Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ingeniería en Ciencias y Sistemas

MEMORIA DISTRIBUIDA COMPARTIDA

Eddie Josué González Pineda

Asesorado por el Ing. Juan Antonio Cabrera Aguirre

Guatemala, octubre de 2006

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

MEMORIA DISTRIBUIDA COMPARTIDA

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA

POR

EDDIE JOSUÉ GONZÁLEZ PINEDA

ASESORADO POR EL ING. JUAN ANTONIO CABRERA AGUIRRE

AL CONFERÍRSELE EL TÍTULO DE

INGENIERO EN CIENCIAS Y SISTEMAS

GUATEMALA, OCTUBRE DE 2006

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERIA

NÓMINA DE JUNTA DIRECTIVA

DECANO	Ing. Murphy Olympo Paiz Recinos
VOCAL I	Inga. Glenda Patricia Garcia Soria
VOCAL II	Lic. Amahán Sánchez Alvarez
VOCAL III	Ing. Julio David Galicia Celada
VOCAL IV	Br. Kenneth Issur Estrada Ruiz
VOCAL V	Br. Elisa Yazminda Vides Leiva
SECRETARIO	Inga. Marcia Ivonne Véliz Vargas

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANO	Ing. Sydney Alexander Samuels Milson
EXAMINADOR	Inga. Claudia Liseth Rojas Morales
EXAMINADOR	Ing. Hiram Sean Urrutia Gramajo
EXAMINADOR	Ing. Franklin Barrientos Luna
SECRETARIO	Ing. Pedro Antonio Aguilar Polanco

HONORABLE TRIBUNAL EXAMINADOR

Cumpliendo con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

MEMORIA DISTRIBUIDA COMPARTIDA,

tema que me fuera asignado por la Dirección de la Escuela de Ingeniería en Ciencias y Sistemas, con fecha junio de 2005.

Eddie Josué González Pineda

Guatemala, 02 de Agosto de 2006

Ing. Carlos Alfredo Azurdia Morales
Coordinador Comisión de Trabajos de Graduación
Escuela de Ciencias y Sistemas
Facultad de Ingeniería
Universidad San Carlos de Guatemala

Ing. Azurdia:

Por medio de la presente hago de su conocimiento que he tenido a bien revisar el trabajo de graduación de Eddie Josué González Pineda, titulado “Memoria Distribuida Compartida”, por lo cual me permito recomendar dicho trabajo para la respectiva revisión por parte de la comisión de trabajos de graduación de la escuela de Ciencias y Sistemas.

Sin otro particular, me suscribo atentamente,

Ing. Juan Antonio Cabrera Aguirre

Universidad de San Carlos de Guatemala
Facultad de Ingeniería

Guatemala, x de Septiembre de 2006

Ingeniero
Sydney Alexander Samuels Milson
Decano
Facultad de Ingeniería

Estimado Sr. Decano:

Atentamente me dirijo a usted para informarle que después de conocer el dictamen del asesor del trabajo de graduación del estudiante Eddie Josué Gonzalez Pineda, titulado “Memoria Distribuida Compartida”, procedo a la autorización del mismo.

Sin otro particular me suscribo con las muestras de mi consideración y estima.

Ing. Jorge Armín Mazariegoz
DIRECTOR
ESCUELA DE INGENIERÍA EN CIENCIAS Y SISTEMAS

AGRADECIMIENTOS

Este trabajo representa no solo un esfuerzo personal, sino el aporte de muchas personas que hicieron posible este logro. En especial, agradezco a:

Dios y Virgen Maria	Porque siempre están conmigo y me han ayudado en todo momento. Su gran amor y misericordia han permitido que alcance esta meta.
mis padres, Rafael ✞ y Lidia	Por haberme dado unos padres ejemplares que me han guiado desde pequeño, dándome la fortaleza y ayuda incondicional para convertirme en profesional.
Ettie, mi amada esposa	Porque sin tu cariño, compresión, apoyo y amor, no lo hubiera alcanzado.
mis hijos Stephanie, Monika y Fernando	Porque ustedes son la razón para seguir adelante, perseguir mis sueños y darles un mejor futuro.
mi demás familia	A mis abuelitos Rafael ✞ y Marcelino ✞, a mis abuelitas Rebequita ✞ y Carmencita ✞, a mi hermano Cristian Ivan, a mis suegros Tito y Yoly, a mis cuñados Edwin y Kary, a mis tíos, tías y cuñados; que de una u otra forma me han apoyado para realizar esta meta.
mis compañeros y amigos	Ludwig Muñoz ✞, Rodolfo Loukota, Cresencio Chang, Lester García, Javier Ralda, Álvaro Díaz, Pedro López,

Carlos Quintana, Benjamín Corado, Byron Cifuentes, José Felipe, Alfredo Ochoa, Walter González, Aura Domínguez, Ingrid Figueroa y tantos otros mas con quienes compartí en todo momento, pero que sobretodo me dieron su amistad durante todos estos años de lucha y esfuerzo por lograr un sueño en común; David Gutiérrez, Patrick Hoag y Jorge Yong de *Hewlett-Packard* por su valiosa colaboración en la realización de este trabajo de graduación.

a mi asesor Por sus consejos y recomendaciones en la elaboración de este trabajo de graduación.

en especial Al público que me acompaña.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES

FIGURAS

TABLAS

GLOSARIO

ABCL/1 – Actor-Based Concurrent Language	Lenguaje de programación concurrente para los sistemas ABCL MIMD, creado en 1986 por Akinori Yonezawa, del departamento de las ciencias de la información en la universidad de Tokio. ABCL/1 utiliza mensajes asincrónicos entre objetos. Se apoya en instrucciones lisp básicas para su funcionamiento.
Aeolus	Lenguaje concurrente con transacciones atómicas.
ASIC - Application-Specific Integrated Circuit	Circuito integrado modificado para un uso especifico. Lo contrario a un microprocesador.
Backplane	Circuito impreso con ranuras para la inserción de tarjetas. Típicamente, es un concentrador que, usualmente, no tiene componentes activos. Eso es lo opuesto a un *motherboard*.

Benchmark	Programa o un conjunto de programas que pueden funcionar en diversas plataformas para dar una medida exacta de funcionamiento, la cual es comparable entre ellas para medir el rendimiento. Estas mediciones contemplan mediciones de gráficos, sistema de entrada/salida y cálculos a nivel de cpu, entre otros.
CAM - Content-addressable memory	Tipo especial de memoria usado en sistemas que necesitan búsquedas de alta velocidad. A diferencia de la memoria RAM en la cual se aplica un direccionamiento, dando como resultado el contenido de esa dirección, CAM esta diseñada para buscar el contenido en toda la memoria para ver si se encuentra almacenada. Sí el contenido es encontrado, retorna una lista de direcciones donde los valores están almacenados.
Carrier Sense Multiple Access (CSMA)	Protocolo probabilístico de control de acceso, en el cual un nodo verifica la ausencia de tráfico antes de transmitir a un medio físico compartido. El medio portador describe el hecho de que un transmisor escucha la onda portadora antes de enviar, es decir, intenta detectar

	la presencia de una señal codificada de otra nodo antes de iniciar su transmisión. Si detecta un portador, el nodo espera la finalización en curso antes de iniciar la propia. El acceso múltiple describe que nodos múltiples envían y reciben por el medio.
Chunker	Programa que parte el contenido de un archivo de entrada en partes más pequeñas, usualmente en tamaños predeterminados.
CIRCAL - CIRcuit CALculus	Proceso algebraico usado para modelar y verificar las correcciones de diseños concurrentes, tales como la lógica digital.
Cluster	Es un grupo de múltiples ordenadores unidos mediante una red de alta velocidad, de tal forma que el conjunto es visto como un único ordenador más potente.
Concurrent Smalltalk	Una variante concurrente del lenguaje de programación Smalltalk.
CORBA - Common Object Request Broker Architecture	Especificación que provee una interfaz estándar de mensajeria entre objetos distribuidos facilitando la invocación de métodos remotos bajo un paradigma

	orientado a objetos.
CSP - Communicating Sequential Processes	Notación de concurrencia basado en el paso de mensajes sincrónicos y comunicaciones selectivas, diseñado por Anthony Hoare en 1978.
DSM - Data Structure Manager.	Lenguaje de programación orientado a objetos, diseñado por *J. E. Rumbaugh y M. E. Lommis*, similar a C++. Esta implementación es utilizada en el diseño de software CAD/CAE.
Emerald	Lenguaje de programación orientado a objetos distribuido, desarrollado por la Universidad de Washington a mediados de 1980.
Firewall code	Código que es insertado en el sistema utilizado para asegurar que los usuarios no comentan algún daño en el resto de sistemas.
Flynn's taxonomy	Clasificación de las arquitecturas de computadora, propuesta por Michael J. Flynn en 1972. Las cuatro clasificaciones definidas por Flynn se basan sobre el número de instrucciones concurrentes, y de los flujos de datos disponibles en la

	arquitectura.
Latencia	Es el retraso entre el momento que se inicia una acción y el momento uno de sus efectos se comienzan a manifestar.
Multiple instruction, multiple data streams - MIMD	Procesadores autónomos múltiples que ejecutan, simultáneamente, diversas instrucciones en diversos flujos de datos. Los sistemas distribuidos se reconocen generalmente para ser arquitecturas de MIMD; explotando una sola memoria compartida o una memoria distribuida.
Multiple instruction, single data stream - MISD	Combinación inusual debido al hecho de los flujos de instrucciones múltiples requieran flujos múltiples para ser efectivos. Sin embargo, este tipo es usando cuando se requiere redundancia en paralelismo.
Parlog	Variante del leguaje de programación Prolog, diseñado por Clark y Gregory de la Universidad Imperial en 1983. Lenguaje de programación lógico en el cual cada definición y consulta pueden ser interpretados como un predicado lógico. A diferencia de Prolog, este incorpora modos de evaluación en paralelo.

Pipeline	Consiste en descomponer la ejecución de cada instrucción en varias etapas para poder empezar a procesar una instrucción diferente en cada una de ellas y trabajar con varias a la vez.
Prolog	Lenguaje de programación ideado a principios de los años 70 en la universidad de Aix-Marseille por los profesores *Alain Colmerauer y Phillipe Roussel* en Francia. Inicialmente, se referia de un lenguaje, totalmente, interpretado hasta que a mediados de los 70, *David H.D. Warren* desarrolló un compilador capaz de traducir Prolog en un conjunto de instrucciones de una máquina abstracta denominada *Warren Abstract Machine*, o abreviadamente, WAM. Desde entonces Prolog es un lenguaje semi-interpretado.
Single instruction, multiple data streams - SIMD	Computadora que explota los flujos de datos múltiples contra una sola corriente de instrucciones para realizar operaciones que pueden ser hechas, paralelamente, de forma natural.
Single instruction, single data stream - SISD	Computadora secuencial que no explota ningún grado de paralelismo en la ejecución de una instrucción o flujo de

	datos.
Single Program, multiple data streams - SPMD	Procesadores múltiples autónomos ejecutando simultáneamente el mismo juego de instrucciones pero con flujos de datos distintos.
Sloop	Programación paralela en un espacio de objetos virtuales.
Smalltalk	Sistema pionero de programación orientada a objetos desarrollado en 1972 por Software Concepts Group, bajo la supervisión de Alan Kay en los laboratorios Xerox PARC entre 1971 y 1983. En este lenguaje se pueden enviar mensajes a cualquier objeto definido en el sistema.
SNOOPS	Leguaje de programación orientado a objetos que soporta reconfiguración de forma dinámica.
Task Control Block	Bloque de control utilizado para llevar el control de los procesos que están asignados dentro de un espacio de direcciones.
THE	Sistema multiprogramación diseñado por Edsger Dijkstra y publicado en 1968.

	Utiliza el concepto de capas como estructura base.
Write-back	Modelo de memoria cache en la cual se escribe en la memoria principal y, posteriormente, es grabada al dispositivo destino.
Write-through	Modelo de memoria cache en la cual se escribe en la memoria principal y, al mismo tiempo, es grabada al dispositivo destino.

RESUMEN

El trabajo describe las tendencias de tecnología que, actualmente, en la industria van planteando nuevos retos para desarrollar sistemas que sean más productivos y, a la vez, económicamente competitivos.

Se definen conceptos que son necesarios para entender las razones por las cuales la arquitectura de un sistema informático posee la habilidad de comunicarse con sus componentes en forma ordenada para completar una tarea en el menor tiempo posible de forma segura y confiable.

Finalmente, se describe generalidades del funcionamiento actual de uno de los sistemas comerciales de alto rendimiento que existen en el mercado, el cual satisface las necesidades de empresas grandes que requieren un alto poder de cómputo, confiabilidad y disponibilidad, aplicado a empresas de manufactura y telecomunicaciones, entre otras.

OBJETIVOS

General

Conforme la necesidad de obtener, constantemente, más poder de cómputo, los sistemas con procesadores múltiples se están volviendo una necesidad. Sin embargo, la programación de tales sistemas requiere de habilidad y conocimiento significativo para su implementación y diseño. El éxito comercial de sistemas multiprocesadores y los sistemas distribuidos estará altamente bajo la dependencia de los paradigmas que la programación ofrece. Esto ha conducido a la elaboración documental que proporcione las nociones relacionadas con los sistemas en cuestión.

Específicos

1. Proporcionar los conceptos básicos de los sistemas DSM.
2. Conocer los diferentes tipos de algoritmos utilizados en la implementación de un sistema DSM
3. Mostrar los diferentes tipos de implementación de Software y Hardware fundamentales que originaron los sistemas DSM.
4. Presentar un caso de estudio de un sistema comercial

INTRODUCCIÓN

Los sistemas de memoria distribuida compartida representan el éxito de dos clases de computadoras paralelas: los multiprocesadores de memoria compartida y los sistemas de cómputo distribuidos. Estos proveen una abstracción compartida de la memoria en los sistemas cuya memoria físicamente esta distribuida y consecuentemente, combina las ventajas de ambas estrategias. A raíz de esto, el concepto de memoria distribuida compartida es reconocido como una de los avances más atractivos para diseño e implementación de sistemas multiprocesadores de alto rendimiento, a gran escala. Esta es importancia creciente de este tema, lo cual impone la necesidad de su comprensión cuidadosa.

