportamiento de los métodos, en cuanto al tratamiento de lecturas y modificaciones de los datos. El soporte de simulación elegido es Java, dado que permite una mayor versatilidad en las construcciones de aplicaciones.

Objetivos Específicos

Los objetivos específicos de este proyecto de tesis son:

1. Implementar cuatro técnicas de replicación basadas en diferentes esquemas de regulación y propagación de actualizaciones sobre las réplicas.

2. Medir las características de performance de dichas técnicas.

3. Comparar las técnicas evaluando las mediciones de los resultados obtenidos.

4. Evaluar los beneficios e inconvenientes de la replicación de datos en general.

Desarrollo del proyecto

Para alcanzar esos objetivos, se realizaron los siguientes pasos:

1. El campo de sistemas de procesamiento distribuido y Bases de Datos Distribuidas fue brevemente estudiado para lograr una comprensión de los problemas y modelos involucrados.

2. Las técnicas de actualización de réplicas fueron cuidadosamente estudiadas para determinar sus características y requerimientos de implementación. Este estudio tubo como pilar una taxonomía proporcionada por Gray en [GHOS96] basada en quién puede actualizar que datos (regulación de las actualizaciones), y una vez actualizado el dato para una copia como será propagado a las demás (propagación de las actualizaciones).

3. Se realizo un análisis y diseño de la modelización de un sistema en red con Bases de Datos Distribuidas.

4. Se especificaron los modelos de aplicación a estudiar y los parámetros a considerar.

5. Se implementó el modelo, en base a las técnicas de actualización especificadas.

6. Las técnicas fueron validadas y los experimentos fueron conducidos parar obtener mediciones de performance.

7. Las técnicas fueron comparadas mediante el análisis de los resultados.

Organización del contenido

La organización del documento consta de 8 capítulos con el siguiente contenido:

El capítulo 1 introduce los conceptos fundamentales de la replicación en Bases de Datos Distribuidas, conceptos necesarios e indispensables para ubicar la naturaleza del proyecto.

El capítulo 2 presenta los esquemas de replicación; siguiendo con un análisis comparativo de las características y funcionalidades de cada uno de ellos.

Los capítulos 3 y 4 describen de manera detallada cada uno de los métodos de propagación y regulación, respectivamente, de las actualizaciones existentes en la integración de métodos de control de réplicas. Se especifican sus principales características, ventajas, costos y aplicación de los mismos, entre otros.

En el capítulo 5 se presenta y describe la combinación de los diferentes esquemas de propagación y regulación de las actualizaciones de las réplicas. Se define un framework

como modelo funcional que permite describir cada uno de los protocolos mediante una secuencia de fases. La metodología y cada una de las fases que la componen se describen de manera detallada, puesto que se consideran elementos fundamentales que sustentan el aspecto teórico del proyecto de investigación.

El capítulo 6 se dedica a la descripción del diseño de un modelo general dedicado a la experimentación en el contexto de actualizaciones sobre réplicas de datos. También se presentan algunos puntos importantes que tienen relación con la implementación de los diferentes esquemas de actualización de réplicas, adaptándolos a sus respectivas combinaciones de regulación y propagación de las actualizaciones.

En el capítulo 7 se presentan y discuten los parámetros e índices utilizados para la evaluación y comparación de los esquemas propuestos. Por último en este capítulo se describen y analizan los resultados experimentados.

En el capítulo 8 resumimos los resultados obtenidos, las aportaciones del proyecto y las opciones para trabajos futuros.

Los apéndices presentan información referente a los informes generados por la herramienta, modelado y estructura de datos que utiliza, además de su carga e inicialización. También se muestra el código de la implementación realizada.

Finalmente se proporciona la bibliografía a la cual se hace referencia a lo largo de todo el material presentado.

Capítulo 1

Replicación de Datos: Conceptos y Definiciones

1.1 Replicación de datos

La *replicación* puede definirse como el proceso de copiar y mantener múltiples objetos. Estos objetos pueden ser datos, procesos y objetos (de Tecnologías Orientada a Objetos), entre otros.

La replicación de diferentes tipos de objetos tienen similitudes pero difieren en muchos aspectos: modelos, asunciones, mecanismos, garantías, e implementación [WPS+00a]. Esta ha sido estudiada en muchas áreas, especialmente en sistemas distribuidos (principalmente con propósitos de tolerancia a las fallas) y en base de datos (principalmente por razones de performance). Nosotros nos centraremos en este último, es decir, en la replicación de datos.

La *replicación de datos* es mucho más que una simple copia de datos entre diferentes almacenamientos. Este abarca el análisis, diseño, implementación, administración, y monitoreo de un servicio que garantiza consistencia de los datos replicados [Bur97].

1.1.1 Ventajas y costos

Las principales ventajas de almacenar las copias de datos en diferentes localidades son [ÖV91, SKS98, Van00]:

Mayor disponibilidad Si falla una de las localidades que contiene una determinada réplica, puede disponerse de esta en otra localidad. Así, el sistema puede continuar procesando consultas que impliquen esta réplica a pesar de haber fallado una localidad. Además, las réplicas restantes pueden seguir funcionando, en contraste con los sistemas de base de datos centralizados, que si ocurre una falla los datos no están disponibles.

Mejor performance La carga del procesamiento de transacciones puede ser distribuida sobre todas las réplicas (máquinas) en el sistema. Esto nos lleva a dos mejoras:

Mayor productividad Las réplicas pueden ejecutar independientemente operaciones de lectura, a causa de que ellas no cambian el estado de la base de datos. Solo las operaciones de escritura necesitan ser ejecutadas sobre todas las réplicas. Debido al hecho de que una parte de la carga transaccional puede ser manejada de manera descentralizada, las réplicas pueden procesar mas transacciones por unidad de tiempo que un sistema centralizado (que siempre tiene que ejecutar todas las operaciones). En el caso en que la mayor parte de los accesos a un ítem resulten

solo en la lectura del mismo, varias localidades podrán procesar consultas que involucren a dicho ítem en paralelo.

Tiempos de respuesta mas cortos Los tiempos de respuestas son mas cortos para las consultas porque pueden ser ejecutadas sobre una réplica, cerca del usuario, sin ninguna comunicación entre las réplicas.

Sin embargo, todas las ventajas que esto puede representar, en términos de rendimiento se desvirtúa frente a las operaciones de actualización. Sus implicaciones son:

Procesamiento agregado y sobrecarga de comunicación Las réplicas deben comunicarse para asegurar que las modificaciones son aplicadas a todas las copias de la base de datos. Esto incrementa la carga sobre las máquinas (mas precisamente, el subsistema de comunicación) y sobre la red de comunicación, que puede degradar la performance global del sistema.

Mayor complejidad del sistema Las réplicas ejecutan asincrónicamente sobre diferentes máquinas y asincrónicamente reciben requerimientos que modifican la base de datos. Una sincronización fiable de las copias de la base sobre las réplicas requiere una mejor comunicación, algoritmos de procesamiento transaccionales más avanzados y un control de las actualizaciones mucho más complejo. El procesamiento de las transacciones y la recuperación de datos se hace más difícil. Por ejemplo, cuando se procese una transacción, ésta puede provocar un requisito de lectura y modificación de datos en los diferentes sitios y de transmitir los mensajes respectivos entre ellos. Después de completada la transacción, el sistema de base de datos distribuida debería asegurar que todos los sitios relevantes hayan completado su procesamiento. Sólo si el procesamiento concluyó normalmente en cada lugar, debería cometerse la transacción. De lo contrario, la transacción debería deshacerse en cada uno de los sitios participantes.

1.2 Grados de consistencia entre las réplicas

Mantener la consistencia de las réplicas, o controlar la calidad del contenido de la réplica, es la función más importante de cualquier servicio de replicación. El propósito de proveer garantías de consistencia es mantener la semántica de los datos. Replicación de datos introduce el problema de mantener la consistencia sobre las diferentes copias de un ítem de dato. Desde el punto de vista de los usuarios, el sistema debe presentarles siempre el estado más actual del dato incluso bajo fallas de comunicación.

Varios autores han considerado diferentes nociones de correctness o grados de consistencia para tratar con la replicación de datos. Entre ellas tenemos consistencia firme o mutua, consistencia floja y consistencia eventual.

1.2.1 Consistencia firme o mutua.

Un criterio fundamental de consistencia para sistemas replicados es garantizar que todas las actualizaciones son aplicadas a cada copia de una manera que asegure consistencia mutua, la cual requiere que las copias de un objeto concuerden en el valor actual para el objeto, otorgando la ilusión que las copias tengan el mismo valor. Decimos ilusión debido a que fallas de sitos o red, o simples consideraciones de performance, las copias físicas pueden contener diferentes valores.

Para garantizar que todas las réplicas se comportasen como una simple copia (propiedad llamada Equivalencia de Una Copia), debemos tener en cuenta que:

- las operaciones físicas de lectura accedan a apropiadas copias para determinar el valor actual (el último valor escrito) de un objeto, y

- las operaciones físicas de escritura actualicen las copias apropiadas para que subsecuentes operaciones de lectura retornen el valor corriente.

De esta manera, logramos que cada acceso brinde datos consistentes, comportándose como un sistema no replicado pero con mayores características de disponibilidad y performance, con lo cual podemos decir que brinda una consistencia fuerte o firme [Bur97, BK97b]

Lamentablemente, estos esquemas requieren un cierto grado de cooperación y coordinación entre las copias del objeto, que puede restringir la disponibilidad, performance y autonomía del sistema. Esta iteración también, puede verse afectada por la comunicación, debido a que generalmente requieren una comunicación fiable para brindar una alta disponibilidad de los datos. Cuando la comunicación falla entre los sitios que contienen copias del mismo dato replicado, la consistencia mutua entre las copias se hace más complicada de asegurar. La más descriptiva de esas fallas de comunicación es la falla de partición, la cual fragmenta la red en subredes aisladas llamadas particiones. Por ejemplo, si tenemos dos réplicas r1 y r2 en diferentes particiones P1 y P2 correspondientemente, si actualizamos la partición P1, no podemos usar la partición P2 disminuyendo la disponibilidad de los datos.

1.2.2 Consistencia floja

En los sistemas donde los factores mencionados anteriormente tales como disponibilidad y performance son críticos, puede ser deseable permitir algún criterio de consistencia más floja que la consistencia mutua.

Son esquemas similares a los de consistencia fuerte con la diferencia que ignoran las dependencias de lectura, con lo cual no aseguran que toda transacción lea datos consistentes. En cambio, las escrituras actualizan la última copia del dato, dando consistencia “instantánea” con respecto a las mismas. La ventaja es que no sacrifican la disponibilidad de los datos para las lecturas, sin embargo, si la disponibilidad de las

escrituras es esencial, sufren los mismos problemas que la consistencia fuerte. Generalmente este grado de consistencia es impuesto por esquemas que ponen la disponibilidad sobre la consistencia. En esta aproximación, cada schedule transforma un estado consistente de la base en otro consistenete. Esto es, un schedule debe preservar la integridad de la base, pero no necesita asegurar que toda transacción lea datos consistentes.

1.2.3 Consistencia eventual

En algunos esquemas, una consistencia mínima, o la intervención externa para restaurar la semántica de los datos es a menudo considerada más aceptable que prohibir completamente el acceso a los datos, especialmente cuando existen problemas de comunicación, como por ejemplo en ambientes móviles.

En este esquema cada acceso puede dar datos inconsistentes, no da ninguna garantía de consistencia durante el acceso, pero el dato eventualmente converge a un estado consistente. Se basa en la propiedad de Convergencia [GHOS96] a partir de la cual se garantiza que cualquiera sea el estado actual de las réplicas, si no hay nuevas actualizaciones emitidas, no hay fallas en las réplicas, y las réplicas son capaces de comunicarse libremente en un determinado tiempo, todas los contenidos de las réplicas son idénticos. Un buen ejemplo de convergencia es Lotus Notes o Microsoft Access [Ham].

Con esta propiedad logramos que el estado resultando contenga los cambios más recientes, pero puede pasar que, además de lecturas sobre valores de datos viejos, aparezcan conflictos de actualizaciones, o actualizaciones hechas a copias viejas, teniendo la consiguiente perdida de actualizaciones. Esas inconsistencias son detectadas y resueltas cuando el dato es integrado, más que cuando es generado. La resolución es hecha automáticamente o manualmente dependiendo de la semántica de los datos. Esta integración y resolución de datos a otros sitios asegura que se cumpla la eventual convergencia de los datos.

Podríamos decir que este grado de consistencia es el requisito mínimo de cualquier algoritmo de replicación; sin esta garantía, el contenido de la réplica puede corromperse, haciendo el sistema prácticamente inútil.

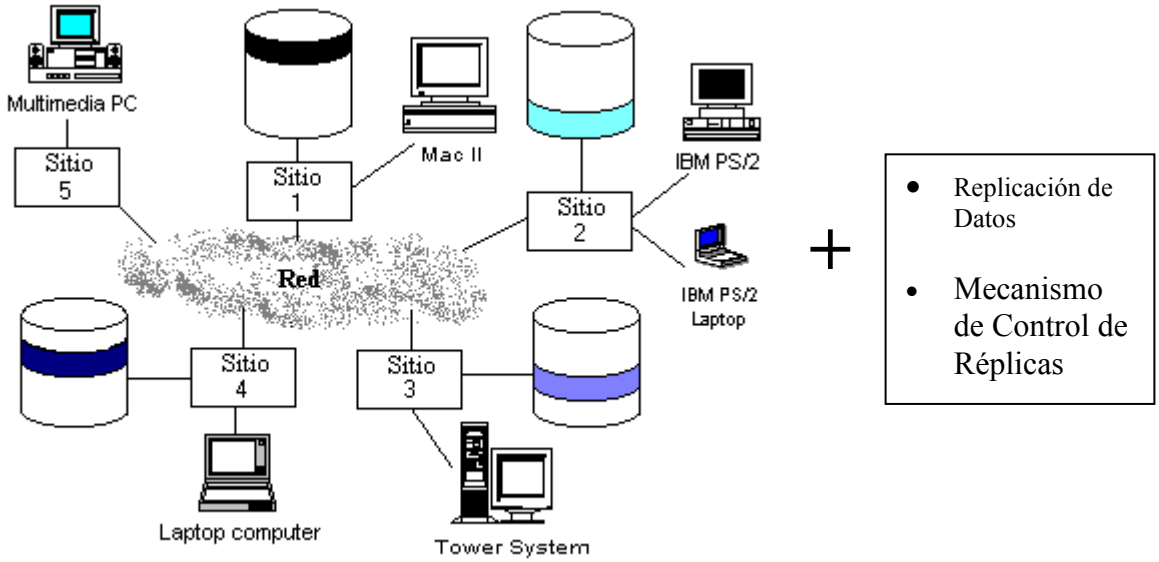
Debido a que las actualizaciones pueden ser hechas en cualquier orden nos da una alta disponibilidad, siendo una buena alternativa para las situaciones en las cuales es esencial dar acceso a los datos, y la conectividad de la red es inestable o no esté presente.

1.3 Sistema de bases de datos replicados.

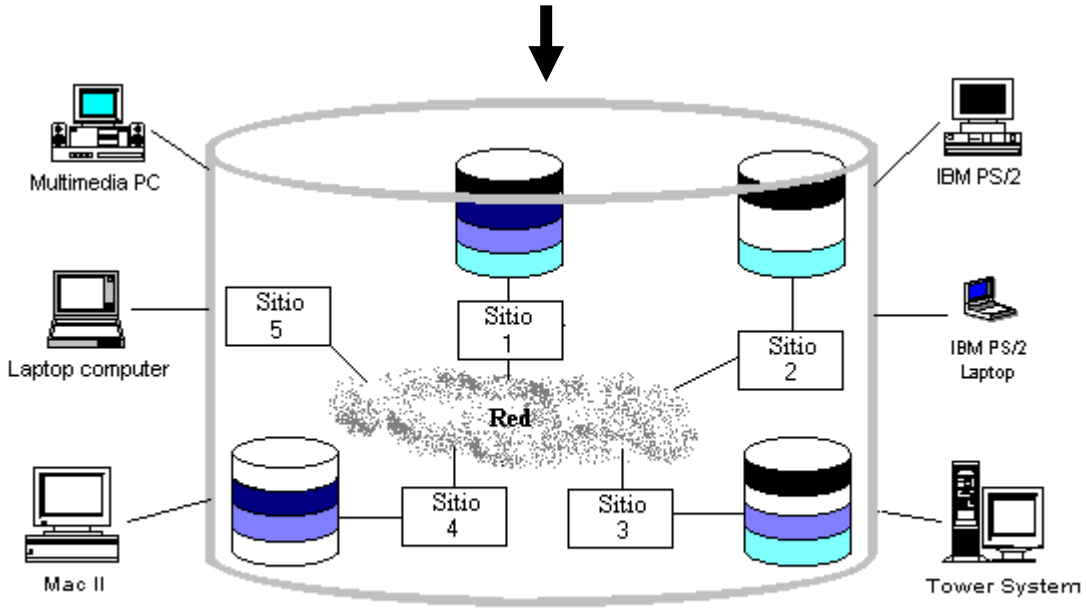
Básicamente, un sistema de base de datos replicado es un SBDD donde los datos tienen copias almacenadas redundantemente en más de un sitio [ÖV91]. Podríamos asumir la existencia de un dato D con copias d1, d2,..., dn. Nos referimos a D como el *dato lógico* y sus copias como *datos físicos*. En estos sistemas, una *operación lógica* (sobre un dato

lógico, a nivel de aplicación) puede resultar en una o más *operaciones físicas* sobre los datos físicos, por ejemplo una escritura requerida por un usuario sobre D consistiría en un conjunto de escrituras físicas sobre todas las copias físicas d1, d2,..., dn. Por lo cual deben contar con algún mecanismo para controlar tales réplicas, mapear las operaciones lógicas en físicas y por supuesto de una manera transparente al usuario.

Un sistema de replicación podríamos representarlo mediante la Figura 1.1, donde los usuarios del sistema no deben preocuparse por el hecho que los datos están replicados, lógicamente actúan como en un sistema de base de datos no replicado.



Sistema de Bases de Datos Distribuidas



Sistema de Bases de Datos Replicadas

Figura 1.1: Sistema de replicación

El hecho que los usuarios actúen lógicamente como en un sistema de base de datos no replicado (ver Figura 1.1) es una de las principales funciones de un sistema de base de datos replicados, es decir, debe otorgar *transparencia sobre la réplicas de datos* [ÖV91]. Esta trasparencia considera si los usuarios deben ser consientes de la existencia de las copias o si el sistema es quien maneja la administración de las copias y el usuario actúa como si hubiese una simple copia de los datos. Se debe tener en cuenta que si el usuario se encargara de manejar las réplicas en un sistema, hace que el manejo transaccional sea más simple para el Sistema Manejador de Bases de Datos Distribuidas (SMBDD). Sin embargo, el usuario tendría que estipular el hecho que ciertas acciones pueden y/o deben realizarse sobre múltiples copias pudiéndose olvidar de mantener la consistencia de las réplicas y así tener datos diferentes, obviamente, prefiere no involucrarse con el manejo de las copias. Dada estas consideraciones, si existen réplicas de objetos de la base de datos, su existencia debe ser controlada por el sistema y no por el usuario. Una tarea adicional en la mayoría de ellos es asegurar Serializabilidad de Una copia, propiedad que definiremos a continuación.

1.3.1 Serializabilidad de Una Copia (One-Copy Seriability)

Esta propiedad esta relacionada con el control de concurrencia; asegura que la ejecución concurrente sobre datos replicados es equivalente a una ejecución serial sobre datos no replicados [ÖV91].

Surge de la combinación de las propiedades de *Serializabilidad* en base de datos no replicadas y *Equivalencia de Una Copia* en base de datos replicadas, que definiremos a continuación:

Serializabilidad: Informalmente, la serializabilidad asegura que cualquier ejecución concurrente de transacciones que son permitidas por un algoritmo de control de concurrencia correcto es equivalente a alguna ejecución serial de ellas.

Equivalencia de Una Copia (One-Copy Equivalence): es un criterio de consistencia de datos replicados, el cual sostiene que múltiples copias de un objeto aparecen como un simple objeto lógico [CP92]. Podríamos decir que es el caso ideal en la consistencia de los datos lograda por lo diferentes algoritmos de control de réplicas debido a que da la ilusión de que no hay datos replicados.

Este requerimiento adicional, en el cual los objetos replicados se comportan como si ellos no lo estuvieran, es el que extiende la Teoría de la Serializabilidad a los sistemas de base de datos replicados.

Es una propiedad que ha atraído la atención y la mayoría de los esquemas de replicación de datos adoptan Serializabilidad de Una Copia como criterio de consistencia [HAE, KA, LCC, WCYC, BK97a, ABKW98]. Preservarla es una condición suficiente para la exactitud o correctness de un algoritmo de manejo de réplicas. Implementada en un sistema distribuido combinando un mecanismo de control de concurrencia que asegure

serializabilidad con un método de control de réplica para hacer la base de datos Equivalente de Una Copia. Podríamos usar 2PL para implementar la aislación y ROWA para mapear las operaciones lógicas a operaciones físicas [BHG87].

Tiene la ventaja que los métodos que utilizan este criterio pueden ser considerados como manejadores de todo tipo de conflictos, no permitiendo divergencias y asegurando que todas las réplicas de un objeto aparezca consistente en todo momento.

Por otro lado, para poder garantizarla se requiere que las réplicas colaboren, típicamente involucrando múltiples rondas de intercambio de mensajes que implican un costo significante. La necesidad de tal colaboración tiene dos consecuencias importantes.

- la performance del sistema degrada significativamente porque muchas réplicas necesitan ser accedidas sincronizadamente antes de cometer la transacción.

- ellos no pueden tratar con fallas de partición de la red.

Estos puntos traen problemas para aplicar esos métodos en la práctica siendo las principales razones por la falta de base de datos replicadas que satisfagan el criterio de Serializabilidad de Una Copia.

1.3.2 Protocolo de control de réplicas

Estos protocolos son el mecanismo necesario en los sistemas de base de datos replicados para controlar el funcionamiento correcto de las réplicas. Estos protocolos tienen principalmente 3 funciones [CP92]:

- Como serán mapeadas las operaciones lógicas en operaciones físicas sobre los datos: se definirá el número de copias que serán leídas o escritas físicamente por cada operación lógica como por ejemplo ROWA (Read One Write All), ROWAA (Read One Write All Available) [BHG87] , entre otros.

- El grado de consistencia que será mantenido entre las réplicas: el criterio fundamental sería que el sistema de la ilusión que todas las copias tienen el mismo valor, propiedad llamada *Equivalencia de Una Copia*, pudiéndose comportar como un sistema no replicado. Sin embargo, una opción es reducir este grado de consistencia, ya sea por razones de disponibilidad debido a fallas de sitios o de red, o simplemente consideraciones de performance.

- El grado de reconfiguración dinámica: muchos protocolos de control de réplica asumen que las copias físicas son alocadas en los sitios de manera estática [KA, BHG87, AES97, BKR+99]. Otros en cambio [TS, WJ92, ZA98], modifican dinámicamente el número y localización de las réplicas, para poder por ejemplo regenerar copias cuando un sitio falla para brindar mayor disponibilidad.

En la implementación de este mecanismo, comúnmente denominado Protocolos de Control de Réplicas, o simplemente protocolos de replicación, pueden ser integrados con servicios de otros sistemas tales como [CP92]:

- Protocolos de comunicación para poder transmitir los mensajes entre las réplicas.

- Protocolos de acuerdo distribuido, utilizado en muchas situaciones que los participantes necesitan llegar a un acuerdo sobre el valor o estado de las réplicas.

- Protocolos de control de concurrencia distribuida, utilizados para preservar la serializabilidad en la base de datos [ÖV91, BG92]. Esta integración, varía mucho de las asunciones realizadas. Un protocolo de replicación puede asumir que un algoritmo de control de concurrencia manejara los conflictos, o recíprocamente, un algoritmo de control de concurrencia puede asumir que cada operación lógica es atómica, pudiendo trabajar para base de datos replicadas y no replicadas, obligando al protocolo de replicación que maneje los conflictos entre las operaciones físicas dentro de una transacción.

1.3.2.1 Diseño de un sistema de base de datos replicados

Para aclarar estos conceptos, mostraremos el diseño de un sistema de base de datos replicados estudiado en [BH93]. El mismo consiste de siete servidores, cada uno de los cuales encapsula un subconjunto de la funcionalidad del sistema. (ver Figura 1.2). Los servidores son:

User Interface (UI) Es un front-end que permite a los usuarios invocar consultas sobre una base de datos relacional.

Action Driver (AD) Este parse traduce las consultas en una secuencia de operaciones de lectura y escritura de bajo nivel.

Access Manager (AM) Es el responsable del almacenamiento de los datos, índices, y la recuperación de información del dispositivo físico.

Concurrency Controller (CC) Chequea que las acciones de lectura y escritura de diferentes transacciones no tengan ningún tipo de conflicto.

Atomicity Controller (AC) es el responsable de asegurar que las transacciones cometan o aborten atómicamente en todos los sitios.

Replication Controler (RC) Maneja las múltiples copias de los datos para otorgarle al sistema mayor confiabilidada y consistencia mutua de los datos replicados. Es la parte del manejo transaccional responsable de asegurara Serializabilidad de Una Copia. Su implementación requiere el uso de un directorio de datos así como información para determinar que sitios involucrar en la realización de cierta operación lógica.

Surveillance Controller (SC) Recoge infamación de conectividad sobre los sitios, e informa de los cambios al Replicación Controller.

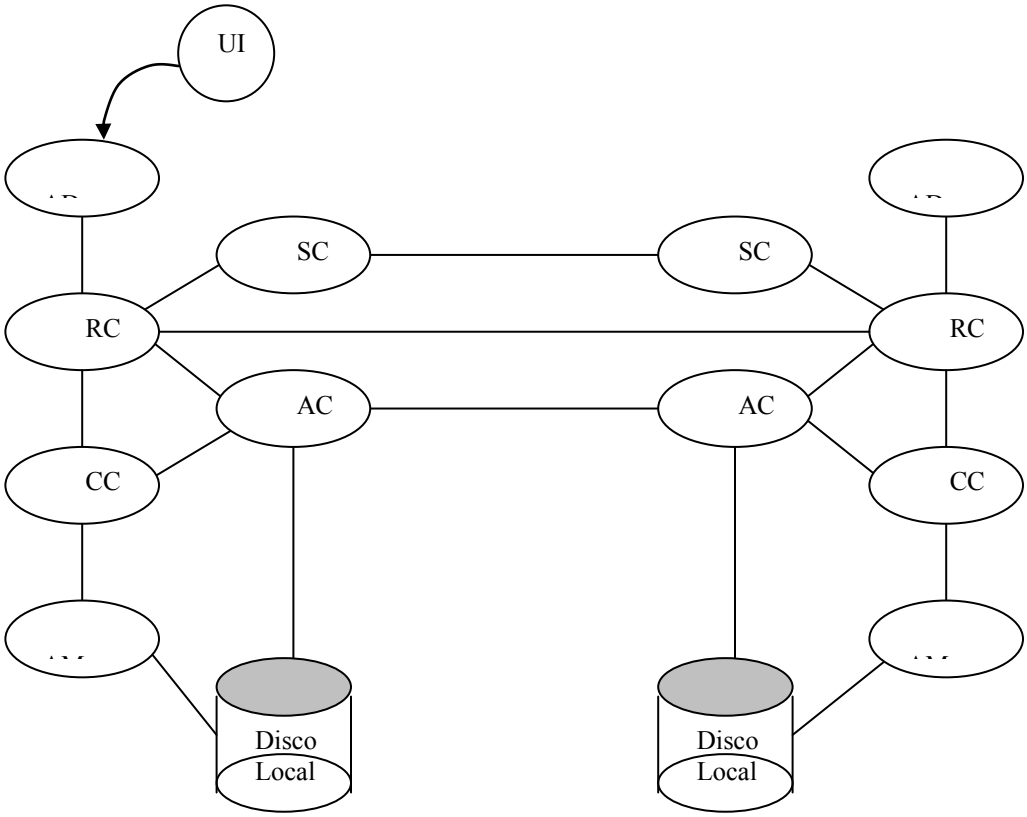


Figura 1.2: Diseño de un sistema de base de datos replicados

La iteración entre los servers puede ser vista claramente en la Figura 1.2. Generalmente, como en este modelo, el control de réplicas (RC) es creado encima de un componente de control de concurrencia (CC) que garantice serializabilidad de manera equivalente a las bases de datos no replicados. Por lo tanto, en la implementación del Transaction Manager, el control de réplicas esta jerárquicamente mas arriba que el control de concurrencia. Una operación lógica sobre un objeto de la base de datos es testeado por el Replication Controller para asegurar Equivalencia de Una Copia y entonces ser transformadas en operaciones físicas sobre las réplicas disponibles, para luego ser pasadas al CC para chequear la serializabilidad.

Capítulo 2

Esquemas de Replicación

2.1 Una definición

El *esquema de replicación* de datos de una base de datos distribuida determina cuántas réplicas de cada ítem de dato[1] son creadas y a qué procesador son asignadas [WJ92].

2.2 Distribución de datos

La distribución de datos en un SBDD es una decisión crucial de diseño que implica establecer cómo cada fragmento o copia de un fragmento se debe asignar a un sitio determinado, de esta manera de define para cada nodo el conjunto de datos que va a administrar o controlar localmente.

Esta decisión se realiza en forma estática al momento del diseño o de la ampliación de la red, pero no interviene la performance real del sistema, sino que se basa en el comportamiento previsto para las transacciones [ZA98]. A su vez, tenemos dos alternativas: sin redundancia de datos o con redundancia de datos [BG92, Bur97].

2.2.1 Sin redundancia de datos

Estos modelos no soportan replicación, cada fragmento reside en un solo sitio (es único en la BD, lo que implica que deben ser disjuntos). Incluyen el modelo *centralizado* y el *particionado*.

2.2.1.1 Modelo centralizado

El modelo centralizado no involucra distribución de datos. Todos los datos están almacenados en un simple manejador de recursos. El uso de los datos puede ser distribuido vía acceso local o remoto, pero el dato reside en una localidad central (ver figura 2.2).

[1] Un ítem de dato puede ser un objeto, una relación etc.

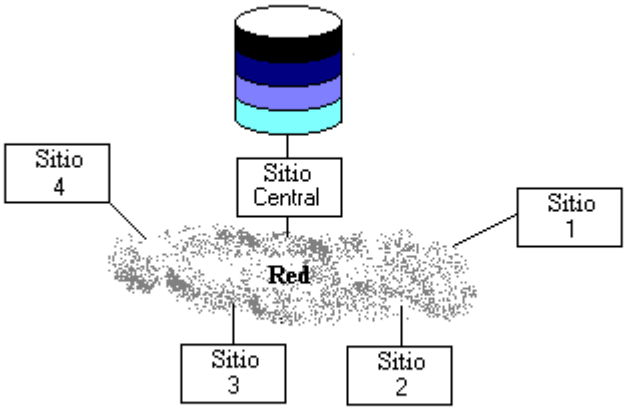


Figura 2.2: Modelo centralizado

Este modelo toma ventajas de la centralización de los datos, tales como:

- Los datos están siempre consistentes y actualizados. El algoritmo de actualización es muy simple. Solo se envía la actualización al sitio central, el cual manejara la serialización para proveer consistencia de datos. Debido a que no existe redundancia, todos los usuarios siempre ven exactamente el mismo dato.

- Esta estrategia tiene el efecto de minimizar los costos de almacenamiento pues existe una sola copia de la BD en un único sitio, minimizando así los costos de almacenamiento secundario.

- La falta de distribución hace el control, la seguridad, y el mantenimiento muy simple.

- Las actualizaciones de la BD pueden manejarse en forma relativamente fácil. Esto implica que un diseñador que utiliza este método de distribución esta relevado de considerar algoritmos de sincronización de actualizaciones muy caros, dado que no se requieren.

- Otro aspecto positivo es que el procesamiento de consultas se simplifica en los sitios remotos. No se requiere optimización porque todos los pedidos se envían a un único sitio y se procesan secuencialmente.

  Sin embargo, esta centralización tiene algunos problemas asociados:

- La confiabilidad del sistema es dependiente de la disponibilidad de la BD para servir usuarios. El mayor inconveniente de éste modelo es que hay un simple punto de falla. Si una falla ocurre y los datos no están disponibles, podría resultar que todo el sistema dejara de funcionar si éste depende totalmente de la BD para su funcionamiento.

- Otro punto a tener en cuenta son los altos costos de los accesos remotos en términos de demoras de comunicación y contención. Si todos los usuarios requieren información del sitio central, serán encolados basándose en la prioridad de sus pedidos. El sitio central

tiene solo la capacidad de servir serialmente, y cuantos más usuarios entren, el tiempo de servicio promedio puede mantenerse constante, pero las demora pueden volverse intolerable debido a la estadía en la cola.

Esta aproximación es factible cuando el volumen de actualizaciones es mucho mayor que la de consultas.

2.2.1.2 Modelo particionado

Esta opción utiliza la noción de tener una sola copia de la BD como en el caso centralizado, pero distribuye las partes de ella en los deferentes sitios (ver Figura 2.3). La idea principal es buscar un balance natural del procesamiento entendiendo que la mayoría de las transacciones accederán solamente a datos locales y por lo tanto tener dichos datos físicamente en el mismo nodo parece ser una buena opción.

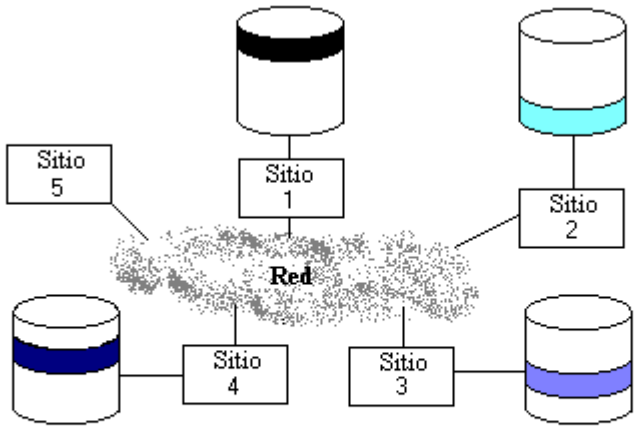


Figura 2.3: Modelo particionado

Como el modelo centralizado, la mayor ventaja es que el dato esta siempre consistente y actualizado. Pero a su vez tiene otras consideraciones que surgen del particionamiento de los datos, ellas son:

- Las actualizaciones serán más baratas y rápidas que los casos centralizado y replicado porque todos los locks, procesamiento, etc., deben ser realizados solo en un sitio que generalmente no esta sobrecargado como en el caso centralizado, y no requiere sincronización de actualización sofisticada, como en el caso replicado.

- Este método, como en la versión centralizada, minimiza el volumen de almacenamiento porque existe una sola copia, pero puede tener un alto costo de hardware al necesitar almacenamiento secundario en todos los sitios.

- Con este tipo de distribución, la confiabilidad es mas alta que en la versión centralizada, pero menos que en la replicada. El caso de un simple punto de falla ha sido reducido pero no eliminado. El sistema completo no se pierde si un nodo falla, solo las aplicaciones que referencian datos en el sitio fallado son afectadas.

- En este modelo, el control, la seguridad, y el mantenimiento son ligeramente más complejo que en el modelo centralizado porque múltiples localidades son usadas para almacenar los datos.

  Este método es adecuado para sistemas de computadoras distribuidas donde la mayoría de los pedidos de datos son solo de nodos específicos. Esto es, existe una alta localidad de referencia. En la fase de diseño, los costos totales de recuperación se reducen requiriendo localidad de referencia como criterio de particionado. Solo los usuarios que piden datos de sitio remotos sufren sobrecarga por comunicaciones.

2.2.2 Con redundancia de datos

Estos modelos soportan replicación, e incluyen los modelos Totalmente Replicados o Parcialmente Replicados

2.2.2.1 Modelo totalmente replicado

Es el caso extremo de los diferentes grados de replicación (replicación total), en el cual se replica toda la BD en todos los sitios del sistema distribuido (ver Figura 2.4).

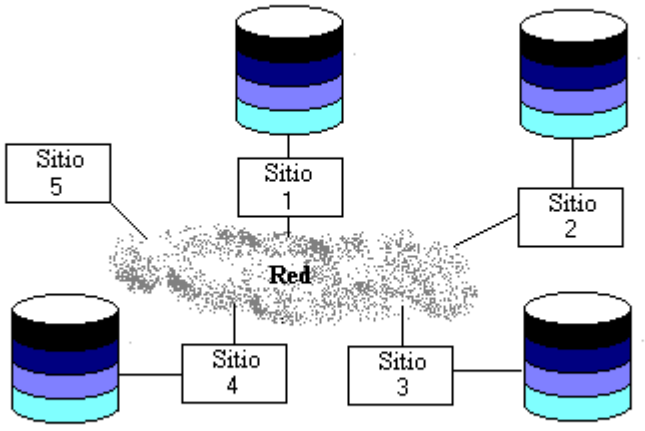


Figura 2.4: Modelo totalmente replicado

Entre los beneficios que nos da la replicación total tenemos:

- El mayor beneficio es para los usuarios finales en términos de confiabilidad aumentada debido a las múltiples copias y tiempo de respuesta reducido para las consultas. La mejora de confiabilidad es causada por la habilidad del sistema distribuido para continuar las operaciones en caso de fallas y el hecho de que en el momento en que hay una copia de la BD disponible, la BD puede servir a pedidos de usuarios.

- El costo de las consultas se minimiza porque no hay overhead por comunicaciones. Esto implica que todos los pedidos pueden ser manejados localmente y que todas las BD son corrientes y consistentes.

Sin embargo este esquema tiene dos costos, el primero relacionado con el hardware, y el segundo, quizás el más importante para nosotros que tiene que ver con las actualizaciones de los datos:

- Este método de distribución posee el costo mas alto asociado con el hardware debido a las N copias (si hay N sitios).

- El problema asociado a este modo de distribución está en las actualizaciones, que son muy caras para realizar. Cuando un usuario actualiza su copia local, debe modificarse todas las otras para mantener la consistencia de los datos. Esto requerirá algoritmos de sincronización de actualización especializados para realizar la modificación de manera lógica y correcta. La complejidad de los algoritmos requeridos y las comunicaciones involucradas están directamente relacionadas con el número de copias en la red y los requerimientos de consistencia para la información. Esto implica que las técnicas de control de concurrencia y recuperación se tornen más costosas de lo que serían sin replicación.

2.2.2.2 Modelo parcialmente replicado

Este modelo otorga replicación parcial, lo cual significa que cada ítem de dato tiene una o más copias residiendo en sitios arbitrarios (ver Figura 2.5).

Tiene su base en los tres modelos anteriores. Se permite que la BD tenga cualquier combinación de los tres modelos previos. Entre estos extremos, tenemos una amplia gama de replicación parcial de los datos, es decir, algunos fragmentos pueden estar replicados y otros no.

Este modelo representa la selección de las mejores características de los anteriores minimizando los efectos de las porciones costosas de cada uno, teniendo como mayor costo que la propagación de datos y la serializabilidad pueden resultar mucho más complejas [KA00].

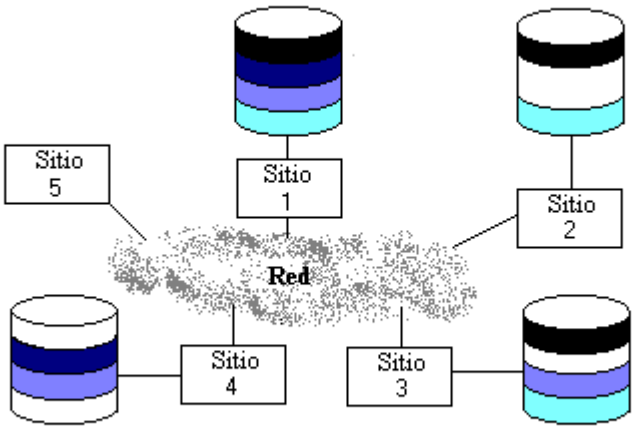


Figura 2.5: Modelo parcialmente replicado

2.2.3 Comparación de los diferentes grados de replicación

En la Figura 2.6 se comparan la complejidad de implementar o tomar ventaja de las diferentes alternativas de replicación, respecto de los diferentes aspectos importantes en bases de datos distribuidas [ÖV91].

	PARTICIONAMIENTO	**REPLICACIÓN PARCIAL**	**REPLICACIÓN COMPLETA**
Procesamiento de Consultas	Moderado	Moderado	Fácil
Control de Concurrencia	Fácil	Dificil	Moderado
Confiabilidad	Bajo	Alto	Muy Alto
Realidad	Aplicación Posible	Realista	Aplicación Posible

Figura 2.6: Comparación de los grados de replicación

Si los datos están totalmente replicados, el procesamiento de consultas va a ser fácil, vamos a tener una complejidad moderada en el control de concurrencia y una confiabilidad

alta. En cambio si la replicamos parcialmente, el procesamiento de consultas y el control de concurrencia será mas complicado, y la confiabilidad no será tan alta. A pesar de esto, sabemos que el caso realista es realizar una replicación parcial.

2.3 Esquemas estáticos

Actualmente, el esquema de replicación de datos es normalmente establecido de una manera *estática*, cuando se diseña la base de datos distribuida, y permanece hasta que el diseñador intervenga manualmente para cambiar el número de réplicas o su localización.

Los esquemas tradicionales de replicación basados en métodos estáticos (copia primaria, ROWA, entre otros, [BHG87]) no cambian con la ejecución de las transacciones y son diseñados para trabajar en ambientes estacionarios. Generalmente asumen ciertas estructuras lógicas (tales como árboles, grillas, entre otros) y una cierta localización de réplicas dentro de ésta estructura lógica. Una vez que la estructura y localización son especificados, ellos deben permanecer inalterados a lo largo de la vida de los objetos replicados que son manejados por el método de replicación. Estos esquemas trabajan bien porque los host son fijos y los patrones de acceso (lecturas y escrituras emitidas por cada procesador) son conocidos a priori y relativamente estáticos. Sin embargo, si este patrón cambia dinámicamente, de maneras imprevisibles, un esquema de la replicación estático, puede llevar a severos problemas de performance.

Este prerequisito de tener una configuración “inmutable” alivia el esquema de replicación del manejo de sobrecargas adicionales resultando algoritmos más simples y menos costosos.

2.4 Esquemas dinámicos

Los esquemas dinámicos son aquellos que de acuerdo a las variaciones del patrón de acceso y carga del sistema, cambian dinámicamente el esquema de replicación de un dato. Este cambio consiste en crear nuevas réplicas o eliminar algunas:

Creaciones de nuevas réplicas. Cuando tenemos nodos que leen frecuentemente réplicas remotas, es conveniente generar copias locales de dichas réplicas para reducir el intenso trafico en la red. Así, beneficia el acceso a costa de mayor espacio de almacenamiento local.

Eliminaciones de réplicas. Si una réplica local es accedida muy poco, no tiene mucho sentido mantenerla, evitando costos de almacenamiento y costos asociados con posibles accesos remotos a dicho dato, como por ejemplo para garantizar su consistencia.

En [ZA98] el protocolo está basado en el desempeño del sistema a través de las operaciones de lectura y escritura. De este modo, el algoritmo va cambiando el esquema de

replicación, sujeto a que nunca debe reducirse el número de copias por debajo de un dado valor k.

Otras investigaciones como [WJ92, WJH97] presentan algoritmos de replicación adaptativos para mejorar la performance de los sistemas distribuidos ajustando el esquema de replicación de un objeto (es decir, el nro. de copias del objeto, y su localización) al actual patrón de acceso en la red. Los algoritmos continuamente mueven el esquema de replicación hacia uno óptimo.

El problema de este tipo de esquema son los mayores costos en tiempo de ejecución y el diseño complejo de los procesos. Por ejemplo, en [ZA98] utiliza tablas de replicación que cambian dinámicamente, al tiempo que el esquema de replicación se modifica. Para ello, cuando un nodo s decida crear una copia en otro nodo s' [2], se compromete a propagar sus escrituras hacia s'. Inversamente, cuando un nodo s decida eliminar una copia de un dato, informara al resto de los nodos de esta novedad. Debido a estos costos, hasta el momento han sido desarrollados pocos esquemas de replicación dinámica [TS].

2.4.1 Algoritmos centralizados vs. dinámicos

Generalmente en un esquema de replicación dinámica tenemos dos tipos de algoritmos [WJH97]:

Algoritmos centralizados. Cada procesador transmite periódicamente la información pertinente (normalmente estadísticas) hacia algún procesador predeterminado x, a su vez, x computa la función y ordena el cambio del esquema de replicación.

Algoritmos distribuidos. Cada procesador toma decisiones para cambiar localmente el esquema de replicación, basado en estadísticas locales. Un ejemplo de un cambio local al esquema de replicación sería: si un procesador x abandona su réplica, entonces indica a sus vecinos que escriben el objeto, que no deberían propagarle los cambios. Estos algoritmos tienen dos ventajas:

- Responden a cambios en el modelo de lectura-escritura de una manera más oportuna, ellos evitan el retraso involucrado en la colección de estadística, cómputo, y decisiones de brodcast o transmisión.

- Su sobrecarga es menor porque ellos eliminan los mensajes extras requeridos en el caso centralizado.

Un algoritmo ideal sería aquel que tiene un completo conocimiento de todas las demandas futuras de lectura-escritura, y su orden. Como sabemos los cambio de este modelo no pueden conocerse a priori, con lo cual este algoritmo es poco realista.

[2] El hecho de que los nodos ordenen a otro que cree una copia viola en algún sentido el principio de autonomía. Sin embargo, esta violación se ve compensada por un mejor desempeño del sistema [ZA98].

2.4.2 Ambientes móviles

En ambientes móviles, las máquinas desconectadas o pobremente conectadas dependen principalmente de los recursos locales. Puesto que los host móviles deben enfrentar una reducida disponibilidad, una medida apropiada para tener datos es almacenarlos localmente. Si el dato es compartido entre varios usuarios móviles o entre usuarios móviles y fijos, la replicación es a menudo la mejor opción. Pero, en un ambiente móvil, donde los usuarios no están en localizaciones fijas y los patrones de acceso observan cambios, los esquemas de replicación tradicionales dificultan la movilidad y causan problemas en operaciones móviles. Por lo tanto, la replicación en ambientes móviles requieren nuevos esquemas, entre los que se destacan aquellos que toman los beneficios de los esquemas dinámicos.

Es importante que las computadoras móviles (palmtops, notebook computer) accedan a sus datos minimizando la comunicación (esta claro que los usuarios que realizan cientos de accesos cada día, la comunicación inalámbrica puede ser muy costosa). Esto puede ser logrado mediante una apropiada localización de los datos. Por ejemplo, si un usuario frecuentemente lee un ítem de dato x, y x es actualizado con poca frecuencia, entonces es beneficioso para el usuario alocar una copia de x en su computadora móvil. En otras palabras, el usuario móvil se subscribe a recibir todas las actualizaciones de x. De esta manera el acceso de lectura es sobre la copia local, y no requiere comunicación. Las actualizaciones infrecuentes, son transmitidas de la base de datos "online" a la computadora móvil. En cambio, si el usuario lee x infrecuentemente comparado a la proporción de actualizaciones, entonces una copia de x no debería ser alocada en la computadora móvil. En cambio, el acceso debería ser hecho por demanda; todo requerimiento de lectura debe ser enviado a la computadora estacionaria que almacena la base de datos online.

En [HSW94] diseñaron un *método de alocación dinámica* donde las copias son creadas y destruidas en base al desempeño del sistema. El objetivo básico de este algoritmo es mantener una alta tasa de accesibilidad y un bajo tráfico en la red, entendiendo esto último como el mínimo acceso posible a copias remotas. En cada momento que un usuario móvil trata de leer un dato, son examinados sus últimos 'k' requerimientos. Si, en esos 'k' requerimientos, el número de lecturas es mayor que el número de escrituras y el usuario móvil no posee una copia del dato que trata de leer, entonces una copia del dato es alocada en el usuario móvil. Por otro lado, si el número de escrituras excede el número de lecturas y el usuario móvil tiene una copia del dato que esta intentando leer, entonces la copia es retirada del usuario móvil.

Resumiendo, la replicación es necesaria cuando un usuario móvil frecuentemente accede a datos que son poco actualizados. La alocación de una copia de tal dato a un usuario móvil puede reducir costos involucrados en tales transacciones.

2.5 Esquema ideal

Uno podría preguntarse ¿cual es el esquema ideal?. En cierto sentido, esto no puede responderse sin un amplio conocimiento del entorno en el cual funcionara el sistema. La elección de sitios y el grado de replicación dependen de los objetivos de rendimiento y disponibilidad para el sistema y de los tipos y secuencias de transacciones introducidas en cada sitio.

Por ejemplo, si se requiere de una alta disponibilidad y la mayoría de las transacciones son de obtención de datos, pudiéndose realizar en cualquier sitio, entonces una base de datos totalmente replicada sería una buena opción.

Por otro lado, si ciertas transacciones tiene acceso específico a sectores de la base de datos y se efectúan en un solo sitio, se podría asignar el conjunto de fragmentos correspondiente exclusivamente en ese sitio.

Los datos que se utilizan en múltiples sitios se pueden replicar en esos sitios. Si se efectúan muchas actualizaciones, puede ser conveniente limitar la replicación. Encontrar una solución óptima, o siquiera buena, es un problema de optimización muy complejo.

Lo que es muy claro, que las decisiones que tomemos tienen implicaciones cruciales para la performance del sistema, leer un dato localmente es más rápido y menos costoso que leerlo de un procesador remoto, y cualquier actualización de datos normalmente consiste en escribir todas las réplicas. Con lo cual debemos tener en cuenta que si estamos en un ambiente donde se realizan muchas lecturas la replicación debe ser ampliamente distribuida para aumentar el número de lecturas locales y disminuir la carga en un servidor central; en cambio si la mayoría de las operaciones son de escritura, debemos optar por un esquema mas estrecho, debido a que una distribución amplia reduciría la velocidad de cada escritura, y aumentaría su costo de comunicación [WJH97].

Como regla general se debe considerar que la replicación de fragmentos es de utilidad cuando el número de consultas de solo lectura es (mucho) mayor que el número de consultas para actualizaciones.

Podemos concluir que un esquema de replicación óptimo depende del patrón de acceso, es decir, el número de lecturas y escrituras emitidas por cada procesador.