Con este fin, este estudio documental cubre varios temas pertinentes y presenta un estudio general de ambos esfuerzos. Los problemas en esta área, lógicamente, se parecen a los problemas relacionados a la coherencia de la memoria *cache* en multiprocesadores de memoria compartida. La relación más cercana puede ser establecida entre temas algorítmicos de estrategias de mantenimiento de coherencia y sus implementaciones en memoria compartida.

Este estudio consta de cinco capítulos. El capítulo 1 da una introducción general para el campo DSM y provee una prospección generosa de sistemas y modus operandis diversos, que representa los conceptos básicos DSM,

mecanismos, tópicos del diseño y los sistemas. El capítulo 2 se concentra en los algoritmos básicos DSM, sus mejoras y su rendimiento. El capítulo 3 presenta implementaciones desarrolladas en hardware. El capítulo 4 presenta implementaciones en el nivel del software, como alternativa a las implementaciones de hardware. El capitulo 5 presenta un caso de estudio, presentando el modelo comercial de *Hewlett-Packard* (HP) el *AlphaServer* GS320.

1. ORÍGENES Y CONCEPTOS DE MEMORIA DISTRIBUIDA COMPARTIDA (DSM)

1.1. Modelos predecesores

La industria hoy en día no ha explotado las ventajas ofrecidas por la arquitectura DSM. Esto se debe particularmente a dos razones: la primera, mucha investigación del tema se ha publicado como una comprensiva colección de referencia; y la segunda, la existencia de tecnologías cliente / servidor que obviamente no satisfacen todas las necesidades de las aplicaciones distribuidas robustas. Las aplicaciones de arquitectura asimétrica predominan en el mercado. Estas arquitecturas no muestran las ventajas ofrecidas por la distribución: mejor rendimiento, fiabilidad, y disponibilidad a través de la replicación de los recursos y los servicios.

Es solo cuestión de tiempo y experiencia antes que los sistemas DSM reemplacen el estilo cliente / servidor orientado al paso de mensajes, visto como un modelo relacional, aunque se considere como un modelo ineficaz. Otra perspectiva seria una coexistencia entre ambos paradigmas, donde cada uno aumente el potencial del otro. El rendimiento es claramente un prerrequisito para un sistema aceptable DSM. El diseño de software concurrente esta basado en el estilo enviar / recibir. Toda la información requerida por dos procesos concurrentes debe ser explicita cada vez que la necesitan o es cambiada.

Los sistemas distribuidos de objetos ofrecen una mejor abstracción de una llamada remota relacionada en la invocación de un método desde otro objeto remoto.

Esto podría ser la razón principal del por qué los artículos no académicos comunican las experiencias de la aplicación con DSM. Para convencer a la industria, se necesita una argumentación para los principios siguientes: (i) El modelo de DSM es conceptualmente más sencillo que el modelo del paso de mensajes y así reducir tiempo de desarrollo. Esta argumentación incluye la longitud de código y elegancia de la solución. (ii) Las soluciones con DSM pueden ser técnicamente superiores con respecto al comportamiento de la memoria *cache*, la escalabilidad, la flexibilidad, y la tolerancia a fallas. Claramente, esto depende de los requerimientos de la aplicación pero en situaciones típicas para DSM, esto no debería ser un punto en contra para su realización.

1.1.1. Arquitecturas simétricas:

La replicación no puede ser soportada por sistemas distribuidos de objetos, porque la implementación de un objeto es una caja negra, desde el punto de vista del administrador de objetos. El administrador no puede saber que estados internos han sido cambiados por la llamada del procedimiento remoto, y así se deben de replicar todos los datos y las acciones del objeto, incluyendo todos los datos entrantes y salientes del sitio remoto. La coordinación de los accesos concurrentes a las replicas de los objetos distribuidos tiene que ser implementadas con mecanismos de sincronización adicionales, como son los semáforos.

Por lo tanto, para aprovechar el paso de mensajes, un solo servidor mantiene la información relevante, y todos los servidores clientes, requieren esa información. Esto naturalmente relaciona la invocación de métodos remotos. La ventaja principal es la administración de un solo servidor, el único inconveniente es un cuello de botella con respecto al desempeño, la disponibilidad, y la confiabilidad sí la aplicación crece. Las arquitecturas multi-capas pueden presentar este problema. La lógica de la aplicación es movida del cliente a otro nivel, el cual se llamará *middleware*, el cual asume también las tareas que también habían sido echas por un solo servidor, pero los datos permanecen en el servidor centralizado. Dichas arquitecturas se consideran actualmente las mejores soluciones con tecnologías cliente / servidor. Sin embargo, tales arquitecturas introducen muchas capas, que pueden aumentar el número de pasos de comunicación entre cliente y servidor, y deben ser coordinados por el programador. El cuello de botella del servidor con respecto a la tolerancia a fallas, no puede ser evitado; requeriría programación extra para replicar el servidor.

La replicación es una propiedad básica del modelo DSM, en el cual la arquitectura tiende a ser simétrica. No existe ninguna diferencia entre los papeles de cliente y servidor. Todos los procesos participantes tienen los mismos derechos de comunicarse, utilizando los segmentos de datos compartidos.

El modelo DSM es el responsable del rendimiento y fiabilidad de la aplicación. Esto significa que la necesidad del programador de la aplicación se despreocupe por tales asuntos, y esos procesos - que claramente puede jugar también el papel de servidor - pueden ser sencillos.

La abstracción ofrecida por el modelo DSM es la perspectiva de que la memoria de la computadora local se extiende para incluir la memoria de otras computadoras en otros sitios. El hardware fundamental se oculta completamente y la lógica de la aplicación no es un impedimento para agregar una nueva dimensión de envió y recepción de mensajes. El diseñador comienza con una definición llamada "estructura de datos coordinados" en el cual definirá como los procesos concurrentes la utilizaran.

1.1.2. Productor – consumidor

El modelo productor / consumidor muestra la cooperación de los procesos concurrentes mientras compiten por recursos comunes. Algunos procesos producen información que otros consumen. El flujo de datos es unidireccional: de uno o más productores a uno o más consumidores. Toda la información producida es relevante y no debe ser perdida por los consumidores. Ellos representan una cronología de eventos / valores.

En una versión sencilla, cada consumidor observa toda la información producida, mientras que en una versión complicada, cada uno observa información exclusiva. En la variante de recursos variables, el espacio del buffer es limitado a N entradas de información.

En esta variante se usa una estructura llamada *stream*, la cual es una lista de celdas, donde la primera celda representa la primera lista de elementos de información, la segunda, representa la segunda lista de elementos de información, y así sucesivamente hasta llegar a N. Asumiendo

que cada lista de celdas esta representada de forma única, la segunda parte de esa lista de celdas contiene los identificadores. La raíz del *stream* es también un identificador, el cual es trasladado a todos los procesos que representan ese segmento en la primera lista de celdas, convirtiéndose en un recurso compartido por ellos. Los procesos que son creados posteriormente, también hacen referencia a ellos.

Los procesos productores compiten para tener acceso a la cola de celdas del *stream* actual. El acceso concurrente es realizado a través de DSM. Un proceso puede escribir en el segmento referenciando a la cola de celdas lo cual producirá un conflicto en los procesos restantes y tendrán la necesidad de actualizar el estado actual de la cola utilizada, rescribiendo los segmentos a la siguiente cola de celdas. Utilizando constantemente la comunicación entre los objetos, los conflictos automáticamente ocurren sí se realiza una escritura de un objeto previamente definido. Con los objetos de comunicación variable, una prueba tiene que ser realizada para determinar sí el objeto esta en estado indefinido antes de sobrescribirlo. Los procesos leen sincrónicamente el segmento de la raíz, y todos los procesos subsecuentes hacen referencia a los segmentos de la cola de celdas. La escritura de nueva información por uno de los procesos productores concurrentemente despertará a todos los procesos restantes en una manera controlada.

Una versión completa puede estar basada en la misma estructura de coordinación, el *stream*. Solo la primera lista de celdas debe ser modificada en su contenido, además de producir información, por medio de una bandera se indica cuando el elemento ha sido consumido o no. Adicionalmente en una lectura sincrónica, el consumidor debe completar la escritura de la

bandera asociada al elemento de entrada. Análogamente, la producción de información sé acceso concurrentemente a través de la memoria compartida. Cuando un consumidor obtiene los elementos de entrada, todos los procesos restantes observaran un conflicto y tendrán que continuar accesando el *stream* asignado. Esta descripción tiene un inconveniente: el *stream* crece infinitamente. Para que eso no suceda, se tiene que aplicar algún algoritmo de recolección de basura: cada proceso debe de reiniciarse después de haber producido / consumido una cantidad finita de elementos de información en la raíz de la cola asignada.

De esta manera, el proceso abandona las referencias a los segmentos de datos. Otra posibilidad, deberá restringir la memoria a N localidades, lo cual requerirá una modificación a la estructura de coordinación, donde las N listas de celdas formaran un buffer circular el cual será liberado con comunicación variable entre objetos y un consumo total por los procesos consumidores. El acceso concurrente es manejado por medio de DSM, el cual requiere una sincronización adicional sí el buffer esta lleno. Cuando esto ocurra, todos los procesos productores entraran en conflicto. La estructura de listas de celdas de un consumidor puede ser reutilizada. Sí un proceso productor encuentra una celda con el elemento de entrada consumido, puede sobrescribirla por una nueva entrada actualizando la bandera respectiva.

Ambos modelos descritos soportan escalabilidad ilimitada. La adición dinámica de nuevos procesos productores / consumidores pueden ser distribuidos en sitios remotos sin necesidad de reprogramarlos. La tolerancia a fallas puede ser agregada, si el modelo DSM provee objetos persistentes y procesos de recuperación.

Este modelo fue el primero en ser usado como una solución independiente, para permitir un número ilimitado de comunicación y sincronización de procesos. Como consecuencia de ello, en el modelo observa:

(i) El espacio del buffer podía ser infinito o limitado a N posiciones.

(ii) Los elementos podían ser asignados a un solo consumidor o a un grupo de consumidores. Los procesos restantes, no pueden ver esa información.

(iii) La relación entre los productores / consumidores podía ser: 1:1, k:1, i:m y k:m, el cual depende de los requerimientos de la aplicación. Estos procesos pueden ser adicionados de forma local o remota.

(iv) El sistema es reactivo y perpetuo, pero puede ser aplicado una cantidad limitada de procesos productores / consumidores. Un proceso productor puede producir un meta elemento, el cual puede indicar el final de un *stream*.

(v) Acceso rápido al valor más reciente en el *stream* producido, el cual, en la estructura de coordinación, contiene el acceso a la lista actual de celdas. Para eliminar la lectura de todas las celdas entre la raíz y la cola de la lista de celdas, un segundo segmento a la raíz puede ser administrado, el cual es compartido por todos los procesos en lugar del segmento de la raíz original, la cual contiene la referencia a la ultima lista de celdas con la mas reciente información.

(vi) Cada productor contiene un *stream* separado. Todos los segmentos raíz deben ser compartidos como usualmente se lleva a cabo, ya que algún otro proceso, pude requerirlos.

1.1.3. Requerimiento – respuesta:

El modelo de requerimiento / respuesta sincroniza los requerimientos concurrentes de uno o más clientes en el servidor. El flujo de datos es bi-direccional: la respuesta se comunica del cliente al servidor, da inicio a una acción, y finalmente, el resultado es comunicado de vuelta. Los clientes pueden ser locales en el mismo servidor o remotos en otros sitios.

Este modelo provee un segmento de datos a cada proceso cliente en el servidor. Cada requerimiento conlleva una invocación al servidor, pasando su información al segmento de datos asignado en él. El requerimiento es almacenado, posteriormente procesado, y la respuesta es colocada en el mismo lugar donde fue tomada la información. Una variante de este modelo, es la siguiente: El segmento de datos asignado al cliente en el servidor esta compuesta por dos segmentos.

Por cada proceso cliente, existe un segmento con significado “requerimiento” y el otro segmento con el significado “resultado”. El servidor y todos los clientes son procesos concurrentes, donde el cliente solo tiene acceso a sus dos segmentos de datos y el servidor tiene acceso a todos los segmentos pares de los clientes. Esta estructura separa los accesos de escritura, lo cual es una ventaja del DSM. Como los segmentos son escritos varias veces, estos son liberados con una comunicación variable entre objetos.

Cuando un requerimiento por un proceso cliente es iniciado cuando escribe en su segmento de datos, posteriormente, emite una lectura sincrónica en su segmento resultante. El proceso en el servidor ejecuta una espera sincronizada por todos los segmentos de datos que son compartidos, esperando que alguna construcción lingüística libere la espera.

Mientras un segmento recibe el siguiente valor, el proceso servidor es despertado y calcula una respuesta, la cual es escrita en el segmento de datos del proceso correspondiente. Este evento hace que se levante el proceso cliente y tome su resultado.

El segundo modelo se convierte en una solución más elegante y eficiente. Este modelo es escalable a un número ilimitado de procesos cliente sin realizarse ninguna modificación. Solo se tiene que inicializar el servidor para que tenga acceso a todos los segmentos pares definidos. Esta iniciación puede ser reemplazada por una adición dinámica de nuevos segmentos pares por cada cliente que necesite del servidor.