Capítulo 3
Métodos de Propagación de Actualizaciones

Para clasificar los Métodos de Propagación de Actualizaciones, Gray en [GHOS96] a utilizado un parámetro basado en *cuando* tiene lugar la propagación de la actualización. La propagación de las actualizaciones puede ser realizada dentro o fuera de los límites de la transacción origen[1]. En el primer caso, tenemos los esquemas *Eager o Sincrónicos,* en el otro los esquemas *Lazy o Asincrónicos*. Wiesmann en [WPS+00a] los define como el momento en el cual el usuario es informado del resultado de la actualización, después (Eager) o antes (Lazy) que todas las réplicas hayan sido sincronizadas y actualizadas.

3.1 Esquemas Eager

3.1.1 Una definición

El *Esquema Eager o Sincrónicos* conserva todas las réplicas exactamente *sincronizadas* en todos los nodos, actualizándolas como parte de una transacción atómica, es decir, se aplican actualizaciones a todas las réplicas de un objeto dentro de los límites de la transacción original (ver Figura 3.1)

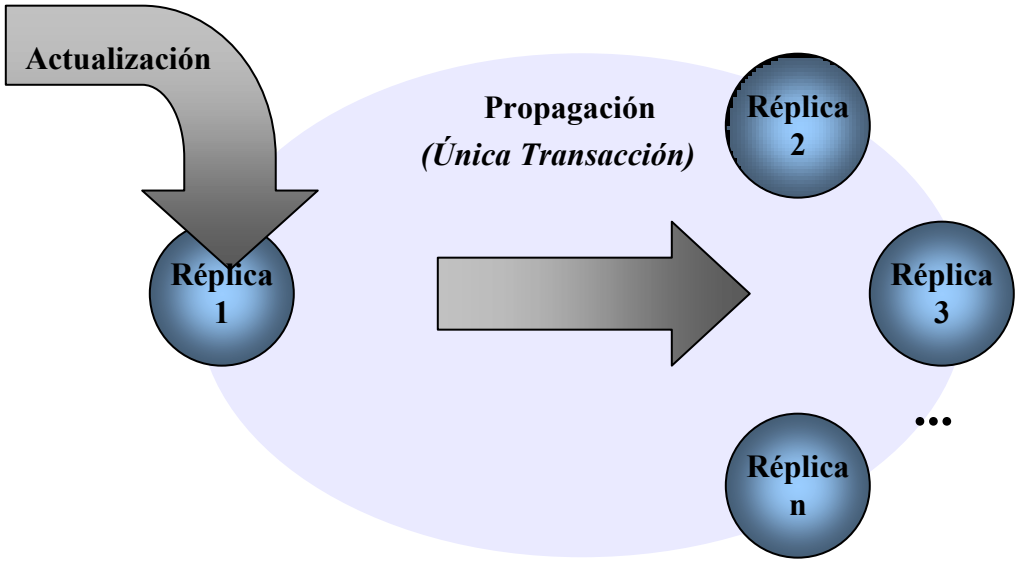


Figura 3.1: Replicación Eager

[1] Transacción que actualiza el ítem de dato local, a partir de la cual deben ser propagados los cambios.

3.1.2 Ventajas y costos

La *replicación sincrónica* nos proporciona las siguientes ventajas:

Serializabilidad. Debido a que todas las actualizaciones dentro del proceso de replicación ocurren como una unidad lógica de trabajo, las transacciones exhiben lo que comúnmente se llaman propiedades ACID, dándonos una ejecución serializable, sin ningún tipo de inconsistencias. Específicamente otorgan la propiedad de *Serializabilidad de Una Copia.*

Consistencia firme. Esta segunda ventaja se desprende de la anterior. En la misma, la actualización original se hace esperar hasta que todas las copias se actualicen, originando una transacción global. De esta manera, cuando una transacción comete, todos las copias tienen el mismo valor, logrando que la latencia antes que se logre la consistencia de los datos es cero.

Sin embargo, estas ventajas tienen los siguientes costos:

Peor performance. Al tener que sincronizar cada acceso a datos con las demás réplicas durante la ejecución de la transacción, lo convierten en un esquema muy caro, llevando consigo las siguientes implicaciones:

Mayores tiempos de respuestas. Esta sobrecarga de mensajes tiende a reducir la performance de actualización e incrementar los tiempos de respuestas transaccionales, no pudiendo darle una respuesta al usuario hasta que la actualización halla sido cometida por completo.

No son fácilmente escalables. Podría decirse, que la mayor desventaja de ellos, es que el número de operaciones por transacción aumenta con el grado de replicación (O(N) donde N es el número de nodos), y debido a que la probabilidad de deadlock, y por consiguiente, fallos en las transacciones suben muy rápidamente con el tamaño de la transacción y con el número de nodos, son inseguros para escalar mas allá de un pequeño número de sitios.

No son apropiados para ambientes móviles. Por un lado reducen la performance de las actualizaciones con sincronizaciones extras, y por el otro, en un ambiente móvil los nodos están normalmente desconectados. De aquí que una actualización sincronizada no sea válida para un ambiente móvil

3.1.3 Técnicas de propagación

En las diferentes implementaciones de técnicas basadas en el esquema Eager, han considerado principalmente dos tipos de protocolos dependiendo si usan *Bloqueos Distribuidos* o *Broadcast Atómicos* para ordenar las operaciones que están en conflicto.

3.1.3.1 Protocolos basados en Bloqueos Distribuídos

Los *protocolos de bloqueos* son usados en la mayoría de los DBMSs para implementar niveles de aislamiento, detectando y regulando la ejecución concurrente. La idea es asegurar que los datos que tienen operaciones en conflicto, sean accedidos por una operación en cada momento. Esto se logra asociando un "lock" con cada unidad de bloqueo. Este bloqueo es realizado antes de acceder al dato, y el liberado luego de usarlo. Obviamente un dato bloqueado no puede ser accedido por una operación si esta bloqueado por otra. Existen muchas variantes, pero todas comparten una característica fundamental, para ejecutar una operación de lectura o escritura sobre un ítem de dato, deben obtener sobre el mismo, un read-lock (compartido) o write-lock (exclusivo) correspondientemente [ÖV91, BG92].

Los protocolos basados en bloqueos distribuídos son aplicados en la mayoría de los protocolos de replicación tradicionales (ROWA, ROWAA [BHG87]). Estos algoritmos coordinan cada operación individualmente, usando el 2PL (Two Phase Locking Protocol) para ordenar las operaciones en conflicto, y el 2PC (Two Phase Commit Protocol) para garantizar atomicidad. Al usar bloqueos distribuidos, por cada operación de la transacción debe obtenerse los correspondientes locks sobre los datos. En el caso especifico de ROWA, cuando una transacción requiere una actualización, el sito origen debe adquirir write-locks (bloqueos exclusivos) por cada sitio que tiene la réplica. Cuando el sitio origen recibe los ACK de todos los sitios, permite que la transacción actualice el dato y proceda con la próxima operación. Al final de todas las operaciones se ejecuta un protocolo 2PC para que cometa o aborte la transacción en todos los sitios [HAE]. Adusumilli y Osborne integran el procotolo 2PL con 2PC, los resultados se muestran en [AO93].

Gray en [GHOS96] ha señalado que esos protocolos que coordinan cada operación individualmente, usando bloqueo distribuido y 2PC tienen significativos problemas de performance y escalabilidad. Por un lado el procesamiento de las réplicas puede ser bloqueado en el caso de fallas de nodos o de la red. Además, no es una solución escalable debido a la cantidad de mensajes requeridos entre el nodo que inicia la actualización y los demás que tienen una réplica del mismo. Supongamos por ejemplo que tenemos un sistema con n nodos y donde cada transacción consiste de m operaciones, una productividad de k transacciones por segundo requiere k.m.n mensajes por segundo. Tales aproximaciones nunca son escalables. Como resultado, cuando el número de nodos aumente, el tiempo de respuesta, la probabilidad de conflictos y la proporción de deadlock aumentan significativamente [OZS96, KA00].

Una clasificación de las técnicas de replicación tradicionales las podemos ver en [CP92, CHKS94].

3.1.3.2 Protocolos basados en Broadcast Atómicos

Entre las principales áreas que han estudiado la replicación tenemos, los Sistemas Distribuidos, generalmente para propósitos de tolerancia a fallas, y las Base de Datos por

razones de performance. Gran parte de la comunidad de sistemas distribuidos han desarrollados sistemas que otorgan primitivas de *Group Communication* para soportar aplicaciones distribuidas, garantizando en su mayoría las propiedades de atomicidad y preservación del orden. Entre éstos mecanismos utilizados, encontramos el Atomic Broadcast (ABCAST), que nos da atomicidad y orden total[2] [Fla96, PGS97, GS97].

Tanto en sistemas distribuidos como en base de datos, las técnicas y mecanismos utilizados son similares. Uno podría emitir un simple comando al sistema de comunicación Group para mandar mensajes a todos los sitios participantes, actualizando todas las réplicas de una vez y reduciendo los costos de actualización [WPS+00a].

Recientemente, algunos autores han estudiado con detenimiento en varios trabajos [Alo97, KA98, Dra98, HAE99] la posibilidad de implementar sistemas de bases de datos replicadas sincrónicamente usando primitivas Group Communication, con el fin de simplificar el movimiento de mensajes y la resolución de conflictos. Todos apuntan a poder realizar un esquema de replicación más eficiente y mejorar así la performance del sistema. Una clasificación de estas técnicas basadas en Broadcast Atómicos podemos encontrarla en [WPS99].

3.1.4 Trabajos relacionados

Las primeras investigaciones han apuntado a la replicación Eager debido a que otorgan consistencia firme [KA]. Sin embargo estos protocolos, comúnmente denominados tradicionales prácticamente no han sido usados, y a pesar de los resultados de Gray y su conclusión de que la replicación Eager no es práctica, en algunas investigaciones han empezado a valerse de diferentes técnicas para evitar las desventajas anteriormente mencionadas. Entre estas técnicas tenemos:

Primitivas de Group Communication. Mediante éstas primitivas un nodo N broadcast un mensaje a todos los demás nodos de un grupo para simplificar la carga de mensajes y los conflictos de actualización, garantizando las propiedades de orden total y atomicidad.

Writes Sets. En éste caso las operaciones de actualización de una transacción Ti son demoradas hasta que todas las lecturas de Ti hayan sido ejecutadas. Luego, el conjunto WSi con todas las operaciones de escritura es empaquetado en un mensaje para ser enviado a los demás nodos.

Copias Shadow. Mediante ésta técnica primero realizan la ejecución de las transacciones localmente sobre dichas copias privadas para chequear la consistencia y posponen la propagación al final de la transacción.

Todas las técnicas anteriormente mencionadas pueden ser combinadas con el fin de reducir la sobrecarga de los mensajes implicados en la actualización de réplicas y a su vez mejorar el perfil de las transacciones conflictivas [ABKW98, KA00].

[2] Esta propiedad de orden total nos asegura que si dos sitios reciben mensajes m y m', los recibirán en el mismo orden

En este contexto, para los sistemas comerciales que deben soportar ambientes distribuidos, móviles y desconectados, los esquemas de replicación sincrónica son considerados muy costosos y poco aplicables. Hay una fuerte creencia en los diseñadores de BD que tales soluciones no son factibles debido a los problemas de performance y escalabilidad. Los usuarios son a menudo advertidos del uso de replicación Eager cuando se requiere una sincronización total de las réplicas. Pocos sistemas comerciales han implementado este esquema. *Oracle Advanced Replication* provee un protocolo Eager en cual primero ejecuta una actualización localmente y luego usa Store Procedures activados por triggers para propagar sincrónicamente los cambios y a su vez bloquear las copias correspondientes [Ora, Syb].

3.1.5 Aplicación

La base necesaria para estos esquemas es tener un sistema con una infraestructura lo suficientemente robusta para tolerar la dependencia que tienen con la disponibilidad del sistema y de la red.

La replicación Eager debe utilizarse cuando la consistencia de datos se requiere en tiempo real para todas las réplicas. Si usamos replicación sincrónica, es importante conservar el número de recursos involucrados en un límite razonable [Bur97].

3.2 Replicación Lazy

3.2.1 Una definición

En los esquemas *Lazy o Asincrónicos,* la transacción origen comete localmente sin esperar por el cometido en los otros nodos, propagando *asincrónicamente* las actualizaciones de la réplica, típicamente como una transacción separada para cada nodo, denominadas *transacciones de refresh* [PSM98] (ver Figura 3.2).

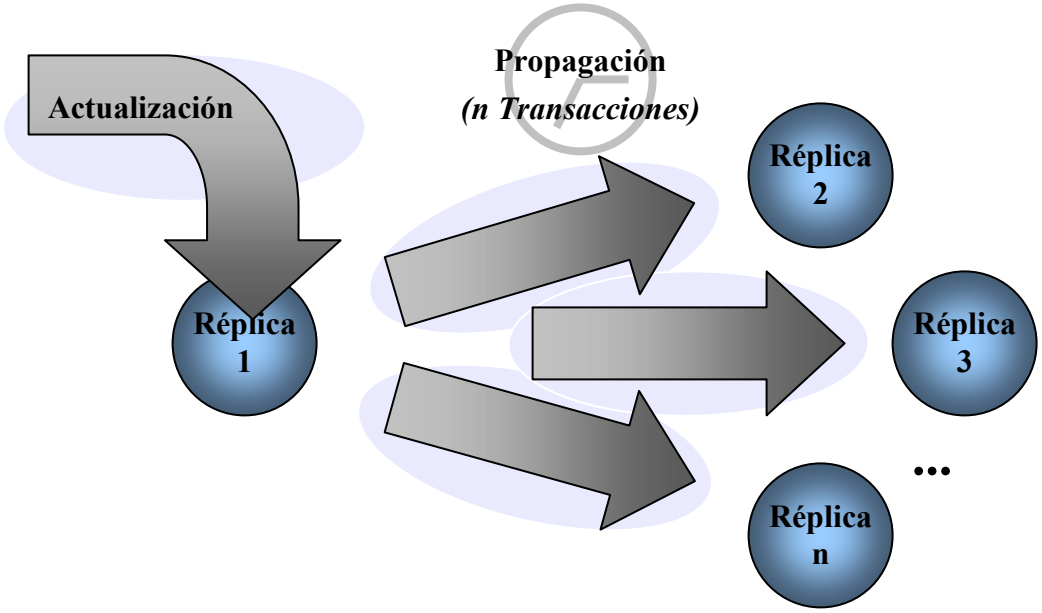


Figura 3.2: Replicación Lazy

3.2.2 Ventajas y costos

La *replicación asincrónica* proporciona las siguientes ventajas:

Mayor performance. El tamaño de las transacciones es reducido y son menores los bloqueos, mejorando la performance global del sistema. Esto lleva a dos puntuales mejoras:

Menores tiempos de respuesta. Algunos sistemas continuamente conectados utilizan esquemas Lazy para mejorar los tiempos de respuesta.

Mayor facilidad de escalabilidad. Tiene mucho menos deadlock que los sistemas Eager debido a que las transacciones son mas cortas. Sin embargo, cada actualización genera N-1 transacciones para actualizar las demás réplicas, esto lleva a que el número de transacciones concurrentes aumente en O(N) (donde N es el número de nodos)

Aptos para ambientes móviles. Estos esquemas dan una respuesta inmediata al usuario, los cambios de la propagación son realizados luego; este tipo de respuesta es de gran importancia dada la proliferación de aplicaciones para *usuarios móviles*, donde una copia no esta siempre conectada al resto del sistema y no tiene sentido esperar hasta las actualizaciones se hayan cometido para permitir al usuario ver los cambios. Con lo cual, los algoritmos basados en este esquema son una buena opción para aplicaciones móviles,

ajustándose a las necesidades de estos ambientes, donde la sincronización ocurre después que la transacción de actualización cometa [Shr].

Por otro lado, podemos apreciar los siguientes costos:

Consistencia Floja. Debido a que cada transacción ejecuta localmente e independientemente, el sistema no requiere protocolos de commit multisitio como el protocolo 2PC (Two-Phase Commit), los cuales tienden a introducir bloqueos y no son fácilmente escalables. Consecuentemente, este esquema hace más laxa la propiedad de consistencia mutua asegurada por 2PC y nos proporciona una consistencia más débil, en la cual la latencia antes de lograr la consistencia de los datos es siempre mayor que cero, el proceso de replicación ocurre asincrónicamente a la transacción origen. En otras palabras, hay siempre algún grado de retraso entre el tiempo de cometer la copia origen y el tiempo para disponerla en las demás réplicas, permitiendo que las copias puedan divergir y se produzcan posibles inconsistencias en los datos.

Acceso a datos viejos. Surge como consecuencia de la consistencia débil. Existe la posibilidad de que la propagación asincrónica puede causar que una transacción de actualización lea valores viejos de algún dato, resultando una ejecución que genera un estado inconsistente de la base de datos. Por ejemplo: supongamos tener dos cuentas bancarias C1 y C2 en dos sitios diferente S1 y S2. El banco requiere que la suma de las cuentas sea positiva. Las dos cuentas están replicadas en ambos sitios. Supongamos que dos personas P1 y P2 comparten esas cuentas con $ 300 en C1 y $ 700 en C2. P1 extrae $900 de la cuenta C1 desde el sitio S1 y, aproximadamente al mismo tiempo, P2 extrae $900 de la cuenta C2 en el sitio S2. Debido a la demora de propagación de los resultados, ambas transacciones pueden cometer. A pesar de que las transacciones fueron propagadas, ambas cantidades tienen un balance negativo, violando el requerimiento bancario (ver Figura 3.3).

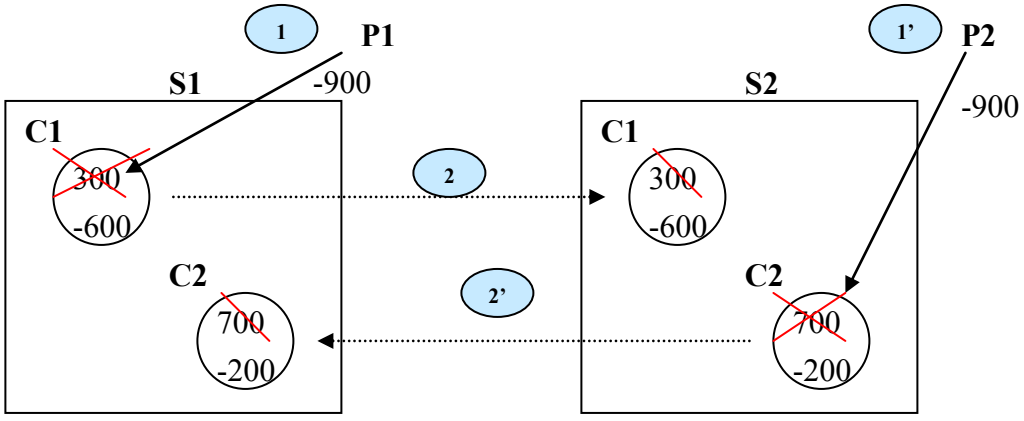


Figura 3.3: Acceso a datos viejos.

Como podemos observar los puntos en los cuales los esquemas Eager tienen como ventajas, estos esquemas lo tienen como desventaja y viceversa, notándose la gran diferencia en sus finalidades de conseguir consistencia o performance respectivamente.

3.2.3 Técnicas de propagación

En esta sección presentaremos técnicas de propagación asincrónicas basadas en diferentes criterios: técnicas según la *oportunidad*, la *granularidad* y la *semántica transaccional.*

3.2.3.1 Técnicas según la oportunidad

En la práctica, hay un intervalo de tiempo entre la ejecución de la transacción de actualización original y las correspondientes transacciones de refresh, pudiendo ser una demora significante debido al tiempo necesitado para propagar y ejecutar las mismas. Una chance es propagarlas inmediatamente después de cometida la transacción origen o también podría ser que el sistema use periodos "ociosos" o de bajo funcionamiento para realizar dicha tarea.

Debido a que estas demoras pueden afectar la performance, la propagación debe ser hecha de manera oportuna, teniendo las siguientes opciones [BK97b]:

Término temporal. Asegura que las actualizaciones son totalmente propagadas dentro de *t* unidades de tiempo desde que la transacción original haya cometido.

Punto de sincronización temporal o de tiempo fijo. Asegura en ciertos momentos, que todas las réplicas de un determinado ítem son idénticas. Ese momento puede ser por ejemplo una vez por hora.

Término basado en eventos. Asegura que las actualizaciones son totalmente propagadas antes que ocurran *n* eventos de tipo *x*, donde x puede ser una transacción de actualización y/o lectura sobre el ítem. Otro ejemplo sería antes que el 10 % de los datos sean antiguos.

Punto de sincronización basado en eventos. Asegura que después de ciertos eventos, todas las réplicas de un ítem dado son idénticas. Por ejemplo, basados en la frecuencia de acceso de los ítems, después que se realicen *n* accesos sobre la réplica *r*, se realiza la propagación.

Sincronización por demanda. Los efectos de sincronización de un determinado ítem están dados por el requerimiento de un sitio, comúnmente denominada Pull. IBM Data Propagator usa una estrategia pull en la cual las actualizaciones son propagadas solo cuando el cliente requiere [Bur97].

3.2.3.2 Técnicas según la granularidad

Estas estrategias nos definirán cuando las réplicas son actualizadas, y como son realizadas dependiendo de la granularidad de información que se utiliza en la propagación, pudiendo enviar transacciones cometidas o simplemente operaciones realizadas en la copia origen [PSM98]:

Propagación demorada. Es la comúnmente utilizada en esquemas Lazy [Gol95]; en la cual, las actualizaciones son propagadas hacia las otras réplicas solamente una vez que se haya cometido la transacción origen. En el mismo la granularidad de propagación generalmente son transacciones *Ti* (w1, w2, ..., wn, commit), pudiendo ser también un súper conjunto de ellas (ver Figura 3.4).

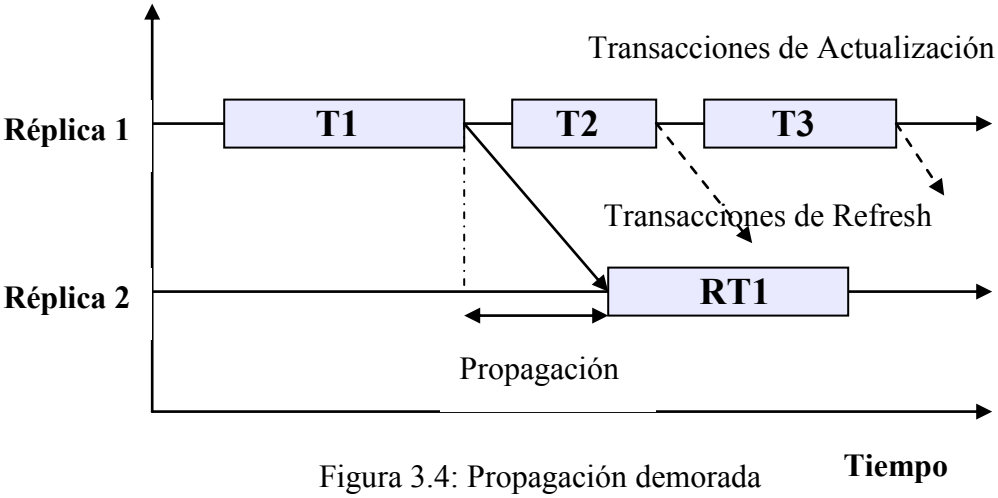


Figura 3.4: Propagación demorada

Propagación inmediata. Las actualizaciones de la copia origen son propagadas hacia las réplicas tan pronto como ellas son detectadas en el nodo origen sin esperar que cometa la transacción de actualización. En este caso la granularidad de propagación son operaciones individuales *wj* (ver Figura 3.5). A su vez podemos dividirla en dos, dependiendo del inicio de la transacción refresh:

Inmediata e inmediata. Una transacción de refresh comienza en la réplica tan pronto como la primer operación de actualización es recibida del nodo origen.

Inmediata y espera. La transacción refresh comienza en las réplicas después de recibir completamente todas las actualizaciones (de la misma transacción) del nodo origen.

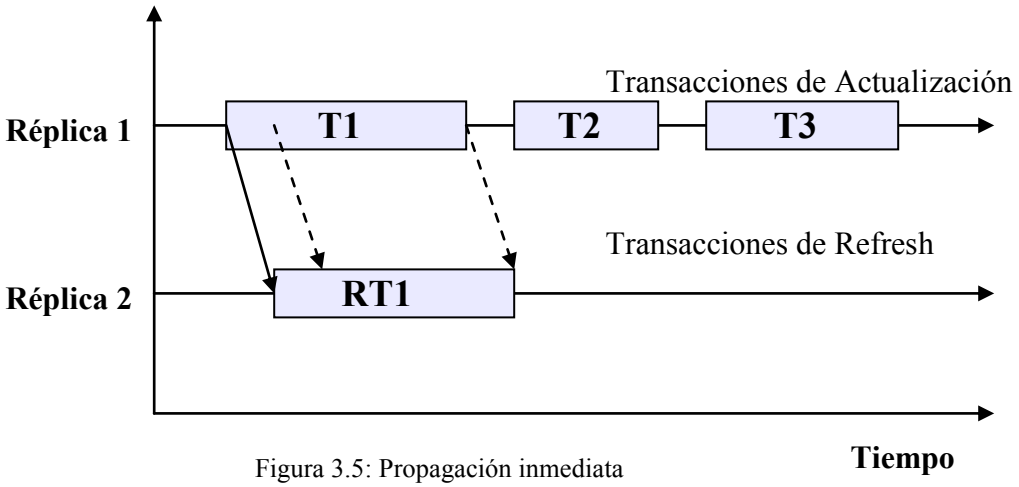


Figura 3.5: Propagación inmediata

3.2.3.3 Técnicas según la semántica transaccional

Conservar la *semántica transaccional* sobre las réplicas, significa que la unidad lógica de trabajo que fue ejecutada como la transacción original es también ejecutada en las demás réplicas, un concepto que se hace más importante a medida que la replicación asincrónica se acerca a tiempos reales de ejecución.

Otra manera de clasificar los métodos de propagación Lazy es dependiendo si conservan o no la semántica transaccional [Bur97]:

Transaccionales o propagación de eventos. Generalmente estos "eventos" que son propagados son transacciones o mensajes que representan las transacciones o algún subconjunto o súper conjunto de ellas. La característica más importante de estos modelos es la conservación de la semántica transaccional siendo transaccionalmente consistente con el sitio donde se origino la actualización (ver Figura 3.6). Por ejemplo, puede ser utilizada en la mayoría de los ambientes OLPT, donde la preservación de la semántica transaccional es critica.

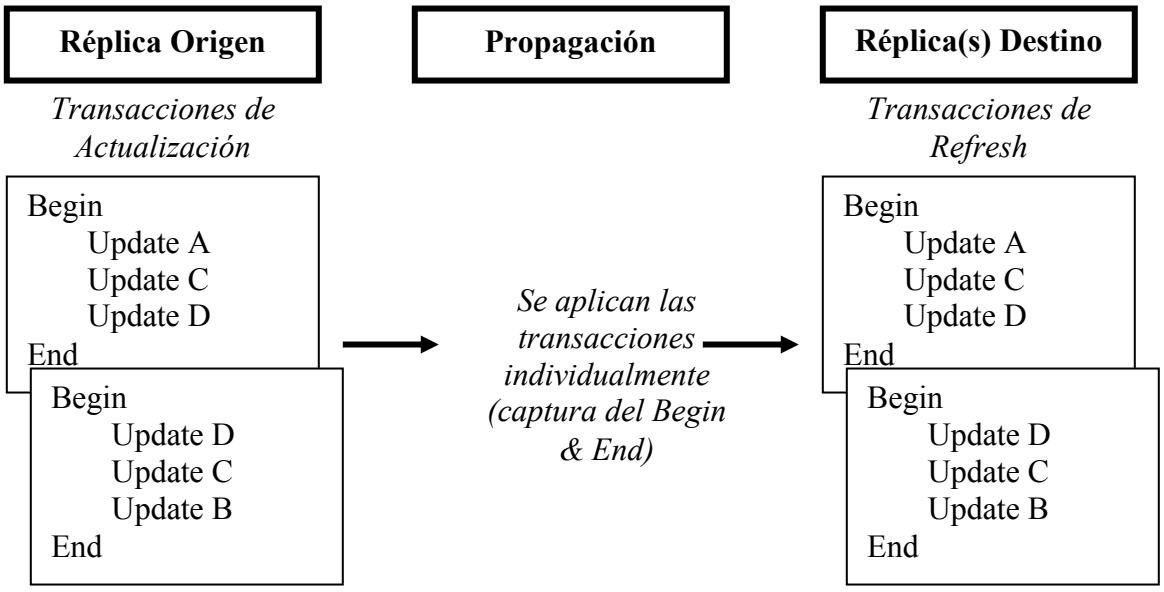


Figura 3.6: Propagación transaccional

Refresh. Esta técnica usa un proceso batch mediante el cual aplica los cambios a las réplicas, generalmente estando los usuarios fuera del sistema, siendo el sistema de replicación el único que tiene acceso a los datos (ver Figura 3.7).

Podemos tener dos tipos de refresh:

Completo. En este se extraen los datos considerados originales y son cargados en las diferentes réplicas, como su nombre lo indica se hace una "copia" completa del dato a replicar.

Incremental. Ocurre el mismo proceso, excepto que son propagados solamente aquellos cambios que han ocurrido desde el último refresh. Es más eficiente que el completo cuando hay pocos cambios debido a que se deben replicar menos datos.

Para los ambientes OLAP (DSS), donde la mayoría de las actualizaciones ocurren como parte de procesos batch, la tecnología de refresh puede ser una buena opción.

Oracle periódicamente puede refrescar los snapshot, esto lo realiza mediante una operación batch que hace que los snapshot reflejen el estado más actual de la copia origen. El usuario puede decidir como y cuando es apropiado realizar un refresh, indicando sobre cuales y en que intervalos de tiempo. Brindando los dos tipos de refreshes mencionados anteriormente [Ora].

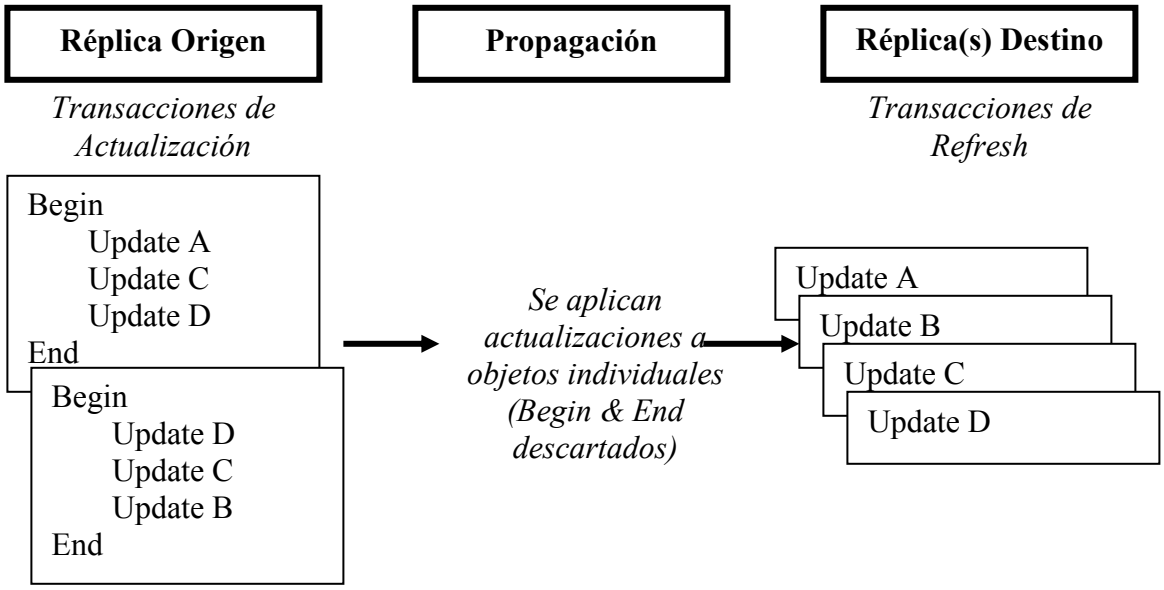


Figura 3.7: Propagación refresh

3.2.4 Trabajos relacionados

Es importante mencionar que a partir de los resultados publicados en [GHOS96] sobre los problemas de los esquemas Eager, muchas de las investigaciones fueron enfocadas hacia los esquemas Lazy [HAE , LLG90, PL91, BK97a].

A medida que la necesidad de replicación fue creciendo, esta técnica ha sido adoptada por muchos sistemas de base de datos comerciales debido a los beneficios de otorgar una mayor performance. Sin embargo, esas técnicas no son seguras, la mayoría no garantizan la consistencia y serializabilidad necesitada por la semántica transaccional, o imponen restricciones sobre la localización de datos y que datos pueden ser actualizados.

3.2.5 Aplicación

Estos esquemas son una buena opción cuando los requerimientos no demandan una consistencia fuerte. Tenemos que tener en cuenta que si la usamos y manejamos correctamente, podremos mejorar la disponibilidad de los datos y la performance de la aplicación. Si la usamos inapropiadamente, este puede corromper los datos hasta tal punto que la reparación sea compleja, si no es francamente imposible [Bur97].

Capítulo 4

Métodos de Regulación de Actualizaciones

Para clasificar los Métodos de Regulación de Actualizaciones, Gray en [GHOS96] a utilizado un parámetro basado en *quien* puede realizar las actualizaciones o mejor dicho que copias pueden ser actualizadas. Hay dos posibilidades: los esquemas *Master-Slave* (actualizaciones centralizadas) y los esquemas *Group* (actualizaciones distribuidas).

4.1 Esquemas Master-Slave

4.1.1 Una definición

La idea es designar una copia determinada de cada ítem de dato como copia master, distinguida o también denominada copia primaria, las demás copias son copias slave o secundarias. El nodo que almacena la copia primaria de un dato es llamado master para este dato, mientras que los nodos que almacenan copias secundarias son llamados slaves.

El nodo master del dato, es la responsable de realizar las actualizaciones sobre este dato. Si un nodo slave desea modificarlo, el requerimiento de actualización debe enviárselo al nodo master, donde se hace el cambio. Cuando una copia primaria es actualizada, el nuevo valor debe ser propagado a todas las copias secundarias (ver Figura 4.1).

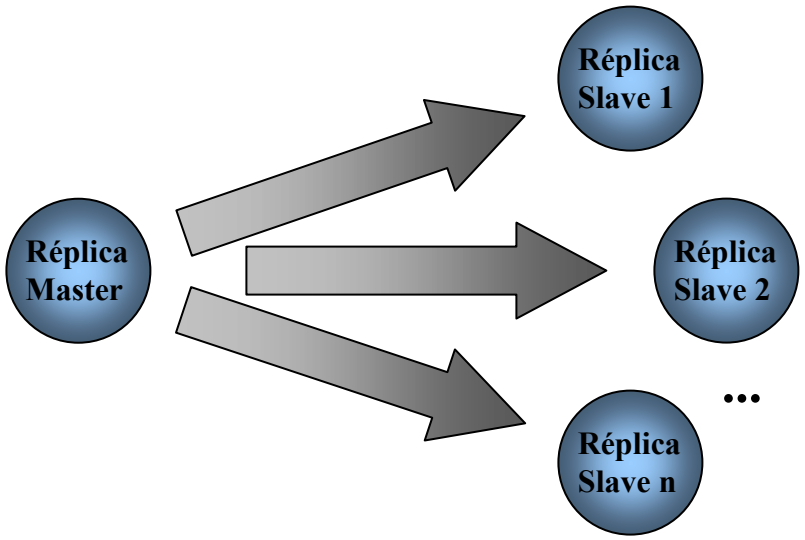


Figura 4.1: Esquema Master-Slave

4.1.2 Ventajas y costos

La principal ventaja de este modelo es la simplicidad:

Simplicidad. Todos aquellos protocolos que se basan en este esquema tienen el beneficio de una implementación relativamente fácil. Esto se debe a la simplicidad del modelo que permite que las actualizaciones sean aceptadas solo en un lugar, logrando que las mismas fluyan de una sola manera. También evita actualizaciones concurrentes a diferentes copias y simplifica el control de la concurrencia [KA].

Esta ventaja trae consigo los siguientes costos:

Funcionalidad limitada. Un modelo simple implica, normalmente, limitar la funcionalidad: una réplica slave es de lectura y sólo pueden realizarse modificaciones a la copia primaria, debiendo sincronizar directamente con el nodo master. Esto puede ser un inconveniente para situaciones donde los usuarios requieren tener la habilidad de actualizar directamente sus datos locales como es el caso de ambiente móviles.

Diferenciación de réplicas. Demostrar ser más simple no significa que el sistema en conjunto puede hacerse más complicado por la presencia de dos clases de réplicas. Se deben manejar de una manera diferente los cambios en datos de nodos master, de los cambios sobre copias slave. Además del retraso que puede significar esto, puede también hacer a los usuarios agudamente conscientes de su estado como un nodo secundario.

Disponibilidad. La disponibilidad de los datos puede verse afectada por:

Cuello de botella. Todos los pedidos de actualización están asociados al sitio que contiene la copia primaria, con lo cual logramos que se genere un cuello de botella

Punto simple de falla. Si un nodo master no puede ser accedido (falla del nodo o link hacia el mismo), las copias master no pueden aceptar actualizaciones hasta la recuperación de los mismos, introduciendo un único punto de falla. Dicha recuperación puede consistir desde la simple espera de su disponibilidad o la búsqueda de un nuevo propietario del dato como veremos más adelante. Esto afecta directamente la disponibilidad y hace un esquema difícil de usar en aplicaciones móviles [GHOS96].

Escalabilidad. La escalabilidad de un sistema replicado esta determinado por el Método de Regulación de Actualizaciones [Sai00]. A medida que aumentan las réplicas, aumenta la demora de la propagación de las actualizaciones y a veces crea cargas desequilibradas entre las réplicas, especialmente en los esquemas Master-Slave en los cuales el master es el responsable de propagar las actualizaciones a las otras réplicas. Estos esquemas experimentan O(N) conflictos de actualización (N es el número de réplicas). Una solución es conectar las réplicas en una estructura de árbol, localizando el master en la

root, y las actualizaciones se van propagando hacia abajo desde la raíz. Esto acorta el retraso de propagación de O(N) a O(log N) (N es el número de réplicas) y reduce la carga sobre el master a un nivel constante.

4.1.3 Variaciones del modelo

En esta sección daremos una breve descripción de diferentes métodos basados en este modelo, pero que de alguna manera difieren en la manipulación de los datos. Entre ellos tenemos: *sitio primario, con respaldo, copia primaria y con migración del propietario*.

4.1.3.1 Sitio primario

En la técnica de sitio primario, todas las copias distinguidas se guardan en el mismo sitio, denominado sitio coordinador para todos los elementos de la BD. También conocido como modelo Master-Slave con primario no fragmentado, debido a que todas las actualizaciones se realizan en un solo sitio (el primario o master) y los cambios son subsecuentemente propagados a una o mas réplicas (slave). Una réplica puede estar situada local o remotamente con respecto a la copia master, y hacer referencia a todos o un subconjunto de los cambios que ocurren en el sitio primario (ver Figura 4.2).

Es el método más simple de implementar, pero tiene la gran desventaja que todas las solicitudes se envían a un mismo sitio, con lo cual probablemente se sobrecargue y origine un cuello de botella del sistema. También, un fallo del sitio primario paralizaría el sistema.

Una aplicación simple de este modelo es para propósitos de validación, en la cual se replica un almacenamiento de datos centralizado en diferentes nodos para que aplicaciones distribuidas puedan realizar las validaciones teniendo solamente acceso de lectura sobre los nodos slave.

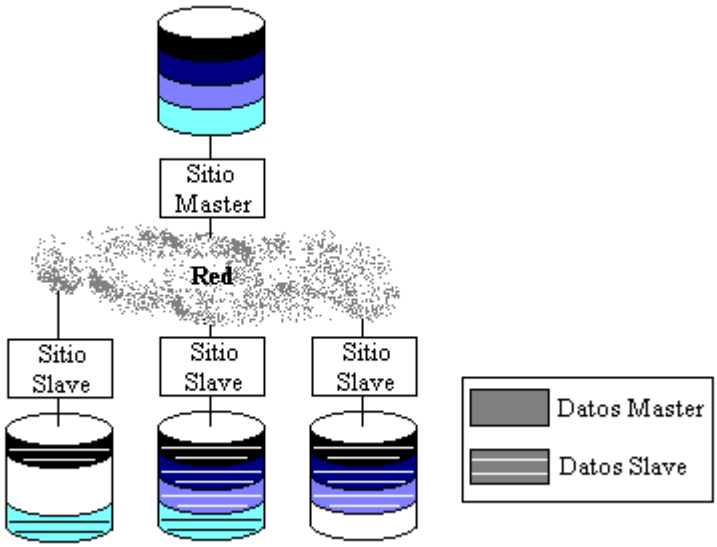


Figura 4.2: Método del sitio primario

4.1.3.2 Sitio con respaldo

Se designa un segundo sitio como sitio de respaldo, con el fin de ser mas tolerantes a las fallas, pudiendo cambiar a este respaldo si ocurre algún problema en el sitio primario (ver Figura 4.3).

En los métodos que usan sitios de respaldo, el procesamiento de transacciones se suspende mientras el sitio de respaldo se designa como nuevo sitio primario

La desventaja que tiene es la sobrecarga que implica mantener actualizado el sitio respaldo, quedando latente el problema de que los sitios se sobrecarguen.

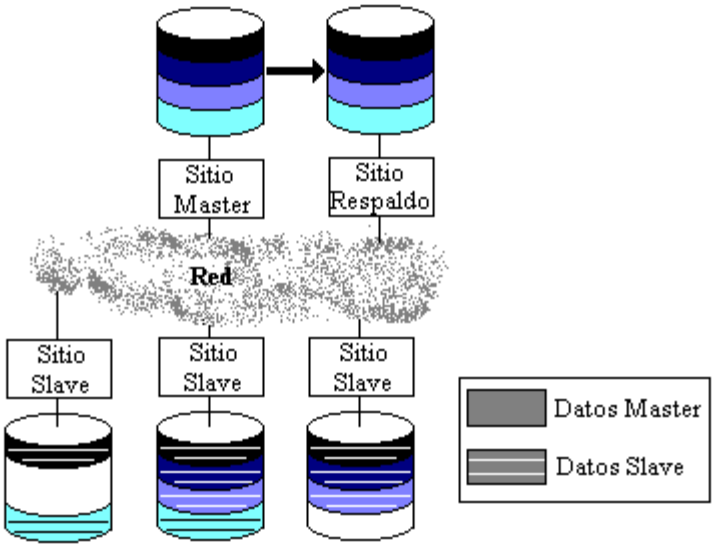


Figura 4.3: Método de sitio con respaldo

4.1.3.3 Copia primaria

Este método intenta distribuir la carga entre varios sitios, las copias maestras de cada ítem de dato pueden almacenarse en diferentes sitios. Cada sitio no solo actúa como master para un conjunto particular de datos, debido a que también puede tener réplicas de otros sitios para los cuales es slave de dichos datos (ver Figura 4.4).

Un sitio que incluye una copia distinguida de un elemento de información actúa básicamente como sitio coordinador para el control de concurrencia de ese elemento. El fallo de este sitio sólo afecta los pedidos hacia el y no un paro del sistema como puede ocurrir en el de sitio primario.

Generalmente este modelo se usa donde las actualizaciones se realizan sobre una porción de datos almacenados localmente de los cuales se es master, pero a su vez el procesamiento de queries requiere una vista total.

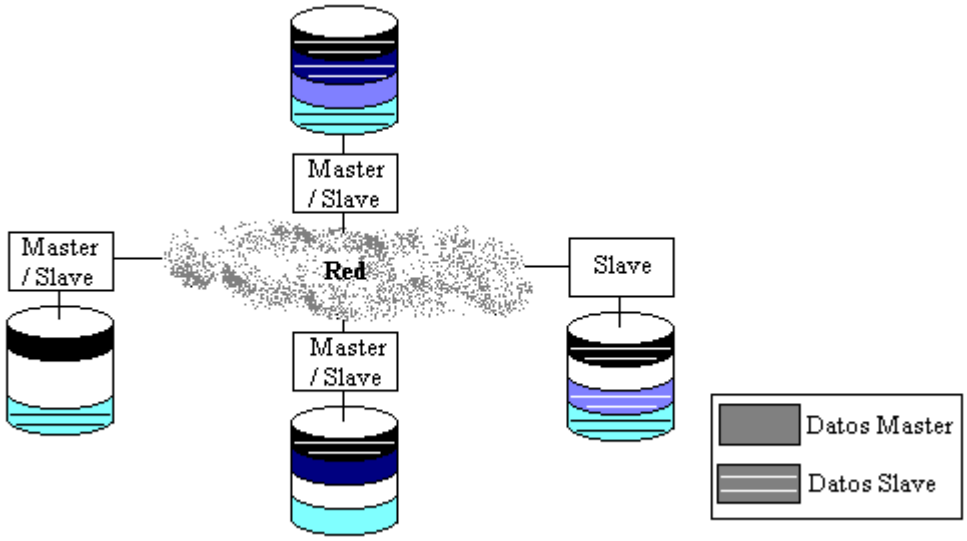


Figura 4.4: Método de copia primaria

4.1.3.4 Con migración del propietario

Otro enfoque es el método de migración del propietario, el cual consisten en migrar el master del dato sobre diferentes réplicas a partir de un determinado criterio, basado frecuentemente en un factor de tiempo, por Ej. con una frecuencia de días (ver Figura 4.5). La propiedad dinámica hace que este modelo sea menos restrictivo que el de sitio primario, la capacidad para actualizar una réplica de datos va moviéndose de sitio en sitio, asegurando que en cada momento haya sólo un sitio master para un determinado dato. Para esto es necesario tener información adicional para identificar el actual propietario del dato, lo cual implica un mayor costo de almacenamiento y mantenimiento debido a que debe ser replicada en todos los nodos [Bur97, Ora].

Una aplicación de este modelo puede ser la implementación de un Modelo Workflow (se caracteriza por ir cambiando dinámicamente el propietario del dato). Es importante tener en cuenta que cuando se usa la replicación para implementar un Workflow, el dato cambia de estado, no cambia de localización. Por ejemplo, aplicaciones departamentales pueden leer el código de estado de un pedido para determinar cuando pueden o no, actualizar su número de orden.

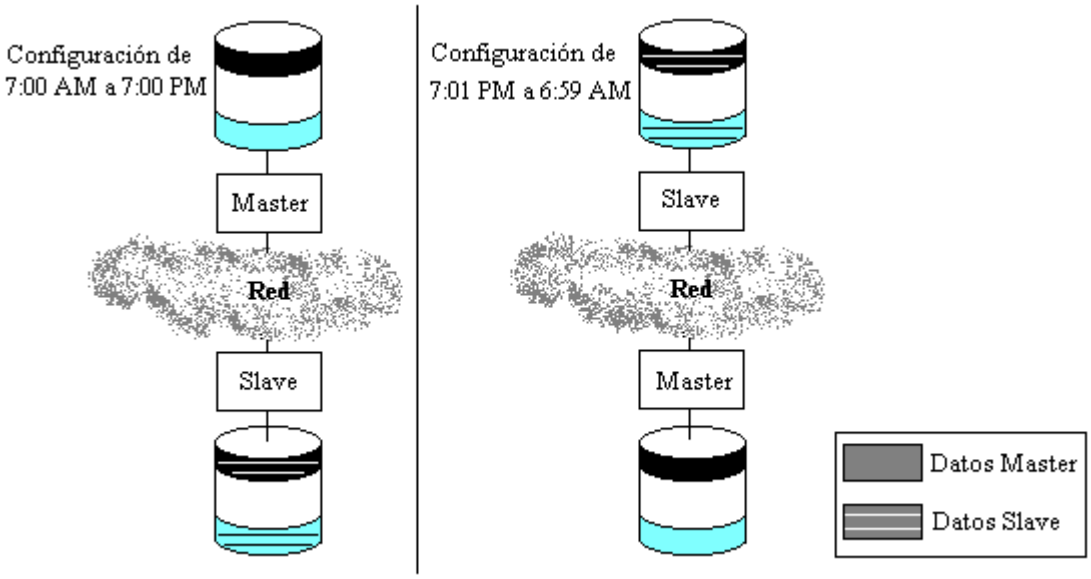


Figura 4.5: Método de con migración del propietario

4.1.4 Trabajos relacionados

Debido a que hay solo un sitio (master) que ejecuta transacciones de actualización, ha sido ampliamente usado en trabajos como [PKL, PSM98, BKR+99], en su mayoría para minimizar los conflictos entre las transacciones ejecutadas sobre datos replicados. Ampliaremos este punto en el próximo capítulo.

4.1.5 Aplicación

La mayoría de los protocolos usados en la práctica son técnicas de copia primaria [WPS+00b]. Una de las principales utilidades de este modelo es la distribución de información como puede ser el caso de una simple distribución de precios de productos de una cadena de negocios donde la Casa Matriz (master) puede actualizar y propagar los precios a las Sucursales (slaves) para que estas puedan solo leerlos.

Sin embargo, tales aplicaciones varían de acuerdo al método de propagación usado, como veremos en el siguiente capítulo.

4.2 Esquemas Group

4.2.1 Una definición

Los esquemas Group o comúnmente llamados Update-Everywhere o Multi-Master [Sai00], permiten que cualquier nodo pueda realizar actualizaciones sobre sus réplicas locales, logrando tanto acceso de lectura como de escritura sobre los mismos. Cuando una copia es actualizada, el nuevo valor debe ser propagado a las demás (ver Figura 4.6).