Sí los procesos del servidor pueden acceder uno o más segmentos con el significado “registro”, cada nuevo cliente puede pasar por referencia sus segmentos de datos y convertirse en un proceso registrado. La distribución de los servidores y los clientes es totalmente transparente. Como consecuencia de ello, en el modelo observa:

(i) Todos los segmentos resultantes podían ser nombrados artificialmente, y muchos clientes podían usar el mismo segmento de resultados. Tomando como referencia al segmento resultante en el servidor, el cliente puede indicar en cual de los segmentos resultantes se entregará la respuesta. De esa forma, si el proceso cliente usa repetidamente el mismo segmento de resultados y la información contenida en ese segmento esta bien estructurada, las siguientes llamadas del cliente pueden beneficiarse con la información obtenida anteriormente. Análogamente, los diferentes clientes localizados en el mismo sitio pueden obtener la ventaja de los resultados obtenidos por el requerimiento de otros procesos cliente.

(ii) Si el servidor cae en una sobrecarga por el incremento del numero de clientes, este pude ser fácilmente distribuido. Los servidores dividen a los clientes en dos partes y cada servidor requiere los servicios de una parte. De acuerdo al número de servidores, este pude ser incrementado a tres o más.

1.2. Conceptos

Los sistemas con múltiples procesadores, usualmente son clasificados en dos grandes grupos, de acuerdo a la forma en que organizan la memoria: (i) Sistemas con memoria compartida (SM) y (ii) Sistemas con memoria compartida distribuida (DSM).

Los sistemas con memoria compartida, la memoria física total es accesible por todos los procesadores. Una ventaja importante de estos sistemas, es el modelo general y conveniente que programación, esto da la habilidad a la información de ser manipulada a través de un mecanismo uniforme de lecturas y escrituras compartidas en la memoria común. El costo del desarrollo del software paralelo, es reducido debido a la comodidad de programación y portabilidad. Sin embargo, los procesadores sufren un incremento de contención y de latencias más largas accesando la memoria compartida, dando como resultado, picos de rendimiento y escalabilidad limitada comparada con sistemas distribuidos.

Los sistemas con memoria distribuida consisten en un conjunto de nodos independientes de procesamiento con módulos de memoria local conectados entre sí a través de algún dispositivo de red de interconexión. Los sistemas con memoria distribuida son escalables dando así altos niveles de procesamiento.

Sin embargo, la comunicación entre los procesos que reside en diferentes nodos, es manejada a través del modelo de paso de mensajes, el cual

requiere llamadas explicitas de las primitivas de enviar / recibir; lo cual genera en los programadores una forma más difícil de administrarlo, ya que tienen que manejar la distribución de la información y la forma de comunicarla entre los ellos.

Un sistema DSM lógicamente implementa el modelo de memoria en una distribución física. El mecanismo específico para llevarlo a cabo es mediante una abstracción de la memoria compartida, la cual puede ser implementada por medio de hardware y/o software en una variedad de formas.

Los sistemas DSM ocultan los mecanismos de comunicación remota del escritor de la aplicación, por lo que la programación y la portabilidad de la memoria compartida son preservadas. Las aplicaciones existentes para sistemas de memoria compartida, pueden ser relativamente de fácil modificación y ejecutas eficientemente en sistemas DSM, preservando la inversión del software y maximizando el desempeño resultante. Además, la escalabilidad y eficacia en función de los costos de sistemas distribuidos de memoria, se heredan también. Consecuentemente, la importancia del concepto de memoria distribuida compartida, viene del hecho que lo parece ser una elección viable para la construcción de sistemas a gran escala de multiprocesamiento. El objetivo principal en los sistemas DSM, es el desarrollo general que minimice el tiempo medio de acceso a los datos compartidos, mientras se mantiene la consistencia de los mismos.

Algunas soluciones, implementan una capa de software encima del sistema de paso de mensajes; mientras otras, extienden las estrategias del

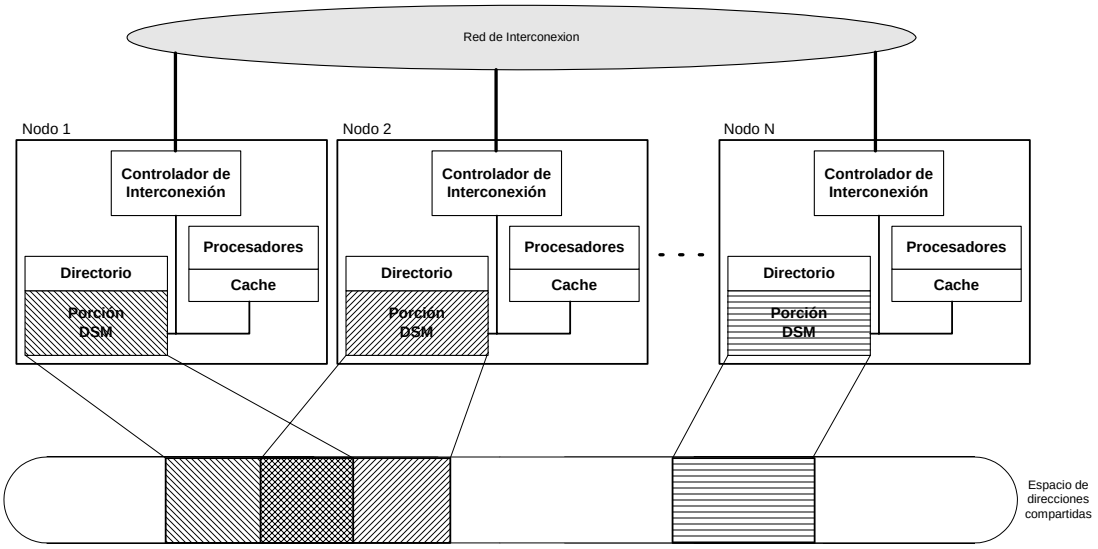
multiprocesamiento compartido de la memoria por medio de memoria *cache* privada.

1.3. Estructura general de sistemas DSM

Un sistema DSM puede ser generalmente visto como un conjunto de nodos o *clusters*, interconectados por una red (figura 1). Un cluster puede ser un sistema uní procesador o multiprocesador, usualmente organizado en un bus compartido.

La memoria *cache* adyacente al procesador es inevitable para la reducción de la latencia de la memoria. Cada cluster en el sistema contiene módulos de memoria física local, la cual es parcial o totalmente asignada al espacio global de direcciones. A pesar de la topología de la red (por ejemplo, en bus, anillo, estrella) es necesario un controlador de interconexión dentro de cada cluster, para poder conectarlo dentro del sistema. La información acerca de estados y ubicaciones actuales de bloques particulares de datos, se mantiene generalmente en la forma de una tabla del sistema o directorio.

Figura 1. Estructura y organización de un sistema DSM



El almacenamiento y la organización del directorio son los más importantes para la toma de decisiones en el diseño, dando como resultado un gran impacto en la escalabilidad del sistema. La organización del directorio puede variar desde un mapa completo del almacenamiento a diferentes organizaciones dinámicas, tales como listas simples o doblemente encadenadas y árboles. No importa el tipo de organización utilizada, el *cluster* debe proveer el almacenamiento completo para el directorio o almacenar una parte de él. De esta forma, el sistema de directorio puede ser distribuido alrededor del sistema como una estructura flotante o jerárquica. En topologías jerárquicas, si existen niveles intermedios en el *cluster*, usualmente solo contienen directorios y la correspondiente controladora de interconexión. La organización del directorio y el almacenamiento semántico de la información depende del método para mantener la consistencia de la misma.

1.4. Clasificación de los sistemas DSM:

Los diseñadores de los sistemas DSM toman como base los principios de la memoria virtual y el principio de la conservación consistente del *cache* en sistemas multiprocesadores de memoria compartida.

Las redes de estaciones de trabajo han llegado a ser cada vez más populares y poderosas actualmente, así que ellas representan la plataforma más adecuada para muchos programadores que entran al mundo de la computación en paralelo. Sin embargo, la velocidad de la comunicación es todavía el obstáculo principal que previene que estos sistemas alcancen el nivel del desempeño esperado.

Los diseñadores de sistemas multiprocesadores de memoria compartida están esforzándose para lograr más escalabilidad a través de un diseño sofisticado de distribución física de la memoria compartida, tales como agrupamientos (*clustering*) y diseños jerárquicos. Para estas razones, la diferencia entre multiprocesadores y multicomputadores es mínima, y ambas clases de sistemas se acercan aparentemente uno al otro en ideas básicas y desempeño, y cada vez mas los sistemas se pueden encontrarse en una familia grande de sistemas modernos DSM. A pesar de muchas equivocaciones y confusión de terminología en esta área, se asume una definición más general, la cual asume que todos los sistemas proveen una abstracción de la memoria compartida, la cual la hace pertenecer a una categoría DSM en especial. Esta abstracción es lograda por acciones específicas para el acceso de la información en el espacio global de direcciones, el cual es compartido entre todos los nodos. Tres temas

importantes están relacionados en el desempeño de estas acciones, las cuales están relacionadas al sitio donde sé acceso la información, mientras mantienen la consistencia (con respecto a otros procesadores); por lo que cada una de estas acciones clasifica a los sistemas DSM en una de las siguientes categorías:

(i) ¿Cómo es ejecutado el acceso? – Algoritmos DSM
(ii) ¿Dónde es implementado el acceso? – Nivel de implementación de mecanismo DSM
(iii) ¿Cuál es la semántica precisa de la palabra consistente? – Modelo consistente de memoria

1.4.1. De acuerdo al algoritmo DSM:

Los algoritmos DSM pueden ser clasificados de acuerdo permisibilidad de la existencia de múltiples copias de la misma información considerando los derechos de acceso a esas copias. Las dos estrategias utilizadas para la distribución de la información compartida mas frecuentemente utilizadas son la replicación y la migración.

La replicación permite múltiples copias de la información simultáneamente en memorias locales diferentes, para aumentar el paralelismo lógico del acceso a la información. La migración implica una sola copia de la información, la cual debe ser movida al sitio que la accesa en forma exclusiva. Los algoritmos DSM se clasifican de la siguiente forma:

(i) LSES – Lector simple / Escritor Simple (*SRSW – Single reader / Single Writer*).
   a. Sin migración
   b. Con migración
(ii) MLES – Múltiples lectores / Escritor simple (*MRSW – Multiple reader / Single writer*)
(iii) MLME – Múltiples lectores / Múltiples Escritores

La replicación es prohibida en la clasificación LSES, mientras si es permitida en los modelos MLES y MLME. La complejidad del mantenimiento de la consistencia depende fuertemente en las condiciones introducidas.

1.4.2. De acuerdo al nivel de implementación del mecanismo

El nivel en el cual el mecanismo DSM será implementado, es uno de las decisiones más importantes para la construcción y diseño de los sistemas DSM, porque implica programación, rendimiento y costo. Las posibles implementaciones son:

(i) Software
   a. A nivel de librería en tiempo de corrida
   b. A nivel del sistema operativo
      i. Dentro del *kernel*
      ii. Fuera del *kernel*
(ii) Hardware
(iii) Híbrido, una combinación de hardware / software.

La implementación de software se basa en la idea de ocultar el mecanismo de pasos de mensajes. Algunas de estas implementaciones se basan en librerías en tiempo de corrida, las cuales son enlazadas con la aplicación que utiliza información compartida.

Otras implementaciones se llevan a cabo a nivel del lenguaje de programación, dado que el compilador puede detectar el acceso compartido e insertar las rutinas apropiadas de sincronización y consistencia dentro del código ejecutable. Otras implementaciones incorporan el mecanismo dentro de un sistema operativo distribuido, dentro o fuera del *kernel*. El sistema operativo y las librerías de tiempo de corrida con frecuencia integran el mecanismo DSM con la administración de la memoria virtual.

Algunos sistemas DSM delegan esta responsabilidad en el hardware, siendo estas: localización, copia y consistencia de la información compartida. Estas soluciones extienden las técnicas tradicionales de *caching* en el multiprocesamiento compartido de la memoria convirtiendo a los sistemas DSM en redes de interconexión escalable. Esta responsabilidad, genera una clasificación de tres grupos, de acuerdo a la arquitectura del sistema de memoria utilizado: *CC-NUMA (cache coherent nonuniform memory access), COMA (cache only memory architecture)* y *RMS (reflective memory system).* En la arquitectura *CC-NUMA*, el espacio de direcciones es estático, distribuido a través de todos los módulos de memoria local de los nodos o *clusters*, los cuales pueden ser accesados tanto por procesadores locales como procesadores remotos, generando latencias de acceso considerables. La arquitectura *COMA* provee mecanismos dinámicos de particionamiento de la información en forma distribuida, organizando un segundo nivel de cache.

La arquitectura *RMS* utiliza un mecanismo de actualización implementado en hardware para propagar inmediatamente cada cambio a todos los sitios compartidos, utilizando multidifución o transmisión de mensajes. Este tipo de sistema de memoria es llamado memoria espejo (*mirror memory*).

A causa del posible rendimiento / complejidad, la integración de métodos de software y hardware parece ser los enfoques más prometedores en el futuro de DSM. Algunos enfoques del software incluyen aceleradores de hardware para las operaciones mas frecuentes mejorando así el rendimiento, mientras algunas soluciones de hardware manejan los eventos poco frecuentes en el software para aminorar la complejidad.

Para obtener mejor rendimiento en los sistemas DSM, las implementaciones más recientes han usado múltiples protocolos dentro del mismo sistema, e incluso integran los mensajes en el mecanismo DSM. Para manejar esta complejidad, básicamente la solución de software, se puede añadir un protocolo programable especial al sistema.

Mientras las soluciones de hardware brindan total transparencia en los mecanismos DSM a los programadores y capas de software, logrando menores latencias de acceso, las soluciones de software pueden tomar mejor ventaja de las características de la aplicación a través del uso de varias sugerencias proveídas por el programador.

1.4.3. De acuerdo al modelo consistente de memoria

El modelo consistente de memoria define el orden valido de las referencias a la memoria emitidas por algún procesador, así como los otros procesadores las observan.

Los diferentes tipos de aplicaciones paralelas requieren inherentemente varios modelos de consistencia. El rendimiento de los sistemas que ejecutan estas aplicaciones se ven muy influenciadas por las restricciones del modelo. Las formas más fuertes del modelo de la consistencia, aumentan típicamente la latencia del acceso de la memoria y los requisitos de ancho de banda, mientras la programación se simplifica.