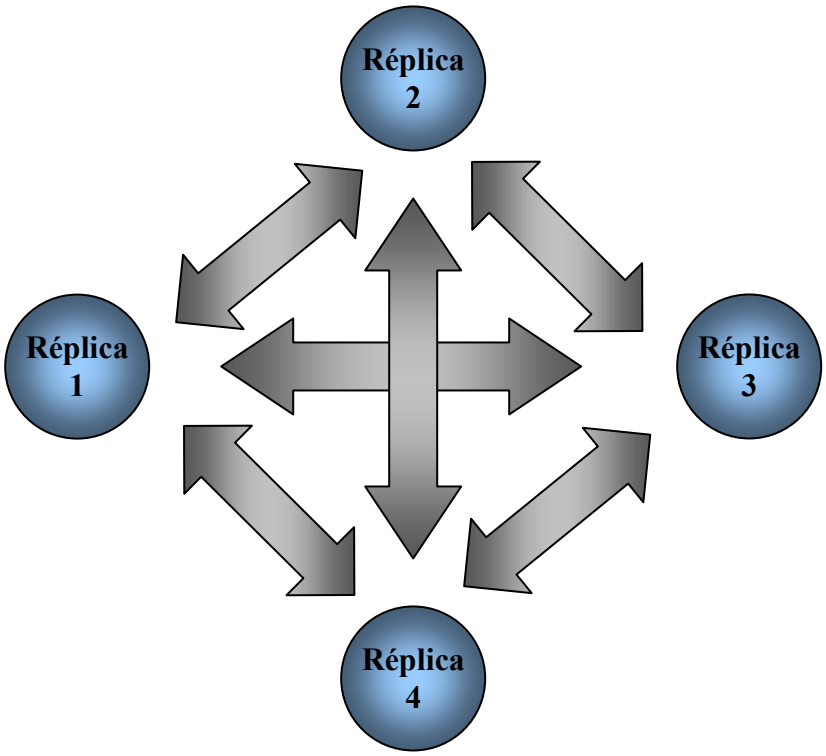


Figura 4.6: Esquema Group

4.2.2 Ventajas y costos

Las ventajas que tiene este esquema mejoran los costos asociados con el modelo Master-Slave, entre ellas tenemos:

Igualdad de réplicas. La propiedad y la actualización de las réplicas es compartida por cualquiera de las réplicas. No hay concepto de master o sitio primario, todas las réplicas son igualas, sin ningún derecho, propiedad o tratamiento especial.

Mayor disponibilidad. Surge de la posibilidad que cualquier réplica puede sincronizar con cualquier otra réplica, y cualquier actualización pueda aplicarse a cualquier réplica accesible. Debido a esta propiedad, son más fáciles de tratar con fallas debido a que no necesitan un protocolo de elección para continuar procesando. Similarmente, en principio, podría decirse que no introducen cuellos de botella, otorgando esquemas más robustos a las fallas y facilitando la distribución de las cargas en los diferentes sitios.

Mayor funcionalidad. Se logra una mayor funcionalidad en la habilidad para proporcionar una comunicación más rica y robusta, que permite a cualquier par de réplicas comunicarse e intercambiar actualizaciones, haciéndo un modelo estrictamente más poderoso.

Mayor flexibilidad. Quita restricciones de tiempo de diseño en las que uno se pregunta dónde y que operaciones pueden realizar los usuarios, y cualquier distinción innecesaria entre los usuarios de nodos centrales contra los nodos remotos.

Sin embargo, este modelo tiene asociado dos costos importantes:

Escalabilidad. En estos esquemas las actualizaciones pueden arribar concurrentemente a dos copias diferentes del mismo ítem (lo cual no sucede en copia primaria), pudiendo llevar a un *conflicto de actualizaciones*. Gray [GHOS96] sostiene que los algoritmos de replicación basados en este esquema no pueden soportar muchas réplicas porque ellos experimentan $O(N^2)$ de conflictos de actualización, llevando a un notable incremento en los conflictos de actualización a medida que aumenta el grado de replicación. Esto significa que al menos que haya una significante cantidad de operaciones de lectura (son locales) en la carga total, el sistema puede no ser escalable a medida que se agregan nuevos nodos .

Diseño. Si uno no es cuidadoso con el diseño, puede afectar la performance mucho más que los esquemas primarios. Al tener la posibilidad que todos los sitios actualicen datos, puede suceder que se produzca una sobrecarga en cualquier lugar, originada por el desequilibrio de las cargas.

4.2.3 Trabajos relacionados

Basados en los esquemas Update-Everywhere se han realizado investigaciones que van desde los tradicionales protocolos ROWA o ROWAA [BHG87], hasta trabajos más recientes tales como [LCC, KA98, HAE99, KA00]. Ampliaremos este punto en el próximo capítulo.

4.2.4 Aplicación

Este modelo es útil para aplicaciones que requieren múltiples puntos de acceso a la información con el propósito de distribuir la carga de los datos entre los diferentes nodos, asegurando alta disponibilidad y proporcionando acceso local a los datos.

Como en esquemas Master-Slave, su aplicación concreta depende del método de propagación utilizado (será tratada en el Capítulo 5).

Capítulo 5

Combinaciones entre Esquemas

En este capítulo veremos la clasificación de los protocolos de replicación en base de datos usando los dos parámetros definidos por Gray en [GHOS96]. Para esto combinamos los diferentes métodos de propagación y regulación de actualizaciones, resultando la taxonomía ilustrada en la Figura 5.1.

cuando / donde	**Lazy**	**Eager**
Master Slave	N transacciones 1 propietario (Sybase / IBM / Oracle)	1 transacción 1 propietario (Ingres)
Group	N transacciones N propietarios (Oracle Advanced Replication)	1 transacción N propietarios (ROWA / ROWAA / Quorums / Oracle Synchr. Repl.)

Figura 5.1: Combinación entre esquemas

5.1 Descripción de un Framework como modelo funcional.

El modelo funcional que utilizaremos para comparar las diferentes combinaciones de esquemas fue definido por Wiesmann en [WPS+00a]. El mismo permite describir un protocolo de replicación mediante una secuencia de cinco fases (ver Figura 5.2), de los cuales alguno puede ser omitido, ordenados de diferente manera, con iteraciones sobre alguno de ellos, o unidos en una simple secuencia. De esta manera podemos compararlos según ellos implementan cada una de esas fases y como las combinan.

Las fases son:

1. **Fase del Requerimiento** En esta fase el usuario requiere al sistema una operación sobre alguna réplica.

2. **Fase de Coordinación de los Sitios** Cada uno de los sitios coordina con los demás para sincronizar la ejecución de la operación, es decir, ordenar las operaciones concurrentes. En esta fase, las diferentes réplicas tratan de encontrar un orden en el cual las operaciones necesitan ser realizadas. Esto es muy importante debido al rol que juega la semántica de las operaciones en la replicación de base de datos: una operación que solo lee un dato no es lo mismo que una operación que lo modifica ya que las dependencias de datos introducidas no son las mismas en ambos casos. Mediante esta coordinación logramos que todas las operaciones tengan las mismas dependencia de datos en todos los sitios.

3. **Fase de la ejecución** Representa la ejecución actual de la operación, la aplicación de las actualizaciones a las demás copias pueden llegar a ser realizadas en la fase de Coordinación del Acuerdo. Esta fase permite observar como cada esquema trata y distribuye sus operaciones.

4. **Fase de la Coordinación del Acuerdo** los diferentes sitios llegan a un acuerdo del resultado de la ejecución (garantizar atomicidad). Como veremos mas adelante, esta fase plantea alguna de las fundamentales diferencias entre los protocolos.

5. **Fase de la Respuesta** Representa el momento en cual el usuario recibe una respuesta del sistema sobre la operación realizada. Esta fase depende del método de propagación, si usamos propagación sincrónica la respuesta será enviada después que todas las actualizaciones hayan sido realizadas, en cambio con la asincrónica, la respuesta es inmediata, dejando para después la propagación de los cambios y coordinación sobre las réplicas.

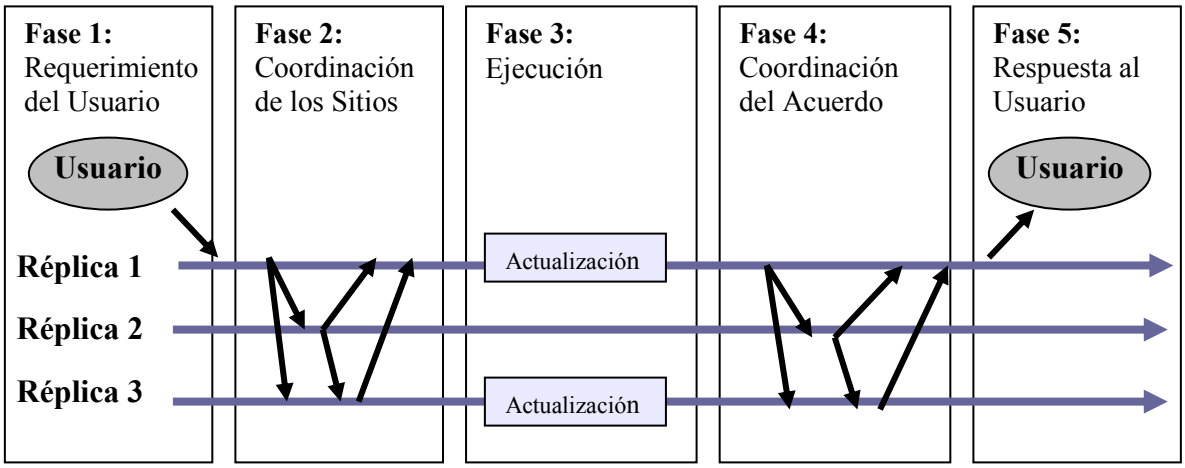


Figura 5.2: Fases para la descripción de un protocolo de replicación

5.2 Esquemas Lazy Master-Slave

5.2.1 Descripción

Como en cualquier esquema Master-Slave, tenemos un sitio o nodo master para cada ítem de datos. Las actualizaciones son primero realizadas por el master y luego propagadas a las demás réplicas. Las lecturas pueden ser realizadas sobre cualquier sitio, si se realizan sobre copias master siempre verán el valor correcto del ítem, en cambio, si las realizan sobre copias slave pueden leer valores viejos. Esto se debe a la naturaleza de la replicación asincrónica. Puesto que las copias slave son solo de lectura, existe un grado de latencia antes que se logre la consistencia de los mismos con el master.

5.2.2 Fases del protocolo en términos del modelo funcional

Las fases involucradas en este protocolo son los siguientes (ver Figura 5.3):

1. Requerimiento del Usuario: este requerimiento es enviado al sitio primario.

-- La transacción comienza en el sitio primario --

2. Coordinación de los sitios: desaparece ya que la ejecución hace lugar solo en el primario.

3. ejecución: se ejecuta la operación local.

-- La transacción termina en el sitio primario --

4. Respuesta al Usuario.

5. Coordinación del acuerdo: esta fase es relativamente sencilla ya que las réplicas secundarias necesitan solo aplicar los cambios como el primario los propaga. Esto se debe a que la necesidad de coordinación y ordenación entre transacciones sucede en el sitio primario.

En la mayoría de los protocolos basados en propagación asincrónica, las actualizaciones no se propagan hasta que cometa la transacción origen, enviando todas las actualizaciones como una unidad (en algunos trabajos como [PSM98] se propagan las operaciones individualmente para que las demás réplicas vayan realizando algún trabajo). De esta manera no hacemos diferencia si la transacción tiene una o más operaciones.

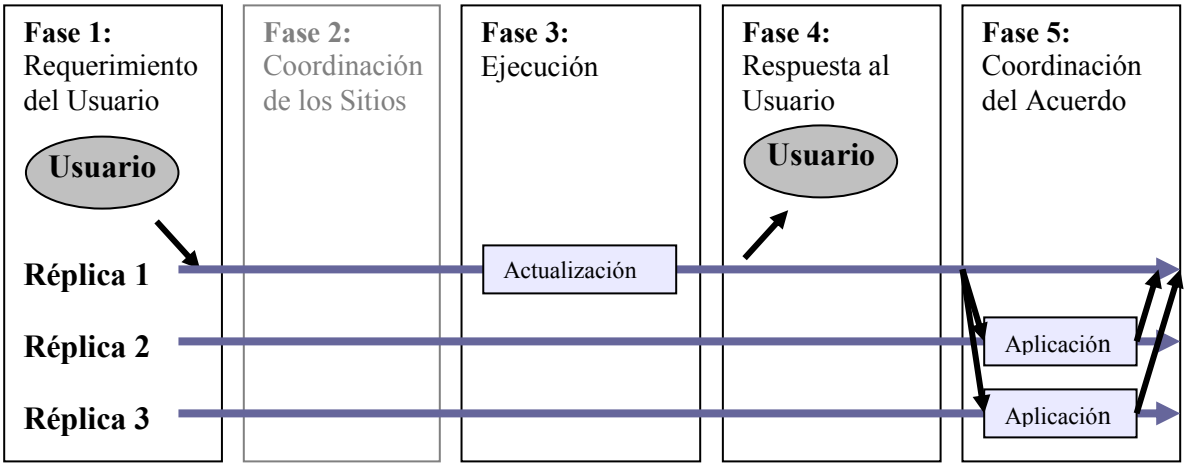


Figura 5.3: Esquema Lazy Master-Slave

5.2.3 Trabajos relacionados

Muchos de los protocolos existentes para base de datos ignoran la complejidad y sobrecarga introducida por las comunicaciones, haciendo por eso muchas soluciones teóricas pero no aplicables a la realidad [Alo97]. Por esta razón y como consecuencia de los problemas de deadlock y performance que pueden aparear los esquemas Eager y Group, la mayoría de las soluciones de replicación en base de datos son asincrónicas y basadas en Master-Slave [BK97a, BKR+99].

Entre los primeros protocolos diseñados encontramos el ASAP (As Soon As Possible) RBP (Remote Backup Procedure), RDF (Remote Duplicate Database Facility), la mayoría de ellos con propósitos de backup [CHKS94].

Mas allá que este acercamiento pueda criticarse, algunos de los argumentos contra las soluciones tradicionales están justificados, sobre todo del punto de vista de productos comerciales donde generalmente la performance toma precedencia ante cualquier otra consideración. Por ejemplo, en Sybase Replication Server las actualizaciones son propagadas a los otras copias inmediatamente después que comete la transacción. (Esta es una estrategia push [1] en un esfuerzo de minimizar el tiempo que las copias son inconsistentes). Oracle brinda esquemas Lazy Master-Slave, utilizando snapshots para ser modificados localmente y más tarde sean enviados a las otras réplicas. Para reducir la proporción de abortos, pueden ser aceptadas lecturas viejas, violando consecuentemente la Serializabilidad de Una Copia [Ora, Syb].

Gray en [GHOS96] implementa un protocolo Lazy Master-Slave, garantizando serializabilidad. Para esto asume que cada DBMS local usa un protocolo 2PL estricto[2]. Cada

[1] En estos sistemas comerciales se refieren a dos estrategias de propagación: push cuando son propagadas inmediatamente, y pull cuando son propagadas por demanda.
[2] El 2PL estricto exige que se posean todos los bloqueos hasta que se cometa la transacción.

operación de lectura debe requerir un bloqueo de lectura sobre el sitio primario de cada elemento que desea leer, evitando con esto la lectura de valores viejos. Las transacciones de actualización son realizadas por el master, bloqueando cada una de las réplicas que debe actualizar. De esta manera sincroniza los conflictos de lectura/escritura(escritura/lectura). En cambio para sincronizar las operaciones de escritura sobre las copias secundarias (conflictos escritura/escritura) utiliza la Thomas Write Rule (TWR)[3] [BHG87].

Trabajos mas recientes, han realizado soluciones que garantizan serializabilidad con el costo de poner restricciones sobre como son seleccionadas las copias primarias [CRR96]. Por ejemplo, en [BK97a, ABKW98] ponen la restricción que una transacción T puede actualizar un ítem de dato d solo si T es originada en el sitio primario de d. Ellos dicen que a pesar que estas restricciones limiten significativamente el conjunto de datos que pueden ser actualizados por una transacción, pueden ser útiles en aplicaciones donde por lo general existe el concepto de "propietario" para cada ítem, adhiriéndose a esta restricción, como por ejemplo una aplicación de stock de precios. Estas soluciones otorgan mejores tiempos de respuesta (no se intercambian mensajes durante la ejecución de una transacción) y garantizan serializabilidad. Sin embargo, el conjunto de posibles configuraciones y ejecuciones de transacciones son severamente restringidas [KA00].

5.2.4 Aplicación

Este esquema de replicación es ampliamente utilizado en aplicaciones tales como data warehouses, almacenamiento de datos operacionales y aplicaciones financieras [PSM98]. También se usa para sistemas tolerantes a las fallas como son las soluciones Stand-By o mecanismos de backup. En este, un sitio primario ejecuta todas las operaciones y un sitio secundario esta listo para tomar el lugar del primario en caso que este falle. Específicamente denominadas Cold-Standby o Warm-Stanby, debido a que los cambios al backup no son aplicados inmediatamente cuando los cambios llegan al sitio primario (Hot-Standby) [WPS+00b].

Una de las principales utilidades de este modelo es la distribución de información. Esta distribución puede ser desde una simple distribución de precios de productos, hasta Sistemas de Soporte de Decisión (DDS). En el primer caso, una cadena de negocios puede distribuir sus precios desde una casa matriz (master) que los actualiza y propaga a las sucursales (slaves). En el segundo caso, la combinación del esquema Master-Slave con propagación asincrónica los hace muy útiles para los DSS, replicando datos entre ambientes OLPT (online transaction processing) y OLAP (online analytical processing) . En estos sistemas uno puede necesitar que antes de llegar las propagaciones al ambiente OLAP se realicen algún tipo de transformación sobre los datos, hecho posibilitado por la propagación asincrónica [Bur97].

[3] Esta regla se refiere al caso donde la transacción T actualiza d pero el timestamp de T es menor que el timestamp de escritura de d. En lugar de abortar T, T continua como si la escritura hubiese sucedido, sin embargo, en realidad, la escritura es ignorada.

5.3 Esquemas Lazy Group

5.3.1 Descripción

Como cualquier esquema Lazy, cuando la transacción origen comete, una transacción es enviada a cada uno de los otros nodos para aplicar las actualizaciones a las réplicas, la diferencia esta en que al ser Group, permite que cualquier nodo actualice cualquier dato local, de esta manera, cualquier nodo podrá realizar tanto operaciones de lectura como de escritura sobre cualquier réplica que posea en su localidad.

Al tener una propagación asincrónica, como en los esquemas Lazy Master-Slave, puede suceder que las lecturas lean valores viejos. En cuanto a las actualizaciones surge un nuevo problema debido a los conflictos que pueden suceder entre ellas, como veremos en la sección 5.3.3.

5.3.2 Fases del protocolo en términos del modelo funcional

Los cinco Fases son: (ver Figura 5.4):

1. Requerimiento del usuario: el requerimiento es realizado en su sitio local.

-- La transacción comienza en el sitio local --

2. Coordinación de los sitios: no existe debido a que la coordinación hace lugar solo después de realizada la actualización.

3. ejecución: se ejecuta la operación local.

-- La transacción termina en el sitio local --

4. Respuesta al Usuario.

5. Coordinación del acuerdo: Durante esta fase, las copias son llevadas a un estado consistente mediante la propagación de todos los cambios y la decisión sobre como aplicarlos. Debido a la posibilidad que se produzcan conflictos de actualización, en esta etapa debe utilizarse un mecanismo de Reconciliación para que las réplicas logren ese estado de consistencia. En la sección 5.3.3. discutiremos estos conceptos.

Como dijimos anteriormente, el hecho que una transacción tenga una o más operación no afecta el modelo funcional para los protocolos Lazy.

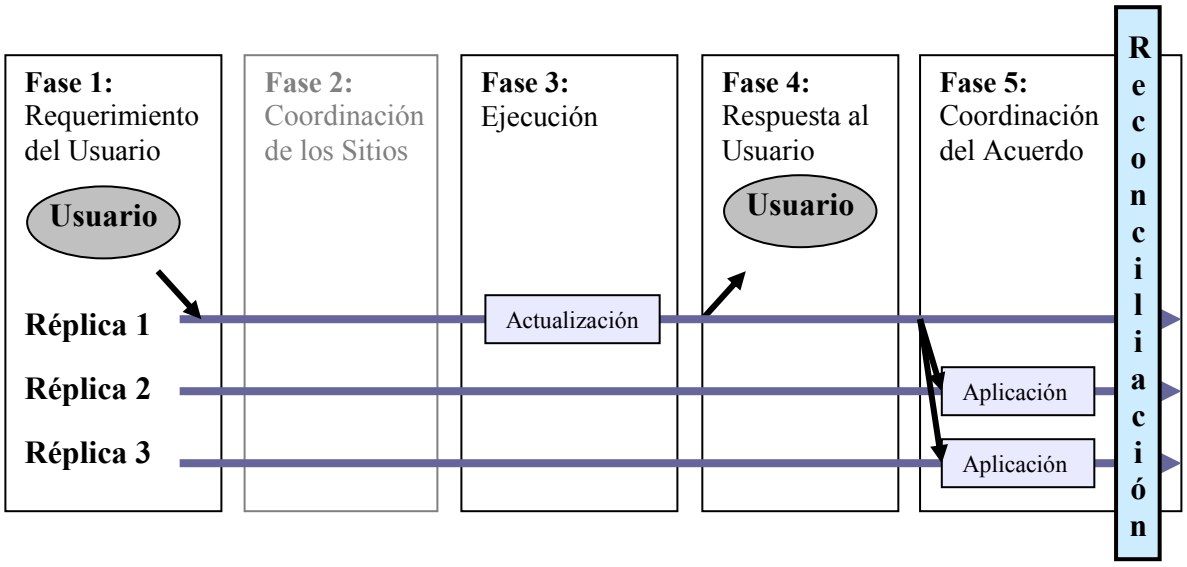


Figura 5.4: Esquema Lazy Group

5.3.3 Conflictos de actualización y su tratamiento

Conflictos de actualización

Esta combinación de esquemas, crea la posibilidad de que dos o más transacciones cometidas que operan sobre distintas réplicas de un dato, tengan *conflictos de actualización* sobre dicho ítem. Esto puede verse claramente en el caso que dos nodos actualizan el mismo objeto casi en el mismo momento y cada uno "corre" para realizar sus actualizaciones a los demás nodos. Por Ej. T1 debe actualizar el ítem d usando las réplicas en el sitio S1 mientras T2 actualiza la réplica d en S2. Asumimos que ambas transacciones cometen. El mecanismo de replicación debe descubrir esta situación y reconciliar ambas transacciones como veremos mas adelante.

Estos conflictos se deben a que cuando usamos este modelo, no hay aislamiento[4] (isolation) de las transacciones desde una perspectiva global. Cada transacción individual en su punto de ejecución lo hace con aislamiento; sin embargo, las transacciones que ejecutan en paralelo están en conflicto sin ninguna garantía que una transacción use el estado más actual de la base antes de realizar la actualización (las copias sobre los diferentes sitios no solo pueden ser viejas sino inconsistentes).

Reconciliación

Este hecho que posibilita que dos réplicas que independientemente aceptan actualizaciones al mismo dato y tengan diferentes valores, necesita un *mecanismo de reconciliación* para asegurar que todas las réplicas concuerden en los contenidos de sus datos. Las transacciones que esperarían bajo las necesidades Eager, aquí se reconciliaran [Ham].

[4] Isolation: la "I" de las propiedades ACID. Significa que cada transacción ejecuta como si no hubiese ninguna otra transacción concurrente.

Las actualizaciones locales siempre son inmediatamente visibles, sin embargo, las colisiones de actualización se descubren solo cuando las actualizaciones son propagadas.

Estos conflictos pueden ser reparados manualmente (tarea delegada a un administrador o simplemente a los usuarios), o mediante transacciones de compensación generadas por el sistema, que deshagan las transacciones cometidas que necesitan reconciliación, para mantener la consistencia de los datos sobre todas las réplicas [KA00].

El uso de transacciones compensatorias significa que a veces las actualizaciones de una transacción no son *durables*[5]. La perdida de durabilidad crea un efecto en el cual un usuario puede tomar una decisión sobre información que es subsecuentemente la perdedora en la resolución de un conflicto, es decir, las transacciones que son posteriormente deshechas, son vistas por otras transacciones antes de que el proceso undo ocurra. Dependiendo del grado de consistencia deseado, puede suceder que las transacciones secundarias también requieran acciones compensatorias para asegurar la consistencia de la base, y así sucesivamente [Bur97].

Este modelo presenta el desafío de incorporar un mecanismo para detectar los conflictos de actualización dentro del ambiente asincrónico y generar un acercamiento para la resolución de conflictos de actualización de datos.

Detección de conflictos → Mecanismo de resolución→ Consistencia de los datos

En el caso de base de datos comerciales han sido desarrolladas reglas de reconciliación para la resolución de conflictos. Además, hay paradigmas de meta-reglas para especificar que reglas pueden aplicar y en que orden si más de una regla es aplicable. Usando esas dos técnicas debería ser posible crear un procedimiento automatizado de resolución de conflictos [AES97]. La mayoría de los esquemas de reconciliación existentes rompen el concepto de transacción al estar basados en objetos (Ej: fila de un RDBMS). Esto es suficiente para el caso donde una transacción consiste de una operación, sin embargo, no lo son para las transacciones que consisten de mas operaciones sobre diferentes objetos de datos.

5.3.4 Trabajos relacionados

Muchas investigaciones se han basado en este modelo [CHKS, HAE, SAS+96, PGS97] de las cuales en gran parte ha sido de interés usarlo como una solución para soportar usuarios distribuidos, móviles y desconectados [BI, Shr, HSW94, LC98]. Estos usuarios pueden trabajar con las réplicas mientras están desconectados y sincronizarse mas tarde.

Los sistemas comerciales tomaron la ventaja que las transacciones pueden cometerse localmente, ejecutándose libremente y otorgando mayor performance, pero a su vez comprometiendo la integridad de los datos. Existen algunas aplicaciones que han introducido librerías de rutinas que automáticamente emiten transacciones compensatorias, pero el problema es poder identificar todas las situaciones de potenciales conflictos y tener disponible una rutina apropiada. En particular, Oracle trabaja como en esquemas Lazy

[5] Durable: la "D" de las propiedades ACID. Significa que los efectos de una transacción cometida sobreviven a las fallas.

Master-Slave con la diferencia que provee 12 reglas de reconciliación para fusionar conflictos de actualización sobre réplicas de datos [Ora].

5.3.5 Aplicación

Es muy utilizado en aplicaciones móviles, donde el procesamiento debe realizarse usando componentes generalmente desconectados. La idea básica es permitir a los usuarios leer o actualizar los datos mientras están desconectados (Ej. sobre datos replicados en computadoras portátiles), reconciliando las modificaciones con las demás réplicas cuando se reconectan (Ej. Lotus Notes).

Buretta en [Bur97] aconseja usar este modelo donde sea aceptable cierto grado de inconsistencias sobre los datos como por ejemplo en sistemas de reservación distribuidos. En estas aplicaciones, tener una alta disponibilidad normalmente es la preocupación más grande. También es aplicable para aplicaciones de solo inserción.

Si la decisión es aplicarlo, tratar de llevar el número de conflictos que pueden ocurrir a un mínimo. Usarlo solo en sistemas que tienen pocas proporciones de actualizaciones e implementar esquemas de distribución de datos que minimicen el número de réplicas actualizables.

5.4 Esquemas Eager Master-Slave

5.4.1 Descripción

Como mencionamos en los esquemas Lazy Master-Slave, al utilizar un esquema Master-Slave, una operación de actualización debe ser realizada primero en la copia master y luego ser propagada a las demás copias slave. La diferencia radica en que el sitio master debe esperar la confirmación que las copias secundarias han sido actualizadas, generando la actualización de una manera atómica, propiedad necesaria de la propagación sincrónica. Las transacciones de lectura pueden ser realizadas sobre cualquier sitio y siempre verán la ultima versión de cada objeto.

5.4.2 Fases del protocolo en términos del modelo funcional

La Figura 5.5 muestra los siguientes Fases:

1. Requerimiento del Usuario: este requerimiento es enviado al sitio primario.

-- La transacción comienza en el sitio primario --

2. Coordinación de los sitios: debido a que la ejecución tiene lugar solo en el primario, esta fase desaparece. El ordenamiento de operaciones en conflicto esta determinado por el sitio primario y debe ser obedecido por las copias secundarias.

3. Ejecución: se ejecuta la operación.

4. Coordinación de acuerdo: Una vez realizada la propagación de la actualización a las copias secundarias, se ejecuta un Protocolo de Cometido Atómico (Atomic Commitment Protocol) como puede ser el Two-Phase Commitment Protocol (2PC)[6]. A diferencia de los esquemas Lazy Master-Slave, mediante este protocolo podemos garantizar que si el primario falla, todas las transacciones activas serán abortadas.

-- Cada sitio termina la transacción de acuerdo al 2PC --

5. Respuesta al Usuario: esta respuesta es dada finalizado el 2PC.

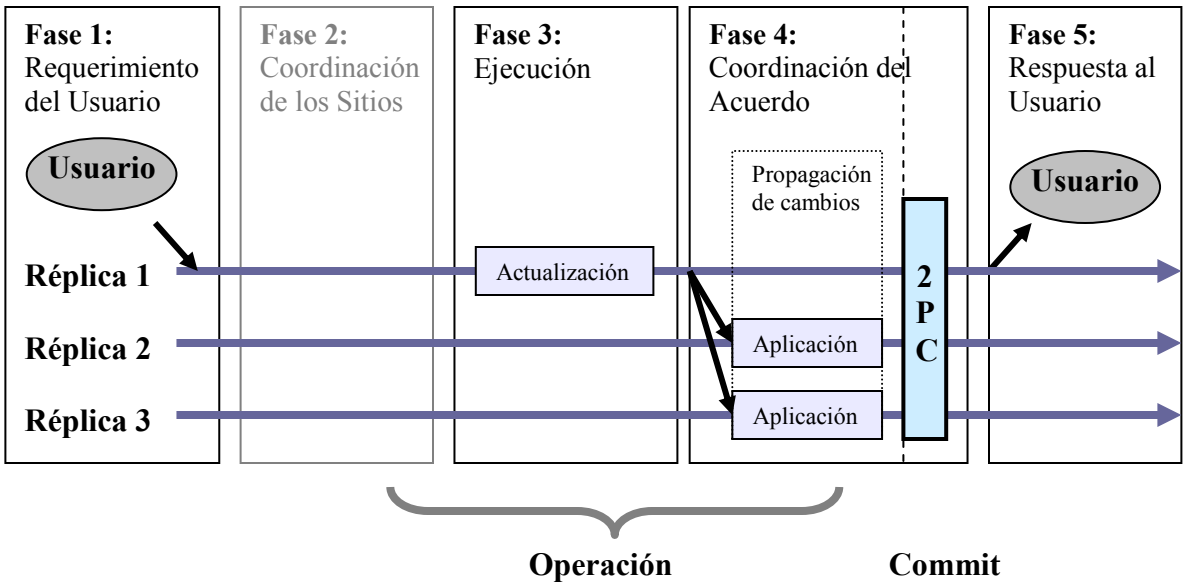


Figura 5.5: Esquema Eager Master-Slave

Si tenemos mas de una operación por transacción se hace un loop en el que interviene la fase 3 para ejecutar la operación en el master, y la fase 4 para propagar los cambios a las copias slave. Una vez ejecutada cada una de las operaciones en este loop, durante la Coordinación del acuerdo se inicia el 2PC para garantizar que todos los sitios cometen la transacción.

[6] Cuando la transacción T originada en el sitio S1 completa su ejecución, - es decir, cuando todos los sitios en que T se ha ejecutado informan al Coordinador de Transacciones de S1, que T se ha completado -, el Coordinador de Transacciones de S1 inicia el protocolo 2PC.

5.4.3 Trabajos relacionados

Entre los primeros protocolos encontramos el método de Copia Primaria, del Sitio Primario y esquemas basados Token[7] [Jin, CHKS94] entre otros; básicamente compuestos de un protocolo de bloqueo distribuido con un protocolo que garantice atomicidad.

INGRES distribuido fue una de las primeras soluciones que usaron este modelo [Sto79], utilizando el método de copia primaria. Gray en [GHOS96] expresa que teniendo un master para cada objeto ayuda a la replicación Eager a evitar deadlock, reduciéndolos tanto como en un sistema de un solo sitio.

5.4.4 Aplicación

Actualmente, son usadas solamente en tolerancia a fallas para implementar soluciones Hot-Standby, donde bajo una operación normal, los requerimientos de los usuarios son tratados por el sitio primario, enviando sus modificaciones de manera inmediata a la copia secundaria o Backup. Debido a la propagación sincrónica, el backup es una exacta réplica del primario, pudiendo tomar su lugar inmediatamente después de una falla.

5.5 Esquemas Eager Group

5.5.1 Descripción

En estos esquemas la regulación de las actualizaciones está basada en el modelo Group, con lo cual tanto lecturas como actualizaciones pueden realizarse localmente. Además, la propagación es sincrónica, con lo cual cada operación es sincronizada con los demás nodos, evitando mediante bloqueos o espera que se produzcan los conflictos de actualización mencionados en el esquema Lazy Group. Esto también evita la lectura de valores viejos, retornando siempre el valor actual.

5.5.2 Fases del protocolo en términos del modelo funcional

En este caso veremos el modelo funcional diferenciando las dos técnicas para propagar las actualizaciones sincrónicamente que vimos en la sección 3.1.3. En la misma vimos que podíamos utilizar protocolos basados en Bloqueos Distribuidos (Ej.: 2PL con 2PC) o basados en Broadcast Atómicos (Ej.: ABCAST)

[7] En estos esquemas, cada ítem de dato tiene asociado un token, permitiendo al portador acceder a dicho dato.

5.5.2.1 Replicación con protocolos basados en Bloqueos Distribuídos

Las fases del protocolo en términos del modelo funcional son (ver Figura 5.6):

1. Requerimiento del usuario: el requerimiento es realizado en el sitio local.

-- La transacción comienza en el sitio --

2. Coordinación de los sitios: en esta fase se utiliza un Protocolo de Bloqueo, tal como el Two Phase Locked[8] (2PL) [ÖV91] mediante el cual se manda un requerimiento de bloqueo a todos los demás sitios que pueden otorgan o no el bloqueo. Si el bloqueo es concedido por todos lo sitios, entonces se puede proceder. Si no, la transacción puede ser demorada y el requerimiento puede ser repetido mas tarde.

3. ejecución: la operación es ejecutada en todos los sitios una vez que todos los bloqueos fueron otorgados.

4. Coordinación del acuerdo: se utiliza un Protocolo de Cometido Atómico (Ej.: 2PC) [BG92] para cometer o abortar la transacción en todos los sitios.

-- Cada sitio termina la transacción de acuerdo al 2PC --

5. Respuesta al Usuario: esta respuesta es dada finalizado el 2PC.

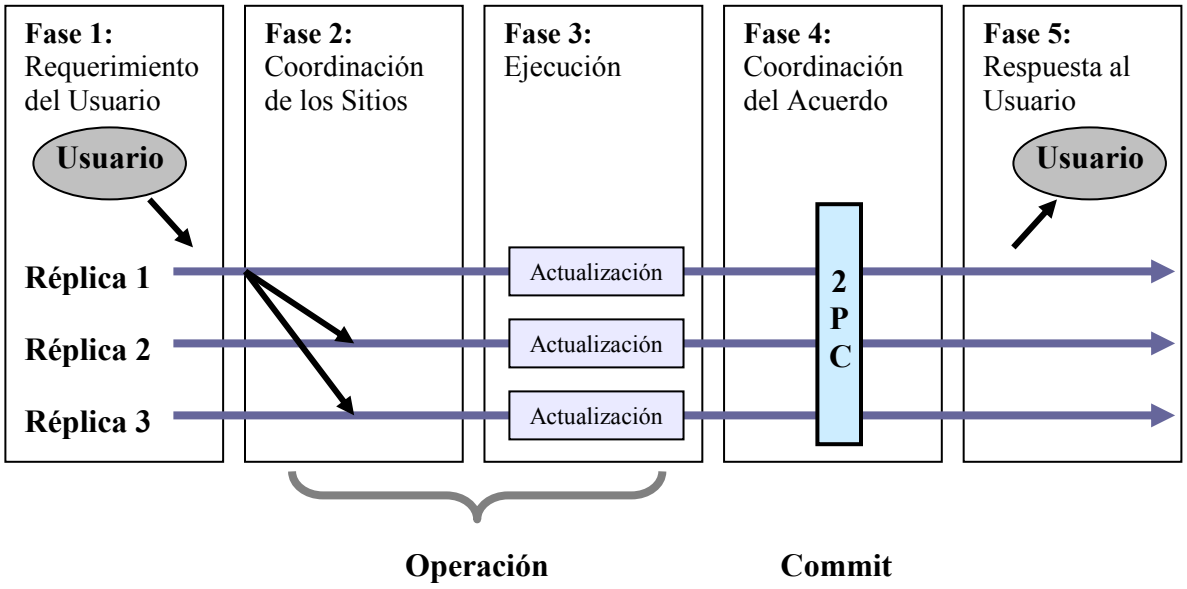


Figura 5.6: Esquema Eager Group con Bloqueos Distribuidos

En el caso que una transacción tenga mas de una operación, se repiten las fases 2 y 3 para cada una de las operaciones, hasta finalizar la transacción. En este loop una operación es sincronizada con las demás réplicas mediante los bloqueos y luego ejecutada en esas réplicas.

[8] Cuando usamos bloqueo distribuido, una réplica puede solo ser accedida después que ha sido bloqueada en todos los sitios.

Una vez realizado el loop para todas las operaciones, al final de la Coordinación del acuerdo, se inicia un 2PC para garantizar atomicidad.

5.5.2.2 Replicación con protocolos basados en Broadcast Atómicos

Para transacciones con una operación, el control de replicación es el siguiente (ver Figura 5.7):

1. Requerimiento del Usuario: el requerimiento es realizado en el sitio local.

-- La transacción comienza en el sitio local --

2. Coordinación de los sitios: el sitio broadcast el requerimiento a todos los demás sitios con los cuales coordina usando el orden dado por el Broadcast Atómico.

3. ejecución: todos los sitios ejecutan la operación. En el caso que dos operaciones entren en conflicto, son ejecutadas en el orden del broadcast atómico.

4. Coordinación del acuerdo: no es necesaria esta fase debido a la propiedad de atomicidad que nos brinda el Broadcast Atómico.

-- Cada sitio termina de la misma manera

5. Respuesta al Usuario.

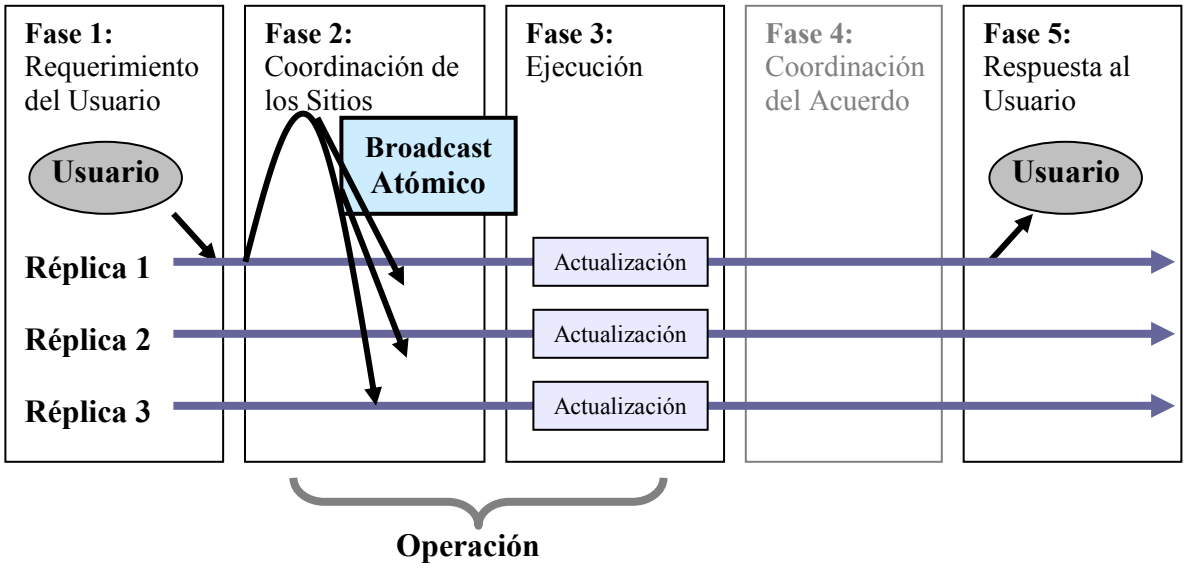


Figura 5.7: Esquema Eager Group basados en Broadcast Atómicos

Cuando usamos transacciones con mas de una operación, no tiene mucho sentido usar el ABCAST para mandar cada operación de una transacción separadamente ya que el orden total resultante del ABCAST no es suficiente para garantizar la serializabilidad necesaria

[WPS+00a]. Para lograr esto, algunas investigaciones [PGS97, KA98] han usado copias shadow en un sitio para realizar las operaciones y luego, cuando la transacción es completada[9], mandar todos los cambios en un simple mensaje, pero antes de cometer, las transacciones tienen que ser certificadas. El procedimiento de certificación chequea si la transacción viola la consistencia de los datos, en que casos deben ser abortadas, o cometidas. Las fases en términos del modelo funcional son (ver Figura 5.8):

1. Requerimiento del Usuario: el requerimiento es realizado en el sitio local.

-- La transacción comienza en el sitio local --

2. Coordinación de los sitios: no existe ya que la coordinación se realiza en la fase 4.

3. ejecución: todas las operaciones requeridas por el cliente son ejecutadas sobre las copias shadow.

4. Coordinación del acuerdo: una vez completada la transacción, se mandan (usando ABCAST) los cambios en un mensaje de certificación. Una vez recibidos estos mensajes, todos los sitios (inclusive el local) deciden si las operaciones pueden ser ejecutadas correctamente. Si no traen problemas de serializabilidad, los cambios son aplicados, garantizando que las transacciones pueden ejecutarse en el orden especificado por el ABCAST.

-- Cada sitio termina de la misma manera --

5. Respuesta al cliente.

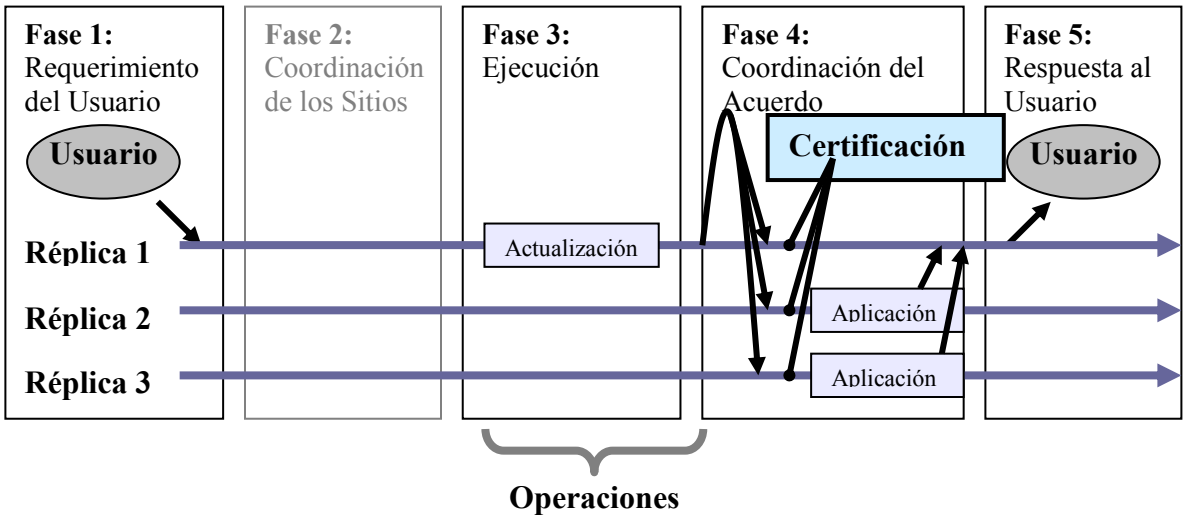


Figura 5.8: Esquema Eager Group con Certificación

[9] Completada no es lo mismo que cometida. Que una transacción haya sido completada significa que se ejecutaron todas sus operaciones, pudiendo luego ser cometida o abortada.

La garantía de Serializabilidad de Una Copia se debe a las propiedades de orden total y atomicidad del ABCAST. Por un lado, nos asegura que todos los sitios certifican todas las transacciones en el mismo orden y por el otro, que si un sitio certifica una transacción, entonces todos los demás también.

5.5.3 Conflictos de actualización

No hay posibilidad que existan conflictos de actualización. Estos esquemas previenen mediante espera o bloqueos que dos actualizaciones generadas en diferentes sitios "choquen" al actualizar la misma réplica.

No olvidemos que son esquemas basados en *Serializabilidad de Una Copia*, donde un objeto debe aparecer como una copia lógica y la ejecución de transacciones concurrentes es coordinada para que sea equivalente a una ejecución serial sobre la copia lógica, evitando de esta manera cualquier tipo de conflicto.

5.5.4 Trabajos relacionados

Esta técnica a sido ampliamente estudiada en la literatura [KA98, HAE99, KA00], todas logran serializabilidad de una copia, pero son implementadas de manera diferente variando las características de performance. Entre las mas conocidas tenemos protocolos tradicionales basados en quorums, como ROWA (read-one/write-all), ROWAA (read-one/write-all-available) o QC (Quorum Consensus). La mayoría de los esfuerzos en esta área han sido dedicado a otorgar diferentes maneras de construir y optimizar el tamaño de los quorums [WCYC, LCC]. En [CHKS94] podemos ver una clasificación y descripción de los mismos .

En el marco de una gran cantidad de investigaciones se sostiene que estas aproximaciones no son muy relevantes en la práctica debido a sus problemas de performance y escalabilidad. Esto ha provocado una fuerte tendencia en los diseñadores, a no utilizarlos, llevando a que estos esquemas no hayan sido prácticamente usados en productos comerciales.

Algunos de los argumentos detrás de esta creencia son, por un lado, el estar propenso a un significante limite de escalabilidad (no son escalables mas allá de unos pocos sitios) debido a las altas probabilidades de deadlocks que introduce la replicación. Gray en [GHOS96] muestra que en algunas configuraciones la probabilidad de deadlock es directamente proporcional a n^3, donde n es la cantidad de nodos. Por el otro, la serializabilidad es la causante de una sobrecarga extremadamente alta (por el número lineal de mensajes), conduciendo a la contención de recursos y largos tiempos de respuestas.

En trabajos mas recientes [Fla96, Alo97, GS97, WPS99], las primitivas de Group Communication han ido surgiendo de manera de resolver estos problemas particulares y dar un soporte para implementar replicación Eager Group.

5.5.5 Aplicación

Mas allá de los resultados presentados por Gray, otros autores [KA] sostienen que Eager Group es factible en un amplio espectro de aplicaciones y configuraciones. Son protocolos totalmente serializables, muy dependiente del sistema y la disponibilidad de la red. Por estas razones, pueden ser usados bajo ciertas condiciones: buena comunicación, baja carga del sistema, bajos proporción de conflictos y un porcentaje de lecturas razonablemente alto.

5.6 Esquemas híbridos

Estas aproximaciones generalmente están formadas por cualquier otra combinación de los esquemas que no sean los cuatro anteriormente mencionado. Puede ser por ejemplo la integración de los dos métodos de propagación, los dos métodos de regulación o los cuatros en uno mismo (ver Figura 5.9).

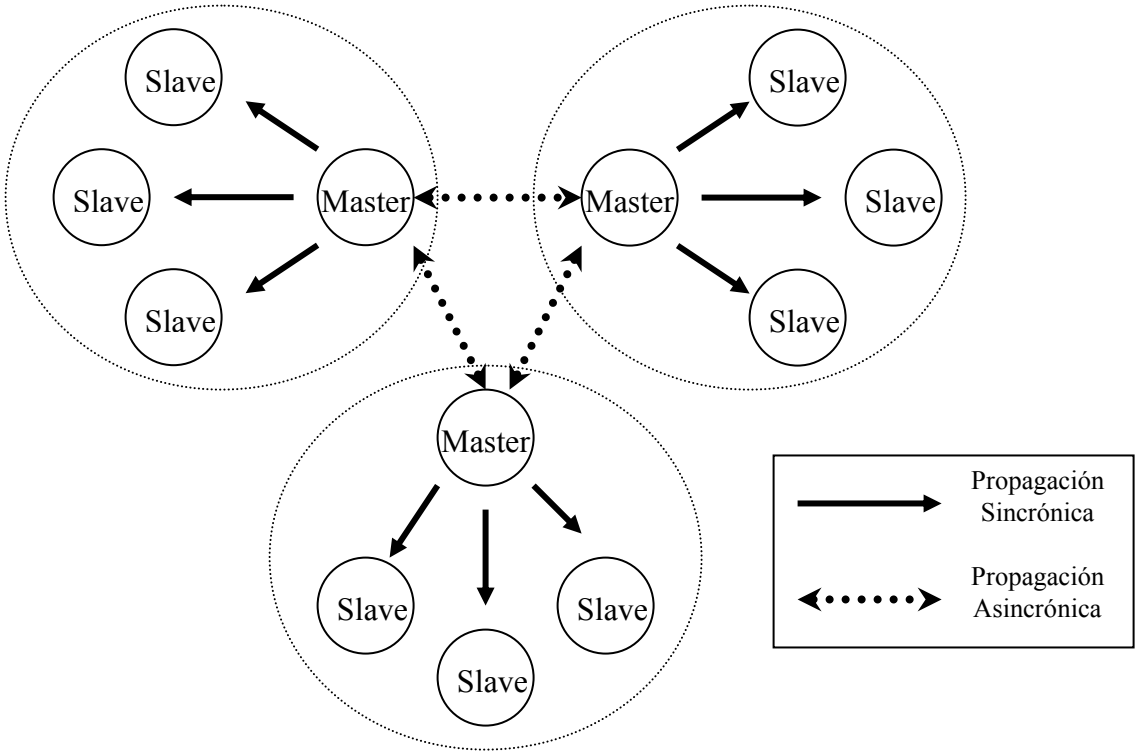


Figura 5.9: Esquema híbrido

Zhou en [ZH99] diseñó un algoritmo basado en copia primaria (Master-Slave) y el de votación por mayoría (Group). Durante la operación normal del sistema, el algoritmo sigue la aproximación de copia primaria, encontrándose el objetivo de proporcionar una sobrecarga

baja en la comunicación , y un tiempo de respuesta bueno. Cuando algún evento de la comunicación falla causando una partición de la red, este se degrada a un algoritmo híbrido de primario/mayoria consiguiendo el objetivo de maximizar la disponibilidad.