La simplificación de restricciones genera modelos más sencillos permitiendo reordenamiento, *pipelining,* y *overlapping* de memoria, dado como resultado un mejor rendimiento pero elevando el costo de la sincronización a los datos compartidos.

Los modelos robustos de memoria consistente sincronizan los accesos, como operaciones ordinarias de lectura y escritura, como secuenciales y consistentes a nivel del procesador. Los modelos más sencillos se distinguen entre accesos sincronizados y ordinarios, pudiendo ser entre ellos: *weak, release, lazy release* y *entry consistensy*.

El modelo de acceso consistente secuencial provee a todos los procesadores del sistema, observar el mismo intervalo de lecturas y escrituras emitidas por un procesador individual. Una implementación sencilla de este modelo es un sistema de memoria compartida de un solo puerto, generando así un servicio de acceso serializado de una cola del tipo primero en entrar, primero en salir.

En los sistemas DSM, la implementación es realizada por medio de la serialización de todos los requerimientos en el servidor central. En ambos casos, no es permitido pasar por alto los requerimientos de lectura y escritura.

El modelo de consistencia de procesadores asume que el orden en cuál las operaciones en la memoria pueden ser vistas por diferentes procesadores, no necesitan ser idénticas, pero la secuencia emitida de cada procesador, debe ser observada por todos los otros procesadores en el mismo orden de emisión.

A diferencia del modelo de acceso consistente secuencial, permite a las lecturas pasar por alto a las colas de escritura. Los modelos de consistencia *weak* se distinguen por ser un acceso ordinario sincronizado a la memoria. En otras palabras, este modelo requiere que la memoria sea consistente solo en el momento de la sincronización. En este modelo, los requisitos para la consistencia secuencial aplican sólo en los accesos de sincronización en sí mismos. Adicionalmente, la sincronización de los accesos deben de esperar a que todos los accesos previos sean ejecutados, mientras las lecturas y

escrituras ordinarias deben de esperar solo por la finalización de su evento previo de sincronización.

Los modelos de consistencia *release* dividen la sincronización del acceso en adquisiciones y liberaciones, protegiendo así el acceso compartido ejecutándolo entre dos eventos pares, adquisición-liberación.

En este modelo, las lecturas o escrituras ordinarias son ejecutadas solo después que todas las adquisiciones previas en el mismo procesador sean finalizadas. Consecuentemente, las liberaciones pueden ser ejecutadas hasta que las lecturas o escrituras ordinarias son finalizadas previamente en el mismo procesador. Finalmente, el acceso sincronizado adquisición-liberación deben cumplir los requerimientos que obligaron la consistencia del procesador en los accesos ordinarios de lectura y escritura.

Una versión extendida del modelo *release*, el modelo *lazy release*, en vez de propagar las modificaciones al espacio de direcciones compartidas en cada liberación, estas se aplazan hasta que una próxima adquisición relevante. Consecuentemente, no todas las modificaciones necesitan ser propagadas en una adquisición, por lo que solo asocia aquellas a la cadena de secciones críticas precedentes. Como resultado de esto, la cantidad de información cambiada es minimizada, mientras el número de mensajes es reducido también por combinación de modificaciones con requerimientos con bloqueo en un mensaje.

Los modelos de consistencia *entry* es una mejora nueva del modelo *release*, el cual requiere que cada variable u objeto compartido debe ser protegido y asociado a una variable de sincronización usando una anotación en el lenguaje de programación. Consecuentemente, una modificación a la variable u objeto es pospuesto hasta la siguiente adquisición asociada a la sincronización de la variable que protege. Puesto que solo el cambio asociado a la variable necesita ser propagado en el momento de la adquisición, el tráfico es disminuido significativamente.

La latencia se reduce también, ya que un acceso compartido no tiene que esperar la terminación de otra adquisición no relacionada. La mejora del desempeño se logra a costa de un compromiso más alto del programador, en especificar la sincronización de la información para cada variable protegida.

1.5. Selecciones importantes en el diseño de sistemas DSM

Adicionalmente a los algoritmos, nivel de implementación del mecanismo, y del modelo consistente de memoria de los sistemas DSM, hay un conjunto de características que pueden afectar fuertemente el rendimiento del sistema entre ellos:

(i) La configuración del *cluster*: uniprocesador o multiprocesador, con memoria *cache* privada o compartida, con *cache* de un nivel o de varios niveles, organización local de la memoria, las interfaces de red, etc.

(ii) Interconexión de la red: bus jerárquico, anillo, malla, hipercubo, etc.

(iii) Estructura de los datos compartidos: objetos estructurados o no estructurados, tipos de lenguajes, etc.

(iv) Granularidad de la unidad de consistencia: palabra, bloque, página, estructuras de datos complejos, etc.

(v) Responsabilidad de la administración del DSM: centralizado, distribuido fijo, distribuido dinámico.

(vi) Política de consistencia: escritura inválida, actualización de escritura, etc.

La configuración del *cluster* varía grandemente entre los diferentes fabricantes de sistemas DSM. Incluye uno o varios procesadores. Puesto que cada procesador posee su propia memoria *cache* o *cache* jerárquico, su consistencia a nivel del cluster debe ser integrada con el mecanismo DSM a un nivel global.

Partes del modulo de memoria local pueden ser configuradas como privadas o compartidas, asignadas al espacio virtual de direcciones compartidas. Además de acoplar el *cluster* al sistema, la interfaz controladora de red algunas veces integra responsabilidades importantes en la administración del DSM.

Casi todos los tipos de redes de interconexión son encontrados en sistemas multiprocesadores y distribuidos, los cuales también pueden ser utilizados en sistemas DSM. La mayor parte del software orientado a

sistemas DSM son independientes de la red, aunque muchos de ellos tienen lugar para ser construidos encima de las redes *ethernet*, fácilmente disponibles en la mayoría de ambientes. Por otro lado, las topologías tales como multinivel en bus, anillo o malla, han sido usadas como plataformas para algunos sistemas de hardware orientados a sistemas DSM. La topología de interconexión de la red puede ofrecer o restringir el potencial de intercambio paralelo de datos relacionado con la administración del DSM. Por estas mismas razones, la topología afecta la escalabilidad. Esto determina el costo de transmisión y multidifución de transacciones, algo muy importante para la implementación del algoritmo DSM.

La estructura de datos compartida representa el diseño global del espacio de direcciones compartidas, así como la organización de los elementos en él. Las soluciones de hardware siempre se encargan de objetos no estructurados, mientras algunas implementaciones de software tienden a usar los elementos que representan en entidades lógicas, para aprovechar la localidad expresada por la aplicación naturalmente.

La granularidad de la unidad de consistencia determina el tamaño del bloque de datos administrado por el protocolo de consistencia. El impacto de este parámetro en el rendimiento general del sistema, esta relacionado estrechamente a la localidad del acceso de los datos en la aplicación. En general, los sistemas basados en hardware usan unidades pequeñas, típicamente llamados bloques, mientras algunas soluciones de software, basan sus mecanismos en la memoria virtual, organizando la data en grandes estructuras de bloques, llamadas páginas. El uso de grandes bloques da como resultado el ahorro de espacio de la estructura de

directorio, pero esto incrementa la probabilidad de que múltiples procesadores requieran el acceso al mismo bloque simultáneamente, aunque ellos acceden partes no relacionadas de ese bloque. Este fenómeno se denomina *false sharing*. Esto puede ocasionar un *thrashing* – un comportamiento caracterizado por el intercambio extensivo de datos entre dos sitios compitiendo por el mismo bloque.

La responsabilidad de la administración del DSM determina que sitio debe manejar las acciones relacionadas al mantenimiento de la consistencia en el sistema; pudiendo ser centralizado o distribuido.

La administración centralizada es la más fácil de implementar, pero esta administración representa un cuello de botella. La responsabilidad a nivel distribuido puede ser definida como estático o dinámico, eliminando los cuellos de botella y proporcionando escalabilidad. La distribución de la responsabilidad para la administración del sistema DSM esta estrechamente relacionada con la información del directorio.

La política de consistencia determina sí la existencia de una copia de un elemento de datos será escrito, actualizándolo en un sitio e invalidándolo en otros sitios. La elección de política de consistencia esta relacionada con la granularidad de los datos compartidos. Por cada granulo fino de datos, el costo del mensaje de actualización es aproximadamente el mismo costo del mensaje de invalidación. Consecuentemente, la política de actualización es frecuentemente basada en mantenimientos de consistencia del tipo *word*. Por otro lado, la invalidación se usa ampliamente en sistemas de granulo

grueso. La eficiencia de una invalidación aprovecha el aumento de las secuencias de acceso de lecturas y escrituras al mismo elemento por varios procesadores que no son altamente intercalados. El mejor rendimiento se puede esperar sí la política de consistencia se adapta dinámicamente al modelo observado.

1.6. Evolución del modelo de objetos distribuidos

Las primeras aplicaciones comerciales para computadora se compusieron de una sola aplicación que corría en un *mainframe* al cual se conectaban un conjunto de terminales (*green screen*). Estas aplicaciones corrían enteramente en un sistema *mainframe* o *clusters* de *mainframes*. Estos tenían varias ventajas, específicamente, el mantenimiento se ejecutaba en una sola maquina y de esa manera, el costo era amortizado por la organización. Además de esto, el costo del desarrollo para estas aplicaciones es bajo, en comparación de otros métodos. El problema de las aplicaciones en estos sistemas es que no tienen una buena escalabilidad.

La creación de nuevos usuarios o la demanda de la aplicación del sistema con frecuencia reducían el rendimiento de todos los usuarios. Realizando un enriquecimiento al sistema, se proveía un alivio temporal, terminando con adquisición de un nuevo *mainframe*, el cual por lo regular era el mejor de la línea. Los sistemas de tiempo compartido caen en la categoría de los *mainframe*, los cuales presentan problemas similares a pesar de proporcionar mejor respuesta interactiva.

El modelo cliente / servidor surgió como una solución al dilema de los *mainframes*. En este modelo, se cuenta con un poderoso servidor el cual provee archivos, base de datos, y algunos otros servicios compartidos a las maquinas clientes de la red. A pesar de la disponibilidad de estos servicios compartidos, los cálculos se ejecutaban en las aplicaciones de los usuarios finales.

Como consecuencia, la adición de nuevos usuarios aumentaba el poder de cómputo, pero eventualmente un límite era alcanzado, y era cuando el servidor llegaba a ser el cuello de botella. Este cuello de botella limitaba la escalabilidad del sistema puesto que no se podían soportar más usuarios y el servidor llegaba a tener una sobrecarga. Para reducir los costos del servidor, se ideo trasladar la lógica de la aplicación al cliente. Este traslado presenta varios inconvenientes significativos. Un inconveniente importante del enfoque cliente / servidor es el factor económico. Sí el software es mejorado, este demanda más procesamiento el cual posiblemente no pueda proveer el cliente, generando así la necesidad de mejorar su hardware. Además de esto, la administración de las maquina clientes, era mucho más costoso que administrar una sola máquina central. Desde que la administración de sistemas es un gasto significativo en las instalaciones, las soluciones que reducen la administración son cada vez más importantes.

Para reducir estos costos emergió el modelo cliente liviano / servidor robusto (*Thin Client / Fat Server*). En este enfoque, el cliente se limito a tener a tener una única interfaz grafica y limitado a algunas aplicaciones que no demanden tanto recurso. La mayor parte de cálculos intensivos, fueron restringidos a ser ejecutados en el servidor. Dado que la aplicación

demanda muy pocos recursos del cliente y la mayor parte de operaciones se ejecutan en el servidor, raramente se necesita realizar alguna mejora de hardware en el cliente. Conociendo este comportamiento y sabiendo que los procesos corren en el servidor, es posible comprar un hardware especifico para esa aplicación. Específicamente, el hardware de bajo costo puede explotar esta configuración para reducir el costo de la administración del sistema.

Infortunadamente, este enfoque hace que la aplicación regrese al servidor, incrementando de nuevo potenciales cuellos de botella en él, limitando la escalabilidad. Para mejorar la escalabilidad de este modelo, los desarrolladores introducen los servidores multi-capas. Un sistema típico multi-capas, por ejemplo, el cliente se comunica con el servidor de aplicaciones quien es responsable de ejecutar un segmento de código de la aplicación. Para el acceso a la base de datos o el servidor de archivos, el servidor de aplicaciones tiene la capacidad de comunicarse donde lo necesite. Teniendo un servidor en el cual se procesan las transacciones, el servidor de aplicaciones se limita a ejecutar el código del cliente.

Puesto que el número total de servidores es mucho menor a la cantidad de clientes, y los servidores están físicamente adyacentes, los precursores de este modelo, proponen que es menos complicada la administración comparada con la del modelo cliente / servidor tradicional, ya que carga de administración se mueven al servidor. Además, la escalabilidad es mejor, porque este modelo usa más paralelismo que el *mainframe* normal o el modelo cliente liviano / servidor robusto.

Haciendo un razonamiento a lo anterior, si tres capas dan buen resultado, agregar mas capas, seria mejor. Tomando la ultima generación de sistemas cliente / servidor, encontramos los llamados “sistemas orientados a objetos” y los “sistemas distribuidos de objetos”. En un sistema distribuido de objetos, los clientes hacen sus requerimientos de servicio a varios servidores independientes.

Estos servidores exportan los objetos al cliente, y el servidor aísla los detalles de la implementación de objetos de los clientes en sí. Los servidores proveen un objeto interfase que puede ser invocado por otros objetos que estén autorizados. La comunicación entre el cliente y las diferentes capas que existan, son invocadas por una secuencia de invocaciones anidadas. Estas invocaciones anidadas constituyen la comunicación que ocurre entre las capas en esta estrategia. Estas llamadas entre las capas pueden ser procesadas por invocaciones sincrónicas o asincrónicas, y por consiguiente, el paralelismo puede ser restringido o no restringido.