Otras aproximaciones combinan esquemas Eager y Lazy [ABKW98, BKR+99]. En [PKL] se define un modelo de replicación (similar a la Figura 5.9) que integra el método de copia primaria con esquemas de propagación sincrónicos y asincrónicos, dependiendo del tipo de réplica que debe ser actualizada. Las réplicas están agrupadas en clusters, dentro del cual puede haber tres tipos de réplicas: primarias, secundarias y terciarias, siendo estas dos últimas copias backup de la primaria. La diferencia esta en que las secundarias están en el mismo cluster que la primaria, con lo cual usa propagación sincrónica, y las terciarias se encuentran en diferentes cluster al primario, siendo actualizadas asincrónicamente.

Capítulo 6

Diseño e Implementación de un Modelo para la Experimentación en Técnicas de Actualización de Réplicas

En este capítulo se mostrará el diseño del modelo para experimentar las diferentes técnicas de actualización, y se describirán algunos puntos relevantes en la implementación de cada una de ellas.

6.1 Componentes básicos en el diseño de la aplicación

6.1.1 Procesos y subprocesos

El diseño de la implementación desarrollada, puede describirse con los procesos de la Figura 6.1 y subprocesos de la Figura 6.2. Algunos de estos procesos se implementan de manera diferente según el método de regulación o actualización que se quiera utilizar, sin embargo, son generales para cualquier combinación de esquemas. Ellos son:

Coordinator (Proceso Coordinator). Se dedica a la carga y distribución de las consultas, también recibe los resultados de las mismas. Esta compuesto por dos subprocesos, el *Manager* y el *Monitor*

Manager (Subproceso administrador de la carga y distribución de consultas). Es el encargado de tomar cada una de las consultas, determinar sobre que réplica se puede realizar la operación, ya sea local o remota, y enviar la consulta con un Identificador de Réplica al proceso *DataServer(MasterSender)*. Es el responsable del procesamiento de las operaciones. También administra el esquema de replicación (como están distribuidas las réplicas) y toda información de conectividad sobre los sitios.

Monitor (Subproceso de Monitorización). Es el encargado de recibir de los subprocesos *SlaveSender y SlaveListener*, los resultados de las consultas, analizarlos, realizar cálculos y escribir en almacenamiento permanente los resultados de la ejecución de la aplicación.

DataServer (Proceso Servidor de Datos). Es el responsable de manejar todos los requerimientos locales de datos que vienen de *Coordinator (Manager)*, y devolver los resultados a *Coordinator (Monitor)*. También manipula los requerimientos remotos de otros sitios. Para esto, esta compuesto por cuatro tipos de subprocesos, el *MasterSender, MasterListener, SlaveSender* y el *SlaveListener.*

MasterSender (Subproceso Master de Envíos). Es el proceso responsable de recibir las consultas que le genera el *Manager* y delegar dicha tarea a un *SlaveSender* si es una consulta local, y si es remota, enviársela a un *MasterListener* remoto. En este último caso se comunica con un *SlaveListener* para que espere el resultado de la consulta remota. Tanto con los *SlaveSender* o *SlaveListener*, lo primero que hace es fijarse si no hay algún proceso que esté manejando la réplica requerida, si lo hay, le asigna la nueva consulta, sino, crea uno nuevo. De esta manera podemos manejar múltiples réplicas simultáneamente.

MasterListener (Subproceso Master de Recepción). Su función es recibir los requerimientos (sobre los datos locales) provenientes de un *MasterSender* remoto, y distribuirlos a los diferentes *SlaveSender* para que realicen tales requerimientos.

SlaveSender (Subproceso Slave de Envíos). Se encarga de gestionar todos los requerimientos de datos de una réplica determinada. Una vez que llega un pedido, ya sea local (enviado por el *MasterSender*) o remoto (enviadas por el *MasterListener*) se comunica con el *DataManager* para realizar la consulta. Luego retorna los resultados a un *SlaveListener* remoto, si fué un requerimiento remoto, o al Monitor, en caso de un requerimiento local.

SlaveListener (Subproceso Slave de Recepción). Es el proceso responsable de recibir los resultados de las consultas remotas sobre una réplica en particular y enviárselos al Monitor.

DataManager (Proceso Administrador de Datos). Es el responsable del almacenamiento y recuperación de los datos. Cuando un *SlaveSender* requiere una lectura sobre una determinada réplica, éste proceso debe recuperar su valor almacenado en el *DataFile*. De igual manera, ante un requerimiento de actualización escribe los datos en el DataFile y si es necesario propagar tal actualización se comunica con el *Propagator*.

Propagator (Proceso Propagador). Es el responsable de realizar las propagaciones hacia las réplicas remotas. Recibe las actualizaciones sobre datos locales provenientes del DataManager y las envía a los procesos *Receiver* remotos.

Receiver (Proceso Receptor). Su función es la de recibir mensajes provenientes de los *Propagators* remotos y realizar los pedidos correspondientes de actualización al *DataManager.*

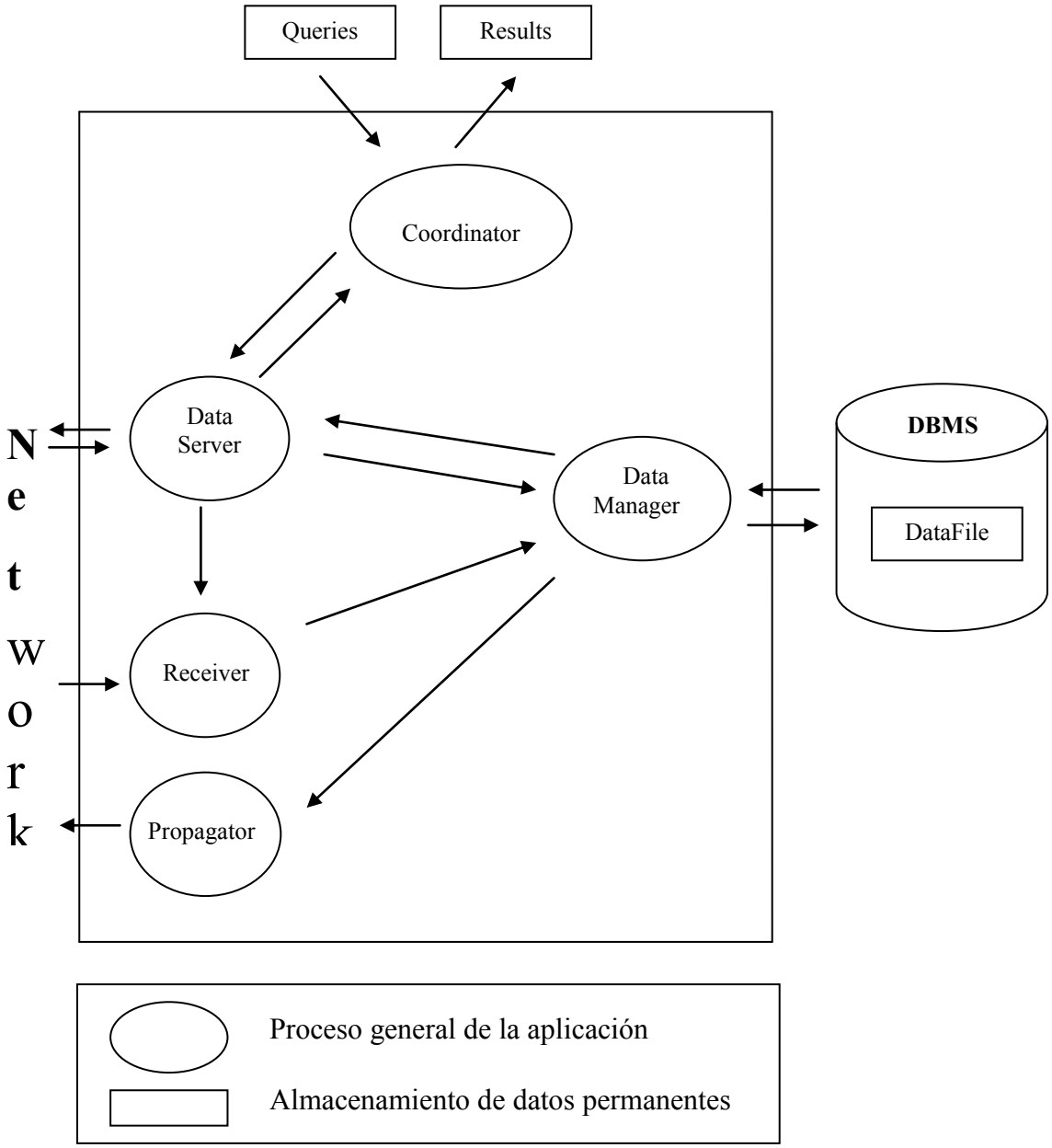


Figura 6.1: Procesos de la aplicación distribuida

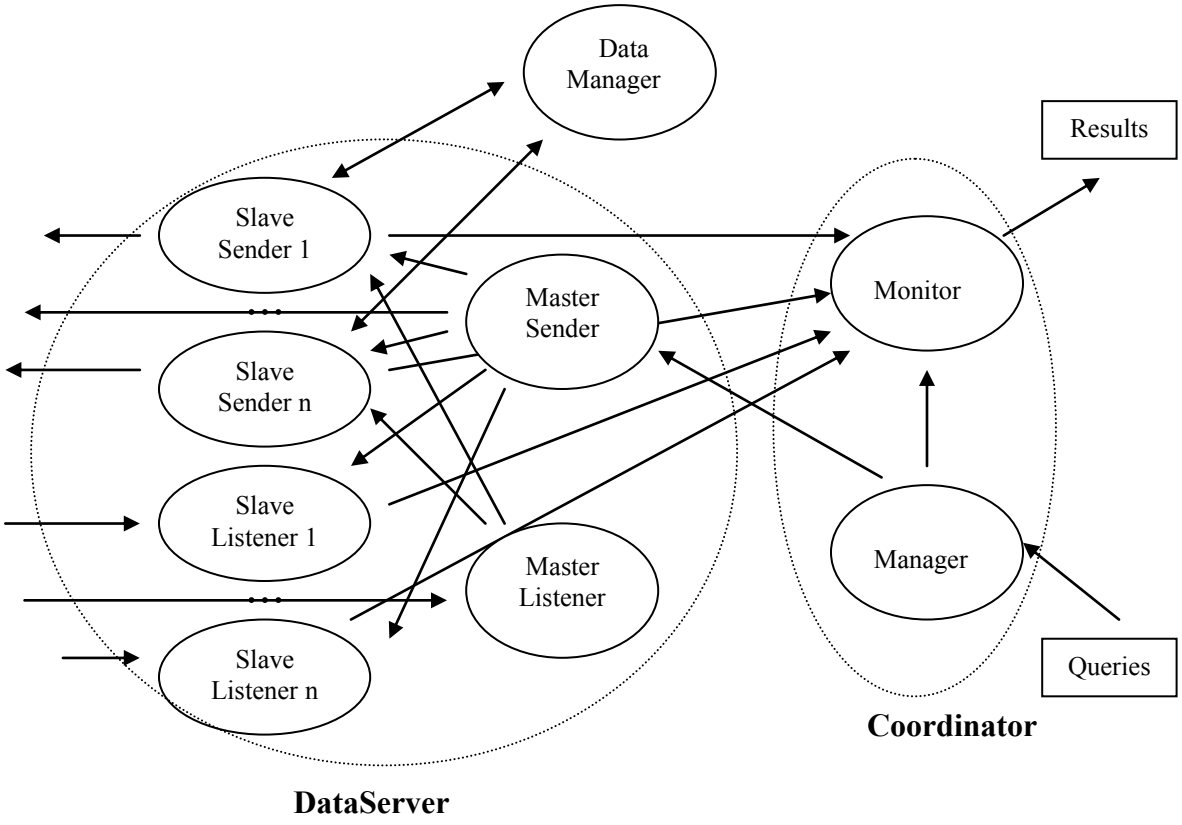


Figura 6.2: Subprocesos de DataServer y Coordinator

6.1.2 Almacenamientos de datos permanentes

Como vemos en las figuras, además de procesos, existen los siguientes almacenamientos de datos permanentes

DataFile. Es un archivo que almacena los datos involucrados en la aplicación. Cumple la función de simular el almacenamiento local de la BD para toda réplica. Cuando un read(X) es ejecutado, se recupera el valor del ítem X en la base de datos. Cuando se ejecuta algún write(X,b), el valor del ítem X se setea a b. Consta de un número constante n de ítems de datos o réplicas de igual tamaño, con la siguiente estructura:

- id: identificador de la réplica
- value: valor de la réplica

Observación: el *DataFile* es implementado simplemente mediante una estructura de datos mantenida en memoria volátil para tener una mayor performance en el acceso a los datos. Los ítems son indexados mediante un entero en el rango de 0...n-1 y la exclusión mutua en el acceso de los mismos, es provista mediante las primitivas de sincronización de los Thread de Java.

Queries. Contiene la traza de ejecución de las operaciones a realizar. Sus datos son:

- id: es el identificador del dato sobre el cual quiero realizar la operación
- operation: indica que tipo de operación se realizará, lectura o actualización (Read,Update)
- new_value: para el caso de las actualizaciones tendrá el nuevo valor del dato a actualizar. Para mejorar el análisis de los resultados, este valor es generado en el Manager con el Id del host mas la hora de inicio de la operación (ver Apéndice A)

Results. Mantiene toda la información referente a los resultados de la aplicación.

6.1.3 Comunicación entre procesos remotos

La forma en que los procesos pertenecientes a diferentes máquinas se comunican se pueden ver en la Figura 6.3.

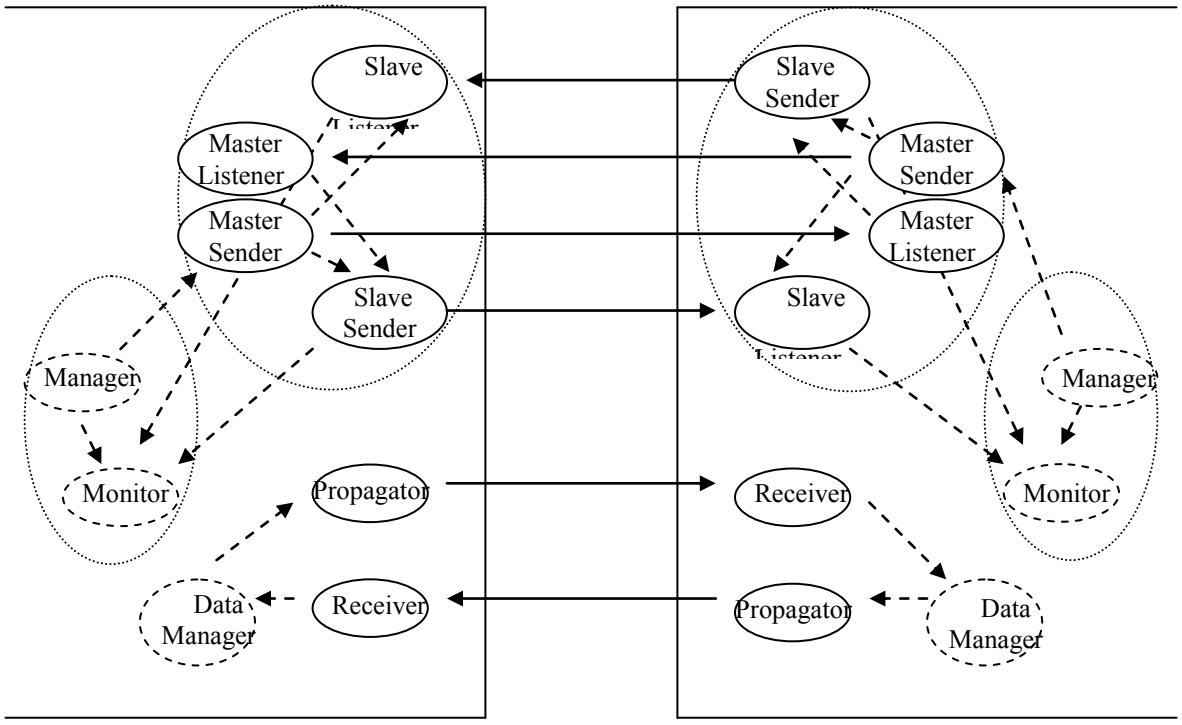


Figura 6.3: Circuitos de información con dos máquinas

6.2 Circuito de información de una operación

6.2.1 Operación de lectura

En una *operación de lectura* (ver Figura 6.4), el Manager primero determina cuál es el host más conveniente sobre el cual realizar la misma, siendo una copia local el mejor de los casos. Por un lado, estos datos iniciales de la operación son enviados al Monitor para que los grabe en almacenamiento permantente (TempIni.txt). Por el otro, envía estos datos al MasterSender quien controla si es una lectura local o remota:

- Si es local otorga el requerimiento al SlaveSender que maneja ésta réplica (si no lo hay lo crea). El SlaveSender requiere el dato al DataManager para luego ser retornado al Monitor, quien graba esos resultados en almacenamiento permanente (TempFin.txt).

- Si es una consulta remota envía el requerimiento al MasterListener del host que contiene la réplica y crea un nuevo SlaveListener (si no existe uno que esté manejando tal réplica) para recibir los resultados remotos. El MasterListener remoto recibe el requerimiento y lo otorga al SlaveSender que maneja ésta réplica (si no lo hay lo crea). El SlaveSender requiere el dato al DataManager para luego ser retornado al SlaveListener remoto. Este último recibe los resultados remotos y se los envía al Monitor, quien graba esos resultados en almacenamiento permanente (TempFin.txt).

6.2.2 Operación de actualización

El circuito de información de una *operación de actualización* es similar al de una lectura, con la diferencia que al realiza una actualización local, el DataManager recibe tal requerimiento, y además de realizar la actualización sobre la réplica se encarga de comunicarse con el Propagator para propagar a los demás host remotos tales cambios. Para realizar esto existen en cada uno de ellos procesos Receivers que aceptarán tales propagaciones y las enviarán a sus respectivos DataManager locales. Si la actualización se debe realizar sobre una réplica remota, ya sea por no poseerla o no ser master de la misma, tal actualización se verá reflejada en la réplica local mediante su propagación (ver Figura 6.5).

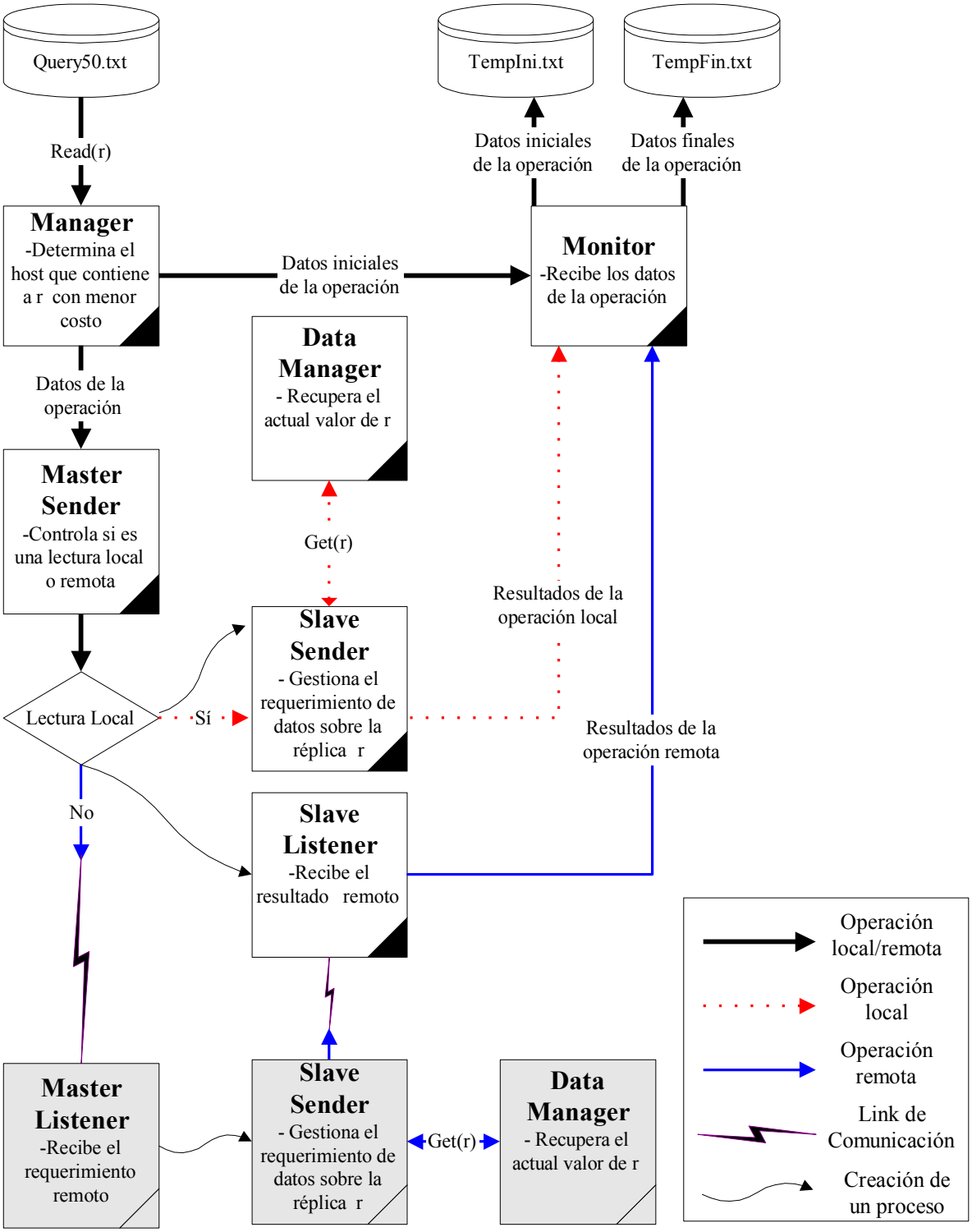


Figura 6.4: Circuito de información de una operación de lectura

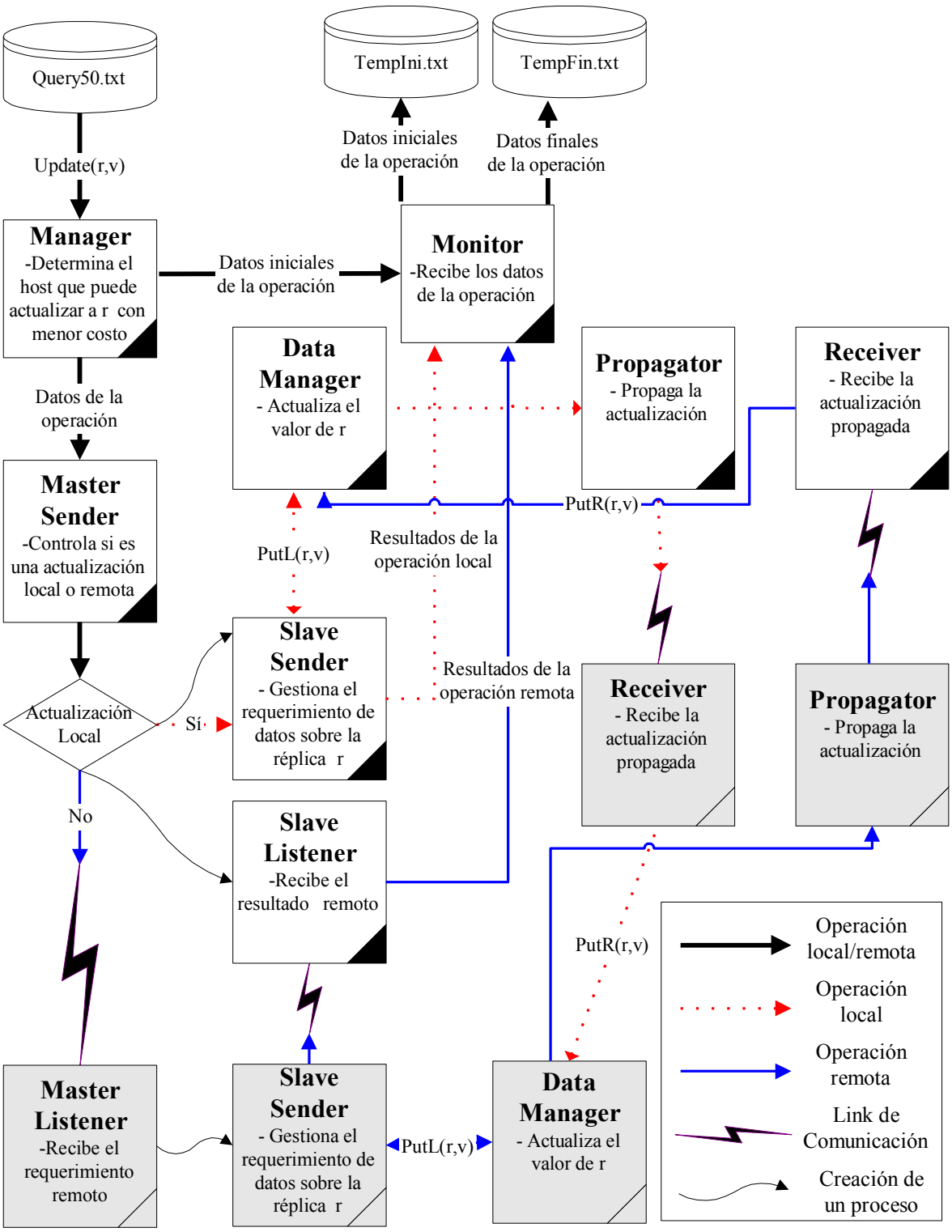


Figura 6.5: Circuito de información de una operación de actualización

6.3 Soporte de implementación

El soporte de simulación elegido es Java [CW96, Jav96, Eck97, WZ99], dado que permite una mayor versatilidad en las construcciones de aplicaciones. El hecho que Java no sea un lenguaje de alta performance no es relevante para nosotros: el modelo solamente es utilizado para comparar la performance relativa de los diferentes esquemas de actualización de réplicas.

6.4 Adaptación de los procesos a cada esquema

En esta sección se describirán algunos puntos importantes que tienen relación con la implementación de los diferentes esquemas de actualización de réplicas.

6.4.1 Métodos de propagación Lazy

Gran parte de los cambios realizados en algunos procesos a la hora de implementar los esquemas Lazy, esta relacionado con la capacidad de detectar y solucionar los conflictos de actualización. Para esto usamos la técnica de *Timestamps*, y garantizamos la *propiedad de convergencia* entre las réplicas.

De esta manera, cada actualización propagada lleva un timestamps de cada réplica original. Si el timestamps de la réplica local viola el orden, existe un conflicto y se necesita alguna forma de reconciliación. En nuestro caso, ésta reconciliación consiste en lograr convergencia entre los datos, y enviar la información de los conflictos ocurridos a un nuevo proceso denominado *Recovery*, con el fin de ser utilizada en los índices de evaluación.

Cambios introducidos:

- Timestamps
- El proceso *Recovery*, tiene como función recibir mensajes provenientes del DataManager de aquellas actualizaciones que tienen conflictos y abría que recuperar, grabando luego los datos de dichas actualizaciones en almacenamiento permanente.

Timestamps

Un *Timestamp* está compuesto de:

- IdHost: identificador del Host
- Lc: contador local para ordenar las propagaciones (logical clock)

Este *reloj lógico* es un contador entero que se incrementa cuando se propaga una actualización. Cada DataManager tiene un reloj lógico utilizado para crear los Timestamps, realizando las siguientes operaciones sobre ellos:

- Cuando un DataManager envía un mensaje al propagador, setea el tiemstamp del mensaje al valor corriente de lc y luego lo incrementa en 1. Así, cada máquina tendrá un timestamp diferente y creciente para cada propagación realizada.

- Cuando un DataManager recibe un mensaje de actualización remoto (enviado por el Receiver) con timestamp ts, setea lc al máximo de lc y (ts.lc + 1). La función de esto es mantener cierto grado de sincronización de los relojes locales entre dos sitios que se comunican.

Estos timestamp además de tener el lc tienen un identificador del Host del cual parte la actualización para inducir un orden total de los mensajes, usando la identidad del host mas bajo para romper empates si timestamps de diferentes host tienen el mismo reloj lógico:

TS1 < TS2 sii TS1.lc < TS2.lc) || ((TS1.lc == TS2.lc) && (TS1.idHost < TS2.idHost)))

6.4.1.1 Detección de conflictos de actualización

En la Figura 6.6 vemos el caso en el cual se realiza una actualización de la réplica 4 en el Sitio 2 sin que se produzcan conflictos de actualización. A partir de un estado inicial (6.6.a) con ambas réplicas consistentes, se realiza una actualización en el sitio 2 (6.6.b). Una vez realizados los cambios locales, se procede a propagar tales cambios con oldTS igual al tsLocal anterior a la actualización, y newTS con el nuevo tsLocal, además de llevar por su puesto el valor actual de la réplica. Cuando la propagación llega al Sitio 1, compara el tsLocal de este sitio, con oldTS, y como son iguales realiza la actualización, llevando las réplicas a un estado consistente (6.6.c).

a)

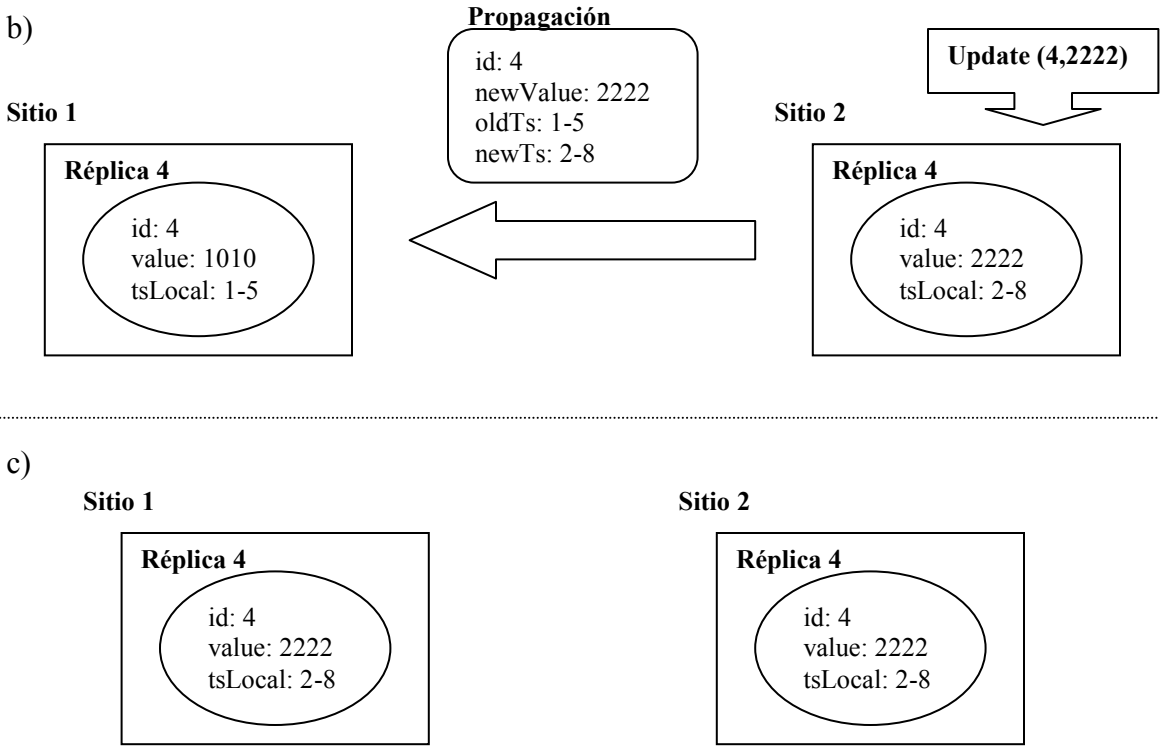
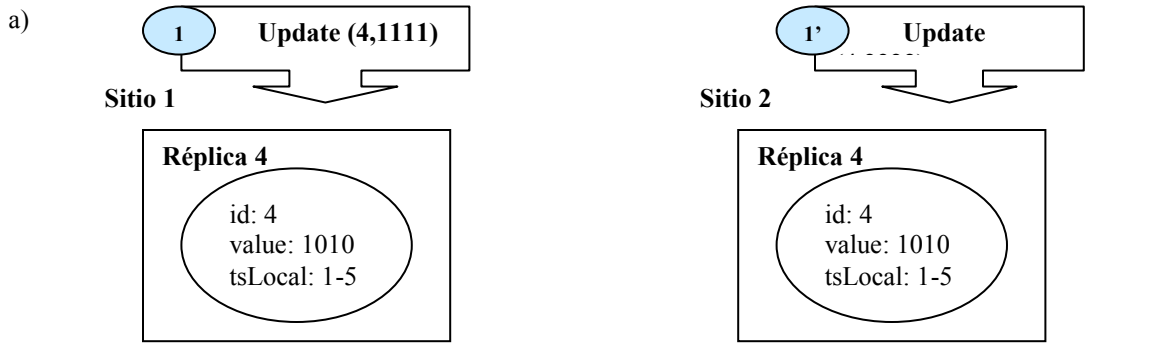


Figura 6.6: Propagación sin conflictos de actualización

También tenemos el caso donde se pueden producir conflictos y es necesario detectarlos (ver Figura 6.7). Supongamos que en el Sitio 1 y el Sitio 2 ocurren dos actualizaciones (1 y 1') casi simultaneas sobre el ítem 4 replicado en cada sitio (6.7.a). Una vez realizados los cambios locales, se procede a realizar las correspondientes propagaciones (6.7.b). Cuando cualquier propagación (2 o 2') quiere actualizar la réplica remota, se encuentra con que su oldTS es diferente al tsLocal, detectando de esta manera un conflicto de actualización y así utilizar un mecanismo para reconciliar ambas actualizaciones.



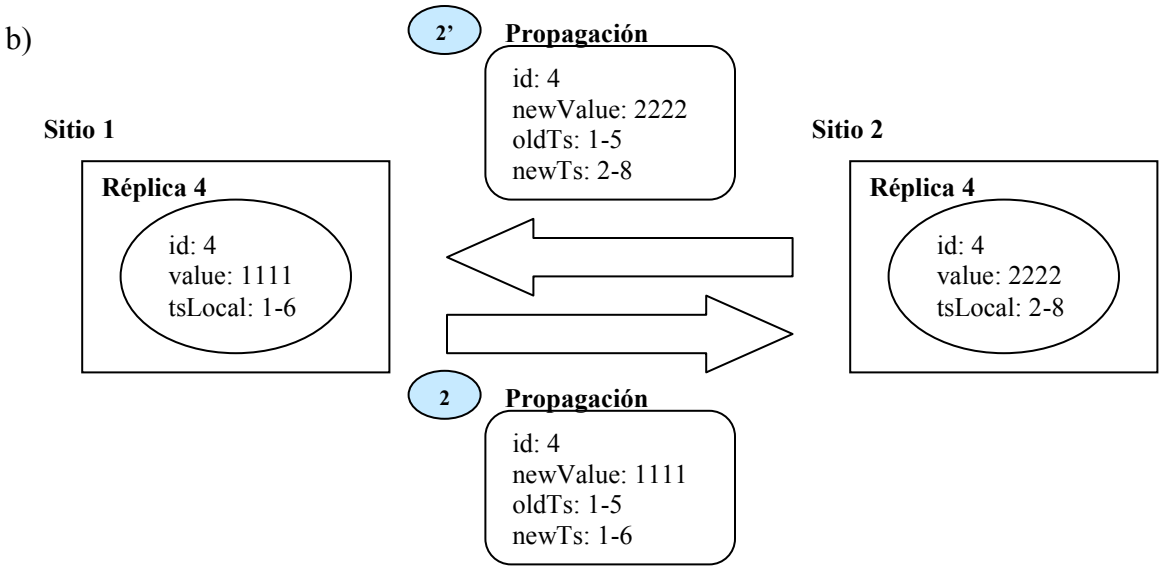


Figura 6.7: Propagación con conflictos de actualización

6.4.1.2 Mecanismo de resolución

Como mecanismo de resolución o reconciliación de los conflictos de actualización, usamos la Thomas Write Rule (TWR) [BHG87]. Esta regla se refiere al caso donde la transacción T actualiza d pero el timestamp de T es menor que el timestamp de escritura de d. En lugar de abortar T, T continua como si la escritura hubiese sucedido, sin embargo, la escritura es ignorada.

En la Figura 6.7, cuando detectamos el conflicto con la propagación (2), vemos que tiene newTS (1-6) menor que el tsLocal del Sitio 2 (2-8), ignorando tal actualización. En cambio, al producirse el conflicto con la propagación (2'), su newTS es mayor que el tsLocal del Sitio 1, realizándose tal actualización. De esta manera ambas réplicas convergen a un mismo valor y así llevadas a un estado consistente mostrado en la Figura 6.8.

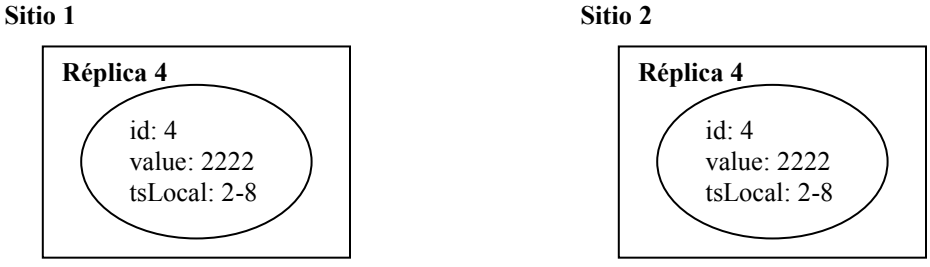


Figura 6.8: Réplicas consistentes

Observación: Estos mecanismos de detección y resolución de conflictos de actualización, a pesar de no ser necesarios en el esquema Lazy Master (si en el Lazy Group) fueron utilizados simplemente para probar esto, es decir que no se producirán conflictos al tener un solo master para cada réplica.

Otras consideraciones

Una *Réplica* no tendrá solamente su id y valor, también tendrá asignado un Timestamp correspondiente al de su última actualización (tsLocal).

El proceso *DataManager* es modificado para manejar el reloj lógico como se indicó anteriormente.

El proceso *Propagator* posee una cola de mensajes de aquellas actualizaciones que debe propagar, esta cola permite manejar la replicación de manera asincrónica, sin que ésta interfiera en los demás procesos. Así, ante una actualización, un *SlaveSender* puede recibir una respuesta desde el *DataManager* (y a su vez enviársela al Monitor o SlaveListener), sin tener que esperar a que se produzca la propagación.

Los procesos *Propagator* y *Receiver* son adaptados para que envíen y reciban respectivamente, los timestamps correspondientes. A los datos de la actualización se le suma el valor anterior del timestamp (oldTS) y el nuevo valor del timestamp (newTS).

6.4.2 Métodos de propagación Eager

En este método utilizaremos *técnicas de bloqueo,* mediante las cuales se detectan las anomalías y se convierten en esperas o bloqueos. Por ejemplo en un esquema Eager Group usaríamos un protocolo 2PL (Two Phase Locking) para ordenar las operaciones en conflicto y 2PC (Two Phase Commit) para garantizar atomicidad de las actualizaciones, o en Eager Master un método de Bloqueo Centralizado y el 2PC.

Para el objetivo de nuestro trabajo, solo usaremos variables de tiempos de espera en el Proceso *Propagator* para simular tales protocolos, y también en *DataManager* para simular las esperas por estar bloqueados los datos que requieren. Básicamente, simula el bloqueo y desbloqueo sobre todo ítem de dato accedido.

Cambio introducidos:

- Variables de tiempo de espera

6.4.2.1 Simulación del bloqueo y desbloqueo de un ítem de dato accedido

Para simular el bloqueo cuando un lock no puede ser concedido utilizamos primitivas de sincronización de los Thread de Java. Desde el punto de vista de la implementación, demorar una operación significa que su Java Thread es bloqueado y luego reiniciado cuando se puede disponer del dato bloqueado. Este mecanismo es implementado en el *DataManager;* cuando se quiere leer o actualizar un dato, se realizan los siguientes pasos:

locked = número randómico entre 0 y 100

if (locked <= probabilidad de que el dato este bloqueado) {

bloqueo el proceso que requirió el dato por n ms, donde n es un número aleatorio entre 0 y 100.

}

actualizo la réplica o retorno su valor en el caso de una lectura

6.4.2.2 Simulación de los protocolos bloqueantes

Si utilizamos un protocolo de bloqueo tendríamos para replicar a n sitios una proporción de mensajes mayores o iguales a 4*n [BG92].

Así, a partir de una matriz de Pesos de Comunicaciones (ver Apéndice A) que contiene un peso de la comunicación existente entre cualquier par de sitios, se calcula el tiempo total de demora que llevaría la sincronización con los demás sitios. Por ejemplo, para calcular el delay de sincronización para actualizar una réplica r tendríamos:

```
for cada host i {
        if i tiene una réplica de r {
            delay = delay + 4 * pesoComunicacion(myHost,i);
        }
}
```

Otras consideraciones

Una consideración muy importante, es que el *Propagator* ya no contiene una cola de pedidos como en el caso asincrónico, sino que atiende sincrónicamente pedido por pedido, bloqueando los demás procesos involucrados. Así, ante una actualización, un *SlaveSender* no recibe una respuesta desde el *DataManager* (y a su vez no puede enviársela al Monitor o SlaveListener), hasta que se produzca la propagación.

A diferencia de los esquemas asincrónicos, no se necesitan Timestamps ya que no existirían los conflictos de actualizaciones. Por lo tanto, una *Réplica* no tendrá asignado un Timestamp, y los procesos *DataManager, Propagator y Receiver* no deberán manejarlos.

6.4.3 Métodos de regulación

En cuanto a la implementación de los procesos implicados en el diseño de la aplicación no existen diferencias entre los métodos de actualización Group y Master. Simplemente en el archivo que contiene el esquema de las réplicas se indica si es una copia Master o Slave, en el caso del método Group todas las réplicas serán Master. El proceso *Manager* será el encargado de decidir si puede o no actualizar un dato en particular.

6.5 Monitorización de los resultados

Cada máquina toma medidas locales (proceso *Monitor)* de manera que no necesitan ninguna comunicación con las demás. Al finalizar la simulación se envían los cálculos en cada máquina a un único proceso denominado *Init*. Este proceso, además de iniciar la ejecución de la aplicación, es el que se encarga de calcular los promedios, dado que recibe toda la información de la ejecución.

Este proceso de recepción de datos esta asignado a una máquina que no realiza ningún tipo de tareas, mas allá de inicializar las demás máquinas, recibir, calcular y escribir los resultados en almacenamiento permanente. Estos resultados serán posteriormente utilizados para calcular los índices de evaluación, analizar y comparar los esquemas propuestos.

El esquema de la Figura 6.9 muestra los procesos de iniciación, recolección y cálculo de las medidas desde la máquina *Inicializadora* a cualquier otra máquina i.

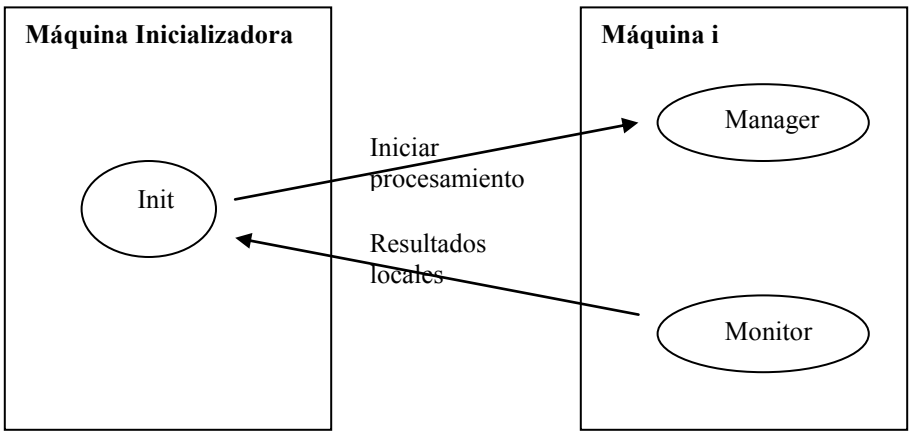


Figura 6.9: Inicialización y recepción de resultados

Capítulo 7

Experimentación

Análisis y Evaluación de los Resultados

En este capítulo comparamos los cuatro esquemas de actualización de réplicas implementados: Lazy Master-Slave, Lazy Group, Eager Master-Slave y Eager Group. Primero definimos los parámetros que son fijos para todos los experimentos, aquellos que varían, y también los índices de evaluación. Finalmente, presentamos los resultados y comparamos las técnicas de replicación.

7.1 Descripción de la Experimentación

7.1.1 Algunos conceptos

Un *experimento* es caracterizado por un número n de *corridas o ejecuciones* del modelo bajo un seteo particular de sus parámetros que pueden afectar la performance de los esquemas. Cada una de estas *corridas* consistirán en la ejecución del modelo bajo un conjunto de operaciones de lectura y actualizaciones sobre los datos replicados, obteniendo en su finalización resultados para los índices de evaluación.

En un mismo experimento los parámetros pueden ir variando en cada una de las corridas, por ejemplo el porcentaje de operaciones de lecturas puede cambiar en cada una de las ejecuciones para experimentar su impacto en cada uno de los esquemas.

Un experimento nos permite ver el impacto del modelo (bajo cierta parametrización) en los índices de evaluación y así comparar los diferentes esquemas de actualización de réplicas.

7.1.2 Suposiciones

- El modelo que asumimos consiste de un conjunto de computadoras independientes, llamadas nodos o sitios, conectadas vía una red de comunicaciones. Los datos son almacenados en todos o alguno de los sitios.

- Los sitios se comunican entre ellos a través de mensajes. Asumimos que los mensajes son intercambiados entre nodos del sistema de replicación a través de una red de comunicación fiable y sincrónica que chequea los errores, y la perdida de mensajes. Los mensajes son recibidos en el mismo orden que son enviados (preservando el orden).

- El esquema de replicación planteado en este estudio se considera estático y preestablecido. Al iniciar un experimento, hay un número fijo de réplicas localizadas en sitios o localidades fijas. Para simplificar los experimentos, realizamos una distribución uniforme de las réplicas, es decir, cada máquina tiene y es master de un número igual de réplicas (ver Apéndice A).

- Las operaciones realizadas sobre tales réplicas pueden ser de dos clases: *update* que modifica pero no observa el valor de una réplica, y *read* que observa o recupera su valor pero no lo modifica.

- Finalmente, no tratamos con el manejo de fallas.

7.1.3 Entorno de experimentación

Los experimentos fueron conducidos utilizando 4 PC:

- Pentium 133 MHZ con 32 MB de RAM. (Máquina Inicializadora)

- Pentium II 233 MHZ con 64 MB de RAM

- K6II 500 MHZ con 64 MB de RAM

- Pentium III MHZ 800 con 64 MB de RAM

Tales máquinas fueron conectadas, usando una red Ethernet-LAN de 10 Mbit/seg y todas las comunicaciones entre los procesos fueron realizas usando sockets y TCP [CS93] como protocolo de transmisión.

7.1.4 Parámetros considerados

Entre los parámetros generales que se consideraron para cada experimento tenemos fijos y variables. Para aquellos que no varían a lo largo de los experimentos se especifica su valor, y en el caso que sus valores vayan cambiando durante los experimentos, damos el rango considerado. Ellos son:

Parámetros fijos

Parámetro	Valores	Observaciones
Tamaño de la base de datos	99 datos	La elección de este parámetro influencia en la cantidad interferencia producida por las operaciones, y por lo tanto afectaría en las técnicas de

		actualización. Sin embargo no es un punto que será investigado en este trabajo.
Número de máquinas	3	Indica la cantidad de máquinas que procesarán consultas (no incluye la inicializadora).
Número de operaciones	1500	Su distribución es de 500 operaciones por máquina.
Tiempo de comunicación entre las máquinas	1 ms.	Utilizado solo en la simulación de los esquemas Eager. Indica el tiempo que tarda una máquina para enviarle un mensaje a otra.
Probabilidad de bloqueo de un dato	50 %	Utilizado solo en la simulación de los esquemas Eager. Indica la probabilidad que existe de que un dato este bloqueado cuando se desea acceder al mismo.

Parámetros variables

Parámetro	Valores	Observaciones
Esquema de propagación de actualizaciones	Lazy Eager	
Esquema de regulación de actualizaciones	Master-Slave Group	
Número de réplicas	1 a 3	1 réplica = Esquema Particionado 2 réplicas = Esquema Parcialmente Replicado 3 réplicas = Esquema Totalmente Replicado
Porcentaje de lecturas	0 a 100	Este parámetro tiene un efecto muy importante sobre cada uno de los esquemas planteados, además de ser quien caracteriza el tipo de aplicación que utilizará tales esquemas.

7.2 Indices de Evaluación

Un *índice de evaluación* es un valor que caracteriza el modelo (en nuestro caso su performance) durante una ejecución, por ejemplo el tiempo de respuesta. Estos índices serán usados para realizar las correspondientes comparaciones.

Entre los índices considerados más importantes para evaluar y comparar los esquemas de actualización de réplicas tenemos el *tiempo de respuesta* y la *productividad*.

Tiempo de respuesta

El tiempo para procesar una operación *o* (Tpo(o)) puede definirse como el tiempo que ocurre entre la emisión de *o* y la recepción de sus resultados.

El tiempo de respuesta para un conjunto de operaciones O, pertenecientes a una corrida C, lo definiremos como:

$$\text{Tpo. de Rta .(C)} = \frac{\sum_{o \in O} \text{Tpo(o)}}{\#C}$$

donde #C es el número de operaciones ejecutadas en la corrida C.

Productividad

La *productividad* la definimos como el número de operaciones cometidas o ejecutadas en una máquina por período de tiempo. Siendo #C el número de operaciones ejecutadas en una corrida C, entonces definimos:

$$\text{Productividad (C)} = \frac{\#\mathbf{C}}{\text{Duración de C}}$$

Duración de C la definimos como $\sum_{i \in M} \frac{\text{Duración de Ci}}{\#\mathbf{M}}$

donde #M el número de máquinas y Duración de Ci esta definido como el tiempo entre la emisión de la primer operación y el momento en el cual se recibe la respuesta de la última operación emitida en la máquina i durante la corrida C.

Conflictos de Actualización

Este índice es utilizado solamente para el esquema Lazy Group donde pueden suceder conflictos de actualización. Es directamente obtenido por el modelo, para luego ser de utilidad al comparar los diferentes grados de replicación.

7.3 Análisis y evaluación de los resultados

En esta sección mostramos y analizamos los resultados obtenidos en las experimentaciones utilizando la variación del porcentaje de lecturas como parámetro fundamental en las mismas.

Estudiamos como la escalabilidad es un punto importante en la replicación de datos y lo hacemos variando el número de réplicas en los experimentos realizados. También probamos la simulación de los esquemas Eager, variando el tiempo de comunicación entre las máquinas y la probabilidad de bloqueo de los datos.

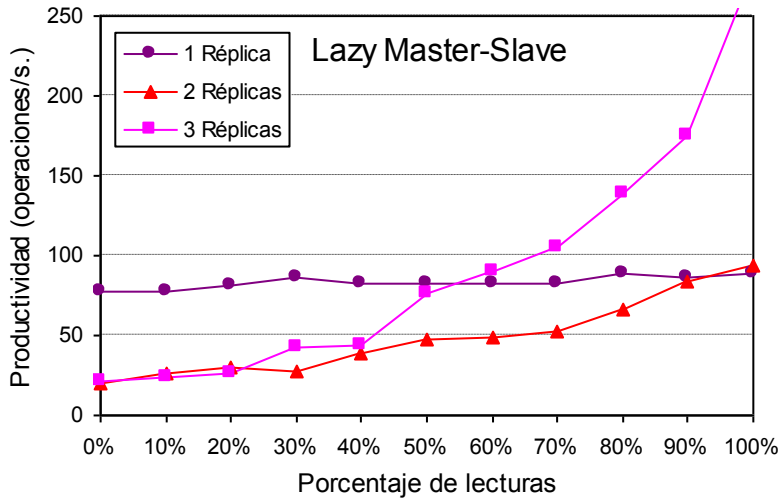
Finalmente, experimentamos la centralización de los datos en una máquina para ser comparados con los casos particionado y replicados.

7.3.1 Porcentaje de lecturas

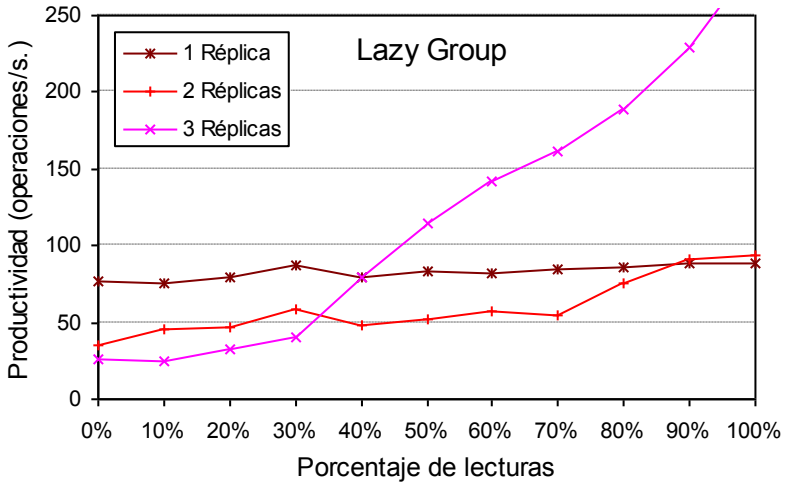
Este parámetro es muy importante por la influencia que tiene en el comportamiento de los diferentes esquemas. Desde el punto de las aplicaciones, este porcentaje de lecturas dependerá de su clase. No es lo mismo pensar un esquema de replicación para aplicaciones con pocas consultas, tales como aplicaciones ATM, a otras con muchas como pueden ser las páginas de Intenet, donde generalmente son leídas por muchos usuarios y cambiadas con muy poca frecuencia, logrando una mayor confiabilidad a través de la replicación. Por ello, este parámetro juega un rol muy importante en los experimentos que comparan las técnicas de replicación.

En todos los esquemas observamos que la *productividad* para altos porcentajes de lectura es mejor que para bajos porcentajes (ver Figura 7.1). Esto se debe a que una operación de lectura se realiza sobre solo una réplica; mientras que las operaciones de actualización se realizan sobre todas las réplicas, produciendo una mayor contención de los datos y recursos, e interfiriendo potencialmente con las demás operaciones. Esto no se ve reflejado para una 1 réplica (caso particionado), ya que no existe replicación y la productividad es similar tanto para altos porcentajes de lecturas como de escrituras, pues en la mayoría de los casos se realizan accesos remotos.

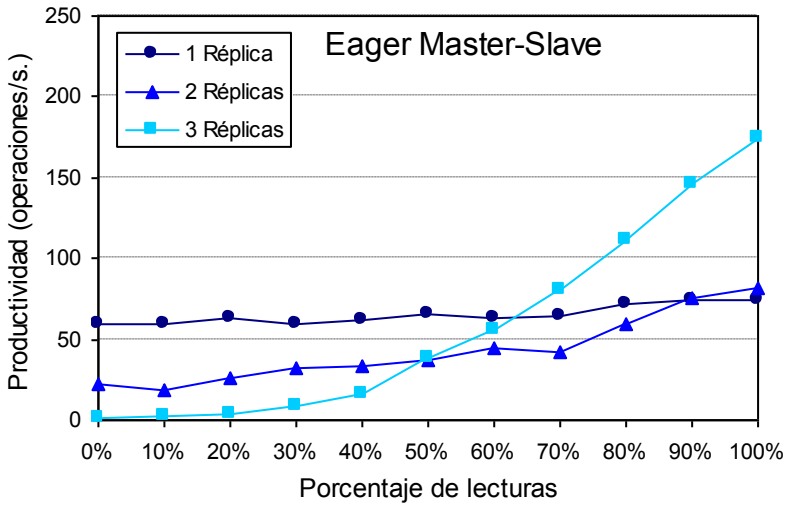
a)



b)



c)



d)

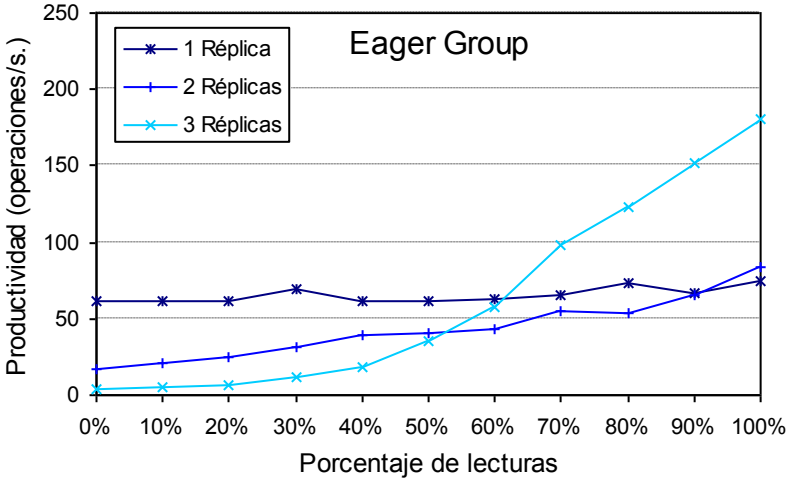
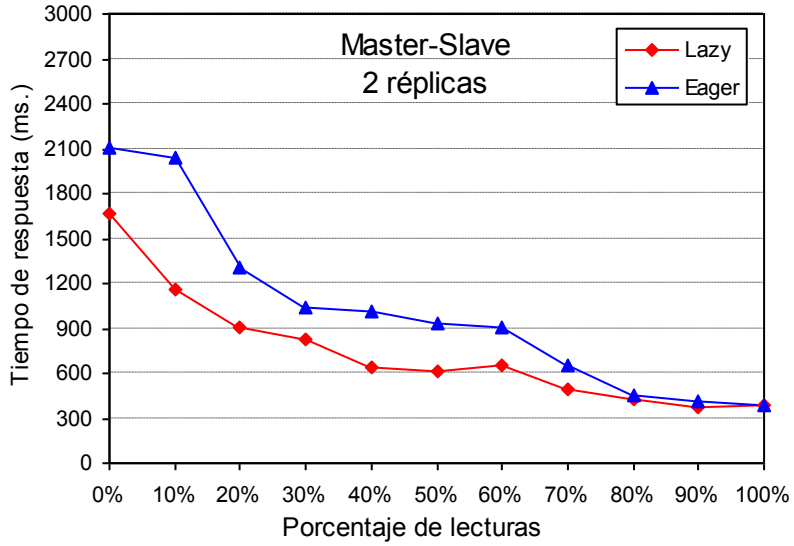


Figura 7.1: Productividad de los esquemas de replicación variando el número de réplicas

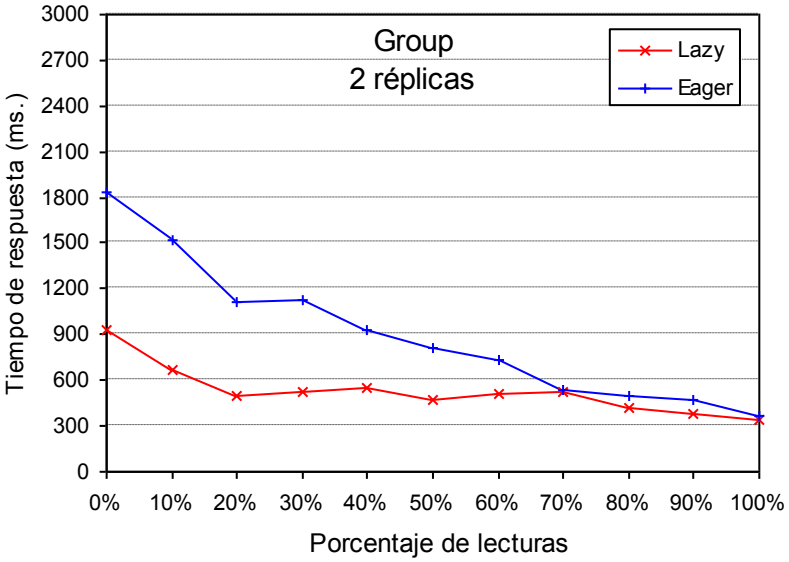
Por idénticas razones se ve afectado el *tiempo de respuesta* que disminuye a medida que el porcentaje de lecturas aumenta (ver Figuras 7.2). Sin embargo, en términos relativos la disminución es mucho mayor en el caso de la replicación total (ver Figuras 7.2.c y 7.2.d) debido a la ejecución de las operaciones de lectura de manera local.

También podemos observar como los esquemas Lazy tienen un mejor tiempo de respuesta que los Eager. Esto se hace aún más evidente al ir aumentando el grado de replicación y la tasa de actualizaciones debido a que en los esquemas Lazy no se espera por la sincronización de tales actualizaciones, primero se retorna una respuesta para luego realizar la propagación.

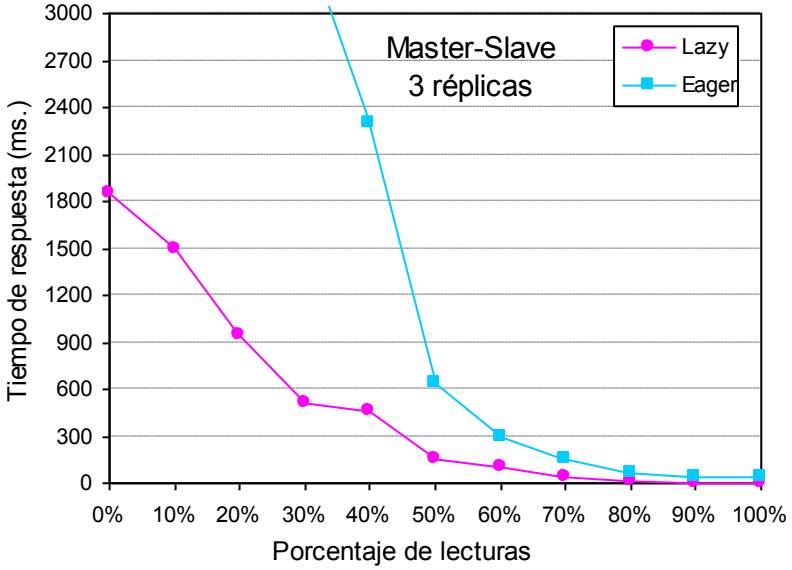
a)



b)



c)



d)

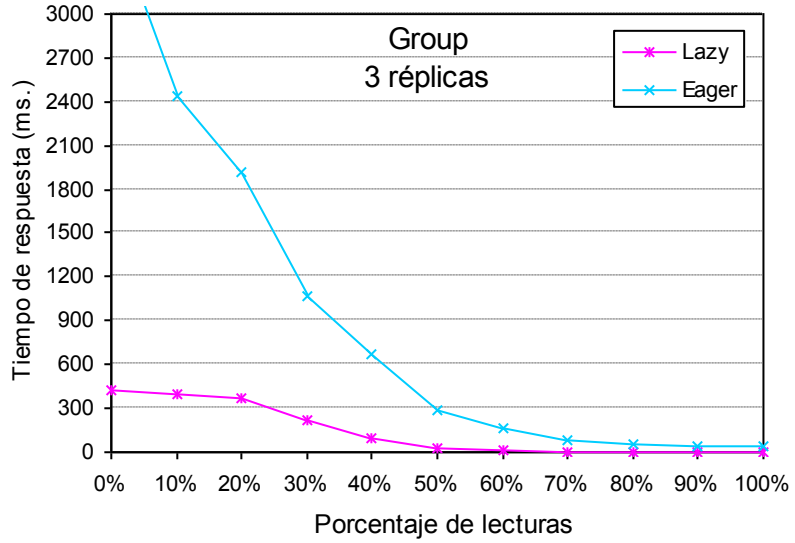


Figura 7.2: Tiempos de respuesta de los esquemas de replicación variando el número de réplicas

En la Figura 7.2 podemos observar que los esquemas Group tienen en general mejores tiempos de respuesta que los Master-Slave, especialmente al ir aumentando las actualizaciones. El problema es que los esquemas Master-Slave necesitan realizar los requerimientos de actualización sobre el master de la réplica, provocando accesos remotos que incrementan el tiempo de respuesta y producen posibles cuellos de botella al tener todos los pedidos de actualización asociados a la máquina que contiene la copia primaria.

En los esquemas Group esto no sucede ya que cualquier actualización puede aplicarse a cualquier réplica balanceando la carga del sistema.

Como mencionamos en un principio, el porcentaje de lecturas caracteriza la aplicación. Razonablemente, la proporción de actualizaciones en aplicaciones prácticas no es extremadamente alta; en bases de datos comerciales, aproximadamente el 70% de las operaciones son de lectura y solo el 30% de actualizaciones. Incluso, en cargas de OLPT, las transacciones normalmente tienen un 50% de operaciones de lectura. Sin embargo, los experimentos realizados también exploran los casos donde el sistema esta bajo altas cargas de actualización. Bajo esta presión, los esquemas sincrónicos fueron los mas afectados, produciendo una mayor contención de los datos y recursos del sistema.

7.3.2 Escalabilidad

A mayor número de réplicas mayor es la disponibilidad del sistema, pero puede ocurrir una sobrecarga en algunos esquemas debido a un aumento de la sincronización entre las mismas. Por otro lado, esta sobrecarga puede ser compensada por el balance de la carga.

La escalabilidad de los esquemas fue analizada variando el número de réplicas. Esta evaluación debió hacerse tanto sobre recuperaciones como actualizaciones de datos (ver Figura 7.1 y 7.2).

Altas tasas de lecturas

En todos los casos vemos que agregando réplicas incrementamos la disponibilidad del sistema. Al replicar los datos se reduce los costos de red, así como los tiempos de respuesta para las operaciones de lectura. Sin embargo, esta ganancia en disponibilidad, se ve afectada por el aumento de sobrecarga de comunicación en las actualizaciones.

En la Figura 7.2 observamos que en todos los esquemas, al aumentar el grado de replicación, bajo altas tasas de lectura el tiempo de respuesta tiende a bajar gracias a la posibilidad de realizarlas localmente.

Claramente, los beneficios potenciales de la replicación aumentan cuando se incrementa el número de accesos de lectura. En el caso extremo que todas sean lecturas y no haya escrituras, la replicación total (ver Figuras 7.2.c y 7.2.d) es indudablemente la mejor aproximación ya que todas las operaciones son locales y no se necesita ningún tipo de sincronización ni intercambio de mensajes.

Altas tasas de actualizaciones

En los experimentos se observa claramente que ante altas tasas de actualizaciones, replicar los datos decrementa la performance; en general, los tiempos de respuesta son mayores (ver Figura 7.2) y la productividad disminuye (ver Figura 7.1). Todo esto se debe al trabajo adicional de realizar actualizaciones sobre un número mayor de réplicas.

Los esquemas Eager muestran un incremento importante en sus tiempos de respuesta debido a una espera de sincronización mucho mayor al aumentar el número de réplicas. Por contrario, los esquemas Lazy no deben esperar por tal sincronización.

Al agregar nuevas réplicas, el esquema Lazy Master-Slave (ver Figura 7.2.a y 7.2.c) tiende a mantener su performance. Su escalabilidad, influencia solamente sobre las operaciones de lectura ya que las actualizaciones siguen realizándose sobre el master de la réplica. En cambio, en el esquema Lazy Group (ver Figuras 7.2.b y 7.2.d), vemos la ganancia en tiempos de respuesta que existe al ir replicando los datos. A mayor grado de replicación, el esquema va tomando ventaja de una mayor disponibilidad para realizar las actualizaciones localmente logrando por su puesto un mejor tiempo de respuesta.

Finalmente diremos que los esquemas Lazy escalan mucho mejor que los Eager, y entre ellos, el Master-Slave escala bastante bien a partir del 20% de lecturas, Lazy Group a partir de un 35%, mientras que para los sincrónicos resultan buenos mas allá del 50% de lecturas. Para menores proporciones, la productividad decrementa significativamente al agregar nuevas réplicas (ver Figura 7.1).

7.3.3 Conflictos de actualización

Otro índice que nos permite evaluar particularmente la escalabilidad de los esquemas Lazy Group es la proporción de *conflictos de actualización*. Esta proporción disminuye cuando

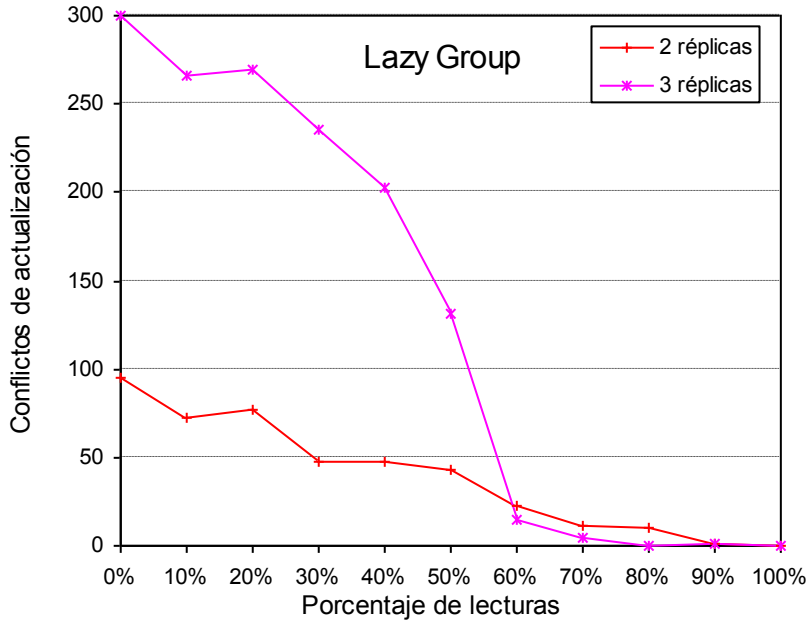


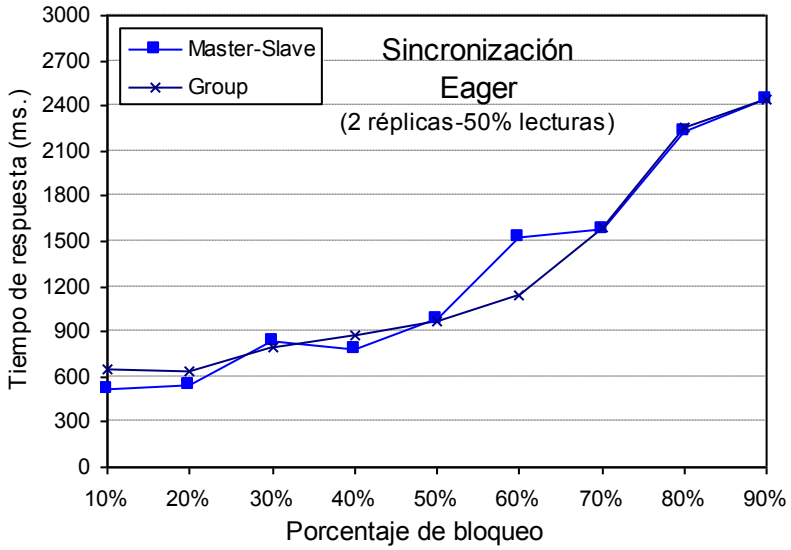
Figura 7.3: Conflictos de actualización para el esquema Lazy Group variando el número réplicas

aumenta el porcentaje de lecturas debido a que solo se producen conflictos con las operaciones de actualización. También observamos que los conflictos tienden a subir abruptamente cuando el número de réplicas aumentan debido a la mayor descentralización del procesamiento que se produce (ver Figura 7.3).

7.3.4 Sincronización en los esquemas Eager

En esta sección, analizamos los tiempos de respuesta para los esquemas Eager, variando los parámetros utilizados para simular la sincronización de los mismos. Ellos son el tiempo de comunicación entre las máquinas (ver Figura 7.4.a) y la probabilidad de que un dato en

a)



b)

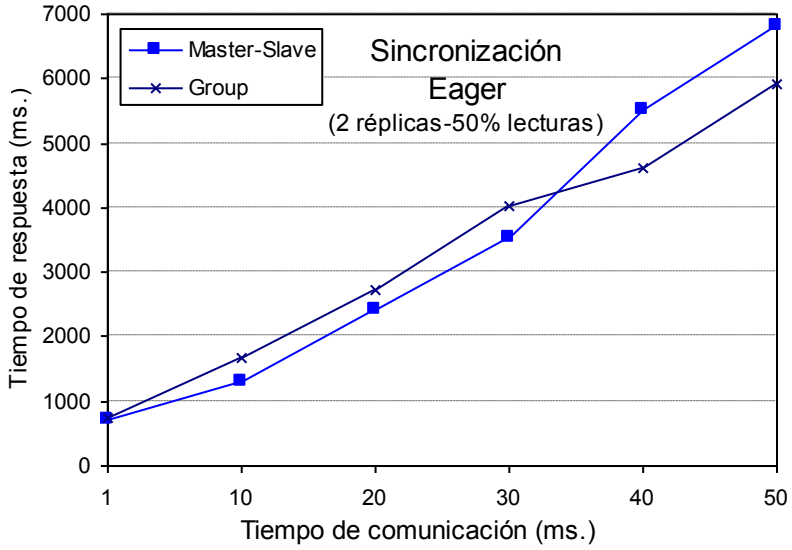


Figura 7.4: Sincronización en esquemas Eager

particular se encuentre bloqueado al momento de ser accedido (ver Figura 7.4.b).

Los resultados indican, que tanto al aumentar el tiempo de comunicación entre las máquinas como la probabilidad de bloqueo de los datos, los tiempos de respuesta aumentan significativamente, cualesquiera sea el esquema Eager empleado. Echo que no ocurre en los esquemas Lazy, ya que el usuario no debe esperar por su respuesta hasta que todos los datos sean actualizados y sincronizados adecuadamente.

7.3.5 Centralizando los datos

En las secciones anteriores comparamos las diferentes técnicas de replicación sobre esquemas de datos totalmente replicados (3 réplicas), parcialmente replicados (2 réplicas) y particionados (1 réplica). Finalmente, centralizamos todos los datos en una sola máquina y los comparamos con los casos anteriormente mencionados (ver Figura 7.5).

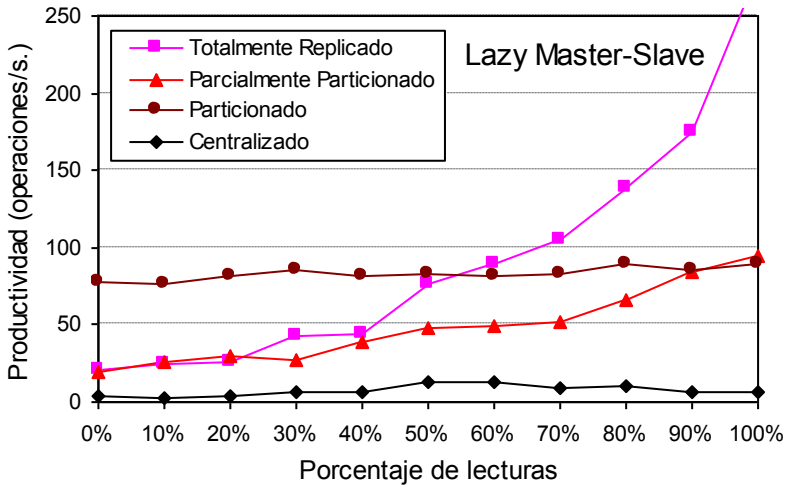


Figura 7.5: Performance para esquemas centralizados

Los modelos replicados otorgan mayor grado de disponibilidad que los sistemas de base de datos no replicados, y en particular un mejor balanceo de la carga que los centralizados. Sin embargo, tienen el procesamiento adicional de que cuando se actualiza una réplica, deben modificarse todas las otras para mantener la consistencia de los datos.

Los esquemas replicados presentan la mejor productividad ante altas tasas de lecturas, mas del 90% replicando los datos parcialmente y más del 55% con la replicación total. Este beneficio lo brinda la minimización del costo de las lecturas al poder manejarlas localmente y no implicar sobrecarga por comunicaciones.

El caso particionado nos da la mayor productividad ante altas tasas de actualizaciones, estas son más baratas y rápidas que en los casos centralizado y replicado debido a que el procesamiento de cualquier operación se realiza en una sola máquina que en general no se encuentra sobrecargada como en el caso centralizado, y no requiere ningún tipo de sincronización o actualizaciones extras como en los casos replicados.

Al centralizar los datos observamos una productividad muy baja con respecto a los casos particionado y replicados. Esto se debe a los altos costos asociados a los accesos remotos en términos de demora de comunicación y contención originada por un cuello de botella en la máquina que centraliza los datos.

Capítulo 8

Conclusiones

La *Replicación sobre Bases de Datos Distribuidas* es un tema que está aumentando día a día con respecto a su importancia. Los ambientes modernos de computación demandan soluciones innovadoras y mecanismos flexibles que puedan soportar diferentes formas de replicación, para lograr mejoras de disponibilidad y performance en sus sistemas de bases de datos.

Por razones de performance, la mayoría de los productos comerciales e investigaciones han direccionado hacia la utilización de esquemas Lazy o Asincrónicos, reduciendo por supuesto la posibilidad que en un futuro los productos puedan soportar replicación Eager. Sin embargo, esta mejora de performance, se ve empañada por la posibilidad de trabajar con datos inconsistentes, hecho que no ocurre en los esquemas Eager.

En este proyecto, hemos considerado y comparado cuatro esquemas de regulación y propagación de actualizaciones. Direccionamos el problema de actualización de réplicas de datos presentando un modelo para los esquemas de regulación (Group y Master-Slave) y propagación (Eager y Lazy). El comportamiento de esas estrategias fué analizado mediante experimentos prácticos mostrando que la decisión de replicar depende potencialmente de la proporción de lecturas y actualizaciones, pudiendo afectar casi todos los algoritmos y funciones de control de un DBMS distribuido.

8.1 Contribuciones

Las principales contribuciones de este Trabajo de Grado son:

Modelo de replicación

Se realizo un análisis y diseño de la modelización de un sistema en red con Bases de Datos Replicadas.

Este se definió, utilizando componentes que son comunes a todas las técnicas, con diferentes implementaciones en alguno de ellos según el esquema utilizado. Los componentes definidos son: Manager, Monitor, DataManager, MasterSender, MasterListener, SlaveSender, SlaveListener, Propagator y Receiver. El Manager, Monitor, DataManager, Propagator y Receiver necesitan ser adaptados para las propagación Lazy o Eager. Los demás componentes son iguales para todos los esquemas.

Evaluación de técnicas de tratamiento de actualizaciones de réplicas en BDD

Este modelo nos permitió comparar y estudiar las características de performance de cuatro técnicas de replicación basadas en diferentes esquemas de regulación y propagación de actualizaciones sobre réplicas.

Resultados de Performance

Entre los indicadores de performance, consideramos el tiempo de respuesta, la productividad y los conflictos de actualizaciones. Los experimentos se realizaron variando las proporciones de lecturas y escrituras realizadas en los mismos.

Los resultados muestran que la replicación Lazy tiene una mejor performance que los esquemas Eager y demuestra ser una opción con mayores posibilidades de escalabilidad.

Los esquemas replicados brindan mayor disponibilidad y balanceo de las cargas que los centralizados logrando una mayor productividad; y ante la presencia de proporciones bajas de actualizaciones, también otorgan una mejor performance que el caso particionado.

Todos los resultados nos llevan a concluir que el principal beneficio ganado en la replicación de datos está en la disponibilidad para realizar operaciones de lectura localmente, sin ningún tipo de comunicación. Incrementar el número de réplicas tiene el efecto negativo esperado sobre la performance ante altas tasas de actualizaciones y uno positivo ante intensivas lecturas.

Mayor experiencia

Mediante este estudio, se logro ganar experiencia en problemas de procesamiento distribuido con datos distribuidos y diferentes modelos de actualización.

8.2 Trabajos futuros

- Este trabajo mantiene consistencia floja sobre los datos para el esquema Lazy Master-Slave, y eventual para el Lazy Group, sería interesante experimentar grados de consistencias más fuertes entre los datos replicados. En el caso de los esquemas Eager, sería bueno implementar los protocolos de bloqueo que sincronizan la actualización de las réplicas.

- También se podría simular un escenario donde ocurran fallas. Una falla puede significar la caída de una o más réplicas y por consiguiente la posible suspensión repentina del procesamiento sobre los datos.

- Otra de las posibles sugerencias para futuros trabajos, con la posibilidad de mejorar este modelo sería variar el esquema de replicación que definimos de manera estática, a uno dinámico, que pudiera ir modificando el grado de replicación en base a los diferentes patrones de lectura/escritura, y así comparar las ventajas de estos esquemas según el tipo de regulación y propagación de actualizaciones utilizado. Por ejemplo cuando agregamos una nueva réplica, el nuevo esquema de la base de datos debe sincronizarse

con las demás réplicas para transferir el nuevo estado. El mecanismo utilizado para tratar con este problema, podría envolver una gran cantidad de sobrecarga en la red afectando la performance, pero a su vez podría mejorarla mediante una localización más cercana de los datos.

Apéndice A

Especificaciones de los Informes y la Herramienta Implementada

A.1 Descripción del informe generado en cada máquina

El esquema de un informe generado localmente por cualquier host que participa de una ejecución, consta de un encabezado, los detalles de las operaciones, el estado final de las réplicas y las recuperaciones de aquellas operaciones que tuvieron conflictos (ver Figura A.1).

```
B.D.D. ESTUDIO DE ACTUALIZACION DE REPLICAS DE DATOS.      Tue Jul 10 12:38:05 GMT-04:00
2001

REGULACION:  Master-Slave     PROPAGACION: Asincrónica.

HOST           ID:2    Dirección: 10.0.0.3    Port: 30000    Cantidad: 3

CONSULTAS      Cantidad: 500  Lecturas al: 80 %

REPLICAS       Cantidad: 99   Replic. al: 50 %

Master de las sig. Réplicas:
 2 5 8 11 14 17 20 23 26 29 32 35 38 41 44 47 50 53 56 59 62 65 68 71 74 77 80 83 86 89
 92 95 98

Slave de las sig. Réplicas:
 1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88
 91 94 97

HORA DE INICIO:              12:38:05.981

ID      TIPO    REPLICA DESTINO        INICIO         FINALIZ.       DURAC.  RESULTADO

0       Read    75      0-10.0.0.1     12:38:05.983   12:38:06.606   623     INI75
1       Read    93      0-10.0.0.1     12:38:06.017   12:38:06.817   800     1-12:32:03.008
2       Read    74      2-LOCAL        12:38:06.041   12:38:06.051   10      INI74
3       Read    26      2-LOCAL        12:38:06.050   12:38:06.056   6       INI26
4       Read    48      0-10.0.0.1     12:38:06.054   12:38:06.952   898     INI48
5       Read    50      2-LOCAL        12:38:06.064   12:38:06.071   7       INI50
6       Read    69      0-10.0.0.1     12:38:06.072   12:38:06.986   914     INI69
7       Read    46      2-LOCAL        12:38:06.075   12:38:06.079   4       INI46
8       Read    67      2-LOCAL        12:38:06.079   12:38:06.082   3       INI67
9       Update  69      0-10.0.0.1     12:38:06.081   12:38:07.001   920     2-12:38:06.081
10      Update  96      0-10.0.0.1     12:38:06.084   12:38:07.038   954     2-12:38:06.084
11      Update  11      2-LOCAL        12:38:06.090   12:38:06.095   5       2-12:38:06.090
12      Read    42      0-10.0.0.1     12:38:06.095   12:38:09.133   3038    INI42
13      Read    34      2-LOCAL        12:38:06.097   12:38:06.106   9       INI34
14      Update  98      2-LOCAL        12:38:06.107   12:38:06.109   2       2-12:38:06.107
15      Read    38      2-LOCAL        12:38:06.111   12:38:06.115   4       INI38
16      Read    74      2-LOCAL        12:38:06.117   12:38:06.123   6       INI74
17      Read    64      2-LOCAL        12:38:06.124   12:38:06.129   5       INI64

                                       . . .
```

```
                                        . . .

481     Read    14      2-LOCAL         12:38:12.300    12:38:12.306    6       INI14
482     Update  1       1-10.0.0.2      12:38:12.305    12:38:12.315    10      2-12:38:12.305
483     Read    25      2-LOCAL         12:38:12.310    12:38:12.326    16      0-12:09:57.942
484     Update  92      2-LOCAL         12:38:12.330    12:38:12.335    5       2-12:38:12.330
485     Read    40      2-LOCAL         12:38:12.346    12:38:12.350    4       2-12:38:11.594
486     Read    50      2-LOCAL         12:38:12.362    12:38:12.367    5       2-12:38:11.573
487     Read    49      2-LOCAL         12:38:12.368    12:38:12.371    3       INI49
488     Update  17      2-LOCAL         12:38:12.374    12:38:12.379    5       2-12:38:12.374
489     Read    61      2-LOCAL         12:38:12.383    12:38:12.393    10      1-12:32:02.993
490     Read    41      2-LOCAL         12:38:12.394    12:38:12.400    6       1-12:32:03.244
491     Read    17      2-LOCAL         12:38:12.402    12:38:12.409    7       2-12:38:12.374
492     Read    24      0-10.0.0.1      12:38:12.412    12:38:12.437    25      1-12:32:03.657
493     Read    95      2-LOCAL         12:38:12.418    12:38:12.423    5       2-12:38:06.494
494     Read    90      0-10.0.0.1      12:38:12.423    12:38:12.454    31      2-12:38:11.377
495     Read    61      2-LOCAL         12:38:12.427    12:38:12.435    8       1-12:32:02.993
496     Read    83      2-LOCAL         12:38:12.437    12:38:12.439    2       2-12:38:10.448
497     Update  36      0-10.0.0.1      12:38:12.441    12:38:12.489    48      2-12:38:12.441
498     Read    40      2-LOCAL         12:38:12.449    12:38:12.458    9       2-12:38:11.594
499     Read    16      2-LOCAL         12:38:12.462    12:38:12.464    2       2-12:38:11.775

HORA DE FINALIZACION: 12:38:16.037

DURACION DE LA EJECUCION:    10056 ms.


ESTADO FINAL DE LAS REPLICAS LOCALES:

REPLICA         VALOR                   TIMESTAMP

0               NO REPLICADO            0-0
1               0-12:10:02.830          1-158
2               2-12:38:11.162          2-97
3               NO REPLICADO            0-0
4               2-12:38:11.569          1-104
5               0-12:10:01.045          2-139
6               NO REPLICADO            0-0
7               2-12:38:11.383          1-100
8               2-12:38:10.140          2-82
9               NO REPLICADO            0-0
10              0-12:09:59.369          1-120


                  . . .


89              2-12:38:11.565          2-104
90              NO REPLICADO            0-0
91              0-12:10:02.179          1-149
92              2-12:38:12.330          2-115
93              NO REPLICADO            0-0
94              0-12:10:02.046          1-146
95              2-12:38:06.494          2-25
96              NO REPLICADO            0-0
97              0-12:09:59.717          1-128
98              2-12:38:10.872          2-91

CONSULTAS A RECUPERAR:

REPLICA TS LOCAL        VALOR LOCAL     OLD_TS_PROP     NEW_TS_PROP     VALOR PROP


CANT. TOTAL DE CONSULTAS A RECUPERAR: 0
```

A.1 Informe generado en cada una de las máquinas

A.1.1 Encabezado

En el encabezado tenemos los siguientes campos:

REGULACION: tipo de regulación empleada (Master-Slave/Group).

PROPAGACION: tipo de propagación empleada (Asincrónica/Sincrónica).

HOST: datos del host local.
ID: identificador. Corresponde a un entero entre 0 y n-1, donde n es el nro. total de máquinas.
Dirección: dirección IP del host.
Port: port a través del cual el MasterListener recibirá los requerimientos remotos.
Cantidad: cantidad de máquinas que participan en el experimento.

CONSULTAS
Cantidad: cantidad total de consultas a ejecutar entre todas las máquinas.
Lecturas al: porcentaje de lecturas que se ejecutaron en este experimento.

REPLICAS
Cantidad: cantidad total de datos replicados.
Replic. al: porcentaje de replicación.

Master de las sig. Réplicas: identificadores de las réplicas que son master en este host.
Slave de las sig. Réplicas: identificadores de las réplicas que son slave en este host.

HORA DE INICIO: hora de inicio de la ejecución de las operaciones.

A.1.2 Detalles de las operaciones

Datos origen de la operación

ID: id de la operación a realizar.

TIPO: indica el tipo de operación a realizar, puede ser de lectura (Read) o de actualización (Update).

REPLICA: id de la réplica sobre la cual se realiza.

DESTINO: host al cual se le requieren los datos. Si es una operación remota figura el id del host remoto mas su dirección por ej: 1-10.0.0.2, en cambio si es una operación local, en vez de la dirección se indicará como LOCAL Ej. : . 2-LOCAL.

INICIO: hora de inicio de su ejecución.

Datos finales de la operación

FINALIZ.: hora de finalización de su ejecución.

DURAC.: duración de su ejecución en ms.

RESULTADO: su valor, es decir, en caso de realizar una lectura se muestra el valor leído sobre la réplica, y si es una actualización se muestra el valor con el cual se realizó tal actualización. Estos valores estarán formados por el id del host más la hora en la cual se generó su última actualización (EJ: 0-12:09:57.942), salvo, que se lea una réplica que nadie haya actualizado, con lo cual retorna su valor de inicialización formado por INI + id de la réplica (Ej.: INI49).

Una vez finalizado los detalles de cada una de las operaciones indicamos la hora de finalización de la ejecución y su duración en ms.:

HORA DE FINALIZACION: 12:38:16.037

DURACION DE LA EJECUCION: 10056 ms.

A.1.3 Estado final de las réplicas locales

Esta información es útil para comprobar que las ejecuciones en los esquemas Lazy terminaron bajo un estado consistente de las réplicas. En ella tenemos:

REPLICA: el id de la réplica.

VALOR: valor final de actualización de la réplica. Si una réplica no está replicada localmente, su valor figura como NO REPLICADO, de lo contrario muestra el id del host y la hora en que se generó la actualización de la réplica.

TIMESTAMP: timestamp de la última actualización de la réplica.

A.1.4 Recuperaciones en los esquemas Lazy

En el informe anterior vemos que no hay conflictos de actualización por ser un ejemplo del esquema Lazy Master-Slave. Ahora mostramos los detalles de los conflictos de actualización ocurridos para el caso de una prueba del esquema Lazy Group (ver Figura A.2):

```
CONSULTAS A RECUPERAR:

REPLICA  TS LOCAL      VALOR LOCAL           OLD_TS_PROP    NEW_TS_PROP    VALOR PROP

33       1-23          1-11:34:06.817        0-0            0-5            2-11:40:09.553
84       1-49          1-11:34:07.175        0-0            0-9            0-11:11:55.949
18       1-42          1-11:34:07.099        0-0            0-16           0-11:11:56.237
72       1-35          1-11:34:06.925        1-1            0-17           0-11:11:56.265
57       1-32          1-11:34:06.870        0-0            0-21           0-11:11:56.330

CANT. TOTAL DE CONSULTAS A RECUPERAR: 5
```

A.2 Informe de conflictos de actualización

Entre estos detalles tenemos:

Datos de la réplica

REPLICA: el id de la réplica local en la cual se produjo un conflicto de actualización.

TS LOCAL: el timestamp local de la réplica.

VALOR LOCAL: el valor local de la réplica antes de producirse el conflicto.

Datos de la propagación

OLD_TS_PROP: valor del timestamp de la réplica remota antes de suceder su actualización.

NEW_TS_PROP: valor del timestamp de la réplica remota después de producirse su actualización.

VALOR_PROP: valor de la actualización producida en la réplica remota.

A.2 Descripción del informe generado en la máquina inicializadora

```
Host: 10.0.0.2    Dur.de Ejec.: 9905 ms.    Cant. Conflic.:  5
Host: 10.0.0.3    Dur.de Ejec.: 9803 ms.    Cant. Conflic.:  9
Host: 10.0.0.1    Dur.de Ejec.: 8794 ms.    Cant. Conflic.: 13

Tiempo de Respuesta: 587 ms.

Throughput: 53 op./seg.

Cantidad Promedio de conflictos:  9
```

A.3 Infome generado en la máquina inicializadora

El informe generado en la máquina inicializadora, está formado por la siguiente información de cada uno de los host: dirección, duración de la ejecución y cantidad de conflictos ocurridos. Luego de ésto se indica el tiempo de respuesta, la productividad y la cantidad promedio de conflictos.

A.3 Almacenamientos de datos permanentes

A.3.1 Queries

Las *Queries* consisten de archivos Query[n].txt donde n indica el porcentaje de lecturas que contendrá este patrón de acceso. Estos archivos son generados mediante una aplicación GenQuery (Args = CantQueries, PorcLecturas, CantDatos) que permite generar un archivo con CantQueries operaciones, de las cuales el PorcLecturas % son operaciones de lectura, todas de forma aleatoria y sobre datos que van desde 0...CantDatos - 1.

La estructura generada tiene la siguiente forma:

- Cantidad de Queries n (Int)
- Operación 1(char) Réplica i(Int)... Operación n (Char) Réplica j(Int)
  $i,j \in 0..$ CantDatos - 1

donde se indica en primer lugar la cantidad de consultas que tiene el archivo. En segundo lugar, se detallan el tipo de operación a realizar (´R´=read, ´U´=update) y sobre que réplica.

Para el caso de las actualizaciones el nuevo valor del dato a actualizar será creado por el Manager y consistirá de: Id – Ts, donde Id es identificador de su host, y Ts es un Timestamp con la hora de inicio de la operación, por ej. 0-12:09:57.942.

Ubicación de los archivos: ../Consultas/

A.3.2 Results

Results, también es implementado con simples archivos de texto denominados R[tr][pr][pl].txt e I[tr][pr][pl].txt donde tr es el tipo de regulación (M = Master, G = Group, C=Centralizado), pr es el porcentaje de replicación de los datos y pl el porcentaje de lecturas para un experimento determinado. Aquellos que comienzan con R son los informes de resultados generados en la máquina inicializadora, y los iniciados con I son los generados en cada uno de las máquinas localmente. Ej: RM5070.txt y IM5070.txt son generados durante un experimento Master-Slave con 50% de los datos replicados y un 70% de lecturas.

Ubicación de los archivos: ../Informes/

A.3.3 Pesos de Comunicaciones

Es un archivo de texto (PesoComunic.txt) que representa una matriz de n x n donde n, es el número de host, y cada [i, j] es el peso que existe entre el host i y el host j. Este peso será un dígito del 0 al 9. Por ejemplo la matriz para tres host, donde el peso entre los host 0 y 1 es 4, entre el 0 y 2 es 6, y entre el 1 y 2 es 8 es la siguiente:

```
0 4 6
4 0 8
6 8 0
```

PesoComunic.txt en cada host

Ubicación del archivo: ../DatosHost/

A.3.4 Direcciones, Ports e Id de los Host

Este archivo (AddrPortHost.txt) contiene las direcciones, id de los host y ports a través de los cuales recibe requerimientos su MasterListener. Una configuración posible sería:

```
Address       Port
10.0.0.01 20000 00
10.0.0.02 25000 XX
10.0.0.03 30000 XX
```

AddrPortHost.txt en el Host **0**

```
Address       Port
10.0.0.01 20000 XX
10.0.0.02 25000 01
10.0.0.03 30000 XX
```

AddrPortHost.txt en el Host **1**

```
Address       Port
10.0.0.01 20000 XX
10.0.0.02 25000 XX
10.0.0.03 30000 02
```

AddrPortHost.txt en el Host **2**

Ubicación de los archivos: ../DatosHost/

A.3.5 Esquema de los datos replicados

Los *Esquemas de Datos* consisten de archivos LRep[tipo][%rep].txt de distribuciones de réplicas con carga balanceada sobre los host a partir de un % de replicación donde:

[tipo] = G (Group), M (MasterSlave)
[%rep] = 10, 20...100

Estos archivos son generados mediante una aplicación GenRepliBal (Args = CantHost, CantDatos, PorcReplicación) que permite generar un archivo para CantHost máquinas, sobre datos que van desde 0...CantDatos-1, y un porcentaje de replicación PorcReplicación. Este programa genera en cada ejecución un archivo para la regulación Master-Slave y otro para la ejecución Group.

La estructura generada tiene la siguiente forma:

- Cantidad de Réplicas n (Int)
- Réplica 1 (Int) Host 1 (Int) Propiedad (Char) ... Réplica 1 (Int) Host n (Int) Propiedad (Char)
  ....
  Réplica n (Int) Host 1(Int) Propiedad (Char) ... Réplica n (Int) Host n (Int) Propiedad (Char)

donde se indica en primer lugar la cantidad de réplicas que tiene el archivo. En segundo lugar, se detallan para cada par de (réplica, host) que propiedad tiene ese host sobre tal réplica (´M´=master, ´S´=slave), en el caso de no tenerla replicada figura como propiedad ´-´

De la misma forma existe un programa GenRepliCen que genera el esquema de datos para centralizar los datos en una sola máquina. El archivo generado es LRepC0.txt.

Ubicación de los archivos: ../EsqDatos/

Distribución de los datos

Los datos replicados se distribuyeron de manera balanceada entre las diferentes máquinas. Teniendo n sitios o máquinas Si, 0<=i<=n-1, y m datos Dj, 0 <=j<=m-1. Si tenemos que distribuir el dato Dj con r réplicas Djr con r<=n, hacemos:

La 1º réplica Dj0 es almacenado en el sitio S(j mod n)

La 2º réplica Dj1 es almacenada en el sitio S(j mod n) y S((j+1) mod n)

La 3º réplica se agrega una copia Dj2 en el sitio S((j+2) mod n)

...

La rº réplica se agrega una copia Dir-1 en el sitio S((j+r-1) mod n)

A.3.6 Archivos temporales para la generación del informe

Existen cuatro archivos temporales que se utilizan para generar el informe local de cada host, ellos son:

- TempIni.txt: archivo generado por el Monitor para almacenar todos los datos iniciales de las consultas generadas.

- TempFin.txt: archivo generado por el Monitor para almacenar todos los datos finales de las consultas generadas.

- DataFile.txt: archivo con el estado final de cada una de las réplicas. Es generado por el DataManager para ser posteriormente leído por el Monitor.

- Recovery.txt: archivo con los detalles de todos los conflictos de actualización ocurridos. Es generado por el Recovery para ser posteriormente leído por el Monitor.

Ubicación de los archivos: ../Temp/

A.4 Carga e inicialización

Una vez generados los archivos de queries, del esquema de replicación, peso de las comunicaciones, direcciones y port de cada host, podemos ejecutar el proceso principal en cada maquina que es el *Manager,* quedando cada uno de ellos a la espera del mensaje de inicio. Para esto, se debe ejecutar el proceso Init (en lo posible en una máquina a parte) el cual se encargara de enviarles tal mensaje para luego esperar los resultados de las ejecuciones. Sus argumentos son:

Tipo de regulación Es un carácter: M para indicar la regulación Master-Slave, G para el método Group o C para el caso Centralizado.

Porcentaje de replicación Es un entero que indica el porcentaje de replicación del experimento a realizar.

Cantidad de consultas Es un entero que indica la cantidad de operaciones que se ejecutarán en cada host durante experimento.

Porcentaje de lecturas Es un entero que indica el porcentaje de lecturas del experimento a realizar.

Por ejemplo al ejecutar Init (M, 50, 100, 90) cada Manager buscara en su localidad los archivos LRepM50.txt y Query90.txt.

Para los experimentos Eager el proceso Inicializador tiene un 5to. parámetro:

Probabilidad de Bloqueo Es un entero que determina la probabilidad que un dato esté bloqueado cuando se quiera acceder al mismo.