Los precursores de los sistemas distribuidos de objetos, argumentan que la administración de este modelo es más fácil de administrar, que el tradicional modelo cliente / servidor. Los servidores de aplicación que se ejecutan en una maquina simple, tienen el potencial de ser migradas a diferentes maquinas para balancear la carga. En muchos casos, el administrador del sistema puede replicar el servidor a otro o más, en varios servidores, si no se presentan problemas de rendimiento y fiabilidad. Los sistemas orientados a objetos, prometen mejor escalabilidad, reduciendo los costos de administración del sistema e implementando la reutilización del

código. El principal inconveniente de este modelo, es que el sistema se sobrecarga. El ancho de banda y la latencia de comunicación afectan al sistema especialmente si utilizan un granulo fino en aplicaciones paralelas o hay muchos usuarios compitiendo por un juego de recursos a través de la comunicación de un enlace de ancho de banda limitado. Como consecuencia de esto, existen dos implementaciones comerciales: *CORBA (Common Object Request Broker Architecture) y DCOM (Distributed Common Object Model)*.

1.7. Memoria distribuida compartida

El modelo cliente / servidor y el modelo de objetos distribuidos, proveen un avance revolucionario sobre el modelo de aplicación en *mainframes* / terminales (*green screen*). Trabajando en paralelo con el auge del modelo cliente / servidor, se ha ido desarrollando en paralelo el modelo de memoria distribuida compartida. Este modelo aprovecha provee una alternativa de bajo costo basada en software sobre los sistemas multiprocesadores de memoria compartida. Específicamente, se ha demostrado que se pueden compartir múltiples procesadores de forma virtual, utilizando redes de estaciones de trabajo de bajo costo. Esta demostración, a generado una nueva investigación de aplicaciones científicas a gran escala en hardware más cómodo. Infortunadamente, los primeros resultados generaron un rendimiento muy pobre en aplicaciones paralelas de granulo fino. La comunidad de investigación respondió con dos enfoques generales de soluciones: espacios de direccionamiento de granularidad fina, y nuevas primitivas de sincronización para los nuevos protocolos de consistencia.

A pesar de que durante una década de investigación con algunos adelantos impresionantes, el perfeccionamiento del rendimiento sigue siendo el enfoque primario de los investigadores de DSM. No obstante, la aceptación del modelo DSM no ha sido muy difundida.

Incluso, en áreas de aplicación propuestas - programas científicos - los sistemas como PVM y MPI, se ha vuelto competidores serios a los sistemas DSM debido a su portabilidad. *Carter, Khandekar & Kamb*, proporcionan varias explicaciones del por qué la comunidad científica tiene no adopta DSM: Los investigadores argumentan que una carencia de los sistemas DSM, tales como la disponibilidad y portabilidad, han generado inconvenientes para su utilización. Otro problema con el DSM, es el pobre rendimiento observado. El DSM provee una solución general de paralelización del software, media vez, la aplicación paralela este cuidadosamente diseñada y sincronizada, utilizando RPC's o paso de mensajes, puede funcionar mejor que la misma aplicación que emplea DSM. Dado que son pocas las aplicaciones científicas paralelas en el mundo real, comparadas con la gran cantidad de aplicaciones comerciales, y dada la necesidad del alto rendimiento, no sorprende la aceptación de sistemas DSM en la comunidad científica. No obstante, los investigadores de DSM han hecho progreso significativo que se orientan a la preocupación del rendimiento. Es ahora seguro decir que los enfoques del DSM son competitivos al RPC y el paso de mensajes; no obstante, la comunidad científica no ha estado enterada de estos desarrollos recientes.

Ajeno a la comunidad científica, la aceptación de los sistemas DSM se están comprendiendo rápidamente. En primer lugar, pocas aplicaciones

comerciales usan multiprocesamiento usando memoria compartida. A pesar de que las primitivas de memoria compartida existen en varios sistemas operativos, se difunden muy poco los métodos de comunicación ínter proceso de memoria compartida.

Para el éxito de los sistemas DSM en los sistemas comerciales, debe haber una curva de aprendizaje baja para el programador de las aplicaciones y el DSM debe exhibir un grado alto de transparencia en las referencias efectuadas. Es decir, el vendedor del sistema operativo, el vendedor del lenguaje de programación, o el vendedor de las bibliotecas de tiempo de corrida, son los que deben incorporar transparentemente el DSM en su sistema, si el DSM se adopta en la vida de programador de las aplicaciones.

1.8. Objetos distribuidos virtuales

El sistema de objetos actual utiliza una solución eficiente (*RPC*) para la invocación de métodos remotos, pero utiliza ineficientemente esta solución para implementar el acceso a los datos. Por otra parte, los sistemas DSM proporcionan un mecanismo conveniente y eficiente para accesar información remota, los cuales generan un alto rendimiento del cache. Sin embargo, estos sistemas no proveen ningún mecanismo para la invocación de métodos remotos. Estos dos enfoques poseen sinergia: sus fortalezas complementan sus debilidades. Esta sinergia se conoce como Objetos Distribuidos Virtuales (del ingles *Virtual Distributed Objects – VDO*). Estos proveen dos mecanismos importantes en una abstracción común: administración de la coherencia y del *cache* de alto rendimiento, y una alta velocidad para la invocación de métodos. Los *VDO's* aprovecha las

fortalezas de ambos métodos y elimina las debilidades individuales de los modelos. Utilizando VDO's, es posible simplificar el modelo de programación de objetos distribuidos y mejorar la eficiencia. Esta combinación, permite al programador tratar el objeto distribuido como un objeto local.

Por ejemplo, un programador de C++ puede tomar la dirección del acceso de un campo de datos de un objeto, y accesar el objeto indirectamente utilizando sus funciones almacenadas. Por contrario, en un sistema de objetos distribuidos estandard, se tienen que utilizar las llamadas "funciones de acceso" para obtener el valor de un miembro en especial. En estos últimos, no se puede tomar la dirección del objeto distribuido y referenciar la información de forma indirecta. El hecho que un objeto es remoto, en lugar de local, no es transparente al programador en el sistema de objetos. Por lo tanto, los sistemas DSM hacen uso de los objetos distribuidos más transparente, ya que el programador de la aplicación tiene una vista uniforme de los objetos remotos y locales. Dado este enfoque, el rendimiento y la escalabilidad tienden a ser mejores. Aunque el rendimiento y escalabilidad deban mejorar, los costos administrativos aumentarán, cuando una aplicación es distribuida en varias maquinas. Dado que los *VDO's* utilizan protocolos estandarizados, se espera el surgimiento de herramientas que ayuden a realizar la administración más sencilla. Las soluciones de alto rendimiento permiten el acceso directo a los campos de un objeto remoto a través de una llamada RPC pero se necesitan algunas técnicas de programación. Infortunadamente, la mayor parte de las aplicaciones no son escritas por programadores expertos en esas técnicas. Mientras el alto rendimiento siempre es una preocupación, los programadores comerciales están normalmente interesados en los desarrollos más rápidos, en lugar de las ejecuciones más rápidas.

2. DISEÑO, IMPLEMENTACIÓN Y ADMINISTRACIÓN DE LA MEMORIA

Uno de los principales temas para la construcción de sistemas DSM esta relacionada con la selección de algún algoritmo apropiado que provea una abstracción de la memoria compartida, el cual pueda encajar con el comportamiento del acceso a la memoria en la aplicación, lo cual determinara el grado de rendimiento que el sistema pueda proveer. La tarea de un algoritmo DSM eficiente es proveer distribución estática y / o dinámica de la información compartida, para minimizar la latencia de los accesos a la memoria. Esto se logra usualmente utilizando estrategias de replicación y migración. La replicación es principalmente utilizada para habilitar accesos simultáneos a la misma información por sitios diferentes, predominando las lecturas compartidas. La migración se prefiere cuando se utiliza un modelo típico secuencial de escrituras compartidas, con el objetivo de disminuir la administración de la consistencia. Ambas políticas tratan de reducir los tiempos de acceso, recuperando los datos del sitio donde se estén utilizando en ese momento. La efectividad de un algoritmo, también refleja su habilidad de reducir el número de traslados de los datos y la cantidad de sobrecarga de consistencia.

2.1. Implementación de algoritmos

Tradicionalmente, la comunicación entre procesos en un sistema distribuido se basa en el modelo de paso de datos.

Los sistemas que usan paso de mensajes o los sistemas que soportan la llamada a procedimientos remotos (RPC) se apegan a este modelo. Este modelo lógicamente y convenientemente, extiende el mecanismo de comunicación fundamental del sistema, utilizando puertos con las primitivas enviar / recibir, los cuales son usados para la intercomunicación de procesos. Esta funcionalidad puede ser ocultada en construcciones a nivel del lenguaje de programación, como mecanismos RPC. En cualquier caso, los procesos distribuidos pasan la información compartida por valor.

El modelo de memoria distribuida compartida, provee procesos de sistema en un espacio de direcciones compartidas. Las aplicaciones pueden usar este espacio de la misma forma como usualmente trabajarían como memoria local. La información en el espacio compartido es accesada mediante operaciones de lectura y escritura, dando como resultado el paso de la información por referencia. El modelo de memoria compartida, es natural para sistemas de computación que corren con multiprocesadores con memoria compartida. Para sistemas distribuidos débilmente acoplados, ninguna memoria física esta disponible para soportar este modelo. Sin embargo, una capa de software puede proveer la abstracción de la memoria compartida para las aplicaciones. Esta capa de software, puede ser implementada en el *kernel* del sistema operativo, con el soporte apropiado del este o con el uso de rutinas de librerías en tiempo de corrida; usan los servicios de un sistema fundamental de comunicación. El modelo de memoria distribuida es aplicado básicamente en los sistemas débilmente acoplados.

2.1.1. Ventajas de la memoria distribuida compartida

La ventaja primaria de memoria distribuida compartida sobre el paso de mensajes, es que proporciona un nivel de abstracción más sencilla al programador de la aplicación. El protocolo de acceso utilizado es consistente con el acceso a la información de las aplicaciones secuenciales, permitiendo una transición más natural de estas a las aplicaciones distribuidas. En principio, se desarrollaba la programación de manera paralela y distribuida para los sistemas multiprocesadores de memoria compartida, y esta podía ser ejecuta por otro sistema de memoria distribuida compartida sin tener necesidad de algún cambio. El sistema de memoria compartida oculta el mecanismo de comunicación remota permitiendo a las estructuras complejas ser pasadas por referencia, simplificando substancialmente la programación de las aplicaciones distribuidas. Además, la información en la memoria distribuida compartida puede permanecer más allá de la vida de un proceso que tiene acceso la memoria compartida.

En contraste, el modelo de paso de mensajes obliga al programador a estar consciente del movimiento de la información entre los procesos todo el tiempo, dado que los procesos usan explícitamente las primitivas de comunicación y canales o puertos para comunicarse. Esto implica, que la información es pasada entre múltiples espacios de direcciones lo cual dificulta el paso de las estructuras complejas de datos. Las estructuras de datos que son manejadas entre los procesos, tienen que ser colocadas y tomadas por la aplicación. Por estas razones, la escritura del código para una aplicación distribuida que utilice memoria distribuida compartida es usualmente más corta y más fácil de comprender que su equivalente usando programas utilizando paso de mensajes.

En algunos casos, las aplicaciones que utilizan memoria distribuida compartida pueden llegar a comportase como sus procesos contraparte de paso de mensajes bajo estas tres posible razones:

(i) Para los algoritmos de memoria compartida que mueven los datos entre máquinas en bloques grandes, la comunicación se amortiza sobre múltiples accesos de la memoria, reduciendo los requisitos completos de comunicación, si la aplicación exhibe un grado suficiente de la localidad en sus accesos de datos.

(ii) Muchas aplicaciones paralelas se ejecutan en fases, donde cada fase del cálculo es precedida por una fase de intercambio de datos. El tiempo necesitado para ejecutar la fase de intercambio de datos es limitada a veces por el rendimiento del sistema de comunicación. Los algoritmos de memoria distribuida compartida típicamente mueven la información en demanda mientras son accesados o eliminados en la fase de intercambio de datos, extendiendo la carga de la comunicación por un periodo largo de tiempo y permitiendo un alto grado de concurrencia.

(iii) El monto total de la memoria puede ser aumentada proporcionalmente, reduciendo el pagineo y la actividad de intercambio (*swapping*).

2.1.2. Sistemas similares

Los sistemas de memoria compartida distribuida tienen objetivos semejantes a aquellos sistemas que utilizan los procesadores con *cache* que utilizan memoria compartida con memoria local con accesos no uniformes.

En particular, tratan de aminorar el tiempo de acceso a los datos que potencialmente son compartidos en un estado consistente. A pesar de que estos sistemas usan algoritmos semejantes, sus detalles e implementación pueden variar significativamente debido a las diferencias en los parámetros del costo y de las maneras ellos se usan. Similarmente, los multiprocesadores de memoria compartida basados en bus, la replicación del *cache* del CPU puede llevarse a cabo a bajo costo, debido a la fiabilidad y propiedades de la transmisión del bus. Por otro lado, en los sistemas distribuidos donde la comunicación es inestable, se utiliza un algoritmo sin replicación la cual puede beneficiar cierto tipo de aplicaciones.

2.2. Algoritmos básicos

Los algoritmos que serán descritos pueden ser clasificados de acuerdo a la forma en que migran o replican la información, la cual es descrita con la tabla I.

Tabla I. Los cuatro algoritmos para la memoria distribuida

	Sin replicación	Con replicación
Sin migración	Central	Replicación Completa
Con migración	Migración Completa	Replicación Parcial

Los algoritmos de migración de información tratan de explotar los accesos en la localidad, tratando de minimizar el número de accesos remotos. Los algoritmos de replicación de información, pueden proporcionar múltiples accesos de lectura al mismo tiempo usando accesos locales.

Las implementaciones de memoria distribuida compartida basada en replicación deberían hacer esta copia transparente para la aplicación. En otras palabras, los procesos no deberían poder observar (las lecturas y escrituras a la información compartida) los accesos a la información dado que estos accesos no son dirigidos a la misma copia de datos. El resultado de usar aplicaciones con datos compartidos debería ser lo mismo como si las operaciones en memoria de todos los servidores serán ejecutadas en algún orden secuencial, y las operaciones de cada servidor individual aparecen consecutivamente en la orden especificado por su programa, en cuál el caso la memoria compartida es debe ser consistente. La memoria compartida en un sistema multiprocesador con memoria compartida – se espera – que se comporte así. Esta definición de consistencia no debería ser confundida con una definición más estricta requiriendo accesos ejecutados para devolver el valor más reciente escrito para la misma localización, lo cual es naturalmente aplicable para la ejecución concurrente de procesos en un sistema uniprocesador pero no necesariamente para esos en multiprocesadores de

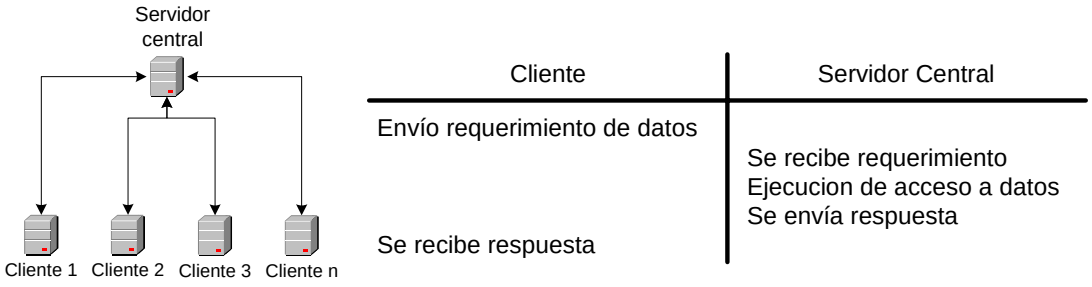
memoria compartida con CPU's con *cache* y *buffers* de escritura retardada. En caso de que mientras la definición sea más estricta para su aplicación, entonces la definición se haga más débil.

2.2.1. Algoritmo de servidor central

La estrategia más simple para implementar memoria compartida distribuida es usar un servidor central, quien es el responsable para servir todos los accesos para los datos compartidos y mantiene la única copia de los mismos.

Ambos operaciones de lectura y escritura involucran el envío de un mensaje de petición para el servidor de datos por el proceso que ejecuta la operación, tal como se describe en la figura 2. El servidor de datos ejecuta la petición y responde ya sea con los datos solicitados en caso de una operación de lectura, o con una confirmación en caso de una operación de escritura.

Figura 2. Algoritmo de servidor central



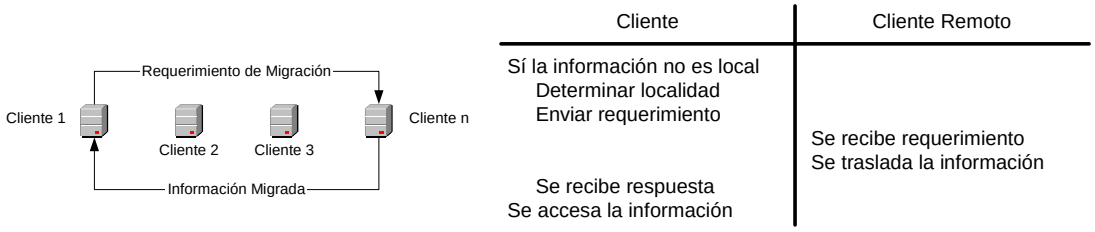
Un protocolo de petición-respuesta simple puede servir para la comunicación en implementaciones de este algoritmo. Para la fiabilidad, una petición es retransmitida después de cada período de tiempo agotado sin respuesta. Esto es suficiente, desde que la petición ejecutada es expirada; para un requerimiento de escritura, el servidor debe conservar un número de secuencia para cada cliente a fin de que pueda detectar transmisiones duplicadas y los pueda reconocer apropiadamente. Una condición de falla es emitida después de varios períodos de tiempos agotados sin respuesta. Por lo tanto, este algoritmo requiere dos mensajes para cada acceso de datos: uno del proceso requiriendo el acceso para el servidor de datos y el otro conteniendo la respuesta del servidor de datos. Además, cada acceso de datos requiere cuatro eventos del paquete: dos en el proceso de petición (uno a enviar la petición y uno a recibir la respuesta) y dos en el servidor.

Un problema potencial con el servidor central es que puede convertirse en un cuello de botella, desde que tenga que atender las demandas múltiples de todos los clientes. Para distribuir la carga del servidor, los datos compartidos pueden ser distribuidos en varios servidores. En ese caso, los clientes deben poder localizar el servidor correcto para el acceso de datos. Un cliente puede repartir sus peticiones de acceso para todos los servidores, pero no disminuiría significantemente la carga en todos los servidores, desde que cada servidor obtendría una sobrecarga de eventos de paquetes por cada petición. Una mejor solución es dividir en partes los datos por direccionamiento y utilizar alguna función simple del mapeo a decidir cuál servidor a contactar.

2.2.2. Algoritmo de migración

En este algoritmo, la información siempre es migrada al sitio donde es accesada, tal como se describe en la figura 3.

Figura 3. Algoritmo de migración



Este algoritmo es un protocolo lector simple / escritor simple (LSES – SRSW), ya que sólo la tarea es ejecutada en un servidor, se puede leer o escribir la información en cualquier momento.

En lugar de migrar los elementos de datos, estos son típicamente migrados entre servidores en una unidad fija de tamaño llamado bloque, para facilitar su administración. La ventaja de este algoritmo, es que ninguno de los costos de comunicaciones es incurrido cuando el acceso de un proceso sé esta ejecutando localmente. Si una aplicación exhibe una alta localidad de referencia, el costo de migración de datos es amortizado sobre accesos múltiples. Sin embargo, con este algoritmo es también posible que páginas se muevan agitadamente entre los servidores, dando como resultando pocas migraciones de accesos en la memoria, y el rendimiento es por consiguiente muy bajo. A menudo, el escritor de la aplicación podrá controlar esta agitación juiciosamente asignando datos por bloques.

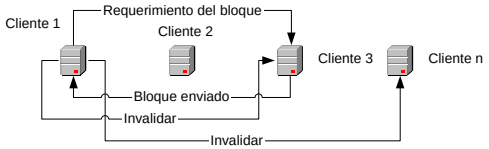
Una segunda ventaja del algoritmo de migración es que puede ser integrada con el sistema de memoria virtual del sistema operativo del servidor, si el tamaño del bloque es igual al tamaño de una página virtual de memoria. Si una página compartida de memoria está sujeta localmente, puede ser de la que se trazó un mapa en el espacio de direcciones virtuales de la aplicación, y es accesada por las instrucciones normales de la máquina para accesarla. Un acceso para un elemento de dato localizado en bloques de datos no mantenidos localmente provoca una falla de la página a fin de que el administrador pueda comunicarse con el servidor remoto para obtener el bloque de datos trazando un mapa de él en el espacio de direcciones de la aplicación. Cuándo un bloque de datos es migrado, es removido de cualquier espacio local de direcciones en el que ha sido trazado.

La localización de un bloque remoto de datos puede ser encontrada por de varios destinatarios a través de un mensaje de petición de migración para todos los servidores remotos, pero no existe un método eficiente conocido. Por ejemplo, estáticamente se puede asignar a cada bloque de datos un servidor que siempre conoce la localización del bloque de datos. Para distribuir la carga, la dirección de todos los bloques de datos está subdividida a través de todos los servidores. Un cliente consulta al servidor que administra el bloque de datos, y ambos no determinan la localización actual del bloque de datos e informar al administrador que se migrarán los bloques de datos.

2.2.3. Algoritmo de replicación de lecturas

Una desventaja de los algoritmos descritos es que sólo los procesos en un servidor pueden acceder a los datos contenidos en el mismo bloque en cualquier tiempo dado. La copia puede reducir el costo de las operaciones de lectura, desde que se permita ejecutar operaciones de lectura simultáneamente localmente (sin sobrecarga de comunicación) en servidores múltiples. Sin embargo, algunas de las operaciones de escritura pueden elevar el costo, desde que las copias pueden tener que ser deshabilitadas o actualizadas para mantener la consistencia. No obstante, sí la tasa de lecturas sobre escrituras es elevado, el costo adicional de las operaciones de escritura puede ser más que la desviación por el costo promedio de las operaciones de lectura. La copia puede agregarse naturalmente al algoritmo de migración permitiendo ya sea un sitio una copia de lectura / escritura de un bloque particular o los sitios múltiples leen sólo las copias de ese bloque. Este tipo de copia es al que se refirió como múltiples lectores / escritor sencillo (MLES, en ingles MRSW). Para las operaciones de lectura de un elemento de datos en un bloque que no es actualmente local, hay que comunicarse con sitios remotos para primero adquirir una copia en modalidad de sólo-lectura de ese bloque, y para cambiar esa modalidad es necesario cambiar los derechos de acceso para cualquier copia escribible sí es necesario, antes que la operación de lectura pueda ser completada. Para una operación de escritura de datos en un bloque ya sea local o remota que no posea permiso de escritura, todas las copias del mismo bloque deben ser mantenido en otros sitios debiendo ser invalidados antes que la escritura puedan proceder, tal como se demuestra en la figura 4.

Figura 4. Algoritmo de replicación de lecturas



Cliente	Cliente Remoto
Sí la información no es local Determinar localidad Enviar requerimiento	
	Se recibe requerimiento Se traslada la información
Se recibe respuesta Emisión de invalidacion	
	Se recibe emisión de invalidación Bloque invalidado
Se accesa la información	

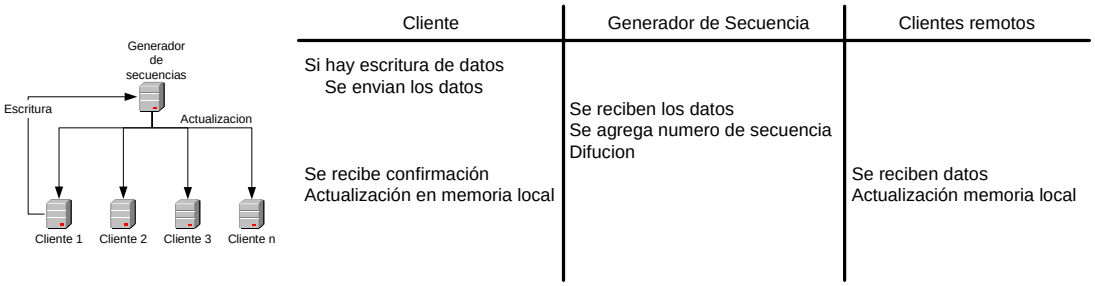
Esta estrategia se parece al algoritmo de invalidación escritura de consistencia del *cache* implementada en hardware en varios sistemas de multiprocesadores. El algoritmo de replicación de lectura es consistente, porque un acceso de lectura siempre retorna el valor de escritura más reciente para la misma localización. Un acceso de lectura (o escritura) en un bloque en el cual un servidor no tiene los derechos de acceso apropiados causa una falla de lectura (o escritura). El administrador de fallas transmite una petición al servidor que tiene en propiedad el bloque apropiado.

Para una falla de lectura, el servidor que lo posee contesta con una copia del bloque, se añade al servidor solicitante la copia determinada del bloque y, sí es necesario, los derechos de acceso de su copia local en sólo modo de lectura. Cuando una falla de escritura ocurre, la propiedad del bloque es transferida al dueño previo para el servidor donde la falla de escritura ocurrió. Después de recibir la respuesta, una petición del administrador de fallas de escritura solicita a todos los servidores que invaliden su copia local, después del cuál los derechos de acceso para ese bloque sean configurados con accesos de escritura por el nuevo propietario y la copia determinada sea liberada.

2.2.4. Algoritmo replicación completa

La replicación completa permite a los bloques de datos estar replicados incluso mientras se están escribiendo. El algoritmo de replicación completa por consiguiente se apega a un protocolo múltiples lectores / múltiples escritores. Para mantener la consistencia de las copias de los datos en algoritmos no-replicados es necesario que el acceso a los datos lleve una secuencia según el orden en el cual ocurren en el lugar donde los datos se mantienen. En caso de la replicación completa, los accesos a los datos deben ser de llevados en algún control o secuencia para asegurar consistencia. Una forma posible de conservar los datos consistentes replicados es poner en secuencia global a las operaciones de escritura, mientras la secuencia de las operaciones de lectura son relativas a las escrituras, estas ocurrirán localmente para el sitio donde esta sea ejecutada. Una estrategia simple basada en secuencias es utilizar una secuencia simple libre de desfases de requerimientos globales, lo cual es un proceso ejecutando en un servidor participando de memoria compartida distribuida, tal como se describe en la figura 5.

Figura 5. Algoritmo de replicación completa



Cuando un proceso intenta una escritura en la memoria compartida, el intento de modificación tiene que ser enviado a ese servidor. La máquina

que controla las secuencias asignara ordenadamente el siguiente número. Cuándo una modificación llega a un sitio, el número de secuencia se verifica para ver sí es la esperada. Si un desfase en los números de secuencia es detectada, entonces una modificación hizo falta o una modificación fue recibida por alguna orden, en cuál caso una retransmisión del mensaje de modificación es demandada (ésta significa que a alguna parte se lleva un control reciente de las peticiones de escritura). En efecto, esta estrategia implementa un protocolo negativo de confirmación. En el caso común dentro del ambiente asumido, los paquetes llegan a todos los sitios en un orden correcto. Por consiguiente, una escritura requiere dos paquetes de eventos para un proceso escritor, dos eventos generados en el servidor de secuencias, y un paquete para cada replica que participa; para un sistema total de S + 2 paquetes de eventos con S sitios participando.