Apéndice B

Código Fuente de la Herramienta Implementada

B.1 Procesos generales

B.1.1 MasterListener.java

```
/**************************************************************************
 *   Es el proceso responsable de recibir los requerimientos
*
 *   provenientes de un   MasterSender remoto, y distribuirlos a los                                    *
 *   diferentes SlaveSender para que realicen dichos requerimientos                                    *
 **************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;

public class MasterListener extends Thread {

  private static final int TIME_WAITING = 60000; //   tiempo maximo (ms.) de
                                                 // espera por un requerimiento
  private Manager myManager = null; //   para comunicarme con el Manager

  public MasterListener(Manager myManager) {
    this.myManager = myManager;
    this.start();
  }



  public synchronized void run() {

    Socket sMListenIn = null;       //   para   comunicarse con el MasterSender
    DataInputStream in = null;      //   para leer de sMListenIn
    ServerSocket listenSocket = null;     // para escuchar los requerimientos
    SlaveSender slvSender = null; //   para procesar un requerimiento remoto
    int myPort; //   pare crear listenSocket

   int myIdHost = myManager.getIdHost();      myPort = myManager.getPortList(myIdHost);
    try {
      // creo el Server Socket para recibir los requerimientos

      try { listenSocket = new ServerSocket(myPort);
            listenSocket.setSoTimeout(TIME_WAITING);}
      catch (IOException e) {System.out.println( "Excepción ocurrida creando "
                                + "MasterListener "); }

      System.out.println("MasterListener: esperando señal para iniciar "
                           + "en PORT: "+ myPort);
```

```
//   espera la "señal" del proceso Init para notificar al Manager que
//   estan todos los MasterListener listos y puede comenzar a
//   procesar las consultas

Socket sInitIn = listenSocket.accept();
sInitIn.close();
sInitIn = null;
synchronized (myManager) { //   notifica al Manager para que comience a
     myManager.notify();     //   procesar las consultas
}

// Por cada iteración, recibe un requerimiento de un MasterSever y
// crea un nuevo SlaveServer (si es necesario), y le entrega el
// nuevo requerimiento remoto

while (true) {
  try {
     System.out.println("MasterListener: esperando ACCEPT en PORT: "+
     myPort);

     //   acepta las conecciones de los MasterSender
     sMListenIn = listenSocket.accept();
     in = new DataInputStream(sMListenIn.getInputStream());

     //   determino los datos de la consulta remota
     //   idConsulta + port + réplica + operación + nuevovalor
     String hostAddr = sMListenIn.getInetAddress().getHostName();
     int idQuery = in.readInt(); //   Id de la consulta remota
     int idHost = in.readInt();      // Id del Host del SlaveListener Remoto que
     int port = in.readInt();      //   Port del SlaveListener Remoto que
                                   //   espera los resultados
     int replica = in.readInt(); //   Réplica sobre la cual se realiza la
                                   //   consulta
     char op = in.readChar();      //   Tipo de operación a realizar
     String newValor = in.readLine();   //   Valor de la operación
                                        //   (para las actualizaciones)

     //   creo un nuevo mensaje con dicha información para enviarsela
     //   a un SlaveSender que se encargue de dicha tarea
     SrvMessage smsg = new SrvMessage(idQuery,idHost,hostAddr,port,replica,op,newValor);
     System.out.println("MasterListener: Procesando Consulta "+
     " Remota " + smsg.getIDQuery() + " del HOST "+ smsg.getIdHost()+"-"+smsg.getAddrHost());



     // controlo si ya tengo un SlaveSender manejando esa réplica
     slvSender =null;
     slvSender = this.myManager.srvSlv.getSlvSend(smsg.getIDRepli());
     if (slvSender == null)   {
          //   creo un nuevo SlaveSender que maneja esa réplica
          slvSender = new SlaveSender(myManager);
          // lo agrego a la tabla de SlaveSender Activos con su réplica
          this.myManager.srvSlv.putSlvSend(smsg.getIDRepli(),slvSender);
     }
     //   le entrego al SlaveSender el nuevo requerimiento y lo notifico
```

```
            //   de esto
            slvSender.putMsg(smsg);
            in.close();
            in = null;
            sMListenIn.close();
            sMListenIn = null;
          } // try
          catch (IOException e) { break;} //   se cumplio TIME_WAITING
        } //while
        listenSocket.close();
        listenSocket = null;
        //   notificar al Manager que no he recibido requerimientos
        //   desde hace TIME_WAITING ms.
        synchronized (this.myManager) {
                this.myManager.notify();
        }
        System.out.println("FIN MasterListener");
      } // try
      catch (IOException e) { System.out.println("MasterListener: "+ e);}
    } //run
}
```

B.1.2 MasterSender.java

```
/**************************************************************************
 *   Es el proceso responsable recibir las consultas que le genera el                    *
 *   Manager y delegar dicha tarea a un SlaveSender si es una consulta                *
 *   local, y si es remota, enviarsela a un MasterListener remoto                        *
 **************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;

public class MasterSender extends Thread {

   private Vector mSendMsg = new Vector(20,20);
      // cola de mensajes recibidos del Manager
      // sobre un requerimiento de datos (Query)
   private Manager myManager = null; //   para comunicarme con el Manager

   public MasterSender(Manager myManager) {
      this.myManager = myManager;
      this.start();
   }

   //   agrega un nuevo mensaje a la cola
   public synchronized void putMsg(SrvMessage msg) {
         this.mSendMsg.addElement(msg);
         this.notify();
   }
   // retorna y elimina el primer mensaje de la cola
   private SrvMessage removeMsg() {
       SrvMessage smsg = (SrvMessage) mSendMsg.firstElement();
```

```
    mSendMsg.removeElementAt(0);
    return smsg;
}

public synchronized void run() {
  // para asignar un port a un SlaveListener
  int lastPortSrv = myManager.getPortList(myManager.getIdHost())+1;
  Socket sMSenderOut;   // para   comunicarse con el MasterListener
  DataOutputStream out;   //para escribir en sMSenderOut

  SlaveListener slvListen = null; //   para esperar resultados remotos
  SlaveSender slvSender = null;    //   para procesar un requerimiento local

  while(true) {
    try {
      if (mSendMsg.isEmpty()) { //   no hay consultas en la cola
          try {
              System.out.println("MasterSender: en espera de consultas...");
              wait(); //   espero hasta que me envíen una consulta
          }              //   o me notifiquen que termine
          catch (InterruptedException e) {;}
      }
      //   recupero el primer mensaje y lo elimino de la cola
      SrvMessage smsg = this.removeMsg();
      if (smsg.getTipo() == SrvMessage.END){break;}   // no hay mas pedidos
      //   proceso el requerimiento
      if (smsg.getIdHost() == myManager.getIdHost()) { //consulta local
          System.out.println("MasterSender: Procesando Consulta Local: "
                                  + smsg.getIDQuery() );
          // controlo si ya tengo un SlaveSender manejando esa réplica
          slvSender = null;
          slvSender =this.myManager.srvSlv.getSlvSend(smsg.getIDRepli());
          if (slvSender == null)   {
              //creo el nuevo SlaveSender.
              slvSender = new SlaveSender(myManager);
              // lo agrego a la tabla de SlaveSender Activos con su réplica
              this.myManager.srvSlv.putSlvSend(smsg.getIDRepli(),slvSender);
          }
          //   le entrego al SlaveSender el nuevo requerimiento y lo notifico
          //   de esto
          slvSender.putMsg(smsg);
      }
      else { // consulta remota
          while (true) {   //   itero hasta que pueda enviar la consulta
                           //   al MasterListener remoto
              out = null;
              sMSenderOut = null;
              try {
                  System.out.println("MasterSender: Procesando Consulta Remota "
                  + smsg.getIDQuery() );
                  //   creo un socket para comunicarme con el MasterListener
                  sMSenderOut = new Socket(smsg.getAddrHost(),smsg.getPort());
                  out = new DataOutputStream(sMSenderOut.getOutputStream());

                  // controlo si ya tengo un SlaveListner manejando esa réplica
                  slvListen = null;
```

```
                    slvListen =this.myManager.srvSlv.getSlvList(smsg.getIDRepli());
                    int portSlv;   //   para el port del SlaveListener
                    if (slvListen == null)   {
                        //   creo un SlaveListener para que espere el resultado
                        //   si es una operacion READ o un Reply del UPDATE
                        portSlv = lastPortSrv++;   //   obtengo el port del SlaveListener
                        slvListen = new SlaveListener(portSlv,this.myManager.monitor);
                        // lo agrego a la tabla de Slave Activos con su réplica
                        this.myManager.srvSlv.putSlvList(smsg.getIDRepli(),slvListen);
                    }
                    else {portSlv = slvListen.getPort();}

                    //   envío la consulta con el port al cual se debe comunicar
                    //   su SlaveSender remoto con el SlaveListener local.
                    out.writeInt(smsg.getIDQuery());          //   id de la consulta
                    out.writeInt(myManager.getIdHost());          //   id del host
                    out.writeInt(portSlv );        //   port del SlaveListener
                    out.writeInt(smsg.getIDRepli()); //   réplica sobre la que se realiza
                    out.writeChar(smsg.getTipo());     //   tipo de op.(read o update)
                    out.writeBytes(smsg.getMsg());     //   nuevo valor de la op. (update)
                    out.writeByte('\n');
                    // realizo el envío
                    out.flush();
                    out.close();
                    sMSenderOut.close();
                    break;
                } //   try
                catch (SocketException e) {
                    // no pude comunicarme con el MasterListener
                    try {wait(0,1);}   // espero 1 nseg.
                    catch (InterruptedException e2) {;}
                    System.out.println("MasterSender: no puedo comunicarme con el MasterListener "
                    +smsg.getIdHost()+"-"+smsg.getAddrHost());
                }
                finally {
                    if (sMSenderOut != null) {
                        sMSenderOut.close();
                        sMSenderOut = null;
                    }
                    if (out != null) {
                        out.close();
                        out = null;
                    }
                } //   finally

            } //   while
          } // else consulta remota
        }//try
        catch (IOException e) { System.out.println("MasterSender "+e); }

    }// while
    System.out.println("FIN MasterSender");
  }
}
```

B.1.3 SlaveListener.java

```
/**************************************************************************
 *   Es el proceso responsable de recibir los resultados de las           *
 *   consultas remotas y enviarselos al Monitor                           *
 **************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.Timestamp;

class SlaveListener extends Thread {

   private int myPort;           // para recibir los resultados remotos
   private Monitor monitor;   //   para retornar los resultados de las consultas remotas
   private static final int TIME_WAITING = 60000; //   tiempo maximo (ms.) de

                                                  // espera por un requerimiento

   public SlaveListener(int port, Monitor m) {
        myPort = port;
        monitor = m;

        this.start();
   }

     //   retorna el port
    public int getPort(){
        return   this.myPort;
    }

   public synchronized void run() {

     ServerSocket listenSocket = null;     // para recibir los resultados
     Socket sSlaveIn = null;     // para   comunicarse con el SlaveSender remoto
     DataInputStream in = null;      //   para leer de sSlaveIn

     try{
       // creo el Server Socket para recibir los resultados
       try { listenSocket = new ServerSocket(myPort);
             listenSocket.setSoTimeout(TIME_WAITING);}
       catch (IOException e) { System.out.println("Excepción ocurrida creando " +
                               "SlaveListener: "+myPort ); }
       while (true) {
         try {
           System.out.println("Slave Listener: esperando ACCEPT en PORT: "+myPort);
           //   acepta las conecciones de los SlaveSender
           sSlaveIn = listenSocket.accept();
           in = new DataInputStream(sSlaveIn.getInputStream());
           // tiempo en el cual recibi los resultados
           Timestamp ts = new Timestamp(System.currentTimeMillis());
           // determino los datos de la consulta remota
           //   replica + nuevovalor
           int idQuery = in.readInt();
```

```
            String valor = in.readLine();
            //mando los resultados al monitor
            MsgMonitor msg = new MsgMonitor(idQuery,valor,ts);
            monitor.putMsg(msg);
            sSlaveIn.close();
            sSlaveIn = null;
            in.close();
            in = null;
          }
          catch (IOException e) {break;}
        } // while
        listenSocket.close();
        listenSocket = null;
        System.out.println("FIN SlaveListener " + myPort);
      } // try
      catch (IOException e) { System.out.println("SlaveListener: "+ e);}

    }//run

}
```

B.1.4 SlaveSender.java

```
/**************************************************************************
 *   Es el proceso responsable de recibir las consultas sobre una                    *
 *   réplica particular, enviadas por el MasterSender (consulta local) o          *
 *   el MasterListener (consulta remota), comunicarse con el DataManager        *
 *   para realizar la consulta y retorna los resultados a un
*
 *   SlaveListener remoto (consulta remota), o al Monitor (consulta local)        *
 **************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.Timestamp;

class SlaveSender extends Thread {

   private Vector slvSenderMsg = new Vector(5,5);
//   cola de mensajes recibidos
   // del MasterSender o el MasterListener sobre un requerimiento de datos
   private Manager myManager = null; //    comunicarme con el Manager para:
   // realizar las consultas sobre una réplica determinada al DataManager
   // retornar los resultados de las consultas locales al Monitor
   // obtener los pesos de comunicación

   private final static int TIME_WAITING = 60000; //   tiempo maximo (ms.) de
                                                  // espera por un   requerimiento

   public SlaveSender(Manager myManager) {
          this.myManager = myManager;
          this.start();
   }
   //   agrega un nuevo mensaje a la cola
```

```
public synchronized void putMsg(SrvMessage msg) {
     this.slvSenderMsg.addElement(msg);
     this.notify();
}
// retorna y elimina el primer mensaje de la cola
private SrvMessage removeMsg() {
    SrvMessage smsg = (SrvMessage) slvSenderMsg.firstElement();
    slvSenderMsg.removeElementAt(0);
    return smsg;
}


public synchronized void run() {

  Socket sSlaveOut; //   para   comunicarse con el SlaveListener a quien hay
                    //   que enviarle los resultados de una consulta remota
  DataOutputStream out;      //para escribir en sSlaveOut
  SrvMessage myMsg; // para la consulta sobre los datos locales
  Timestamp ts;       //   para informarle al monitor en que instante
                    //   se realizo la consulta

  try {
    while(true) {
        if (slvSenderMsg.isEmpty()) {   //   no hay consultas en la cola
            try { System.out.println("SlaveSender: en espera de consultas...");
                  wait(TIME_WAITING);   //   espero hasta que me envíen una consulta
            }                           //   o me notifiquen que termine
            catch (InterruptedException e) {;}
            if (slvSenderMsg.isEmpty()) {break;} // fin de proceso
        }
        //   recupero el primer mensaje y lo elimino de la cola
        myMsg = this.removeMsg();
        String valor = new String(" ");
        if (myMsg.getTipo() == myMsg.READ) { //operación de lectura
            valor = myManager.dataM.get(myMsg.getIDRepli());
            // tiempo de finalización de la lectura
            ts = new Timestamp(System.currentTimeMillis());
        }
        else {   // operación de escritura
                myManager.dataM.putL(myMsg.getIDRepli(),myMsg.getMsg());
                // tiempo de finalización de la escritura
                ts = new Timestamp(System.currentTimeMillis());
                valor = myMsg.getMsg();   //   reply
        }

      if (myMsg.getIdHost() == myManager.getIdHost()) { // consulta local
          // retorno los datos al Monitor
          // id de cosulta, su valor y timestamp en que se realizó
          MsgMonitor msg = new MsgMonitor(myMsg.getIDQuery(),valor,ts);
          myManager.monitor.putMsg(msg);
       }
       else { //   consulta remota
          // retorno los datos al SlaveListener remoto
          while (true) {   //   itero hasta que pueda comunicarme con
                           //   al SlaveListener remoto
              out = null;
```

```
                sSlaveOut = null;
                try {
                  sSlaveOut = new Socket(myMsg.getAddrHost(),myMsg.getPort());
                  out = new DataOutputStream(sSlaveOut.getOutputStream());
                  out.writeInt(myMsg.getIDQuery()); //   id de la consulta
                  out.writeBytes(valor);   //   valor de una lectura o reply de
                  out.writeByte('\n');      //   de una escritura
                  out.flush();
                                                        out.close();
                  sSlaveOut.close();
                  break;
                }
                catch (SocketException e) {
                     // no pude comunicarme con el SlaveListener
                     try {wait(0,1);}   // espero 1 nseg.
                     catch (InterruptedException e2) {;}
                     System.out.println("SlaveSender: no puedo comunicarme con el SlaveListener "
                     +myMsg.getIdHost()+" "+myMsg.getPort());
                }
                finally {
                     if (sSlaveOut != null) {
                          sSlaveOut.close();
                          sSlaveOut = null;
                     }
                     if (out != null) {
                          out.close();
                          out = null;
                     }
                } //   finally

             } //   while
          }   //   else
       } //while
       System.out.println("FIN SlaveSender ");
    } //   try
    catch (IOException e) { System.out.println("SlaveSender: "+ e); }
  } //   run
}
```

B.1.5 SrvSlave.java

```
/**************************************************************************
 *   Es el proceso utilizado como servidor de procesos esclavos, ya sean                    *
 *   SlaveSenders o SlaveListener, que iterectúan sobre una réplica                            *
 *   determinada
*
 *************************************************************************/

import java.io.*;
import java.util.*;

public class SrvSlave extends Thread{

   private Hashtable mySlvSend = new Hashtable(5); //   Tabla de los SlaveSenders
```

```
  // locales activos, con Key = nroReplica y Value = SlaveSender
  private Hashtable mySlvList = new Hashtable(5);//   Tabla de los SlaveListeners
  // locales activos, con Key = nroReplica y Value = SlaveListener

  public SrvSlave() {
        this.start();
  }

  //   agrega un nuevo SlaveSender a cargo de la réplica idRep
  public synchronized void putSlvSend(int idRep, SlaveSender slvSender) {
       mySlvSend.put(new Integer(idRep),slvSender);
  }

  // retorna el SlaveSender que trabaja con la réplica idRep
  public SlaveSender getSlvSend(int idRep) {
      return (SlaveSender) mySlvSend.get(new Integer(idRep));
  }

  //   agrega un nuevo SlaveListener a cargo de la réplica idRep
  public synchronized void putSlvList(int idRep, SlaveListener slvListener) {
       mySlvList.put(new Integer(idRep),slvListener);
  }

  // retorna el SlaveListener que trabaja con la réplica idRep
  public SlaveListener getSlvList(int idRep) {
      return (SlaveListener) mySlvList.get(new Integer(idRep));
  }

  public synchronized void run() {
    try {
      wait();
    }
    catch (InterruptedException e) {System.out.println("FIN SrvSlave");}
  }
}
```

B.1.6 MsgMonitor.java

```
/**************************************************************************
 *   Etructura de los mensajes enviados al Monitor para indicarle los                                *
 *   datos finales de la consulta (id + hora finalización + resultado)                                *
 **************************************************************************/

public class MsgMonitor {

  private int idQuery;   //   para el id de la consulta
  private java.sql.Timestamp ts;   //   para la hora de finalización
  private String valor; //   para el resultado

  // crea un nuevo mensaje
  public MsgMonitor(int id, String valor,java.sql.Timestamp ts) {
       this.idQuery = id;
       this.valor = valor;
       this.ts = ts;
  }
```

```
  //   retorna el id de la consulta
  public int getIDQuery(){
       return   this.idQuery;
  }

  //   retorna el tiempo de finalización de la consulta
  public java.sql.Timestamp getTime(){
       return   this.ts;
  }

  //   retorna el valor de la consulta
  public String getValue(){
       return   this.valor;
  }
}
```

B.1.7 SrvMessage.java

```
/*************************************************************************
 *   Etructura de los mensajes utilizados para enviar datos sobre una                    *
 *   consulta, tales como id de la consulta, host y port destino (donde                  *
 *   se efectuara la misma, sobre que réplica y el tipo y valor de la                      *
 *   operación a realizar
*
 *************************************************************************/

public class SrvMessage {
  private int id;            // nro. que identifica la consulta
  private int idHost;        // id del host al cual se le enviará el mensaje
  private String addrHost;   // dirección del host al cual se le enviará el mensaje
  private int      port;     // port a través del cual se le enviará
  private int      replica; // mro. de réplica sobre la que se hará
                            // la consulta
  private char tipo;   // tipo de mensaje
  private String msg;      // datos del mensaje

  //   constantes de los posibles tipos de operaciones
  static final char READ = 'R';
  static final char UPDATE = 'U';
  static final char END = 'E';

  //   crea un nuevo mensaje
  public SrvMessage(int id,int idHost,String addrHost,int port,int replica,char messageId,String message) {

       this.id = id;
       this.idHost = idHost;
       this.addrHost = addrHost;
       this.port   = port;
       this.replica = replica;
       this.tipo = messageId;
       this.msg    = message;
  }

  //   retorna el Id de la consulta
```

```
  public int getIDQuery(){
       return   this.id;
  }

  //   retorna el Id del host destino
  public int getIdHost(){
       return   this.idHost;
  }

  //   retorna la dirección del host destino
  public String getAddrHost(){
       return   this.addrHost;
  }

  //   retorna el port por al cual se comunicara
  public int getPort(){
       return   this.port;
  }

  //   retorna el id de la réplica
  public int getIDRepli(){
       return   this.replica;
  }

  //   retorna el tipo del mensaje
  public char getTipo(){
       return   this.tipo;
  }

  //   retorna el contenido del mensaje
  public String getMsg(){
       return   this.msg;
  }
}
```

B.2 Procesos para ejecuciones Lazy

B.2.1 Manager.java

```
/**************************************************************************
 *   Es el proceso principal, encargado de inicializar todos los datos                    *
 *   e iniciar los procesos necesarios para procesar las consultas                          *
 **************************************************************************/

import java.lang.*;
import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.Timestamp;
import java.math.*;

public class Manager extends Thread {
```

```
private final static String TIPO_ACTU = "Asincrónica."; // tipo de actualización

private int CANT_HOST = 3; //   cantidad total de Host
private int NUMBER_HOST;    //   nro. ID del host
private String[] addrHost; //   direcciones de los host
private int[] portList;      //   ports de los MasterListener en cada host
private int[] portRecei;    //   ports de los Receiver en cada host
private String addrInit; //   dirección del host iniciador
private int portInit;         //   port del host iniciador
private int[][] pesoComunic; // peso asignado a cada una de
// las aristas del grafo de comunicación entre los host
private char[][] locateRep; // localización estática de las
// réplicas sobre los diferentes Host


private int CANT_DATOS;      //   cantidad de datos sobre los que se realizan
                                //   las consultas
private int PORC_REPLICA;   //   porcentaje de replicación
private char TIPO_REGULACION; //   tipo de regulación de la propagación
private int   PORC_READ;      //   porcentaje de lecturas

// para la localizacion de las réplicas
private final static char MASTER = 'M';   //   master de una réplica
private final static char SLAVE = 'S';    //   slave de una réplica
private final static char NOT_OWNER = '-';        //   no tiene la réplica

private final static char GROUP = 'G'; // regulacion Group

private final static char CENTRAL = 'C'; // para el caso con todos los datos centralizados

MasterSender mSender; //   para enviar las consultas a quien deba procesarlas
MasterListener mListen; //   para recibir los requerimientos remotos
DataManager dataM;   //   para manejar el conjunto de réplicas
Recovery recovery; //   para guardar la información de las consultas que
                        //   tuvieron conflictos en sus actualizaciones
Monitor monitor;      //   para generar el informe final
Propagator propagator;   //   para propagar las actualizaciones
Receiver   receiver; //   para recibir las actualizaciones remotas propagadas
SrvSlave srvSlv;   //   para tener el cto. de SlaveSenders y SlaveListener

public Manager() {
      this.start();
}

 //   inicializa el nro. de ID del Host y las
 //   direcciones y port de los host

private void initAddrPortHost(){
      // para leer el archivo de direcciones de cada uno de los host
      FileReader in;
      File inputFile;
      char[] host = new char[1];
      char[] port = new char[5];
      char[] numHost = new char[2];
      char[] blanco = new char[1];
      char[] salto = new char[1];
```

```
        int blank = 32;
        char NOT_NUMBER = 'X';

      try {
        addrHost = new String[this.CANT_HOST]; // direcciones de los host
        portList = new int[this.CANT_HOST]; // ports de los Listener en cada host
        inputFile = new File("DatosHost/AddrPortHost.txt");
        in = new FileReader(inputFile);
        for (int i = 0; i <= this.CANT_HOST-1;i++) {
            String addr = new String("");
            in.read(host);
            // leo una dirección de un Host
            while (host[0] != blank) {
                addr = addr + new String(host);
                in.read(host);
            }
            // leo el port del listener
            in.read(port);
            in.read(blanco);
            // leo el nro. de ID del Host
            in.read(numHost);
            if (numHost[0] != NOT_NUMBER ) {
                this.NUMBER_HOST = (new Integer(new String(numHost)).intValue());
            }
            in.read(salto);
            addrHost[i] = new String(addr);
            portList[i] = (new Integer(new String(port)).intValue());
        }
        in.close();
      }
      catch (IOException e) {System.out.println("Manager: error al abrir DatosHost/AddrPortHost.txt");}
    }


  // inicializa los pesos de las comunicaciones entre los host
  // a partir de un archivo

  private void initPesoComunic(){
        // para leer el archivo de pesos de comunicaciones entre los host
        FileReader in;
        File inputFile;
        char[] peso = new char[1];
        char[] blanco = new char[1];
        char[] salto = new char[1];

      try {
        pesoComunic = new int[this.CANT_HOST][this.CANT_HOST];
        inputFile = new File("DatosHost/PesoComunic.txt");
        in = new FileReader(inputFile);
        for (int i = 0; i <= this.CANT_HOST-1;i++) {
            for (int j = 0; j <= this.CANT_HOST-1;j++) {
                in.read(peso);
                in.read(blanco);
                pesoComunic[i][j] = (new Integer(new String(peso)).intValue());
            }
            in.read(salto);
```

```
        }
        in.close();
    }
    catch (IOException e) {System.out.println("Manager: error al abrir DatosHost/PesoComunic.txt");}
}

 //   genera mensajes con los datos iniciales del encabezado del informe
 //   y envía dichos mensajes al Monitor

private void initInforme(int cantQueries){

    String msg = null;   //   para generar y enviar un mensajes
    String tipoRegu = null; //   para imprimir el tipo de regulación

    if (TIPO_REGULACION == this.MASTER) { tipoRegu = " Master-Slave ";}
    else {   if (TIPO_REGULACION == this.GROUP) { tipoRegu = " Group ";}
             else { tipoRegu = " Centralizada ";} }

    //armo el encabezado del informe
    msg = "B.D.D. ESTUDIO DE ACTUALIZACION DE REPLICAS DE DATOS.\t"
              + new Date(System.currentTimeMillis())+"\n\n";
    monitor.putMsg(msg);
    msg = "REGULACION: "+tipoRegu+ "\tPROPAGACION: "+TIPO_ACTU+"\n \n";
    monitor.putMsg(msg);
    msg = "HOST\t\tID:"+this.NUMBER_HOST+"\tDirección: "
          + this.addrHost[this.NUMBER_HOST]+"\tPort: "
          + this.portList[this.NUMBER_HOST]+"\tCantidad: "+this.CANT_HOST+"\n \n";
    monitor.putMsg(msg);
    msg = "CONSULTAS\tCantidad: "+cantQueries+"\tLecturas al: "
              + PORC_READ +" % \n \n";
    monitor.putMsg(msg);
    msg = "REPLICAS\tCantidad: "+this.CANT_DATOS+"\tReplic. al: "
              + PORC_REPLICA +" %\n";
    monitor.putMsg(msg);
    //   informo que réplicas tiene el host y si es Master o Slave de la misma
    String mst = "";
    String slv = "";
    int countMst = 0;
    int countSlv = 0;
    for (int i = 0; i <= this.CANT_DATOS -1;i++) {
        if (locateRep[i][this.NUMBER_HOST] == this.MASTER) {
            mst = mst+" "+ i;
             countMst = countMst + 1;
             if (countMst == 30) {mst = mst +"\n";
                                  countMst = 0;
             }
        }
        else {
            if (locateRep[i][this.NUMBER_HOST] == this.SLAVE) {
                slv = slv+" "+ i;
                  countSlv = countSlv + 1;
                  if (countSlv == 30) {slv = slv +"\n";
                                  countSlv = 0;
                  }
            }
        }
```

```
    }
    msg = "\nMaster de las sig. Réplicas:\t\n"+mst+"\n";
    monitor.putMsg(msg);
    msg = "\nSlave de las sig. Réplicas:\t\n"+slv+"\n\n";
    monitor.putMsg(msg);
    Timestamp ts= new Timestamp(System.currentTimeMillis());// times de inicio
    msg = "HORA DE INICIO:\t\t"+(ts.toString()+"000").substring(11,23)
        +"\n\n";

    monitor.putMsg(msg);
    msg ="ID\tTIPO\tREPLICA\tDESTINO\t\tINICIO\t\tFINALIZ.\tDURAC.\tRESULTADO\n \n";
    monitor.putMsg(msg);
    //   informo el tiempo en que se inicia el procesamiento
    monitor.initTS(ts);
}



// inicializa la localización de cada una de las réplicas en los
// diferentes host a partir de un archivo

private void initLocateRep(){

 try {
    DataInputStream inFile = new DataInputStream(new FileInputStream("EsqDatos/LRep"
                          + this.TIPO_REGULACION + this.PORC_REPLICA + ".txt"));
       CANT_DATOS = inFile.readInt();   //   lee la cantidad de datos a replicar

       locateRep = new char[this.CANT_DATOS][this.CANT_HOST];
       for (int i = 0; i <=CANT_DATOS-1;i++) {
            for (int   t = 0; t <=CANT_HOST-1;t++) {
                int nroRep =inFile.readInt();
                int nroHost =inFile.readInt();
                char MstSlv =inFile.readChar();
                locateRep[nroRep][nroHost]=MstSlv ;
            }
       }
       inFile .close();
    }
    catch (IOException e) {System.out.println("Manager: error al abrir "
                                + "EsqDatos/LRep" + this.TIPO_REGULACION +
                                  this.PORC_REPLICA + ".txt");}
}

//   retorna el peso de la comunicación entre idHost1 y idHost2
public int getPesoHost(int idHost1,int idHost2) {
    return this.pesoComunic[idHost1][idHost2];
}

 //   retorna el ID del Host que tiene la réplica idRep con el menor costo
 //   de comunicación, además, se usa el parámetro masterHost para indicar
 //   si es "necesario" que el host retornado sea o no el master de dicha réplica

private int getHostReplica(int idRep, boolean   masterHost){
```

```
    //   para determinar el host de menor peso en comunicación
    int minPeso = Integer.MAX_VALUE;
    int minHost = Integer.MAX_VALUE;

    if (masterHost) {   //   requiere un host Master (para update)
        for (int j = 0; j <= this.CANT_HOST-1;j++) {
            if   ((locateRep[idRep][j] == this.MASTER) &&
                (minPeso > pesoComunic[this.NUMBER_HOST][j])) {
                minPeso = pesoComunic[this.NUMBER_HOST][j];
                minHost =j;
            }
        }
    }
    else { //   puede ser un host Master o Slave (para read)
        for (int i = 0; i <= this.CANT_HOST-1;i++) {
            if   ((locateRep[idRep][i] != this.NOT_OWNER) &&
                (minPeso > pesoComunic[this.NUMBER_HOST][i])) {
                minPeso = pesoComunic[this.NUMBER_HOST][i];
                minHost =i;
            }
        }
    }
    return   minHost;
}

//   retorna true si el host idHost tiene la réplica idRep,
//   sino, retorna falso
public boolean ownerHostReplica(int idRep,int idHost){
    return   locateRep[idRep][idHost]!= this.NOT_OWNER;
}

//   retorna el id del Host local
public int getIdHost(){
    return   this.NUMBER_HOST;
}

//   retorna la dirección del host idHost
public String getAddrHost(int idHost){
    return   this.addrHost[idHost];
}

//   retorna port del MasterListener del host idHost
public int getPortList(int idHost){
    return   this.portList[idHost];
}

//   retorna port del Receiver del host idHost
public int getPortReceiver(int idHost){
    return   this.portRecei[idHost];
}

//   retorna la cantidad de host
public int getCantHost(){
    return   this.CANT_HOST;
}
//   retorna la cantidad de datos que se replican
```

```
public int getCantDatos(){
    return   this.CANT_DATOS;
}

public synchronized void run() {
    String myAddrHost; // para la dirección de mi host local
    int myportList; //   para el port listener de mi host local

    while (true) {
        System.gc();
        initAddrPortHost(); //   inicializo los ID y direcciones de los host
        myAddrHost = addrHost[this.NUMBER_HOST];
        myportList = portList[this.NUMBER_HOST];
        System.out.println("INICIANDO HOST en ADDR: " +myAddrHost+
        " Nro: "+this.NUMBER_HOST+ " PORT: "+myportList);
        //   verifico si quedaron procesos corriendo de anteriores corridas
        if (this.activeCount()==2) {
                System.out.println("Manager: no hay procesos activos.");
        } else {   //   han quedados procesos corriendo
                System.out.println("Manager: hay "+ (this.activeCount()-1)
                                        + " procesos activos.");
                System.exit(1);
        }

        try {
          //   creo un ServerSocket para recibir la señal de inicio
          ServerSocket listenInit;
          Socket sInit;
          listenInit = new ServerSocket(portList[this.NUMBER_HOST]);
          System.out.println("Manager: esperando INICIO en PORT: "
                                + portList[this.NUMBER_HOST]);
          sInit = listenInit .accept();
          DataInputStream in = new DataInputStream(sInit.getInputStream());
          //   leo los datos sobre los cuales hago el procesamiento
          TIPO_REGULACION = in.readChar(); // tipo Regulacion Master-Slave o Group
          PORC_REPLICA = in.readInt();// porcentaje de replicación
          PORC_READ = in.readInt(); // porcentaje de lecturas
          portInit = in.readInt(); // port del host inicializador
           addrInit = in.readLine();   //   dirección del host inicializador
          in.close();
          in = null;
          sInit.close();
          sInit = null;
          listenInit.close();
          listenInit = null;
          if   ((TIPO_REGULACION != this.MASTER) &&
               ( TIPO_REGULACION != this.GROUP ) &&
               ( TIPO_REGULACION != this.CENTRAL )) {break;}; // terminó la ejecución
        }
        catch (IOException e) { System.out.println("Manager: "+e);}
        initLocateRep();   //   inicializo la localización de las réplicas
        initPesoComunic();   //   inicializo el peso de las comunicaciones
        // genero los port para los receivers
        portRecei = new int[this.CANT_HOST];
        for (int i = 0; i <= this.CANT_HOST-1;i++) {
            portRecei[i] = portList[i]-1;
```

```
            }

            recovery = new Recovery();
            propagator = new Propagator(this);
            dataM = new DataManager(this);
            receiver = new Receiver(this);
            srvSlv = new SrvSlave();
            mSender = new MasterSender(this);
            mListen = new MasterListener(this);

            //   espero a que todos los MasterListener remotos esten listos
            try {
              wait();
            }
            catch (InterruptedException e) {;}

            System.out.println("Manager: generando consultas... ");
            //   comienzo a ejecutar las consultas
            GenerarQueries generarQueries = new GenerarQueries();

            //   espero hasta que se hayan ejecutado todas las consultas
            try {
                 wait();
            }
            catch (InterruptedException e) {;}

            //   notifico a cada uno de mis procesos para que terminen
            //   su ejecución
            synchronized (recovery) {recovery.notify();}
            synchronized (dataM) {dataM.notify();}
            synchronized (propagator) {propagator.notify();}
            synchronized (srvSlv) {srvSlv.notify();}
            synchronized (monitor) {monitor.notify();}

            //   espero por sus terminaciones
            try {
                 System.out.println("Manager: terminando...");
                 wait(40000);
            }
            catch (InterruptedException e) {;}

        } // while

        System.out.println("SALIENDO DEL SISTEMA.");
    }


     //   lee cada una de las consultas que hay en un archivo y las
     //   envia al MasterSender local para su procesamiento

class GenerarQueries extends Thread {

   public GenerarQueries() {
          this.start();
   }
```

```
public synchronized void run() {

    final int TIME_WAITING = 1; //   tiempo de espera para
                                // leer una nueva consulta (en ms)


    int cant_Queries;   //   cantidad total de consultas

    char op;   // para leer una operación
    int nroRep; //   para leer sobre qué réplica es la operación
    Timestamp ts; //   para la hora de inicio de la consulta
    String nueValor;   //   para crear el valor de la consulta
    int nroHost;   //   para el nro de host que tiene la réplica
    String hostAddr;   // para la dirección del host que tiene la réplica
    //   para leer el archivo de consultas
    String horaIni; // para la hora de inicio de la consulta
    try {

      DataInputStream inFile = new DataInputStream(new FileInputStream(
                                "Consultas/Query"+PORC_READ+".txt"));
      //   leo la cantidad de consultas que hay en el archivo
      cant_Queries = inFile.readInt();
      monitor = new Monitor("I"+TIPO_REGULACION+PORC_REPLICA+
      PORC_READ+".txt",cant_Queries,addrInit,portInit);
      //   inicializa el encabezado del informe
      initInforme(cant_Queries);

      //   para cada una de las consultas, lee una y la envía
      //   al MasterSender e informa al Monitor con los datos
      //   iniciales de la misma

      for (int i = 0; i <= cant_Queries-1;i++) {
           op =inFile.readChar();
           nroRep =inFile.readInt();
           // armo una nueva query y la envío la MasterSender (mSendMsg)
           ts= new Timestamp(System.currentTimeMillis());
           if (op == SrvMessage.UPDATE){
                //   creo el valor con el nro de host + hora de inicio de la query
       nueValor = NUMBER_HOST+"-"+(ts.toString()+"000").substring(11,23);

        nroHost = getHostReplica(nroRep,true);
           }
           else {
                nueValor =   "X";   //   asigno un valor insignificante si la
                                    //   operación es una lectura
                nroHost =   getHostReplica(nroRep,false);
           }

           //   determino la dirección del host que tiene la réplica
           hostAddr = addrHost[nroHost];
           //   creo un mensaje con la consulta para   el MasterSender
           SrvMessage msgSender = new SrvMessage(i,nroHost,hostAddr,portList[nroHost]
                                ,nroRep,op,nueValor);
           //   envío el mensaje
           mSender.putMsg(msgSender);
```

```
            horaIni = String.valueOf(ts.getTime()+(ts.getNanos()/1000000));
            //   creo un mensaje con la consulta para   el Monitor
            if (nroHost == NUMBER_HOST) {hostAddr = "LOCAL      ";}
            SrvMessage msgMoni = new SrvMessage(i,nroHost,hostAddr,portList[nroHost]
                                    ,nroRep,op,horaIni);
            //   envío el mensaje
            monitor.putMsg(msgMoni);
            // espero TIME_WAITING ms para leer la próxima consulta
            try {sleep(TIME_WAITING);} // duermo el proceso
            catch (InterruptedException e) {;}
         } //   for

         //   creo un mensaje de finalizacion para el MasterSender
         SrvMessage srvMsg = new
SrvMessage(Integer.MAX_VALUE,Integer.MAX_VALUE,"",0,0,SrvMessage.END,"");
         mSender.putMsg(srvMsg);
         inFile.close();
       }
       catch (IOException e) {System.out.println("Manager: error al abrir "
                                + "Consultas/Query"+PORC_READ+".txt");}
   }
 }

   public static void main(String[] args) {
         new Manager();
   }
}
```

B.2.2 DataManager.java

```
/**************************************************************************
 *   Maneja todas las réplicas locales, retornando su valor ante una                    *
 *   lectura, y actualizando su valor ante una escritura.
*
 *   También envía mensajes al Propagador en caso que sea necesario                    *
 *   propagar las actualizaciones.
*
**************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;
import java.math.*;

public class DataManager extends Thread {

  private Vector dataFile;    //   conjunto de datos replicados con su valor actual
  private int lc;   //reloj lógico
  private Manager myManager = null; //   para comunicarme con el Manager

  public DataManager(Manager myManager) {

    this.myManager = myManager;

    //   inicializo los timestamps de todas las réplicas antes
```

```
    //   que empiecen todos los procesos en los diferentes host
    int cantDatos = this.myManager.getCantDatos();
    dataFile = new Vector(cantDatos);
    Replica r;
    for (int i = 0;i <=cantDatos - 1;i++) {
        if (myManager.ownerHostReplica(i,myManager.getIdHost())) {
          r = new Replica(i,"INI"+i,new Timestamp(0,0));
        } else {
          r = new Replica(i,"NO REPLICADO",new Timestamp(0,0));
        }
        dataFile.addElement(r);
    }
    lc=1; //   inicializo mi reloj lógico
    this.start();
}

// retorna el lc corriente y luego lo incrementa en 1
// es utilizado cuando se envía un mensaje
private synchronized int getLCsend(){
      lc = lc + 1;   //   para que si e1 ocurre antes que e2 localmente
                     //   =======> ts(e1) < ts (e2)
      return   (lc - 1);
}

// es para el caso que se recibe un mensaje con timestamp lcProp
// su función es mantener cierto grado de sincronizacion de los relojes
// locales entre dos sitios que se comunican
private   synchronized void setLCReceiv(int lcProp){
      lc = Math.max(lc,lcProp);
}

// retorna el valor de la réplica id
public   String get(int id){
      Replica r;
      r = ((Replica)(dataFile.elementAt(id)));
      return   r.getValue();
}


//   actualización de la réplica id con el valor newValue
//   utilizado para realizar una actualización de una consulta local
//   la cual debe ser propagada
public void putL(int id, String newValue){
      Replica r;
      Timestamp oldTSsend,newTS;
      synchronized (dataFile.elementAt(id)){
            r = ((Replica)(dataFile.elementAt(id)));
            oldTSsend = r.getTS(); // guardo el ts old para propagarlo
            //   calculo el nuevo timestamp
            newTS = new Timestamp(myManager.getIdHost(),this.getLCsend());
            //   actualizo la réplica
            r.setTS(newTS);
            r.setValue(newValue);
            dataFile.setElementAt(r,id);
      }
      //   propago la actualización
```

```
        myManager.propagator.putMsg(new MsgPropag(id,newValue,oldTSsend,newTS));
    }

    //   actualización de la réplica id con el valor newValue
    //   utilizado para realizar una actualización de proveniente de una
    //   propagación remota
    public void putR(int id, String newValue,Timestamp oldTs,Timestamp newTs){

        Replica r;
        Timestamp tsLocal;

        this.setLCReceiv(newTs.getLC()); //   sincronizo los relojes
        synchronized (dataFile.elementAt(id)){
            r = ((Replica)(dataFile.elementAt(id)));
            if (r.getTS().equals(oldTs)) {   //   no hay conflicto
                r.setTS(newTs);
                r.setValue(newValue);
                dataFile.setElementAt(r,id);
            }
            else {   // hay conflicto - reconciliación
                    // notifico al recovery
                    //   tsLocal valorLocal oldTSProp newTSProp newValorProp
                    this.myManager.recovery.putMsg(id+"\t"+r.getTS()
                    .getStrTS()+"\t\t"+r.getValue()+"\t"+oldTs.getStrTS()+"\t\t"+newTs
                    .getStrTS()+"\t\t"+newValue);

                //   actualizo la réplica para lograr convergencia
                if (r.getTS().before(newTs)) {
                    r.setTS(newTs);
                    r.setValue(newValue);
                    dataFile.setElementAt(r,id);
                }
            } //   else
        } //   synchronized
    }


    public synchronized void run() {

        Replica r;
        String valor;
        Timestamp ts;

        try {
            System.out.println("DataManager: listo.");
            wait();
        }
        catch (InterruptedException e) {;}
        try {
          // genero un archivo con el estado final de las réplicas locales
          File outputFile = new File("Temp/DataFile.txt");
          // abre un FileWriter sobre DataFile.txt
          FileWriter out = new FileWriter(outputFile);
          out.write("REPLICA\tVALOR\t\tTIMESTAMP\n\n");
          int cantDatos = this.myManager.getCantDatos();
          for (int i = 0;i <=cantDatos - 1;i++) {
```

```
            r = ((Replica)(dataFile.firstElement()));
            valor = r.getValue();
            ts = r.getTS();
            dataFile.removeElementAt(0);
            out.write(i+"\t"+valor+"\t"+ts.getStrTS()+"\n");
          }
          out.close();
          System.out.println("FIN DataManager");
        }
        catch (IOException e ) { System.out.println("DataManager: "+e);}
    }
}
```

B.2.3 Propagator.java

```
/*************************************************************************
*Propaga las actualizaciones locales provenientes del DataManager                    *
*************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;

public class Propagator extends Thread {

   private Vector propagMsg = new Vector(5,5);//cola de mensajes recibidos
   // del DataManager de las actualizaciones a propagar

   private Manager myManager = null; //    para comunicarme con el Manager

   public Propagator(Manager myManager) {
        this.myManager = myManager;
        this.start();
   }

   //   agrega un nuevo mensaje a la cola
   public synchronized void putMsg(MsgPropag msg) {
        this.propagMsg.addElement(msg);
        this.notify();
   }
   // retorna y elminia el primer mensaje de la cola
   private MsgPropag removeMsg() {
       MsgPropag msg = (MsgPropag) propagMsg.firstElement();
       propagMsg.removeElementAt(0);
       return msg;
   }

   public synchronized void run() {

      Socket sPropagOut = null;   // para   comunicarse con los Receivers
      DataOutputStream out = null;   //para escribir en sPropagOut

      while(true) {
        try {
          if (propagMsg.isEmpty()) { //   no hay actualizaciones en la cola
```

```
        try {
            System.out.println("Propagator: en espera...");
            wait(); //   espero hasta que me envíen una actualiz.
        }               //   o me notifiquen que termine
        catch (InterruptedException e) {;}
        if (propagMsg.isEmpty()) {break;} //   fin de proceso
    }
    //   recupero el primer mensaje y lo elimino de la cola
    MsgPropag smsg = this.removeMsg();
    System.out.println("Propagator: procesando replica: " + smsg.getIDRepli() +
                        "valor: "+ smsg.getValue());

    //   propago la actualización
    int cantHost = this.myManager.getCantHost();
    int myIdHost = this.myManager.getIdHost();
    for (int i = 0; i <= cantHost - 1;i++) {
        //   no lo propago a mi mismo, ni a los que no tienen
        //   una copia de la replica
        if ((this.myManager.ownerHostReplica(smsg.getIDRepli(),i)) &&
            (i != myIdHost)) {
            while (true) {
                try {
                    //   determino la direccion y el port al cual debo realizar
                    //   la propagación
                    int portReceiv = this.myManager.getPortReceiver(i);
                    String addHost = this.myManager.getAddrHost(i);
                    System.out.println("Propagator: port destino: " + portReceiv);
                    sPropagOut = new Socket(addHost,portReceiv);
                    out = new DataOutputStream(sPropagOut.getOutputStream());
                    //   envío la actualización
                    //   oldTS + newTS + nro Replica + nuevoValor
                    out.writeInt(smsg.getOldTs().getIDHost());   //   oldTS
                    out.writeInt(smsg.getOldTs().getLC());
                    out.writeInt(smsg.getNewTS().getIDHost());   // newTS
                    out.writeInt(smsg.getNewTS().getLC());
                    out.writeInt(smsg.getIDRepli());   // nro Replica
                    out.writeBytes(smsg.getValue());   // nuevo Valor
                    out.writeByte('\n');
                    // realizo el envío
                    out.flush();
                     out.close();
                    out = null;
                    sPropagOut.close();
                    sPropagOut = null;
                    break;
                }
                catch (SocketException e) {
                    // no pude comunicarme con el Receiver
                    try {wait(0,1);}   // espero 1 nseg.
                    catch (InterruptedException e2) {;}
                    System.out.println("Propagator: no puedo comunicarme con el Reciever: "
                    + myManager.getAddrHost(i)+" "+myManager.getPortReceiver(i));}
                finally {
                    if (sPropagOut != null) {
                        sPropagOut.close();
                        sPropagOut = null;
```

```
                        }
                        if (out != null) {
                            out.close();
                            out = null;
                        }
                    } //   finally
                } //while
            }   //   if
        } //   for
       }//try
       catch (IOException e) { System.out.println("Propagator: "+e); }
     }// while
     System.out.println("FIN Propagator");
   } // run
}
```

B.2.4 Receiver.java

```
/**************************************************************************
*   Su función es la de recibir mensajes provenientes de los Propagadores                    *
*   remotos y realizar los pedidos correspondiente de actualización al                         *
*                                                                                          DataManager
*
**************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;

public class Receiver extends Thread {

  private static final int TIME_WAITING = 60000; //   tiempo máximo (ms.) de
                                          // espera por un   requerimiento
  private Manager myManager = null; //   para comunicarme con el Manager

  public Receiver(Manager myManager) {
       this.myManager = myManager;
       this.start();
  }

  public synchronized void run() {

       int hostOld,lcOld,hostNew,lcNew,replica;
       String valor;
       ServerSocket listenSocket = null;     // para escuchar los requerimientos
       Socket sRecIn = null;     // para   comunicarse con el Propagador que le
                                 // envía una actualización
       DataInputStream in;   //para leer de sRecIn
       // para crear un socket y recibir los requerimientos remotos
       String myHost;
       int myPort;

     try{
       int idHost = myManager.getIdHost();
       myHost = myManager.getAddrHost(idHost);
```

```
        myPort = myManager.getPortReceiver(idHost);

        try { listenSocket = new ServerSocket(myPort);
              listenSocket.setSoTimeout(TIME_WAITING);}
        catch (IOException e){System.out.println("Excepción ocurrida creando "
                                +"Receiver: " + myPort);}
        while (true) {
           try {
             System.out.println("Receiver: esperando en el PORT: "+myPort);
             //   acepta las conecciones de los MasterSender
             sRecIn = listenSocket.accept();
             in = new DataInputStream(sRecIn.getInputStream());
             //   determino los datos de la actualización remota
             //   OldTS + NewTS + réplica + valor
             hostOld = in.readInt(); //OldTS
             lcOld = in.readInt();
             hostNew = in.readInt(); //NewTS
             lcNew = in.readInt();
             replica = in.readInt(); //réplica
             valor = in.readLine();   //valor
             System.out.println("Receiver: procesando Actualización Remota. "
                                  + "Réplica: " + replica +"Valor: "+valor);
             //   realizo la operacion de actualización
             myManager.dataM.putR(replica,valor,new Timestamp(hostOld,lcOld),
             new Timestamp(hostNew,lcNew));
             in.close();
             in = null;
            sRecIn.close();
             sRecIn = null;
           }
           catch (IOException e) {System.out.println("Receiver: "+ e);
                                    break;}
        } //while
        System.out.println("FIN Receiver ");
        listenSocket.close();
        listenSocket = null;
      } // try
      catch (IOException e) { System.out.println("Receiver: "+ e);}

    } //run

}
```