2.2.5. Comparaciones de rendimiento

Todos los cuatro algoritmos descritos anteriormente, aseguran consistencia en la memoria distribuida compartida. Sin embargo, su rendimiento es sensitivo para el comportamiento de acceso de los datos en la aplicación. Los parámetros descritos en la tabla II representan los costos básicos de los accesos a los datos compartidos y el comportamiento de las aplicaciones.

Tabla II. Parámetros que caracterizan los costos de acceso para datos compartidos

p	El costo de un paquete de acontecimiento, esto es, el costo de procesamiento de envío o recepción de un paquete pequeño, el cuál incluye posibles cambios de contexto de copia de datos, y sobrecarga de manipulación de interrupciones. Los valores típicos para los sistemas reales van de cero a varios milisegundos.
P	El costo de envío o recepción un bloque de datos. Esto es similar a p, excepto que la P es significativamente más alta. Para un bloque de 8 *kilobytes*, dónde a menudo se necesitan múltiples paquetes, los valores típicos se extienden desde 20 para 40 milisegundos.
S	El numero de sitios participantes en el sistema de memoria distribuida compartida
R	La razón de lecturas/escrituras, esto es, hay una operación de escritura por cada r lecturas en promedio. Este parámetro se usa también para referirse al patrón de acceso de bloques enteros. Aunque las razones pueden diferir, se asume que son iguales para simplificar los análisis.
F	Probabilidad de una falla de acceso de un bloque de datos no replicado usando el algoritmo de migración. Esto es igual al inverso del número común de accesos consecutivos para un bloque por un solo sitio, antes de que otro sitio haga un acceso al mismo bloque, generando una falla. f caracteriza la localidad de accesos de datos para el algoritmo de migración.
F'	Probabilidad de una falla de un bloque replicado, usando el algoritmo de replicación de lecturas. Es la inversa al promedio de accesos consecutivos a los elementos de datos en bloques mantenidos localmente, antes que uno de los elementos de datos en un bloque no mantenido localmente es accesado. f´ caracteriza los accesos de datos localmente para el algoritmo de replicación de lecturas.

Entre ellos, los dos tipos de razón de falla de acceso, f y f', tienen el impacto más alto en el desempeño de los algoritmos correspondientes, pero desafortunadamente, es también lo más difícil para evaluar desde que se diferencien ampliamente en la aplicación. Se tiene que señalar que estos parámetros no son enteramente independientes uno del otro. Por ejemplo, el tamaño de un bloque de datos y el costo de transferencia del bloque, P, proporciona una influencia sobre f y f'. Sí el tamaño del bloque aumenta, más accesos en el bloque son posibles antes de que otro bloque sea accesado; sin embargo, las interferencias de acceso entre sitios son más probables. S también tiene impacto directo en las tasas de fallas. Para enfocar la atención en las características esenciales de desempeño de los algoritmos y simplificar los análisis, se asumirán las siguientes condiciones:

(i) La cantidad de tráfico de mensajes no causará congestión en la red. Por lo tanto, sólo consideraremos el costo del procesamiento del mensaje, *p* y *P*, pero no el ancho de banda de la red ocupado por los mensajes.

(ii) La congestión del servidor no es lo suficientemente alta para retrasar acceso remoto significativamente. Esto es razonable para los algoritmos presentados, dado que hay formas efectivas para reducir la carga en los servidores.

(iii) El costo de acceso a un elemento de datos localmente es insignificante comparado con el costo del acceso remoto y por consiguiente, no se tomara en cuenta en los cálculos del costo de acceso.

(iv) Se asume que el paso de los mensajes es confiable, así que el costo de retransmisión no es obtenido. Sin embargo, el costo para los mensajes de confirmación son requeridos para determinar si

una retransmisión es necesaria o no, lo cual es incluido en los modelos.

Para comparar el rendimiento de los algoritmos de memoria distribuida compartida, se va a definir una medida de rendimiento. La memoria compartida distribuida se usa a menudo para soportar aplicaciones paralelas en las cuales las tareas múltiples en ejecución pueden estar en marcha en cierto número de sitios. Por consiguiente, se escoge el costo medio del acceso a los datos para el sistema total como la medida de rendimiento. Por lo tanto, si un acceso de datos involucra a uno o más sitios remotos, entonces el procesamiento de los mensajes entre ambos sitios locales y remotos es incluido.

Usando los parámetros básicos y las suposiciones simplificadoras descritas anteriormente, el costo promedio de acceso de los cuatro algoritmos puede ser expresado como sigue:

Algoritmo de servidor central $C_c = \left(1 - \frac{1}{S}\right) * 4p$

Algoritmo de migración $C_m = f * (2P + 4p)$

Algoritmo de replicación de lecturas $C_{rr} = f' * \left(2P + 4p + \frac{Sp}{r+1}\right)$

Algoritmo de replicación completa $C_{fr} = \frac{1}{r+1} * (S+2)p$

Cada una de estas expresiones, tiene dos componentes. El primer componente, a la izquierda del *, es la probabilidad de un acceso a un

elemento de datos en algún sitio remoto. El segundo componente, a la derecha del * es igual al costo promedio de acceso a los elementos remotos. Dado que el costo del acceso local - se asume - es insignificante, el costo medio de acceso a un elemento de datos, es por consiguiente el producto de estos dos componentes.

En el caso del algoritmo de servidor central, la probabilidad de acceso un elemento remoto de datos es $1-\frac{1}{S}$, en cuyo caso cuatro eventos de paquete son necesarios para el acceso (asumiendo que los datos están uniformemente distribuidos sobre todos los sitios). El costo total C_c, es por consiguiente determinado principalmente por el costo de un evento de paquete, mientras el número de sitios sea más de cuatro o cinco.

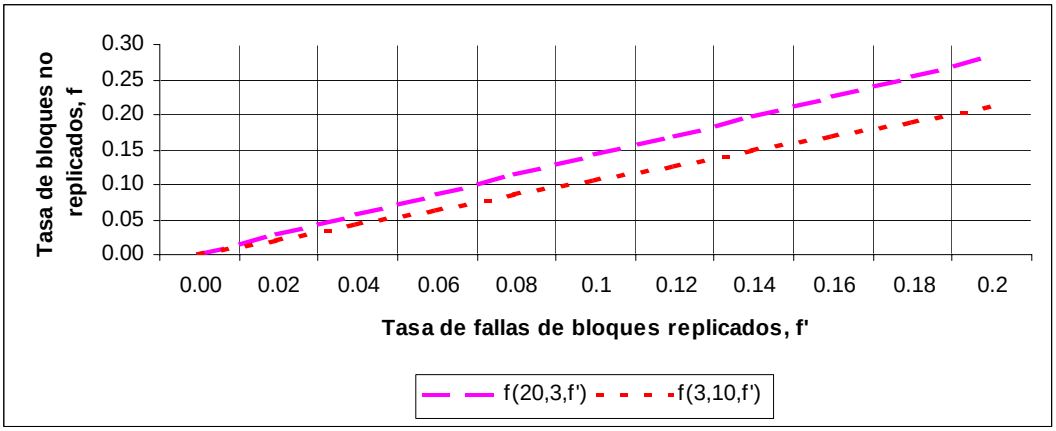
En el caso del algoritmo de migración, f representa la probabilidad de acceso a un elemento de datos no local. El costo de acceso a un elemento de datos en este caso, corresponde al costo de traída de los bloques de datos que contienen los elementos de datos al sitio local, lo cual incluye la transferencia total de un bloque (2P) y cuatro eventos de paquete distribuidos entre el sitio local, el administrador y los sitios remotos. Se asume que el administrador local, y los servidores de los sitios son todos distintos, y el requerimiento es reenviado por el administrador del servidor. La secuencia de eventos de paquetes es envío (en el sitio local), recepción (por el administrador del sitio), reenvió (por el administrador del sitio) y recepción (en el servidor del sitio).

En el caso del algoritmo de replicación de lecturas, el costo de acceso remoto se aproxima al algoritmo de migración excepto en el caso de una falla de escritura (que ocurre con una probabilidad de $\frac{1}{r+1}$), lo cual conlleva a la generación de un paquete de invalidación, el cual debe ser manejado por todos los S sitios. El costo de transferencia en bloque es siempre incluido en la expresión, aunque no puede ser necesario si una falla de escritura ocurre y la copia local (en modo lectura) del bloque está disponible.

Finalmente, en el caso del algoritmo de replicación completa, la probabilidad de un acceso remoto es igual a la probabilidad de un acceso de escritura. El costo asociado para ésta escritura es siempre un mensaje del sitio local para la máquina para establece las secuencias (dos paquetes), seguida por la propagación de un mensaje de actualización para todos otros sitios (S paquetes).

Las descripciones descritas anteriormente generan algunas comparaciones del rendimiento de los algoritmos, lo cual ilustra las condiciones bajo las cuales algún algoritmo podría funcionar mejor que otro. Cada comparación está hecha igualando los costos medios de dos algoritmos relacionados, para derivar una curva a lo largo de la cual produce un rendimiento similar. Esta curva, la cuál llamaremos la curva del rendimiento igual, divide el espacio de parámetro en dos mitades, algo semejante que en cada medio de los algoritmos realizará algo mejor que el otro. Por ejemplo, la siguiente comparación entre el algoritmo de migración y el algoritmo de replicación de lectura, la ecuación en la figura 6, es la curva del rendimiento igual, derivada de con algún arreglo para su ilustración.

Figura 6. Comparación de rendimiento: migración versus replicación de lectura



Dado que todas las fórmulas incluyen el costo del paquete p, sólo la razón entre P y p en los siguientes análisis comparativos. Se va a asumir que el valor de $\frac{P}{p}$ es 20. Basados en estas comparaciones, se realizaran algunos comentarios del rendimiento.

La única diferencia entre los algoritmos de migración y replicación de lecturas radica en que la copia es usada en el algoritmo de replicación de lecturas para permitir interpaginado en varios sitios sin movimientos de bloques, pero a expensas de la petición de invalidación de varios destinatarios en la actualización. Como se observa en la gráfica 1, el tráfico de invalidación no tiene una influencia fuerte en el rendimiento de los algoritmos. Mientras el costo de una transferencia en bloque es sustancialmente más alto que eso de un mensaje pequeño, las curvas para los valores diferentes de S y r se aproximan y esté muy cerca para la línea f y f '. Típicamente, la replicación de lectura efectivamente reduce la tasa de fallas del bloque porque en contraste al algoritmo de migración, los accesos

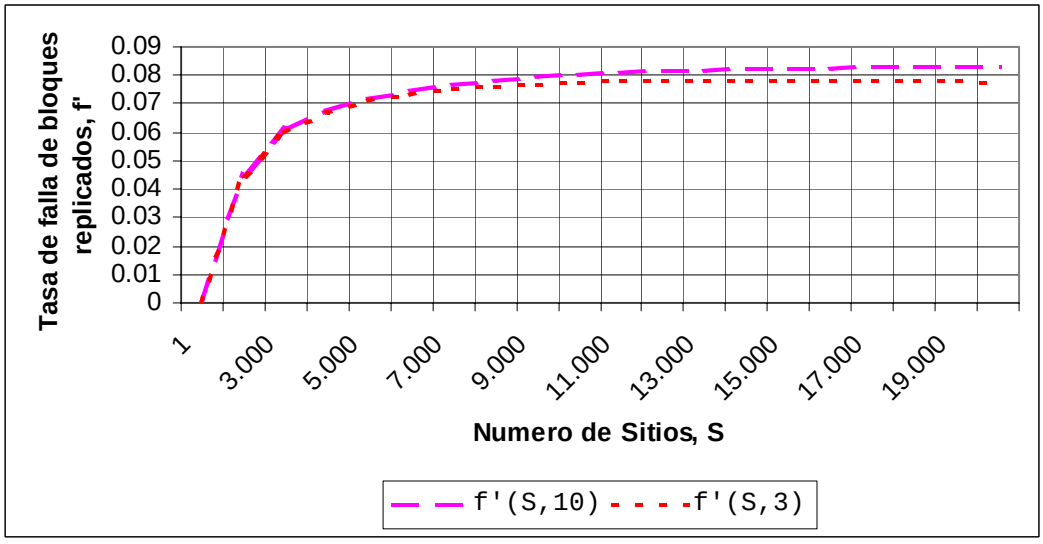
intercalados para la lectura del bloque ya no causan fallas, así el valor de f ' es más pequeño que f. Por consiguiente, uno puede esperar una replicación de lecturas funcione mejor que la migración para una inmensa mayoría de aplicaciones. Los cálculos para la figura 6, son los siguientes:

$$f(S,r,f') = f'\left[1+\frac{S}{44(r+1)}\right]$$

S	r	f'	0.00	0.02	0.04	0.06	0.08	0.1	0.12	0.14	0.16	0.18	0.2
20	3	f(20,3,f')	0.00	0.02	0.04	0.07	0.09	0.11	0.13	0.16	0.18	0.20	0.22
3	10	f(3,10,f')	0.00	0.02	0.04	0.06	0.08	0.10	0.12	0.14	0.16	0.18	0.20

En el caso de comparar los algoritmos de servidor central y replicación de lecturas, la figura 7 muestra la curva del rendimiento igual para estos algoritmos.