B.2.5 Monitor.java

```
/**************************************************************************
*   Toma toda la informacion brindada por los diferentes procesos, y                    *
*   a partir de ella genera un informe final
*
**************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.Timestamp;
```

```
public class Monitor extends Thread {

    private Vector monitorMsg = new Vector(10); //   para los mensajes de resultados
    //   necesarios para generar el detalle del informe

    //   para generar el archivo del informe final
    private FileWriter out;
    private File outputFile;
    private String InfoName;   //   nombre del informe final
    private Timestamp tsIni,tsFin;   //   para los tiempos de inicio y fin de
                                     //   procesamiento
    private long[] iniTime;     //   para los tiempos de inicio de las consultas
    private long[] endTime;     //   para los tiempos de finalización de las consultas
    private int cantQueries;   // para la cantidad de consultas
    private int count = 0;      // para   la cantidad de consultas a recuperar
    private long duracTotal = 0; // para la duración total de las consultas
    //   dirección y port del host que iniciador y al cual se le retorna
    //   el los resultados obtenidos en la ejecución
    private String hostResu;
    private int portResu;

  public Monitor(String InfoName,int cantQueries, String host, int port) {
       this.InfoName=InfoName;
       this.cantQueries = cantQueries;
       this.hostResu = host;
       this.portResu = port;
       iniTime = new long[this.cantQueries];
       endTime = new long[this.cantQueries];
       this.start();
  }

  //   inicializa el timestamp de inicio de procesamiento
  public synchronized void initTS(Timestamp ts) {
       this.tsIni = ts;
  }
  //   agrega un nuevo mensaje a la cola
  public synchronized void putMsg(Object msg) {
       this.monitorMsg.addElement(msg);
       this.notify();
  }
  // retorna y eliminia el primer mensaje de la cola
  private Object removeMsg() {
      Object smsg = monitorMsg.firstElement();
      monitorMsg.removeElementAt(0);
      return smsg;
  }

  //   a partir de los archivos temporarios con los datos iniciales y
  //   finales de las consultas, genero los detalles del informe

  private void JoinDetalles() {

       String ini,fin;
       char car[];
    try {
```

```
      DataInputStream inFile = new DataInputStream(new FileInputStream("Temp/TempIni.txt"));
      //   leo los datos iniciales de una consulta en tempIni.txt
      ini = inFile.readLine();
      while(ini!= null) {
            System.out.print(".");
            DataInputStream finFile = new DataInputStream(new FileInputStream("Temp/TempFin.txt"));
            //   busco los datos finales de la consulta en el archivo tempFin.txt
            fin = finFile.readLine();
            while ((fin!=null) && (((ini.charAt(0))!=(fin.charAt(0))) ||
                                    ((ini.charAt(1))!=(fin.charAt(1))) ||
                                        ((ini.charAt(2))!=(fin.charAt(2)))))
            {
             fin = finFile.readLine();
             }
             if (fin!=null){
                 //   escribo en el informe todos los datos de la consulta
                 out.write(ini+" " + fin.substring(3,fin.length()) + "\n");
            }
            finFile.close();
            ini = inFile.readLine();
      }
      inFile.close();
    }
    catch (IOException e ) { System.out.println("Monitor: "+e);}
}

//   genera el estado final de las réplicas locales con sus
//   correspondientes valores
private void GenEstadoRepli() {

      String ini,fin;
      char car[];

    try {
      out.write("\n");
      out.write("ESTADO FINAL DE LAS REPLICAS LOCALES:\n\n");
      //   recupera los valores finales de las réplicas a partir de los
      //   datos generados por el DataManager (guardados en DataFile.txt)
      DataInputStream inFile = new DataInputStream(new FileInputStream("Temp/DataFile.txt"));
      ini = inFile.readLine();
      while(ini!= null) {
            out.write(ini+ "\n");
            ini = inFile.readLine();
      }
      inFile.close();
    }
    catch (IOException e ) { System.out.println("Monitor: "+e);}
}

//   genero el detalle de las consultas que tuvieron conflictos
private void GenRecovery() {

      String ini,fin;
      char car[];

    try {
```

```
      out.write("\n");
      out.write("CONSULTAS A RECUPERAR:\n\n");
      //   recupera las consultas que tuvieron conflictos a partir de los
      //   datos generados por el DataManager (guardados en Recovery.txt)

      DataInputStream inFile = new DataInputStream(new FileInputStream("Temp/Recovery.txt"));
      ini = inFile.readLine();
      while(ini!= null) {
           out.write(ini+ "\n");
           ini = inFile.readLine();
           count ++;
      }
      //   calculo la cantidad de consultas a recuperar
       count=count-2;//-2 por los encabezados
       if (count<0) {count=0;}
       out.write("\nCANT. TOTAL DE CONSULTAS A RECUPERAR: "+ count+"\n");

      inFile.close();
    }
    catch (IOException e ) { System.out.println("Monitor: "+e);}
  }

  //   Genera el Encabezado del Informe a partir de los mensajes de encabezamiento
  //   del informe, generados por el manager

  private synchronized void GenEncabezado() {
    try{

      if (monitorMsg.isEmpty()) {
           try {wait(); }
           catch (InterruptedException e) {;}
      }
      //   recupero el primer mensaje y lo elimino de la cola
      Object smsg = this.removeMsg();
      String str= "";
      // mientras sean mensajes del encabezamiento
      while (smsg.getClass()== str.getClass() )    {
           //   escribo en el informe
           out.write((String) smsg);
           if (monitorMsg.isEmpty()) {
                try {wait(); }
                catch (InterruptedException e) {;}
           }
           //   recupero el sig. mensaje y lo elimino de la cola
           smsg = this.removeMsg();
      }
        this.putMsg(smsg); // guardo el ùltimo mensaje que no era un String
    }
    catch (IOException e ) { System.out.println("Monitor: "+e);}
  }

  //   genero dos archivos temporarios tempIni.txt y tempFin.txt
  //   con los datos iniciales y finales de las consultas respectivamente

  private synchronized void GenDetalles() {
    try {
```

```
        //   creo dos clases con "basura" para utilizarlas solo para
        //   recuperar obtener sus clases
        SrvMessage srvMsg = new
SrvMessage(Integer.MAX_VALUE,Integer.MAX_VALUE,"",0,0,SrvMessage.END,"");
        MsgMonitor moniMsg =    new MsgMonitor(Integer.MAX_VALUE,"",new Timestamp(0));;

        //   creo los archivos temporarios con los datos iniciales y finales

        FileWriter     tempIni = new FileWriter(new File("Temp/TempIni.txt"));
        FileWriter     tempFin = new FileWriter(new File("Temp/TempFin.txt"));

        tsFin = tsIni;   //   inicializo el timestamp final con el inicial
        if (monitorMsg.isEmpty()) {
             try {wait(); }
             catch (InterruptedException e) {;}
        }
        //   recupero el primer mensaje y lo elimino de la cola
        Object smsg = this.removeMsg();
        while (true)   {
             //mensaje con los datos iniciales de la consulta
             //   iniciales = id + tipo + réplica + destino + hora inicio
             if (smsg.getClass()== srvMsg.getClass()) {
                  srvMsg = (SrvMessage) smsg;
                  // convierto la hora de inicio en milisegundos a h:m:s:ms
                  long miliseg = Long.valueOf(srvMsg.getMsg()).longValue();
                   int idQuery = srvMsg.getIDQuery();
                  iniTime[idQuery]= miliseg;
                  Timestamp tsI = new Timestamp(miliseg);

                  if (srvMsg.getTipo() == SrvMessage.UPDATE) {

                       tempIni.write( idQuery+"     \tUpdate\t" +
                       srvMsg.getIDRepli()+"\t" + srvMsg.getIdHost()+"-"+srvMsg.getAddrHost()
                       +"\t"+(tsI.toString()+"000").substring(11,23)+"\n");
                  }
                  else {   tempIni.write(idQuery
                   +"     \tRead\t"+srvMsg.getIDRepli()

                   +"\t" + srvMsg.getIdHost()+"-" + srvMsg.getAddrHost()
                    +"\t"+(tsI.toString()+"000").substring(11,23)+"\n");

        }
              }
             //   mensaje con los datos finales de la consulta
             //   finales     = id + hora finalizacion + Duración + resultado
             else {
                  moniMsg = (MsgMonitor) smsg;
                  Timestamp tsF = moniMsg.getTime();
                  long miliseg = tsF.getTime()+(tsF.getNanos()/1000000);
                  int idQuery = moniMsg.getIDQuery();
                  endTime[idQuery]= miliseg;
                  long duracion = (endTime[idQuery]-iniTime[idQuery]);
                  if (duracion == 0) {duracion = 1;}
                  duracTotal = duracTotal + duracion;
                  tempFin.write(idQuery + "    \t"+(tsF.toString()+"000")
                  .substring(11,23)+"\t"+ duracion+"\t" + moniMsg.getValue()+"\n");
```

```
                //   calculo la hora de finalizacion del procesamiento
                if (tsFin.before(moniMsg.getTime())){tsFin=moniMsg.getTime();}
            }

            if (monitorMsg.isEmpty()) {
                try {wait(); }   //   espero por un nuevo mensaje o una notificación
                                 //   de que no se terminaro las consultas
                catch (InterruptedException e) {;}
                if (monitorMsg.isEmpty()) {break;} // se terminaron las consultas
            }
            //   recupero el primer mensaje y lo elimino de la cola
            smsg = this.removeMsg();
        }
        tempIni.close();
        tempFin.close();
    }
    catch (IOException e ) { System.out.println("Monitor: "+e);}
}

public synchronized void run() {

    try {
        outputFile = new File("Informes/"+this.InfoName);
        // abre un FileWriter sobre InfoName
        out = new FileWriter(outputFile);

        //   genero el Encabezado del Informe
        GenEncabezado();
        //   genero los archivos temporarios para el Detalle del Informe
        GenDetalles();
        //   genero el detalle del informe a partir de los archivos temporarios
        System.out.print("Generando Informe");
        JoinDetalles();
        //   genero el Pie del Informe
        out.write("\nHORA DE FINALIZACION:\t"+ (tsFin.toString()+"000")
        .substring(11,23)+"\n \n");
        long tpoTotal = ((tsFin.getTime()+(tsFin.getNanos()/1000000))
        -(tsIni.getTime()+(tsIni.getNanos()/1000000)));
        out.write("DURACION DE LA EJECUCION:      \t"+tpoTotal+ " ms.\n \n");
        //   genero el estado final de las réplicas locales con sus
        //   correspondientes valores
        GenEstadoRepli();
        //   genero el detalle de las consultas que tuvieron conflictos
        GenRecovery();
        out.close();
        System.out.println("Monitor: Informe finalizado.");

        //   envío al host iniciador el tiempo total de respuesta y la cantidad
        //   total de conflictos
        while (true) {   //   itero hasta que pueda comunicarme
            Socket sMonitorOut = null;
            DataOutputStream outS = null;
            try {
                sMonitorOut = new Socket(hostResu,portResu);
                outS = new DataOutputStream(sMonitorOut
                .getOutputStream());
```

```
            outS.writeLong(tpoTotal); // envio la duración de la ejecucion
            outS.writeLong(duracTotal); // envio la duración total de las consultas
            outS.writeInt(count);   // envio la cantidad de conflictos de actualiz.
            outS.flush();
            outS.close();
            outS = null;
            sMonitorOut.close();
            sMonitorOut = null;
            System.out.println("FIN Monitor");
            break;
          }   //try
          catch (SocketException e) {
               // no pude comunicarme con el Inicializador
               try {wait(0,1);}    // espero 1 nseg.
               catch (InterruptedException e2) {;}
                        System.out.println("Monitor: no puedo comunicarme con el   "+
                        "proceso Inicializador "+ e);
          }
          finally {
               if (sMonitorOut != null) {
                    sMonitorOut.close();
                    sMonitorOut = null;
               }
               if (outS!= null) {
                    outS.close();
                    outS= null;
               }
          } //   finally
        } //   while
      } //   try
      catch (IOException e ) { System.out.println("Monitor: "+e);}
    } // run
}
```

B.2.6 Recovery.java

```
/*************************************************************************
 *   Su función es la de recibir mensajes provenientes del DataManager                    *
 *   de aquellas actualizaciones que tienen conflictos y tendría que                          *
 *   recuperar, grabando los datos de dichas actualizaciones en un archivo                *
*************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;

public class Recovery extends Thread {

   private Vector recoveryMsg = new Vector(10);   //   mensajes de datos a recuperar

   public Recovery() {
          this.start();
   }
   //   agrega un nuevo mensaje a la cola
   public synchronized void putMsg(String msg) {
```

```
        this.recoveryMsg.addElement(msg);
        this.notify();
    }
    // retorna y elminia el primer mensaje de la cola
    private String removeMsg() {
        String msg = (String) recoveryMsg.firstElement();
        recoveryMsg.removeElementAt(0);
        return msg;
    }

    public synchronized void run() {
      try {
        //   genero un archivo con las consultas que a reconciliar
        File outputFile = new File("Temp/Recovery.txt");
        // abre un FileWriter sobre Recovery.txt
        FileWriter out = new FileWriter(outputFile);
        //replica tsLocal valorLocal oldTSProp newTSProp newValorProp
        out.write("REPLICA\tTS LOCAL\tVALOR     LOCAL\tOLD_TS_PROP\tNEW_TS_PROP\tVALOR
        PROP\n\n");

        while (true) {
            if (recoveryMsg.isEmpty()) {
                try {
                    System.out.println("Recovery: en espera de mensajes...");
                    wait();
                }
                catch (InterruptedException e) {;}
                if (recoveryMsg.isEmpty()) {   // fin de proceso
                                    out.close();
                                break;
                        }
            }
            //   recupero el primer mensaje y lo elimino de la cola
            String smsg = this.removeMsg();
            // escribo el nuevo mensaje al archivo
            out.write(smsg+"\n");
        }
        out.close();
        System.out.println("FIN Recovery");
      }
      catch (IOException e ) { System.out.println("Recovery: "+e);}
    }
}
```

B.2.7 Replica.java

```
/*****************************************************************************
 *   Etructura de las réplicas, las mismas tienen un id, un valor actual,                                    *
 *   y un timestamp de su última actualización
*
*****************************************************************************/

public class Replica   {

   private int id;
```

```
  private String value;
  private Timestamp tsLocal;

  //   crea una nueva réplica
  public Replica(int id,String value, Timestamp tsLocal) {
    this.id = id;
    this.value = value;
    this.tsLocal = tsLocal;
  }

   //   retorna el id de la réplica
   public int getIDRepli(){
       return   this.id;
   }

   //   retorna el timestamp de actualización de la réplica
   public Timestamp getTS(){
       return   this.tsLocal;
   }

   //   retorna el valor de la réplica
   public String getValue(){
       return   this.value;
   }

   //   actualiza el timestamp a newTS
   public void setTS(Timestamp newTS){
       this.tsLocal=newTS;
   }

   //   actualiza el valor a newValue
   public void setValue(String newValue){
       this.value = newValue;
   }

}
```

B.2.8 Timestamp.java

```
/**************************************************************************
 *   Timestamp formado por un id del host + valor de reloj lógico                                    *
 **************************************************************************/

public class Timestamp   {

  private int idHost; // identificador del Host
  private int lc;       // contador local para ordenar las propagaciones
                        // (logical clock)

  //   crea un nuevo Timestamp con el valor idHost-lc
  public Timestamp(int idHost, int lc) {
    this.idHost = idHost;
    this.lc = lc;
  }
```

```
  //   retorna el idHost
  public int getIDHost(){
       return   this.idHost;
  }

  //   retorna el reloj lógico
  public int getLC(){
       return   this.lc;
  }

  //   retorna true si este timestamp es menor que ts,
  //   sino retorna false
  public boolean before(Timestamp ts){
       return ((this.lc < ts.lc)||
        ((this.lc == ts.lc) && (this.idHost < ts.idHost)));
  }

  //   retorna true si este timestamp es mayor que ts,
  //   sino retorna false
  public boolean after (Timestamp ts) {
       return ((this.lc > ts.lc)||
        ((this.lc == ts.lc) && (this.idHost > ts.idHost)));
  }

  //   retorna true si este timestamp es igual que ts,
  //   sino retorna false
  public boolean equals (Timestamp ts) {
       return ((this.lc == ts.lc) && (this.idHost == ts.idHost));
  }

  //   retorna el timestamp formateado en un String
  public String getStrTS() {
      return (this.idHost+"-"+this.lc);
  }
}
```

B.2.9 MsgPropag.java

```
/*************************************************************************
 *   Estructura de los mensajes enviados al Propagador para indicarle los                    *
 *   los datos de una consulta que debe propagar
*
*************************************************************************/

public class MsgPropag   {

  private int replica;   //   para el id de la réplica que se propaga su actualización
  private Timestamp oldTS;   //   para el valor anterior del timestamp
  private Timestamp newTS;   //   para el nuevo valor del timestamp
  private String valor; //   para el nuevo valor de la réplica

  //   crea un nuevo mensaje
  public MsgPropag(int replica, String valor,Timestamp oldTS,Timestamp newTS) {
       this.replica = replica;
       this.valor = valor;
```

```
        this.oldTS = oldTS;
        this.newTS = newTS;
    }

    //   retorna el id de la réplica que hay se propaga
    public int getIDRepli(){
        return   this.replica;
    }

    //   retorna el nuevo valor de la réplica
    public String getValue(){
        return   this.valor;
    }

    //   retorna el timestamp anterior
    public Timestamp getOldTs(){
        return   this.oldTS;
    }

    //   retorna el nuevo timestamp
    public Timestamp getNewTS(){
        return   this.newTS;
    }
}
```

B.2.10 Init.java

```
/**************************************************************************
 *   Sincroniza el inicio de procesamiento de todos los host
*
 *************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.Timestamp;

public class Init extends Thread {

    private static char TIPO_REGULACION; // Tipo Regulacion M Master-Slave G Group
    private static int PORC_REPLICA;// porcentaje de   replicación
    private static int CANT_QUERY; // cantidad de consultas
    private static int PORC_READ; // porcentaje de lecturas
    private final static int CANT_HOST = 3;
    private int myPort= 10000; //   pare crear listenSocket y recibir los resultados
    private static String myAddr = "10.0.0.4"; //   dirección local para enviarla a los demás host
    private String[] addrHost; // direcciones de los host
    private int[] portHost; // ports de los Listener en cada host
    private final static char MASTER = 'M';   //   regulación Master-Slave
    private final static char GROUP = 'G'; // regulación Group
    private final static char CENTRAL = 'C'; // regulación Centralizada

   public Init() {
        initAddrPortHost();
        this.start();
```

```
    }

    //   inicializa las direcciones y port de los host
    private void initAddrPortHost(){
          // para leer el archivo de direcciones de cada uno de los host
          FileReader in;
          File inputFile;
          char[] host = new char[1];
          char[] port = new char[5];
          char[] numHost = new char[2];
          char[] blanco = new char[1];
          char[] salto = new char[1];
          int blank = 32;

        try {
          addrHost = new String[this.CANT_HOST]; // direcciones de los host
          portHost = new int[this.CANT_HOST]; // ports de los Listener en cada host
          inputFile = new File("DatosHost","AddrPortHost.txt");
          in = new FileReader(inputFile);
          for (int i = 0; i <= this.CANT_HOST-1;i++) {
               String addr = new String("");
               in.read(host);
               // leo una dirección de un Host
               while (host[0] != blank) {
                    addr = addr + new String(host);
                    in.read(host);
               }
               // leo el port del listener
               in.read(port);
               in.read(blanco);
               // leo el nro. de ID del Host
               in.read(numHost);
               in.read(salto);
               addrHost[i] = new String(addr);
               portHost[i] = (new Integer(new String(port)).intValue());
          }
          in.close();
        }
        catch (IOException e) {System.out.println("Init: error al abrir DatosHost /     AddrPortHost.txt");}
    }


     public synchronized void run() {

          Socket sListen = null;        //   para   comunicarse con cada uno de los host
          DataInputStream in = null;       //   para leer de sListen
          ServerSocket listenSocket = null;     // para recibir los resultados
          if (TIPO_REGULACION == this.CENTRAL) {PORC_REPLICA = 0;}
          //   envío los datos de inicio a cada uno de los Manager
          for (int i = 0; i <= this.CANT_HOST -1;i++) {

               while (true) {   //   itero hasta que pueda enviar los datos
                                //   al a los host remotos remoto
                 try {
                   System.out.println("REGULACION: "+this.TIPO_REGULACION+" PORC. REP.:
"+this.PORC_REPLICA+" PORC. LECT.:"+this.PORC_READ);
```

```
            System.out.println("Iniciando Host: "+portHost[i]);
            Socket sInitOut = new Socket(addrHost[i],portHost[i]);
            DataOutputStream out = new DataOutputStream(sInitOut.getOutputStream());
            //   envío los datos
            out.writeChar(this.TIPO_REGULACION);   //   tipo de regulación
            out.writeInt(this.PORC_REPLICA);   //   porcentaje de replicación
            out.writeInt(this.PORC_READ);     //   porcentaje de op. de lecturas
            out.writeInt(myPort);     //   port para recibir los resultados
            out.writeBytes(myAddr);   //   mi dirección IP
            out.writeByte('\n');
            out.flush();
             out.close();
            out = null;
                 sInitOut.close();
            sInitOut = null;
            break;
          }
          catch (IOException e) { System.out.println("No puedo comunicarme con el host: "
          +addrHost[i]+" "+portHost[i]);}
       } ///while
}
if   ((TIPO_REGULACION != this.MASTER)&& // exit
   ( TIPO_REGULACION != this.GROUP )&&
   ( TIPO_REGULACION != this.CENTRAL )) { System.exit(0);}

// espero 10 seg. para que inicializen todos los host
try {wait(10000);}
catch (InterruptedException e) {;}

//   sincronizo el inicio de todos los host
//   enviándoles una señal a cada uno de los MasterListeners
for (int i = 0; i <= this.CANT_HOST -1;i++) {

      while (true) {   //   itero hasta que pueda enviar la señal
          try {
            System.out.println("Iniciando Listener: "+portHost[i]);
            Socket sInitOut = new Socket(addrHost[i],portHost[i]);
            DataOutputStream out = new DataOutputStream(sInitOut.getOutputStream());
            //   envío la señal
            out.flush();
             out.close();
            out = null;
            sInitOut.close();
            sInitOut = null;
            break;
          }
          catch (IOException e) { System.out.println("No puedo comunicarme con el host: "
          +addrHost[i]+" "+portHost[i]);}
       } ///while
}

//   espero los resultados de cada uno de los Host

try {listenSocket = new ServerSocket(myPort);}
catch (IOException e) { System.out.println( "Excepción ocurrida creando Init" + myPort); }
System.out.println("Esperando resultados...");
```

```
        try {
          long durEjecTot = 0; // duración total de las ejecuciones de los n host en ms
          long durQueryTot = 0; // duración total de las consultas de los n host en ms
          long conflicTot = 0; // cant.de conflictos totales de los n host

          //   escribo todos los resultados en un archivo
          File outputFile = new
File("Informes/R"+this.TIPO_REGULACION+this.PORC_REPLICA+this.PORC_READ + ".txt");
          FileWriter out = new FileWriter(outputFile);
          for (int i = 0; i <= this.CANT_HOST -1;i++) {
               sListen = listenSocket.accept();
               //   acepto los resultados de un host
               in = new DataInputStream(sListen.getInputStream());
               long durEjec = in.readLong();//duración de la ejecucion del host i
               long durQuery = in.readLong();//duración de las consultas del host i
               int conflic = in.readInt();//cant de conflictos del host i
               System.out.println("Host: "+sListen.getInetAddress().getHostAddress()+"\tDur.de Ejec.: "+
               durEjec +" ms.\tCant. Conflic.: "+conflic + "\n");
               //   escribo los resultados en el archivo
               out.write("Host: "+sListen.getInetAddress()+"\tDur.de Ejec.: "+ durEjec
               +"ms.\tCant. Conflic.: "+conflic + "\n");
               durEjecTot =durEjecTot+ durEjec;
               durQueryTot = durQueryTot+ durQuery;
               conflicTot =   conflicTot + conflic;
               in.close();
               sListen.close();
          }
          listenSocket.close();
          //   determino los resultados
          float totQuery = (float)(CANT_QUERY*CANT_HOST); // total de consultas
          float tpoSeg = (float)(((float) durEjecTot)/1000.0); // tpo ejec.total en seg.
          int prod = (int) Math.round(totQuery/tpoSeg);// productividad o Throughput
          // tiempo de respuesta
          int tpoRta = (int) Math.round(((float)durQueryTot)/((float)totQuery));
          // cantidad promedio de conflictos
          int conflicProm   =   (int) Math.round(((float)conflicTot)/((float)CANT_HOST));

          System.out.println("Tiempo de Respuesta: "+tpoRta+" ms.");
          System.out.println("Throughput: "+prod+" op./seg.");
          System.out.println("Cantidad Promedio de Conflictos: "+conflicProm );
          out.write("Tiempo de Respuesta: "+tpoRta+" ms.\n");
          out.write("Throughput: "+prod+" op./seg.\n");
          out.write("Cantidad Promedio de conflictos: "+conflicProm + "\n");
          out.close();
        }
        catch (IOException e) { System.out.println( "Init " + e); }
    }


    public static void main(String[] args) {
     try {
      TIPO_REGULACION =args[0].charAt(0);
      PORC_REPLICA=((new Integer(args[1])).intValue());
      CANT_QUERY =((new Integer(args[2])).intValue());
      PORC_READ =((new Integer(args[3])).intValue());
      new Init();
```

```
      }
      catch (Exception e) {System.out.println("Args = Tipo de regulación(M,G o C), "
      +" Porcentaje de replicación, Cantidad de consultas, Porcentaje de lecturas.");}

     }

}
```

B.3 Procesos para ejecuciones Eager

B.3.1 Manager.java

```
/**************************************************************************
 *   Es el proceso principal, encargado de inicializar todos los datos                    *
 *   e iniciar los procesos necesarios para procesar las consultas                          *
**************************************************************************/
import java.lang.*;
import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.Timestamp;

public class Manager extends Thread {


    private final static String TIPO_ACTU = "Sincrónica."; // tipo de actualización
    private   int   PORC_LOCK;// probabilidad de bloqueo de un dato
    private int CANT_HOST = 3; //   cantidad total de Host
    private int NUMBER_HOST;     //   nro. ID del host
    private String[] addrHost; //   direcciones de los host
    private int[] portList;      //   ports de los MasterListener en cada host
    private int[] portRecei;     //   ports de los Receiver en cada host
    private String addrInit;      //   dirección del host iniciador
    private int portInit;         //   port del host iniciador
    private int[][] pesoComunic; // peso asignado a cada una de
    // las aristas del grafo de comunicación entre los host
    private char[][] locateRep; // localización estática de las
    // réplicas sobre los diferentes Host


    private int CANT_DATOS;      //   cantidad de datos sobre los que se realizan
                                 //   las consultas
    private int PORC_REPLICA;   //   porcentaje de replicación
    private char TIPO_REGULACION; //   tipo de regulación de la propagación
    private int   PORC_READ;      //   porcentaje de lecturas

    // para la localizacion de las réplicas
    private final static char MASTER = 'M';   //   master de una réplica
    private final static char SLAVE = 'S';     //   slave de un réplica
    private final static char NOT_OWNER = '-';        //   no tiene la réplica

    private final static char GROUP = 'G'; // regulación Group
```

```
private final static char CENTRAL = 'C'; // para el caso con todos los datos centralizados

MasterSender mSender; //   para enviar las consultas a quien deba procesarlas
MasterListener mListen; //   para recibir los requerimientos remotos
DataManager dataM;   //   para manejar el conjunto de réplicas
Monitor monitor;      //   para generar el informe final
Propagator propagator;   //   para propagar las actualizaciones
Receiver   receiver; //   para recibir las actualizaciones remotas propagadas
SrvSlave srvSlv;   //   para tener el cto. de SlaveSenders y SlaveListener

public Manager() {
      this.start();
}

 //   inicializa el nro. de ID del Host y las
 //   direcciones y port de los host

private void initAddrPortHost(){
      // para leer el archivo de direcciones de cada uno de los host
      FileReader in;
      File inputFile;
      char[] host = new char[1];
      char[] port = new char[5];
      char[] numHost = new char[2];
      char[] blanco = new char[1];
      char[] salto = new char[1];
      int blank = 32;
      char NOT_NUMBER = 'X';

    try {
      addrHost = new String[this.CANT_HOST]; // direcciones de los host
      portList = new int[this.CANT_HOST]; // ports de los Listener en cada host
      inputFile = new File("DatosHost/AddrPortHost.txt");
      in = new FileReader(inputFile);
      for (int i = 0; i <= this.CANT_HOST-1;i++) {
            String addr = new String("");
            in.read(host);
            // leo una dirección de un Host
            while (host[0] != blank) {
                  addr = addr + new String(host);
                  in.read(host);
            }
            // leo el port del listener
            in.read(port);
            in.read(blanco);
            // leo el nro. de ID del Host
            in.read(numHost);
            if (numHost[0] != NOT_NUMBER ) {
                  this.NUMBER_HOST = (new Integer(new String(numHost)).intValue());
            }
            in.read(salto);
            addrHost[i] = new String(addr);
            portList[i] = (new Integer(new String(port)).intValue());
      }
      in.close();
```

```
    }
    catch (IOException e) {System.out.println("Manager: error al abrir DatosHost/AddrPortHost.txt");}
  }


// inicializa los pesos de las comunicaciones entre los host
// a partir de un archivo

private void initPesoComunic(){
      // para leer el archivo de pesos de comunicaciones entre los host
      FileReader in;
      File inputFile;
      char[] peso = new char[1];
      char[] blanco = new char[1];
      char[] salto = new char[1];

    try {
      pesoComunic = new int[this.CANT_HOST][this.CANT_HOST];
      inputFile = new File("DatosHost/PesoComunic.txt");
      in = new FileReader(inputFile);
      for (int i = 0; i <= this.CANT_HOST-1;i++) {
           for (int j = 0; j <= this.CANT_HOST-1;j++) {
                in.read(peso);
                in.read(blanco);
                pesoComunic[i][j] = (new Integer(new String(peso)).intValue());
           }
           in.read(salto);
      }
      in.close();
    }
    catch (IOException e) {System.out.println("Manager: error al abrir DatosHost/PesoComunic.txt");}
}

 //   genera mensajes con los datos iniciales del encabezado del informe
 //   y envía dichos mensajes al Monitor

private void initInforme(int cantQueries){

    String msg = null;   //   para generar y enviar un mensajes
    String tipoRegu = null; //   para imprimir el tipo de regulación

    if (TIPO_REGULACION == this.MASTER) { tipoRegu = " Master-Slave ";}
    else {   if (TIPO_REGULACION == this.GROUP) { tipoRegu = " Group ";}
             else { tipoRegu = " Centralizada ";} }

    //armo el encabezado del informe
    msg = "B.D.D. ESTUDIO DE ACTUALIZACION DE REPLICAS DE DATOS.\t"
              + new Date(System.currentTimeMillis())+"\n\n";
    monitor.putMsg(msg);
    msg = "REGULACION: "+tipoRegu+ "\tPROPAGACION: "+TIPO_ACTU+"\n \n";
    monitor.putMsg(msg);
    msg = "HOST\t\tID:"+this.NUMBER_HOST+"\tDirección: "
          + this.addrHost[this.NUMBER_HOST]+"\tPort: "
          + this.portList[this.NUMBER_HOST]+"\tCantidad: "+this.CANT_HOST+"\n \n";
    monitor.putMsg(msg);
    msg = "CONSULTAS\tCantidad: "+cantQueries+"\tLecturas al: "
```

```
                + PORC_READ +" % \n \n";
    monitor.putMsg(msg);
    msg = "REPLICAS\tCantidad: "+this.CANT_DATOS+"\tReplic. al: "
                + PORC_REPLICA +"%   - Lock: "+ PORC_LOCK +" %\n";
    monitor.putMsg(msg);
    //   informo que réplicas tiene el host y si es Master o Slave de la misma
    String mst = "";
    String slv = "";
    int countMst = 0;
    int countSlv = 0;
    for (int i = 0; i <= this.CANT_DATOS -1;i++) {
         if (locateRep[i][this.NUMBER_HOST] == this.MASTER) {
              mst = mst+" "+ i;
               countMst = countMst + 1;
               if (countMst == 30) {mst = mst +"\n";
                                    countMst = 0;
               }
         }
         else {
              if (locateRep[i][this.NUMBER_HOST] == this.SLAVE) {
                   slv = slv+" "+ i;
                     countSlv = countSlv + 1;
                     if (countSlv == 30) {slv = slv +"\n";
                                    countSlv = 0;
                     }
              }
         }

    }
    msg = "\nMaster de las sig. Réplicas:\t\n"+mst+"\n";
    monitor.putMsg(msg);
    msg = "\nSlave de las sig. Réplicas:\t\n"+slv+"\n\n";
    monitor.putMsg(msg);
    Timestamp ts= new Timestamp(System.currentTimeMillis());// times de inicio
    msg = "HORA DE INICIO:\t\t"+(ts.toString()+"000").substring(11,23)
        +"\n\n";

    monitor.putMsg(msg);
    msg ="ID\tTIPO\tREPLICA\tDESTINO\t\tINICIO\t\tFINALIZ.\tDURAC.\tRESULTADO\n \n";
    monitor.putMsg(msg);
    //   informo el tiempo en que se inicia el procesamiento
    monitor.initTS(ts);
}



// inicializa la localizacion de cada una de las réplicas en los
// diferentes host a partir de un archivo

private void initLocateRep(){

 try {
       // apertura del archivo del esquema de datos
         DataInputStream inFile = new DataInputStream(new FileInputStream("EsqDatos/LRep"
        + this.TIPO_REGULACION + this.PORC_REPLICA + ".txt"));
```

```
        CANT_DATOS = inFile.readInt();   //   lee la cantidad de datos a réplicar

        locateRep = new char[this.CANT_DATOS][this.CANT_HOST];
        for (int i = 0; i <=CANT_DATOS-1;i++) {
            for (int   t = 0; t <=CANT_HOST-1;t++) {
                int nroRep =inFile.readInt();
                int nroHost =inFile.readInt();
                char MstSlv =inFile.readChar();
                locateRep[nroRep][nroHost]=MstSlv ;
            }
        }
        inFile .close();
    }
    catch (IOException e) {System.out.println("Manager: error al abrir "
    + "EsqDatos/LRep" + this.TIPO_REGULACION + this.PORC_REPLICA + ".txt");}
}

//   retorna el peso de la comunicación entre idHost1 y idHost2
public int getPesoHost(int idHost1,int idHost2) {
    return this.pesoComunic[idHost1][idHost2];
}

 //   retorna el ID del Host que tiene la réplica idRep con el menor costo
 //   de comunicación, además, se usa el parámetro masterHost para indicar
 //   si es "necesario" que el host retornado sea o no el master de dicha réplica

private int getHostReplica(int idRep, boolean   masterHost){
    //   para determinar el host de menor peso en comunicación
    int minPeso = Integer.MAX_VALUE;
    int minHost = Integer.MAX_VALUE;

    if (masterHost) {   //   requiere un host Master (para update)
        for (int j = 0; j <= this.CANT_HOST-1;j++) {
            if   ((locateRep[idRep][j] == this.MASTER) &&
                (minPeso > pesoComunic[this.NUMBER_HOST][j])) {
                minPeso = pesoComunic[this.NUMBER_HOST][j];
                minHost =j;
            }
        }
    }
    else { //   puede ser un host Master o Slave (para read)
        for (int i = 0; i <= this.CANT_HOST-1;i++) {
            if   ((locateRep[idRep][i] != this.NOT_OWNER) &&
                (minPeso > pesoComunic[this.NUMBER_HOST][i])) {
                minPeso = pesoComunic[this.NUMBER_HOST][i];
                minHost =i;
            }
        }
    }
    return   minHost;
}

//   retorna true si el host idHost tiene la réplica idRep,
//   sino, retorna falso
public boolean ownerHostReplica(int idRep,int idHost){
    return   locateRep[idRep][idHost]!= this.NOT_OWNER;
```

```
}

//   retorna el id del Host local
public int getIdHost(){
    return   this.NUMBER_HOST;
}

//   retorna la dirección del host idHost
public String getAddrHost(int idHost){
    return   this.addrHost[idHost];
}

//   retorna port del MasterListener del host idHost
public int getPortList(int idHost){
    return   this.portList[idHost];
}

//   retorna port del Receiver del host idHost
public int getPortReceiver(int idHost){
    return   this.portRecei[idHost];
}

//   retorna la cantidad de host
public int getCantHost(){
    return   this.CANT_HOST;
}
//   retorna la cantidad de datos que se replican
public int getCantDatos(){
    return   this.CANT_DATOS;
}

public synchronized void run() {
    String myAddrHost; // para la direccion de mi host local
    int myportList; //   para el port listener de mi host local

    while (true) {
        System.gc();
        initAddrPortHost(); //   inicializo los ID y direcciones de los host
        myAddrHost = addrHost[this.NUMBER_HOST];
        myportList = portList[this.NUMBER_HOST];
        System.out.println("INICIANDO HOST en ADDR: " +myAddrHost+
        " Nro: "+this.NUMBER_HOST+ " PORT: "+myportList);
        //   verifico si quedaron procesos corriendo de anteriores corridas
        if (this.activeCount()==2) {
                System.out.println("Manager: no hay procesos activos.");
        } else {   //   han quedados procesos corriendo
                System.out.println("Manager: hay "+ (this.activeCount()-1)
                                        + " procesos activos.");
                System.exit(1);
        }

        try {
          //   creo un ServerSocke para recibir la señal de inicio
          ServerSocket listenInit;
          Socket sInit;
          listenInit = new ServerSocket(portList[this.NUMBER_HOST]);
```

```
   System.out.println("Manager: esperando INICIO en PORT: "
                           + portList[this.NUMBER_HOST]);
   sInit = listenInit .accept();
   DataInputStream in = new DataInputStream(sInit.getInputStream());
   //   leo los datos sobre los cuales hago el procesamiento
   TIPO_REGULACION = in.readChar();
   PORC_REPLICA = in.readInt();// porcentaje de replicación
   PORC_READ = in.readInt(); // porcentaje de lecturas
   PORC_LOCK = in.readInt(); // probabilidad de bloqueo
   portInit = in.readInt(); // port del host inicializador
    addrInit = in.readLine();   //   dirección del host inicializador
   in.close();
   in = null;
   sInit.close();
   sInit = null;
   listenInit.close();
   listenInit = null;
   if   ((TIPO_REGULACION != this.MASTER) &&
        ( TIPO_REGULACION != this.GROUP ) &&
        ( TIPO_REGULACION != this.CENTRAL )) {break;}; // termino la ejecución
}
catch (IOException e) { System.out.println("Manager: "+e);}
initLocateRep();   //   inicializo la localización de las réplicas
initPesoComunic();   //   inicializo el peso de las comunicaciones
// genero los port para los receivers
portRecei = new int[this.CANT_HOST];
for (int i = 0; i <= this.CANT_HOST-1;i++) {
     portRecei[i] = portList[i]-1;
}

propagator = new Propagator(this);
dataM = new DataManager(this,PORC_LOCK);
receiver = new Receiver(this);
srvSlv = new SrvSlave();
mSender = new MasterSender(this);
mListen = new MasterListener(this);

//   espero a que todos los MasterListener remotos esten listos
try {
  wait();
}
catch (InterruptedException e) {;}

System.out.println("Manager: generando consultas... ");
//   comienzo a ejecutar las consultas
GenerarQueries generarQueries = new GenerarQueries();

//   espero hasta que se hayan ejecutado todas las consultas
try {
     wait();
}
catch (InterruptedException e) {;}

//   notifico a cada uno de mis procesos para que terminen
//   su ejecución
```

```
            synchronized (dataM) {dataM.notify();}
            synchronized (propagator) {propagator.notify();}
            synchronized (srvSlv) {srvSlv.notify();}
            synchronized (monitor) {monitor.notify();}

            //   espero por sus terminaciones
            try {
                 System.out.println("Manager: terminando...");
                 wait(40000);
            }
            catch (InterruptedException e) {;}

        } // while

        System.out.println("SALIENDO DEL SISTEMA.");
    }


     //   lee cada una de las consultas que hay en un archivo y las
     //   envia al MasterSender local para su procesamiento

class GenerarQueries extends Thread {

   public GenerarQueries() {
          this.start();
   }


   public synchronized void run() {

        final int TIME_WAITING = 1; //   tiempo de espera para
                                    // leer una nueva consulta (en ms)


        int cant_Queries;   //   cantidad total de consultas

        char op;   // para leer una operación
        int nroRep; //   para leer sobre que réplica es la operación
        Timestamp ts; //   para la hora de inicio de la consulta
        String nueValor;   //   para el crear el valor de la consulta
        int nroHost;   //   para el nro. de host que tiene la réplica
        String hostAddr;   // para la dirección del host que tiene la réplica
        //   para leer el archivo de consultas
        String horaIni; // para la hora de inicio de la consulta
        try {

           DataInputStream inFile = new DataInputStream(new FileInputStream(
                                    "Consultas/Query"+PORC_READ+".txt"));
           //   leo la cantidad de consultas que hay en el archivo
           cant_Queries = inFile.readInt();
           monitor = new Monitor("I"+TIPO_REGULACION+PORC_REPLICA+
           PORC_READ+".txt",cant_Queries,addrInit,portInit);
           //   inicializa el encabezado del informe
           initInforme(cant_Queries);

           //   para cada una de las consultas, lee una y la envia
```

```
            //   al MasterSender e informa al Monitor con los datos
            //   iniciales de la misma

            for (int i = 0; i <= cant_Queries-1;i++) {
                op =inFile.readChar();
                nroRep =inFile.readInt();
                // armo una nueva query y la envío la MasterSender (mSendMsg)
                ts= new Timestamp(System.currentTimeMillis());
                if (op == SrvMessage.UPDATE){
                    //   creo el valor con el nro de host + hora de inicio de la query
             nueValor = NUMBER_HOST+"-"+(ts.toString()+"000").substring(11,23);

              nroHost = getHostReplica(nroRep,true);
                 }
                else {
                    nueValor =   "X";   //   asigno un valor insignificante si la
                                        //   operación es una lectura
                    nroHost =   getHostReplica(nroRep,false);
                }

                //   determino la dirección del host que tiene la réplica
                hostAddr = addrHost[nroHost];
                //   creo un mensaje con la consulta para   el MasterSender
                SrvMessage msgSender = new SrvMessage(i,nroHost,hostAddr,portList[nroHost]
                                    ,nroRep,op,nueValor);
                //   envío el mensaje
                mSender.putMsg(msgSender);
                horaIni = String.valueOf(ts.getTime()+(ts.getNanos()/1000000));
                //   creo un mensaje con la consulta para   el Monitor
                if (nroHost == NUMBER_HOST) {hostAddr = "LOCAL      ";}
                SrvMessage msgMoni = new SrvMessage(i,nroHost,hostAddr,portList[nroHost]
                                    ,nroRep,op,horaIni);
                //   envío el mensaje
                monitor.putMsg(msgMoni);
                // espero TIME_WAITING ms para leer la próxima consulta
                try {sleep(TIME_WAITING);} // duermo el proceso
                catch (InterruptedException e) {;}
            } //   for

            //   creo un mensaje de finalización para el MasterSender
            SrvMessage srvMsg = new
SrvMessage(Integer.MAX_VALUE,Integer.MAX_VALUE,"",0,0,SrvMessage.END,"");
            mSender.putMsg(srvMsg);
            inFile.close();
          }
          catch (IOException e) {System.out.println("Manager: error al abrir "
                                    + "Consultas/Query"+PORC_READ+".txt");}
      }
    }

      public static void main(String[] args) {
            new Manager();
      }
}
```

B.3.2 DataManager.java

```
/**************************************************************************
 *   Maneja todas las réplicas locales, retornando su valor ante una                    *
 *   lectura, y actualizando su valor ante una escritura.
*
 *   También envía mensajes al Propagador en caso que sea necesario                    *
 *   propagar las actualizaciones.
*
 **************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;
import java.math.*;

public class DataManager extends Thread {

  private Vector dataFile;    //   conjunto de datos replicados con su valor actual

  private Manager myManager = null; //   para comunicarme con el Manager
  private int porcLock; //   para la probabilidad de que un dato este lockeado

  public DataManager(Manager myManager, int porcLock) {
    this.myManager = myManager;
    this.porcLock = porcLock;
    //   inicializo todos datos
    int cantDatos = this.myManager.getCantDatos();
    dataFile = new Vector(cantDatos);
    Replica r;
    for (int i = 0;i <=cantDatos - 1;i++) {
        if (myManager.ownerHostReplica(i,myManager.getIdHost())) {
          r = new Replica(i,"INI"+i);
        } else {
          r = new Replica(i,"NO REPLICADO");
        }
        dataFile.addElement(r);
    }
    this.start();
  }

  // retorna el valor de la réplica id
   public   String get(int id){
        int time_wait;
        Replica r;
        int locked;
        locked = (int)(Math.random()*100);
        if (locked <= porcLock) {
            synchronized (this) {
                 time_wait = (int)(Math.random()*100);
                if (time_wait != 0) {
                          try {wait(time_wait); }
                         catch (InterruptedException e){; }

                }
```

```
          }
        }
         r = ((Replica)(dataFile.elementAt(id)));
         return   r.getValue();

    }


  //   actualización de la réplica id con el valor newValue
  //   utilizado para realizar una actualización de una consulta local
  //   la cual debe ser propagada

     public   void putL(int id, String newValue){
        int time_wait;
        int locked;
        Replica r;

        locked = (int)(Math.random()*100);
        if (locked <= porcLock) {
            synchronized (this) {
                  time_wait = (int)(Math.random()*100);
                 if (time_wait != 0) {
                           try {wait(time_wait); }
                          catch (InterruptedException e){; }
                 }
            }
        }
        //   actualizo la réplica
        synchronized (dataFile.elementAt(id)) {
               r = ((Replica)(dataFile.elementAt(id)));
               r.setValue(newValue);
               dataFile.setElementAt(r,id);
         }
          //   propago la actualización
         myManager.propagator.propagarMsg(new MsgPropag(id,newValue));

    }

  //   actualización de la réplica id con el valor newValue
  //   utilizado para realizar una actualización de proveniente de una
  //   propagación remota
       public   void putR(int id, String newValue){
         int time_wait;
         int locked;
         Replica r;
         locked = (int)(Math.random()*100);
         if (locked <= porcLock) {
            synchronized (this) {
                  time_wait = (int)(Math.random()*100);
                 if (time_wait != 0) {
                           try {wait(time_wait); }
                          catch (InterruptedException e){; }
                 }

            }
        }
```

```
        //   actualizo la réplica
        synchronized (dataFile.elementAt(id)) {
                r = ((Replica)(dataFile.elementAt(id)));
                r.setValue(newValue);
                dataFile.setElementAt(r,id);
        }
    }


  public synchronized void run() {

        Replica r;
        String valor;

        try {
             System.out.println("DataManager: listo.");
             wait();
        }
        catch (InterruptedException e) {;}
        try {
          // genero un archivo con el estado final de las réplicas locales
          File outputFile = new File("Temp/DataFile.txt");
          // abre un FileWriter sobre DataFile.txt
          FileWriter out = new FileWriter(outputFile);
          out.write("REPLICA\tVALOR\n\n");
          int cantDatos = this.myManager.getCantDatos();
          for (int i = 0;i <=cantDatos - 1;i++) {
               r = ((Replica)(dataFile.firstElement()));
               valor = r.getValue();
               dataFile.removeElementAt(0);
               out.write(i+"\t"+valor+"\n");
          }
          out.close();
          System.out.println("FIN DataManager");
        }
        catch (IOException e ) { System.out.println("DataManager: "+e);}
  }
}
```

B.3.3 Propagator.java

```
/*************************************************************************
 *   Popagar las actualizaciones locales provenientes del DataManager                    *
 *************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;

public class Propagator extends Thread {


    private Manager myManager = null; //   para comunicarme con el Manager

  public Propagator(Manager myManager) {
```

```
        this.myManager = myManager;
        this.start();

    }

    //   agrega un nuevo mensaje a la cola
    public synchronized void propagarMsg(MsgPropag msg) {
            Socket sPropagOut = null;   // para   comunicarse con los Receivers
            DataOutputStream out = null;   //para escribir en sPropagOut
        try {
            MsgPropag smsg = msg;
            System.out.println("Propagator: Procesando replica: " + smsg.getIDRepli() +
                                " Valor: "+ smsg.getValue());

            //   realizo el un delay para simular la sincronización
            //   para replicar a n sitios con un protocolo bloqueante cant Msg >= 4*n
            long time;
            time = 0;
            int cantHost = this.myManager.getCantHost();
            int myIdHost = this.myManager.getIdHost();
            for (int i = 0; i <= cantHost -1;i++) {
                if ((this.myManager.ownerHostReplica(smsg.getIDRepli(),i)) && (i != myIdHost)) {
                    time = time +   4 * myManager.getPesoHost(myIdHost,i)+ 5;
                }
            }
            try {
                System.out.println("PROPAGATOR SINCRONIZANDO POR: "+time+" ms");
                sleep(time); // duermo el proceso por el tiempo que simula
                // realizar el protocolo bloqueante
            }
            catch (InterruptedException e) {;}

            //   propago la actualización

            for (int i = 0; i <= cantHost - 1;i++) {
                //   no lo propago a mi mismo, ni a los que no tienen
                //   una copia de la réplica
                if ((this.myManager.ownerHostReplica(smsg.getIDRepli(),i)) && (i != myIdHost)) {
                    while (true) {
                      try {
                        //   determino la dirección y el port al cual debo realizar
                        //   la propagación
                        int portReceiv = this.myManager.getPortReceiver(i);
                        String addHost = this.myManager.getAddrHost(i);
                        System.out.println("Propagator: port destino: " + portReceiv);
                        sPropagOut = new Socket(addHost,portReceiv);
                        out = new DataOutputStream(sPropagOut.getOutputStream());
                        //   envío la actualización
                        //   nro. Réplica + nuevoValor
                        out.writeInt(smsg.getIDRepli());   // nro. Réplica
                        out.writeBytes(smsg.getValue());   // nuevo Valor
                        out.writeByte('\n');
                        // realizo el envío
                        out.flush();
                         out.close();
                        out = null;
```

```
                    sPropagOut.close();
                    sPropagOut = null;
                    break;
                  }
                  catch (SocketException e) {
                       // no pude comunicarme con el Receiver
                       try {wait(0,1);}    // espero 1 nseg.
                       catch (InterruptedException e2) {;}
                       System.out.println(e+"Propagator: no puedo comunicarme con el Receiver "
                       + myManager.getAddrHost(i)+" "+myManager.getPortReceiver(i)); }
                  finally {
                       if (sPropagOut != null) {
                            sPropagOut.close();
                            sPropagOut = null;
                       }
                       if (out != null) {
                            out.close();
                            out = null;
                       }
                  } //   finally
                } //while
             }   //   if
          } //   for
       }//try
       catch (IOException e) { System.out.println("Propagador: "+e); }

  }


  public synchronized void run() {


    System.out.println("Propagator: en espera...");
    try {
      wait();
    }
    catch (InterruptedException e) {System.out.println("FIN Propagator.");}

  } // run
}
```

B.3.4 Receiver.java

```
/**************************************************************************
 *   Su función es la de recibir mensajes provenientes de los Propagadores                    *
 *   remotos y realizar los pedidos correspondiente de actualización al                          *
 *   DataManager
*
 **************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;

  public class Receiver extends Thread {
```

```
private static final int TIME_WAITING = 300000; //   tiempo maximo (ms.) de
                                                    // espera por un requerimiento

private Manager myManager = null; //   para comunicarme con el Manager

public Receiver(Manager myManager) {
    this.myManager = myManager;
    this.start();

}


public synchronized void run() {

    int hostOld,lcOld,hostNew,lcNew,replica;
    String valor;
    ServerSocket listenSocket = null;     // para escuchar los requerimientos
    Socket sRecIn = null;     // para   comunicarse con el Propagador que le
                              // envía una actualización
    DataInputStream in;   // para leer de sRecIn
    // para crear un socket y recibir los requerimientos remotos
    String myHost;
    int myPort;

  try{

    int idHost = myManager.getIdHost();
    myHost = myManager.getAddrHost(idHost);
    myPort = myManager.getPortReceiver(idHost);

    try { listenSocket = new ServerSocket(myPort);
          listenSocket.setSoTimeout(TIME_WAITING);}
    catch (IOException e){System.out.println("Excepción ocurrida creando "
                          +"Receiver: " + myPort);}

    while (true) {
     try {
        System.out.println("Receiver: esperando en el PORT: "+myPort);
        //   acepta las conecciones de los MasterSender
        sRecIn = listenSocket.accept();
        in = new DataInputStream(sRecIn.getInputStream());
        //   determino los datos de la actualización remota
        //   réplica + valor
        replica = in.readInt(); //réplica
        valor = in.readLine();   //valor
        System.out.println("Receiver: Procesando Actualizacion Remota. "
                           + " Replica: " + replica +" Valor: "+valor);
        //   realizo la operación de actualización
        myManager.dataM.putR(replica,valor);
        in.close();
        in = null;
        sRecIn.close();
        sRecIn = null;
      }
```

```
          catch (IOException e) {System.out.println("Receiver "+e); break;}
        } //while
        System.out.println("FIN Receiver.");
        listenSocket.close();
        listenSocket = null;
      } // try
      catch (IOException e) { System.out.println("Receiver: "+ e);}

    } //run

}
```

B.3.5 Monitor.java

```
/**************************************************************************
 *   Toma toda la informacion brindada por los diferentes procesos, y                    *
 *   a partir de ella genera un informe final
*
 *************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.Timestamp;

public class Monitor extends Thread {

    private Vector monitorMsg = new Vector(10); //   para los mensajes de resultados
    //   necesarios para generar el detalle del informe

    //   para generar el archivo del informe final
    private FileWriter out;
    private File outputFile;
    private String InfoName;   //   nombre del informe final
    private Timestamp tsIni,tsFin;   //   para los tiempos de inicio y fin de
                                     //   procesamiento
    private long[] iniTime;     //   para los tiempos de inicio de las consultas
    private long[] endTime;     //   para los tiempos de finalización de las consultas

    private int cantQueries;   // para la cantidad de consultas
    private long duracTotal = 0; // para la duración total de las consultas
    //   dirección y port del host que iniciador y al cual se le retorna
    //   el tiempo total de respuesta o procesamiento y la cantidad de
    //   consultas que tuvieron conflictos
    private String hostResu;
    private int portResu;


  public Monitor(String InfoName,int cantQueries, String host, int port) {
      this.InfoName=InfoName;
      this.cantQueries = cantQueries;
      this.hostResu = host;
      this.portResu = port;
      iniTime = new long[cantQueries];
```

```
        endTime = new long[cantQueries];
        this.start();
    }

    //   inicializa el timestamp de inicio de procesamiento
    public synchronized void initTS(Timestamp ts) {
        this.tsIni = ts;
    }
    //   agrega un nuevo mensaje a la cola
    public synchronized void putMsg(Object msg) {
        this.monitorMsg.addElement(msg);
        this.notify();
    }
    // retorna y eliminia el primer mensaje de la cola
    private Object removeMsg() {
       Object smsg = monitorMsg.firstElement();
       monitorMsg.removeElementAt(0);
       return smsg;
    }

    //   a partir de los archivos temporarios con los datos iniciales y
    //   finales de las consultas, genero los detalles del informe

    private void JoinDetalles() {

        String ini,fin;
        char car[];
      try {
        DataInputStream inFile = new DataInputStream(new FileInputStream("Temp/TempIni.txt"));
        //   leo los datos iniciales de una consulta en tempIni.txt
        ini = inFile.readLine();
        while(ini!= null) {
            System.out.print(".");
            DataInputStream finFile = new DataInputStream(new FileInputStream("Temp/TempFin.txt"));
            //   busco los datos finales de la consulta en el archivo tempFin.txt
            fin = finFile.readLine();
            while ((fin!=null) && (((ini.charAt(0))!=(fin.charAt(0))) ||
                                   ((ini.charAt(1))!=(fin.charAt(1))) ||
                                       ((ini.charAt(2))!=(fin.charAt(2)))))
            {
             fin = finFile.readLine();
             }
             if (fin!=null){
                //   escribo en el informe todos los datos de la consulta
                out.write(ini+" " + fin.substring(3,fin.length()) + "\n");
            }
            finFile.close();
            ini = inFile.readLine();
        }
        inFile.close();
      }
      catch (IOException e ) { System.out.println("Monitor: "+e);}
    }

    //   genera el estado final de las réplicas locales con sus
    //   correspondientes valores
```

```
private void GenEstadoRepli() {

    String ini,fin;
    char car[];

  try {
    out.write("\n");
    out.write("ESTADO FINAL DE LAS REPLICAS LOCALES:\n\n");
    //   recupera los valores finales de las réplicas a partir de los
    //   datos generados por el DataManager (guardados en DataFile.txt)
    DataInputStream inFile = new DataInputStream(new FileInputStream("Temp/DataFile.txt"));
    ini = inFile.readLine();
    while(ini!= null) {
        out.write(ini+ "\n");
        ini = inFile.readLine();
    }
    inFile.close();
  }
  catch (IOException e ) { System.out.println("Monitor: "+e);}
}


//   Genera el Encabezado del Informe a partir de los mensajes de encabezamiento
//   del informe, generados por el Manager

private synchronized void GenEncabezado() {
  try{

    if (monitorMsg.isEmpty()) {
        try {wait(); }
        catch (InterruptedException e) {;}
    }
    //   recupero el primer mensaje y lo elimino de la cola
    Object smsg = this.removeMsg();
    String str= "";
    // mientras sean mensajes del encabezamiento
    while (smsg.getClass()== str.getClass() )   {
        //   escribo en el informe
        out.write((String) smsg);
        if (monitorMsg.isEmpty()) {
            try {wait(); }
            catch (InterruptedException e) {;}
        }
        //   recupero el sig. mensaje y lo elimino de la cola
        smsg = this.removeMsg();
    }
     this.putMsg(smsg); // guardo el ùltimo mensaje que no era un String
  }
  catch (IOException e ) { System.out.println("Monitor: "+e);}
}

//   genero dos archivos temporarios tempIni.txt y tempFin.txt
//   con los datos iniciales y finales de las consultas respectivamente

private synchronized void GenDetalles() {
  try {
```

```
        //   creo dos clases con "basura" para utilizarlas solo para
        //   obtener sus clases
        SrvMessage srvMsg = new
SrvMessage(Integer.MAX_VALUE,Integer.MAX_VALUE,"",0,0,SrvMessage.END,"");
        MsgMonitor moniMsg =    new MsgMonitor(Integer.MAX_VALUE,"",new Timestamp(0));;

        //   creo los archivos temporarios con los datos iniciales y finales

        FileWriter     tempIni = new FileWriter(new File("Temp/TempIni.txt"));
        FileWriter     tempFin = new FileWriter(new File("Temp/TempFin.txt"));

        tsFin = tsIni;   //   inicializo el timestamp final con el inicial
        if (monitorMsg.isEmpty()) {
             try {wait(); }
             catch (InterruptedException e) {;}
        }
        //   recupero el primer mensaje y lo elimino de la cola
        Object smsg = this.removeMsg();
        while (true)   {
             //mensaje con los datos iniciales de la consulta
             //   iniciales = id + tipo + replica + destino + hora inicio
             if (smsg.getClass()== srvMsg.getClass()) {
                  srvMsg = (SrvMessage) smsg;
                  // convierto la hora de inicio en milisegundos a h:m:s:ms
                  long miliseg = Long.valueOf(srvMsg.getMsg()).longValue();
                   int idQuery = srvMsg.getIDQuery();
                  iniTime[idQuery]= miliseg;
                  Timestamp tsI = new Timestamp(miliseg);

                  if (srvMsg.getTipo() == SrvMessage.UPDATE) {

                       tempIni.write( idQuery+"     \tUpdate\t" +
                       srvMsg.getIDRepli()+"\t" + srvMsg.getIdHost()+"-"+srvMsg.getAddrHost()
                       +"\t"+(tsI.toString()+"000").substring(11,23)+"\n");
                  }
                  else {   tempIni.write(idQuery
                   +"     \tRead\t"+srvMsg.getIDRepli()

                   +"\t" + srvMsg.getIdHost()+"-" + srvMsg.getAddrHost()
                    +"\t"+(tsI.toString()+"000").substring(11,23)+"\n");

        }
              }
             //   mensaje con los datos finales de la consulta
             //   finales     = id + hora finalizacion + duración + resultado
             else {
                  moniMsg = (MsgMonitor) smsg;
                  Timestamp tsF = moniMsg.getTime();
                  long miliseg = tsF.getTime()+(tsF.getNanos()/1000000);
                  int idQuery = moniMsg.getIDQuery();
                  endTime[idQuery]= miliseg;
                  long duracion = (endTime[idQuery]-iniTime[idQuery]);
                  if (duracion == 0) {duracion = 1;}
                  duracTotal = duracTotal + duracion;
                  tempFin.write(idQuery + "    \t"+(tsF.toString()+"000")
                  .substring(11,23)+"\t"+ duracion +
```

```
                "\t" + moniMsg.getValue()+"\n");
                //   calculo la hora de finalización del procesamiento
                if (tsFin.before(moniMsg.getTime())){tsFin=moniMsg.getTime();}
            }

            if (monitorMsg.isEmpty()) {
                try {wait(); }   //   espero por un nuevo mensaje o una notificación
                                 //   de que no se terminaro las consultas
                catch (InterruptedException e) {;}
                if (monitorMsg.isEmpty()) {break;} // se terminaron las consultas
            }
            //   recupero el primer mensaje y lo elimino de la cola
            smsg = this.removeMsg();
        }
        tempIni.close();
        tempFin.close();
    }
    catch (IOException e ) { System.out.println("Monitor: "+e);}
}

public synchronized void run() {

    try {
        outputFile = new File("Informes/"+this.InfoName);
        // abre un FileWriter sobre InfoName
        out = new FileWriter(outputFile);

        //   genero el Encabezado del Informe
        GenEncabezado();
        //   genero los archivos temporarios para el Detalle del Informe
        GenDetalles();
        //   genero el detalle del informe a partir de los archivos temporarios
        System.out.print("Generando Informe");
        JoinDetalles();
        //   genero el Pie del Informe
        out.write("\nHORA DE FINALIZACION:\t"+ (tsFin.toString()+"000")
        .substring(11,23)+"\n \n");
        long tpoTotal = ((tsFin.getTime()+(tsFin.getNanos()/1000000))
        -(tsIni.getTime()+(tsIni.getNanos()/1000000)));
        out.write("DURACION DE LA EJECUCION:      \t"+tpoTotal+ " ms.\n \n");

            //   genero el estado final de las réplicas locales con sus
        //   correspondientes valores
        GenEstadoRepli();
        //   genero el detalle de las consultas que tuvieron conflictos

        out.close();
        System.out.println("Monitor: Informe finalizado.");

        //   envío al host iniciador el tiempo   de respuesta y la cantidad
        //   total de conflictos
        while (true) {   //   itero hasta que pueda comunicarme
            Socket sMonitorOut = null;
            DataOutputStream outS = null;
            try {
                sMonitorOut = new Socket(hostResu,portResu);
```

```
            outS = new DataOutputStream(sMonitorOut
            .getOutputStream());
            outS.writeLong(tpoTotal); // envío la duración de la ejecución
            outS.writeLong(duracTotal); // envío la duración total de las consultas
            outS.writeLong(tpoTotal);
             outS.writeInt(0);   // en un método sincrónico no hay conflictos
                                 // de actualización
            outS.flush();
            outS.close();
            outS = null;
            sMonitorOut.close();
            sMonitorOut = null;
            System.out.println("FIN Monitor");
            break;
          }   //try
          catch (SocketException e) {
               // no pude comunicarme con el Inicializador
               try {wait(0,1);}    // espero 1 nseg.
               catch (InterruptedException e2) {;}
                        System.out.println("Monitor: no puedo comunicarme con el Inicializador "+e);
          }
          finally {
               if (sMonitorOut != null) {
                    sMonitorOut.close();
                    sMonitorOut = null;
               }
               if (outS!= null) {
                    outS.close();
                    outS= null;
               }
          } //   finally
        } //   while
      } //   try
      catch (IOException e ) { System.out.println("Monitor: "+e);}
    } // run
}
```

B.3.6 Replica.java

```
/*************************************************************************
 *   Etructura de las réplicas, las mismas tienen un id y un valor actual.                    *
 *************************************************************************/

public class Replica   {

   private int id;
   private String value;

   //   crea una nueva réplica
   public Replica(int id,String value) {
      this.id = id;
      this.value = value;
   }
```

```
    //   retorna el id de la réplica
    public int getIDRepli(){
        return   this.id;
    }

    //   retorna el valor de la réplica
    public String getValue(){
        return   this.value;
    }

    //   actualiza el valor a newValue
    public void setValue(String newValue){
        this.value = newValue;
    }

}
```

B.3.7 MsgPropag.java

```
/**************************************************************************
 *   Etructura de los mensajes enviados al Propagador para indicarle los                    *
 *   los datos de una consulta que debe propagar
*
 **************************************************************************/

public class MsgPropag   {

   private int replica;   //   para el id de la réplica que se propaga su actualización
   private String valor; //   para el nuevo valor de la réplica

   //   crea un nuevo mensaje
   public MsgPropag(int replica, String valor) {
         this.replica = replica;
         this.valor = valor;
   }

   //   retorna el id de la réplica que se propaga
   public int getIDRepli(){
         return   this.replica;
   }

   //   retorna el nuevo valor de la réplica
   public String getValue(){
         return   this.valor;
   }

}
```

B.3.8 Init.java

```
/**************************************************************************
```

```
 *   Sincroniza el inicio de procesamiento de todos los host
*
**************************************************************************/

import java.io.*;
import java.net.*;
import java.util.*;
import java.sql.Timestamp;

public class Init extends Thread {

    private static char TIPO_REGULACION; // tipo regulación M Master-Slave G Group
    private static int PORC_REPLICA;// porcentaje de   replicación
    private static int CANT_QUERY; // cantidad de consultas
    private static int PORC_READ; // porcentaje de lecturas
    private static int PORC_LOCK; // probabilidad de bloqueo de un dato
    private final static int CANT_HOST = 3;
    private int myPort= 10000; //   pare crear listenSocket y recibir los resultados
    private String myAddr= "127.0.0.1"; //   pare enviarla a los demás host
    private String[] addrHost; // direcciones de los host
    private int[] portHost; // ports de los Listener en cada host
    private final static char MASTER = 'M';   //   regulación Master-Slave
    private final static char GROUP = 'G'; // regulación Group
    private final static char CENTRAL = 'C'; // regulación Centralizada

   public Init() {
        initAddrPortHost();
        this.start();
   }

   //   inicializa las direcciones y port de los host
   private void initAddrPortHost(){
         // para leer el archivo de direcciones de cada uno de los host
         FileReader in;
         File inputFile;
         char[] host = new char[1];
         char[] port = new char[5];
         char[] numHost = new char[2];
         char[] blanco = new char[1];
         char[] salto = new char[1];
         int blank = 32;

       try {
         addrHost = new String[this.CANT_HOST]; // direcciones de los host
         portHost = new int[this.CANT_HOST]; // ports de los Listener en cada host
         inputFile = new File("DatosHost","AddrPortHost.txt");
         in = new FileReader(inputFile);
         for (int i = 0; i <= this.CANT_HOST-1;i++) {
              String addr = new String("");
              in.read(host);
              // leo una dirección de un Host
              while (host[0] != blank) {
                   addr = addr + new String(host);
                   in.read(host);
              }
              // leo el port del listener
```

```
            in.read(port);
            in.read(blanco);
            // leo el nro. de ID del Host
            in.read(numHost);
            in.read(salto);
            addrHost[i] = new String(addr);
            portHost[i] = (new Integer(new String(port)).intValue());
        }
        in.close();
      }
      catch (IOException e) {System.out.println("Init: error al abrir DatosHost/AddrHost.txt");}
    }

    public synchronized void run() {

        Socket sListen = null;        //   para   comunicarse con cada uno de los host
        DataInputStream in = null;      //   para leer de sListen
        ServerSocket listenSocket = null;     // para recibir los resultados
        if (TIPO_REGULACION == this.CENTRAL) {PORC_REPLICA = 0;}
        //   envío los datos de inicio a cada uno de los Manager
        for (int i = 0; i <= this.CANT_HOST -1;i++) {

            while (true) {   //   itero hasta que pueda enviar los datos
                             //   al a los host remotos remoto
              try {
                System.out.println("REGULACION: "+this.TIPO_REGULACION+" PORC. REP.:
"+this.PORC_REPLICA+" PORC. LECT.:"+this.PORC_READ);
                System.out.println("Iniciando Host: "+portHost[i]);
                Socket sInitOut = new Socket(addrHost[i],portHost[i]);
                DataOutputStream out = new DataOutputStream(sInitOut.getOutputStream());
                //   envío los datos
                out.writeChar(this.TIPO_REGULACION);   //   tipo de regulación
                out.writeInt(this.PORC_REPLICA);   //   porcentaje de replicación
                out.writeInt(this.PORC_READ);     //   porcentaje de op. de lecturas
                  out.writeInt(this.PORC_LOCK);     //   probabilidad de bloqueo de los datos
                out.writeInt(myPort);     //   port para recibir los resultados
                out.writeBytes(myAddr);   //   mi dirección IP
                out.writeByte('\n');
                out.flush();
                 out.close();
                out = null;
                     sInitOut.close();
                sInitOut = null;
                break;
              }
              catch (IOException e) { System.out.println("Init: no puedo comunicarme con el host: "
              +addrHost[i]+" "+portHost[i]);}
            } ///while
        }
        if   ((TIPO_REGULACION != this.MASTER)&& // exit
          ( TIPO_REGULACION != this.GROUP )&&
          ( TIPO_REGULACION != this.CENTRAL )) { System.exit(0);}

        // espero 10 seg. para que inicializen todos los host
        try {wait(10000);}
        catch (InterruptedException e) {;}
```

```
        //   sincronizo el inicio de todos los host
        //   enviándoles una señal a cada uno de los MasterListeners
        for (int i = 0; i <= this.CANT_HOST -1;i++) {

             while (true) {   //   itero hasta que pueda enviar la señal
               try {
                  System.out.println("Iniciando Listener: "+portHost[i]);
                  Socket sInitOut = new Socket(addrHost[i],portHost[i]);
                  DataOutputStream out = new DataOutputStream(sInitOut.getOutputStream());
                  //   envío la señal
                  out.flush();
                   out.close();
                  out = null;
                  sInitOut.close();
                  sInitOut = null;
                  break;
               }
               catch (IOException e) { System.out.println("Init: no puedo comunicarme con el host: "
               +addrHost[i]+" "+portHost[i]);}
             } ///while
        }

        //   espero los resultados de cada uno de los Host

        try {listenSocket = new ServerSocket(myPort);}
        catch (IOException e) { System.out.println( "Excepción ocurrida creando Init" + myPort); }
        System.out.println("Esperando resultados...");
        try {
          long durEjecTot = 0; // duración total de las ejecuciones de los n host en ms
          long durQueryTot = 0; // duración total de las consultas de los n host en ms
          long conflicTot = 0; // cant.de conflictos totales de los n host

          //   escribo todos los resultados en un archivo
          File outputFile = new
File("Informes/R"+this.TIPO_REGULACION+this.PORC_REPLICA+this.PORC_READ + ".txt");
          FileWriter out = new FileWriter(outputFile);
          for (int i = 0; i <= this.CANT_HOST -1;i++) {
               sListen = listenSocket.accept();
               //   acepto los resultados de un host
               in = new DataInputStream(sListen.getInputStream());
               long durEjec = in.readLong();//duración de la ejecucion del host i
               long durQuery = in.readLong();//duración de las consultas del host i
               int conflic = in.readInt();//cant de conflictos del host i
               System.out.println("Host: "+sListen.getInetAddress()+"\tDur.de Ejec.: "+
               durEjec +" ms.\tCant. Conflic.: "+conflic + "\n");
               //   escribo los resultados en el archivo
               out.write("Host: "+sListen.getInetAddress().getHostAddress()+"\tDur.de Ejec.: "+ durEjec
               +"ms.\tCant. Conflic.: "+conflic + "\n");
               durEjecTot =durEjecTot+ durEjec;
               durQueryTot = durQueryTot+ durQuery;
               conflicTot =    conflicTot + conflic;
               in.close();
               sListen.close();
          }
          listenSocket.close();
```

```
            //   determino los resultados
            float totQuery = (float)(CANT_QUERY*CANT_HOST); // total de consultas
            float tpoSeg = (float)(((float) durEjecTot)/1000.0); // tpo ejec.total en seg.
            int prod = (int) Math.round(totQuery/tpoSeg);// productividad o Throughput
            // tiempo de respuesta
            int tpoRta = (int) Math.round(((float)durQueryTot)/((float)totQuery));
            // cantidad promedio de conflictos
            int conflicProm   =   (int) Math.round(((float)conflicTot)/((float)CANT_HOST));

            System.out.println("Tiempo de Respuesta: "+tpoRta+" ms.");
            System.out.println("Throughput: "+prod+" op./seg.");
            System.out.println("Cantidad Promedio de Conflictos: "+conflicProm );
            out.write("Tiempo de Respuesta: "+tpoRta+" ms.\n");
            out.write("Throughput: "+prod+" op./seg.\n");
            out.write("Cantidad Promedio de conflictos: "+conflicProm + "\n");
            out.close();
          }
          catch (IOException e) { System.out.println( "Init " + e); }
    }

    public static void main(String[] args) {
     try {
      TIPO_REGULACION =args[0].charAt(0);
      PORC_REPLICA=((new Integer(args[1])).intValue());
      CANT_QUERY =((new Integer(args[2])).intValue());
      PORC_READ =((new Integer(args[3])).intValue());
      PORC_LOCK = ((new Integer(args[4])).intValue());
      new Init();
     }
     catch (Exception e) {System.out.println("Args = Tipo de regulación(M,G o C), "
     +" Porcentaje de replicación, Cantidad de consultas, Porcentaje de lecturas, "
     + "Probabilidad de bloqueo");}
    }
}
```

B.4 Generadores

B.4.1 GenQuery.java

```
/**************************************************************************
 *   genera un archivo Query[% Lecturas].txt de consultas aleatorias,
          *
 *   donde [% Lecturas] = 0, 10, 20...100
          *
 *   La estructura generada es un cto. de enteros y caracteres con la
          *
 *   siguiente forma:
          *
 *   Cant. de Queries n (Int)
          *
 *   Operacion 1(char) Replica 1(Int)... Operacion n (Char) Replica n(Int)
          *
 **************************************************************************/
```

```
import java.io.*;
import java.util.*;
import java.math.*;

public class GenQuery extends Thread{
    //  para guardar los argumentos recibidos como parámetros
    private   static int CANT_DATOS; //  para la cantidad de datos
    private   static int CANT_QUERIES; //  para la cantidad de consultas
    private   static int CANT_READ;   //  para la cantidad de lecturas
    private   static int PORC_READ;   //  para el porcentaje de lecturas

    private static final byte OP_READ = 0;
    private static final byte OP_UPDATE = 1;

  public GenQuery() {
        System.gc();
        this.start();
  }

  public synchronized void run() {
      //  para generar el achivo de consultas
      DataOutputStream outFile ;
      //  para guardar la operación a realizar
      char[] operaciones = new char[CANT_QUERIES];
      //  para guardar sobre que dato se realiza la operación
      int[] datos = new int[CANT_QUERIES];

    try {
      System.out.println("Generando "+CANT_QUERIES+" con el " + PORC_READ
                          + "% de lectutas ("+CANT_READ+" lecturas)");
      outFile = new DataOutputStream(new FileOutputStream("Query"+PORC_READ+".txt"));
      outFile.writeInt(CANT_QUERIES);
      String operacion;
      System.out.println("");
      //  genera la cantidad de operaciones de lecturas requeridas
      for (int i = 0; i <=CANT_READ -1;i++) {
          int dato = (int)(Math.random()*CANT_DATOS);
          operaciones[i]='R';
          datos[i] = dato;
      }

      //  genera el resto de las operaciones con actualizaciones
      for (int i = CANT_READ; i <=CANT_QUERIES -1;i++) {
          int dato = (int)(Math.random()*CANT_DATOS);
          operaciones[i]='U';
          datos[i] = dato;
      }

      //  realizo un intercambio aleatorio de las operaciones
      for (int i = 0; i <=CANT_QUERIES -1;i++) {
          int pos1 = (int)(Math.random()*CANT_QUERIES);
          int pos2 = (int)(Math.random()*CANT_QUERIES);
          // swap pos1, pos2
          char auxOp;   //   intercambio la operación
          auxOp = operaciones[pos1];
```

```
            operaciones[pos1] = operaciones[pos2];
            operaciones[pos2] = auxOp;
            int auxDat;   //   intercambio los datos
            auxDat = datos[pos1];
            datos[pos1] = datos[pos2];
            datos[pos2] = auxDat;
        }

        //   grabo las consultas en el archivo
        for (int i = 0; i <=CANT_QUERIES -1;i++) {
            System.out.print(operaciones[i]);
            System.out.print(datos[i]+" ");
            outFile.writeChar(operaciones[i]);
            outFile.writeInt(datos[i]);
        }
        System.out.println(" ");
        outFile.close();
      }
      catch (IOException e) {System.out.println("IO: error ");}
    }


    public static void main(String[] args) {
     try {
      CANT_QUERIES =((new Integer(args[0])).intValue());
      CANT_DATOS = ((new Integer(args[1])).intValue());
      PORC_READ = ((new Integer(args[2])).intValue());
      CANT_READ = (int) (((CANT_QUERIES)*PORC_READ)/100);
      new GenQuery();
     }
     catch (Exception e)
      {System.out.println("Args = Cantidad de Consultas, Cantidad de Datos, Porcentaje de Lecturas");}

    }
}
```

B.4.2 GenRepli.java

```
/************************************************************************************ *
 * genera archivos LRep[tipo][%rep].txt de distribuciones de réplicas
          *
 * aleatorias sobre los host a partir de un % de replicacion donde:
          *
 * [tipo] = G (Group) o M (MasterSlave) y [%rep] = 0, 10, 20...100
          *
 * genera dos archivos, uno para Group y uno para Master-Slave
          *
 * La estructura generada es un cto. de enteros con la sig. forma:
          *
 * Cantidad de Datos o Réplicas n (Int)
          *
 * Replica1(int)Host1(Int)Propiedad(char)...Replica1(int) Hostn(Int)Propiedad(char)
          *
 * Replican(int)Host1(Int)Propiedad(char)...Replican(int) Hostn(Int)Propiedad(char)
          *
```

```
 ********************************************************************************/

import java.io.*;
import java.util.*;
import java.math.*;

public class GenRepli extends Thread {
    //   para guardar los argumentos recibidos como parámetros
    private static int CANT_HOST;   //   cantidad de host
    private static int CANT_DATOS;   //   para la cantidad de datos
    private static int PORC_REP;   // para el porcentaje de replicación

    private final static char NO_REP = '-';
    private final static char MASTER_REP = 'M';
    private final static char SLAVE_REP = 'S';

  public GenRepli() {
        System.gc();
        this.start();
  }


  public synchronized void run() {
    //   para generar los achivos de distribución de réplicas
    DataOutputStream outFileM ;
    DataOutputStream outFileG ;
    //   para localizar estáticamente las réplicas para una regulación Master-Slave
    char[][] locateRepM = new char[this.CANT_DATOS][this.CANT_HOST];
    //   para localizar estáticamente las réplicas para una regulación Group.
    char[][] locateRepG = new char[this.CANT_DATOS][this.CANT_HOST];

    try {
      // archivo para Master Slave
      outFileM = new DataOutputStream(new FileOutputStream("LRepM"+PORC_REP+".txt"));
      outFileM.writeInt(CANT_DATOS);
      // archivo para Group
      outFileG = new DataOutputStream(new FileOutputStream("LRepG"+PORC_REP+".txt"));
      outFileG.writeInt(CANT_DATOS);

      // cant. de copias o réplicas a realizar segun el % de replicación
      int CANT_REPLICAS = (int) (((CANT_HOST-1)*PORC_REP)/100);
      //   CANT_HOST-1 de host sobre los que se hacen las copias

      System.out.println("Cantidad de Host: "+CANT_HOST);
      System.out.println("Cantidad de Datos: "+CANT_DATOS);
      System.out.println("Porcentaje de Replicación: "+PORC_REP);
      System.out.println("Cantidad de réplicas: "+CANT_REPLICAS);

      //inicializo los vectores para Master-Salve
      for (int i = 0; i <=CANT_DATOS-1;i++) {
          for (int t = 0; t <=CANT_HOST-1;t++) {
              locateRepM[i][t] =   NO_REP;
          }
      }

      //inicializo los vectores para Group
```

```
        for (int i = 0; i <=CANT_DATOS-1;i++) {
            for (int t = 0; t <=CANT_HOST-1;t++) {
                locateRepG[i][t] =   NO_REP;
            }
        }


        System.out.println("DISTRIBUCION MASTER-SLAVE");
        //   para cada uno de los datos
        for (int i = 0; i <=CANT_DATOS-1;i++) {
            boolean setMaster = false; // para ubicar solo un master
            //   distribuyo todas sus copias
            for (int t = 0; t <=CANT_REPLICAS;t++) {
                int idHost;
                //   busco algún host que no tenga una réplica del dato i
                while (true) {
                    idHost =(int)(Math.random()*CANT_HOST);
                    if (locateRepM[i][idHost]== NO_REP) {break;}
                }
                // Master - Slave
                if (!setMaster) { //   el master no fue seteado
                    locateRepM[i][idHost] = MASTER_REP;
                    System.out.println("Replica: "+i+" HOST: "+idHost+" MASTER");
                    setMaster = true; }
                else {                //   el master fue seteado
                    locateRepM[i][idHost] = SLAVE_REP;
                    System.out.println("Replica: "+i+" HOST: "+idHost+" SLAVE");
                }
               // en group todos son Master de sus réplicas
               locateRepG[i][idHost] = MASTER_REP;
           }
        }
        System.out.println("PARA GROUP TIENE LA MISMA DISTRIBUCION, PERO TODOS SON
MASTERS");

        //   guardo en el archivo la localización   master-slave
        for (int i = 0; i <=CANT_DATOS-1;i++) {
            for (int t = 0; t <=CANT_HOST-1;t++) {
                outFileM.writeInt(i); // réplica
                outFileM.writeInt(t); // host
                outFileM.writeChar(locateRepM[i][t]); // propiedad
            }
        }

        //   guardo en el archivo la localización   master-slave
        for (int i = 0; i <=CANT_DATOS-1;i++) {
            for (int t = 0; t <=CANT_HOST-1;t++) {
                outFileG.writeInt(i); //replica
                outFileG.writeInt(t); //host
                outFileG.writeChar(locateRepG[i][t]);
            }
        }
        //   cierro los archivos
        outFileM.close();
        outFileG.close();
    }
```

```
    catch (IOException e) {System.out.println("IO: error " + e);}
  }

  public static void main(String[] args) {
    try {
      CANT_HOST = ((new Integer(args[0])).intValue());
      CANT_DATOS = ((new Integer(args[1])).intValue());
      PORC_REP = ((new Integer(args[2])).intValue());
      new GenRepli();
    }
    catch (Exception e) {System.out.println("Args = Cantidad de Host, Cantidad de Datos,"
    + " Porcentaje de Replicación");}
  }
}
```

B.4.3 GenRepliBal.java

```
/*****************************************************************************
 *   genera archivos LRep[tipo][%rep].txt de distribuciones de réplicas      *
 *   con carga balanceada sobre los host a partir de un % de replicación donde:  *
 *   [tipo] = G (Group) o M (MasterSlave) y [%rep] = 0, 10, 20...100         *
 *   genera dos archivos, uno para Group y uno para Master-Slave             *
 *   La estructura generada es un cto. de enteros con la sig. forma:         *
 *   Cantidad de Datos o Réplicas n (Int)                                    *
 *   Replica1(int)Host1(Int)Propiedad(char)...Replica1(int) Hostn(Int)Propiedad(char)  *
 *   Replican(int)Host1(Int)Propiedad(char)...Replican(int) Hostn(Int)Propiedad(char)  *
 *   El balance se hace de la sig. manera, si tenemos n sitios Si,0<=i<=n-1, y  *
 *   m datos Dj, 0 <=j<=m-1. Si tenemos que distribuir el dato Dj con  r réplicas  *
 *   Djr con r<=n, hacemos:                                                  *
 *   La 1º réplica Dj0 es almacenado en el sitio S(j mod n)                  *
 *   La 2º réplica  Dj1 es almacenada en el sitio S(j mod n)  y  S((j+1) mod n)  *
 *   La 3º réplica se agrega una copia Dj2 en el sitio S((j+2) mod n)        *
 *                ...                                                        *
 *   La rº réplica se agrega una copia Dir-1 en el sitio S((j+r-1) mod n)    *
 *****************************************************************************/

import java.io.*;
import java.util.*;
import java.math.*;

public class GenRepliBal extends Thread {
    //  para guardar los argumentos recibidos como parámetros
    private static int CANT_HOST;  //  cantidad de host
    private static int CANT_DATOS;  //  para la cantidad de datos
    private static int PORC_REP;  // para el porcentaje de replicación

    private final static char NO_REP = '-';
    private final static char MASTER_REP = 'M';
    private final static char SLAVE_REP = 'S';

  public GenRepliBal() {
       System.gc();
       this.start();
  }
```

```
public synchronized void run() {
  //   para generar los achivos de distribución de réplicas
  DataOutputStream outFileM ;
  DataOutputStream outFileG ;
  //   para localizar estáticamente las réplicas para una regulación Master-Slave
  char[][] locateRepM = new char[this.CANT_DATOS][this.CANT_HOST];
  //   para localizar estáticamente las réplicas para una regulación Group.
  char[][] locateRepG = new char[this.CANT_DATOS][this.CANT_HOST];

  try {

    // archivo para Master Slave
    outFileM = new DataOutputStream(new FileOutputStream("LRepM"+PORC_REP+".txt"));
    outFileM.writeInt(CANT_DATOS);
    // archivo para Group
    outFileG = new DataOutputStream(new FileOutputStream("LRepG"+PORC_REP+".txt"));
    outFileG.writeInt(CANT_DATOS);

    // cant. de copias o réplicas a realizar según el % de replicación
    int CANT_REPLICAS = (int) (((CANT_HOST-1)*PORC_REP)/100);
    //   CANT_HOST-1 de host sobre los que se hacen las copias

    System.out.println("Cantidad de Host: "+CANT_HOST);
    System.out.println("Cantidad de Datos: "+CANT_DATOS);
    System.out.println("Porcentaje de Replicación: "+PORC_REP);
    System.out.println("Cantidad de réplicas: "+CANT_REPLICAS);

    //inicializo los vectores para Master-Salve
    for (int i = 0; i <=CANT_DATOS-1;i++) {
        for (int t = 0; t <=CANT_HOST-1;t++) {
            locateRepM[i][t] =   NO_REP;
        }
    }

    //inicializo los vectores para Group
    for (int i = 0; i <=CANT_DATOS-1;i++) {
        for (int t = 0; t <=CANT_HOST-1;t++) {
            locateRepG[i][t] =   NO_REP;
        }
    }

    System.out.println("DISTRIBUCION MASTER-SLAVE");
    //   para cada uno de los datos
    for (int i = 0; i <=CANT_DATOS-1;i++) {
        boolean setMaster = false; // para ubicar solo un master
        //   distribuyo todas sus copias
        for (int t = 0; t <=CANT_REPLICAS;t++) {
            //   determino el host para contener la replica del dato i
            int idHost = (i+t) % CANT_HOST;
            // Master - Slave
            if (!setMaster) { //   el master no fue seteado
                locateRepM[i][idHost] = MASTER_REP;
                System.out.println("Replica: "+i+" HOST: "+idHost+" MASTER");
                setMaster = true; }
```

```
                else {              //   el master fue seteado
                    locateRepM[i][idHost] = SLAVE_REP;
                    System.out.println("Replica: "+i+" HOST: "+idHost+" SLAVE");
                }
               // en group todos son Master de sus réplicas
               locateRepG[i][idHost] = MASTER_REP;
            }
        }
        System.out.println("PARA GROUP TIENE LA MISMA DISTRIBUCION, PERO TODOS SON
MASTERS");

        //   guardo en el archivo la localización   master-slave
        for (int i = 0; i <=CANT_DATOS-1;i++) {
            for (int t = 0; t <=CANT_HOST-1;t++) {
                outFileM.writeInt(i); // réplica
                outFileM.writeInt(t); // host
                outFileM.writeChar(locateRepM[i][t]); // propiedad
            }
        }

        //   guardo en el archivo la localización   master-slave
        for (int i = 0; i <=CANT_DATOS-1;i++) {
            for (int t = 0; t <=CANT_HOST-1;t++) {
                outFileG.writeInt(i); // replica
                outFileG.writeInt(t); // host
                outFileG.writeChar(locateRepG[i][t]); // propiedad
            }
        }
        //   cierro los archivos
        outFileM.close();
        outFileG.close();
      }
      catch (IOException e) {System.out.println("IO: error "+e);}
    }

    public static void main(String[] args) {
      try {
        CANT_HOST = ((new Integer(args[0])).intValue());
        CANT_DATOS = ((new Integer(args[1])).intValue());
        PORC_REP = ((new Integer(args[2])).intValue());
        new GenRepliBal();
      }
      catch (Exception e) {System.out.println("Args = Cantidad de Host, Cantidad de Datos,"
                              + " Porcentaje de Replicación");}
    }
}
```

B.4.4 GenRepliCen.java

```
/*****************************************************************************
 *   genera archivos LRep[tipo].txt de distribuciones de réplicas            *
 *   [tipo] = C (Centralizado)                                               *
 *   La estructura generada es un cto. de enteros con la sig. forma:         *
 *   Cantidad de Datos o Réplicas n (Int)                                    *
 *   Replica1(int)Host1(Int)Propiedad(char)...Replica1(int) Hostn(Int)Propiedad(char)  *
```

```
 *   Replican(int)Host1(Int)Propiedad(char)...Replican(int) Hostn(Int)Propiedad(char)                *
 *   Al ser centralizado Host1 será el host 0 para todas las réplicas, con propiedad                *
 *   master                                                                                         *
 ************************************************************************************************/

import java.io.*;
import java.util.*;
import java.math.*;

public class GenRepliCen extends Thread {
    //   para guardar los argumentos recibidos como parametros
    private static int CANT_HOST;   //   cantidad de host
    private static int CANT_DATOS;   //   para la cantidad de datos

    private final static char NO_REP = '-';
    private final static char MASTER_REP = 'M';

  public GenRepliCen() {
        System.gc();
        this.start();
  }


  public synchronized void run() {
    //   para generar archivo centralizado de réplicas
    DataOutputStream outFileC ;

    //   para localizar estáticamente las réplicas
    char[][] locateRepC = new char[this.CANT_DATOS][this.CANT_HOST];

    try {

      outFileC = new DataOutputStream(new FileOutputStream("LRepC0.txt"));
      outFileC.writeInt(CANT_DATOS);

      // cant. de copias o réplicas a realizar
      int CANT_REPLICAS = 1;


      System.out.println("Cantidad de Host: "+CANT_HOST);
      System.out.println("Cantidad de Datos: "+CANT_DATOS);
      System.out.println("Cantidad de Réplicas: "+CANT_REPLICAS);

      //inicializo los vectores
      for (int i = 0; i <=CANT_DATOS-1;i++) {
          //   el host 0 (central) sera master para todos los datos
          locateRepC[i][0] = MASTER_REP;
          for (int t = 1; t <=CANT_HOST-1;t++) {
              locateRepC[i][t] =   NO_REP;
          }
      }
      System.out.println("Centralización de los datos sobre el host 0, terminada.");

      //   guardo   el archivo de localización centralizada
      for (int i = 0; i <=CANT_DATOS-1;i++) {
          for (int t = 0; t <=CANT_HOST-1;t++) {
```

```
                outFileC.writeInt(i); // réplica
                outFileC.writeInt(t); // host
                outFileC.writeChar(locateRepC[i][t]); // propiedad
            }
        }

        //   cierro el archivo
        outFileC.close();

      }
      catch (IOException e) {System.out.println("IO: error "+e);}
    }

    public static void main(String[] args) {
      try {
        CANT_HOST = ((new Integer(args[0])).intValue());
        CANT_DATOS = ((new Integer(args[1])).intValue());
        new GenRepliCen();
      }
      catch (Exception e) {      System.out.println("Args = Cantidad de Host, Cantidad de Datos");}
    }
}
```

Bibliografía

[ABKW98] T. A. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive?. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 484-495, Seattle, Washington, June 1998.

[AES97] D. Agrawal, A. El Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson, Arizona, May 1997.

[Alo97] G. Alonso. Partial database replication and group communication primitives. Institute for Information Systems, ETH Zentrum, Zürich CH-8092, Switzerland, January 17, 1997.

[AO93] P. R. Adusumilli and L. J. Osborne. An integrated solution for managing replicated data in distributed systems. *ACM*, 1993.

[BG92] D. Bell and J. Grimson. *Distributed Database Systems*. Addison-Wesley, 1992.

[BH93] B. Bhargava and A. Helal. Efficient availability mechanisms in distributed database systems. *ACM*, 1993.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database System*s. Addison-Wesley, 1987.

[BK97a] Y. Breitbart and H. F. Korth. Replication and consistency: Being lazy helps sometimes. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems,* pages 173-184, Tucson, Arizona, May 1997.

[BK97b] Y. Breitbart and H. F. Korth. Replication and consistency in a distributed environment. Technical Report, Bell Labs, 1997.

[BKR+99] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. *ACM SIGMOD*, 1999.

[BI] B. Badrinath and T. Imielinski. Replication and mobility. Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

[Bur97] M. Buretta. *Data replication: Tools and Techniques for Managing Distributed Information.* Wiley, 1997

[CHKS] S. Ceri, M. Houtsma, A. Keller, and P. Samarati. The case for independent updates. Department of Computer Science, Stanford University, Stanford, CA 94305-2140, USA.

[CHKS94] S. Ceri, M. Houtsma, A. Keller, and P. Samarati. A classification of update methods for replicated databases. Technical Report CS-TR-91-1392, Department of Computer Science, Stanford University, May 1991.

[CP92] S. Chen and C. Pu. A structural classification of integrated replica control mechanisms. Technical Report CUCS-006-92, Department of Computer Science ,Columbia University, New York, NY 10027, 1992.

[CRR96] P. Chundi, D. J. Rosenkrants, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *Proceedings of the Twelveth International Conference on Data Engineering*, New Orleans, Louisiana, 1996.

[CS93] D. Comer and D. Stevens. *Internetworking with TCP/IP Vol III : Client-Server Programming And Applications, BSD Socket Version* , Prentice – Hall, Inc., 1993.

[CW96] M. Campione and K. Walrath. *The Java Tutorial.* Addison-Wesley, 1996.

[Dra98] Information & Communcations Systems Research Group, ETH Zürich and Laboratoire de Systèmes d'Exploitation (LSE), EPF Lausanne. DRAGON: Database Replication Based on Group Communication, May 1998.
http://www.inf.ethz.ch/department/IS/iks/research/dragon.html.

[Eck97] B. Eckel. *Thinking in Java*. MindView Inc. 1997.

[Fla96] C. Flaviu. Synchronous and asynchronous group communication. *ACM*, 1996.

[GHOS96] J. N. Gray, P. Helland, P. O'Neil and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173-82, Montreal, Canada, June 1996.

[Gol95] R. Goldring. Things every update replication customer should know. In *Proceeding of the ACM SIGMOD*, (439-440), June 1995.

[GS97] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Compute*r, 30(4):68–74, April 1997.

[HAE] J. Holliday, D. Agrawal, and A. El Abbadi. Database replication: If you must be lazy, be consistent. Department of Computer Science, University of California at Santa Barbara, Santa Barbara, CA 93106.

[HAE99] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group multicast. In *Proceedings of IEEE International Symposium on Fault Tolerant Computing (FTCS29)*, pages 158-165, 1999.

[Ham] B. Hammond. Wigman, a replication service for Microsoft Access and Visual Basic, Microsoft White Paper.

[HSW94] Y. Huang, P. Sistla, and O. Wolfson. Data replication for mobile computers. *ACM SIGMOD*, May 1994.

[Jav96] Java FAQ list and tutoral: a work in progress. http:// sunsite.unc.edu /javafaq/ javafaq.html, 1996.

[Jin] W. Jin. Replication data management in distributed database system. Department of Information and Computer Science, University of Hawaii at Manoa, Honolulu, HI 96826.

[KA] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database System*s. To appear.

[KA98] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proceedings of the 18th Internationnal Conference on Distributed Computing Systems (ICDCS)*, Amsterdam, The Netherlands, May 1998.

[KA00] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of the 26th VLDB Conferenc*e, Caïro, Egypt, 2000.

[LC98] K. Lee and Y. Chin. A new replication strategy for unforeseeable disconnection under agent-based mobile computing system. *IEEE*, 1998.

[LCC] C. Lin,, G. Chiu, and C. Cho. An efficient quorum-based scheme for managing replicated data in distributed systems. Department of Electrical Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan.

[LLG90] R. Ladin, B. Liskov and S. Ghemawat. Providing high availability using lazy replication. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, August 1990.

[Ora] Oracle8 Concepts Release 8.0 A58227-01 Library Product Contents Database Replication. http://technet.oracle.com/doc/server.804/a58227/ch_repli.htm

[ÖV91] M. T. Özsu, and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.

[OZS96] M. T. Özsu and P. Valduriez. Distributed and parallel database systems. *ACM Computing Surveys*, Vol. 28, No. 1, March 1996.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proceedings of the 16th Symposium on Reliable Distributed Systems (SRDS-16)*, Durham, North Carolina, USA, October 1997.

[PKL] C. Park, M. Kim, and Y. Lee. A replica control method for improving availability for read-only transactions. Department of Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusong-dong, Yusong-gu, Taejon, 305-701, Korea.

[PL91] Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 377-386, Denver, May 1991.

[PSM98] E. Pacitti, E. Simon, and R. Melo. Improving data freshness in lazy schemes. In *Proceedings of ICDCS'98*, Amsterdam, Netherlands, May 1998. *IEEE* Computer Society.

[Sai00] Y. Saito. Optimistic replication algorithms. August 17, 2000.

[SAS+96] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data replication in Mariposa. In *Proceedings of the Twelveth International Conference on Data Engineering*, New Orleans, Louisiana, 1996.

[Shr] A. Shrivastav. Analysis and comparison of replication schemes for mobile data management. Depertment of Computer Science and Engineering, University of Texas at Arlington.

[SKS98] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Fundamentos de Bases de Datos*. McGraw-Hill, 1998.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, SE-5:188-194, May 1979.

[Syb] Sybase replication server: a practical architecture for distributing and sharing corporate information. http://www.sybase.com/products/datamove/repsrv.htm ,

[TS] O. Theel and T. Strauβ. An excursion to the zoo of dynamic coterie-based replication schemes. Department of Computer Science, Darmstadt University of Technology, D-64283 Darmstadt, Germany.

[Van00] R. Vandewall. Database replication prototype. Department of Mathematics and Computer Science, University of Groningen, Groningen , The Netherlands.

[WCYC] Y. Wu, Y. Chang, S. Yuan, and H. Chang. A new quorum-based replica control protocol. Dept. of Computer and Information Science, National Chiao-Tung University, Hsinchu, Taiwan.

[WJ92] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *Proccedings of the 11th ACM Principles of Database Systems*, San Diego, June 1992.

[WJH97] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, Vol. 22, Nro. 2, pages 255-314, June 1997.

[WPS99] M. Wiesmann, F. Pedone, and A. Schiper. A systematic classification of replicated database protocols based on atomic broadcast. In *Proceedings of the* 3 *rd Europeean Research Seminar on Advances in Distributed Systems (ERSADS'99*), Madeira Island (Portugal), April 23–28, 1999. BROADCAST Esprit WG 22455.

[WPS+00a] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of* 20 *th International Conference on Distributed Computing Systems (ICDCS'2000*), pages

264–274, Taipei, Taiwan, R.O.C., April 2000. *IEEE* Computer Society Los Alamitos California.

[WPS+00b] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: A three parameter classification. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000*), Nürnberg, Germany, October 2000. *IEEE* Computer Society.

[WZ99] L. Wang and W. Zhou. Primary-backup object replication in Java. *IEEE*,1999.

[ZA98] M. Zanconi y J. Ardenghi. Un protocolo para replicación dinámica de datos. IV Congreso Argentino de Ciencias de la Computación (CACIC98).

[ZH99] W. Zhou and R. Holmes. The design and simulation of a hybrid replication control protocol. *IEEE*, 1999.