Figura 7. Comparación de rendimiento: servidor central versus replicación de lectura



Los cálculos para la figura 7, son los siguientes:

$$f'(S,r) = \frac{4\left(1-\frac{1}{S}\right)}{44+\frac{S}{r+1}}$$

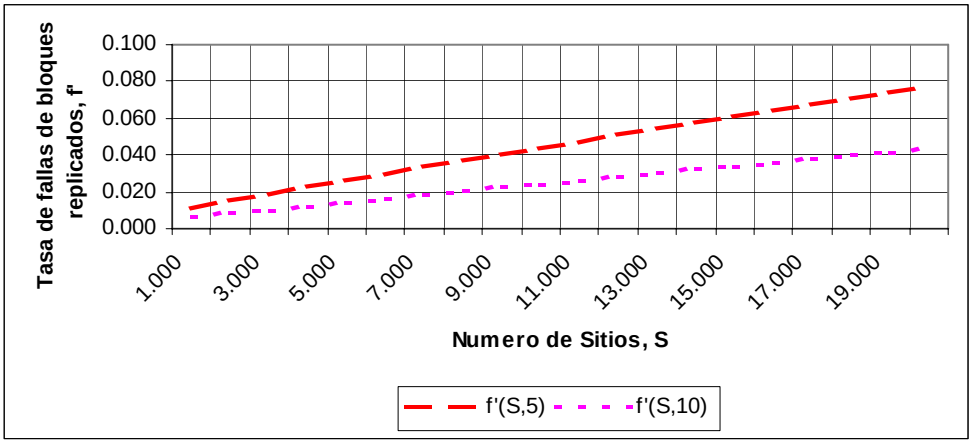
r	S	1	2.000	3.000	4.000	5.000	6.000	7.000	8.000	9.000	10.000	11.000
10	f'(S,10)	0	0.045	0.060	0.068	0.072	0.075	0.077	0.078	0.079	0.080	0.081
3	f'(S,3)	0	0.045	0.060	0.067	0.071	0.073	0.075	0.076	0.077	0.077	0.078

r	S	12.000	13.000	14.000	15.000	16.000	17.000	18.000	19.000	20.000
10	f'(S,10)	0.081	0.082	0.082	0.082	0.083	0.083	0.083	0.083	0.083
3	f'(S,3)	0.078	0.078	0.078	0.078	0.078	0.078	0.078	0.078	0.078

La curva del rendimiento igual es casi plana, lo que implica que es insensible para el número de sitios. Además, la influencia de la taza de lectura / escritura es también mínima. Por lo tanto, la llave a considerar entre los dos algoritmos es la localidad de acceso. Típicamente, una tasa de falla de bloque del 0.07 (14 accesos entre falla) es considerada muy alta (las fallas son muy frecuentes). Por consiguiente, la replicación de lecturas parece ser más favorable para muchas aplicaciones.

En el caso de comparar los algoritmos de replicación de lecturas y replicación completa, ambos tienen como base la replicación de lecturas pero la replicación completa es más agresiva que el de las copias múltiples que son mantenidas, aun para actualización. Como se pude observar en la grafica 3, muestra el rendimiento relativo de los dos algoritmos que dependen de un número de factores, incluyendo el grado de replicación, la tasa de lectura / escritura, y el grado de factibilidad de la localidad para realizar la replicación de lecturas.

Figura 8. Comparación de rendimiento: replicación de lectura versus replicación completa



Los cálculos para la figura 8, son los siguientes:

$$f'(S,r) = \frac{S+2}{S+44(r+1)}$$

R	S	1.000	2.000	3.000	4.000	5.000	6.000	7.000	8.000	9.000	10.000
5	f'(S,5)	0.011	0.015	0.019	0.022	0.026	0.030	0.033	0.037	0.040	0.044
10	f'(S,10)	0.006	0.008	0.010	0.012	0.014	0.016	0.018	0.020	0.022	0.024

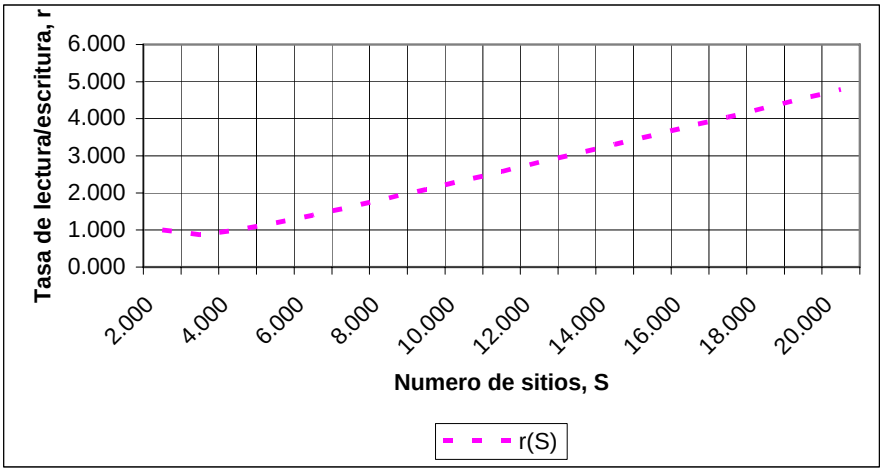
r	S	11.000	12.000	13.000	14.000	15.000	16.000	17.000	18.000	19.000
5	f'(S,5)	0.047	0.051	0.054	0.058	0.061	0.064	0.068	0.071	0.074
10	f'(S,10)	0.026	0.028	0.030	0.032	0.034	0.036	0.038	0.040	0.042

r	S	20.000
5	f'(S,5)	0.077
10	f'(S,10)	0.044

El algoritmo de replicación completa no es susceptible para localidades poco numerosas, dedo que todos los datos están replicados en todos los sitios. Por otra parte, el costo de difusión aumenta con S. Por consiguiente, la replicación completa funciona pobremente para los sistemas grandes y cuándo la frecuencia de actualización es alta (esto es, cuando la r es bajo).

En el caso de comparar los algoritmos de servidor central y replicación completa, representan los extremos para la implementación de un sistema de memoria compartida: mientras uno es completamente centralizado, el otro es completamente distribuido y replicado. Excepto para valores pequeños de S, su comportamiento es casi lineal, tal como se observa en la figura 9. Para valores de S arriba de 20, la agresividad de la replicación completa parece ser ventajosa, mientras la proporción de lectura / escritura es de cinco o más. Para una replicación muy grande, sin embargo, el costo de actualizar una replicación caería en el tiempo que tomaría ponerse al día, y la preferencia se convertirá al uso de un algoritmo simple de servidor central.

Figura 9. Comparación de rendimiento: servidor central versus replicación completa



Los cálculos para la figura 9, son los siguientes:

$$r(S) = \frac{1}{4} * \left[(S-1) + \frac{3}{S-1} \right]$$

S	2.000	3.000	4.000	5.000	6.000	7.000	8.000	9.000	10.000
r(S)	1.000	0.875	1.000	1.188	1.400	1.625	1.857	2.094	2.333

S	11.000	12.000	13.000	14.000	15.000	16.000	17.000	18.000	19.000	20.000
R(S)	2.575	2.818	3.063	3.308	3.554	3.800	4.047	4.294	4.542	4.789

Los dos pares restantes no consideraron y están resumidos como sigue. La comparación entre servidor central y migración se parece a esa entre servidor central y replicación de lecturas, con una curva más bien plana por debajo de f = 0.09. Así, a menos que la tasa de fallas de bloque sea muy alta, la migración funciona mejor. La comparación entre la migración y la reaplicación completa no revela un claro ganador, tal como el caso de replicación de lectura versus replicación completa, con la decisión de los factores S y r.

2.3. Análisis de algoritmos de memoria distribuida compartida

Los modelos principales de comunicación de procesos son el paso de mensajes y la memoria compartida. En el primer modelo, la comunicación y sincronización de los procesos se ejecuta a través del envió y recepción de mensajes, mientras que en el segundo modelo interactúa a través de recuperación y almacenamiento de los datos en un espacio común de direcciones. La memoria compartida puede ser considerada mas flexible

para la intercomunicación de procesos, si por ninguna otra razón la facilidad del paso de mensajes pueda ser implementada dado una memoria compartida. Moverse en dirección opuesta, particularmente de un sistema que sólo soporta paso de mensajes para un sistema de memoria compartida, no es una tarea simple. Requiere que el desarrollo de los algoritmos incluirá una vista consistente del espacio compartido con sobrecarga aceptable. A estos algoritmos que mantienen esta vista del espacio algoritmos de memoria distribuida compartida.

En un sistema distribuido cada procesador tiene su propio espacio de direcciones virtuales que es direccionada a su memoria local. Un procesador puede accesar su memoria local, a diferencia de los sistemas multiprocesadores, en los cuales los diferentes procesadores pueden accesar la misma memoria física. Como optimización de rendimiento, en estas arquitecturas ya sea remotas o locales, se tiene que migrar datos entre cualquiera de las memorias locales relacionadas. Moverse en dirección opuesta, particularmente de un sistema que sólo soporta paso de mensajes para un sistema de memoria compartida, no es una tarea simple. Requiere que el desarrollo de los algoritmos incluya una vista consistente del espacio compartido con una sobrecarga aceptable.

En un sistema distribuido cada procesador tiene su propio espacio de direcciones virtuales que trazó un mapa de su memoria local. Un procesador solo puede accesar su memoria local, a diferencia de los sistemas multiprocesadores los cuales permiten que diferentes procesadores accesen la misma memoria física. Como una optimización de rendimiento en las arquitecturas locales / remotas, uno de ellos puede migrar la información de

la memoria remota a la memoria local. La migración es una necesidad, no una optimización, cuando esta soportando un sistema con memoria distribuida compartida. Un procesador puede leer el contenido de una localización en el espacio compartido que estaba escrito por otro proceso sólo si su contenido ha sido transferido por un mensaje de la memoria local del escritor para la memoria local del lector. Con la ayuda del control de acceso proporcionado por la memoria virtual de hardware, la necesidad para ejecutar la transferencia puede ser detectada.

2.3.1. Cómo trabaja la memoria compartida distribuida

Los algoritmos DSM son implementados en software, pero es muy similar al método de directorios de hardware, el cual es usado para mantener la consistencia en un sistema multiprocesador. Una sólida definición de lo que es un sistema consistente de memoria es la siguiente: un sistema en el cual una lectura en una localización de la memoria compartida devolverá el valor del último valor para esa localización. Una definición más sencilla sería que una lectura eventualmente devolverá el valor de la última actualización. La consistencia del *cache* en hardware mantiene la consistencia en cada bloque del *cache*, la cual es la unidad básica de la memoria controlada por el hardware. La diferencia principal entre DSM y la consistencia del *cache* en hardware es que el costo de mantener un DSM coherente puede ser más alto, dado que la coherencia es mantenida en páginas, no bloques de *cache*, y las transferencias están obligadas a transferir páginas en la red. No sólo son paginas de memoria virtual considerablemente más grandes que un bloque de *cache*, pero la comunicación de la red provocara latencias más altas, que en un bus consistente de hardware.

Los algoritmos DSM utilizan cuatro ideas principales para mantener una imagen consistente de la memoria compartida:

(i) El DSM está subdividido en páginas, la granularidad ofrecida por la memoria virtual.

(ii) La página es la unidad en la cual la consistencia es mantenida. La escritura a una página es estrictamente ordenada y eventualmente se propaga para todos los nodos.

(iii) Las copias de cada página del DSM existirán en uno o más nodos según las restricciones de consistencia. En cualquier momento, al menos una copia de cada página existe.

(iv) El hardware de memoria virtual es usado para restringir el acceso a los datos compartidos. En a cualquier momento un procesador puede tener: Ningún acceso, acceso de lectura, o acceso de lectura / escritura para una página dada del DSM. Los fallas de la página ocurrirán cuando un proceso viole derechos de acceso.

Las reglas implementadas por los algoritmos DSM para mantener eficazmente la consistencia son:

(i) Un nodo deberá tener acceso de lectura y escritura a una página de la memoria compartida si y sólo si el nodo dado tiene la única e incomparable copia de la página dada.

(ii) Un solo nodo deberá tener acceso de lectura a una página sí y sólo si existen copias múltiples de la página en nodos diferentes.

Para mantener a estas invariantes, los algoritmos DSM siguen la pista a la localización de todas las copias de una página dada. La forma más fácil de hacer esto es tener a un propietario para cada página, quien conocerá cuáles nodos retienen una copia de la página. Todos los requerimientos por cambio de derechos de acceso a la página, pasarán a través del propietario de esa página. La propiedad puede ser asignada dinámicamente y ser transferida de un nodo a otro.

Se pueden presentar dos tipos de fallas cuando los derechos de acceso son violados: fallas de lectura y fallas de escritura. Las fallas de lectura ocurrirán cuando un proceso trate de leer una página para la cual no tiene acceso, el nodo no tiene una copia de la página. Las fallas de escritura ocurrirán cuando un proceso trate de escribir una página para la cual no tiene acceso de lectura y de escritura.

El protocolo base seguido sobre una falla de lectura de una pagina DSM es como sigue:

(i) El nodo que falla requiere una copia de la página al propietario de la página.
(ii) El propietario de la página envía una copia al nodo que falla, notando que la localidad de la nueva copia cambiando su derecho de acceso a solo lectura.
(iii) El nodo que falla recibe la página y coloca los derechos de acceso locales a sólo lectura.

El protocolo base seguido sobre una falla de escritura de una página DSM es como sigue:

(i) El nodo que falla requiere accesos de lectura y escritura al propietario de la página.
(ii) El propietario de la página envía la página, sí es necesario, y actualiza la lista de nodos que contienen una copia de la página.
(iii) El nodo que falla envía el mensaje de invalidación a todos otros nodos con una copia de la página dada. Al recibir un mensaje de invalidación, la copia de la página dada se destruye.
(iv) El nodo que falla coloca los derechos de acceso locales para leer y escribir, y se convierte en el nuevo propietario de la página.

Note que con este algoritmo puede ser posible en copias de sólo lectura de una página dada, existan en una longitud del tiempo pequeña después de otro nodo ha obtenido acceso de escritura para la página, según que escriba, puede proceder antes de que las invalidaciones sean completadas. Si la escritura no está permitida hasta que las invalidaciones sean completadas, la definición fuerte de memoria consistente dada, será implementada. Si la ejecución puede continuar antes de que las invalidaciones sean completadas, una eficiencia más alta puede ser obtenida, pero, sólo la definición simple de consistencia puede ser implementa, y la propiedad de consistencia secuencial puede ser violada. Hasta ahora se ha dirigido a la pregunta de cómo localizar el dueño actual de una página durante una falla.

Un mecanismo centralizado usa un administrador conocido para cada página, para localizar el propietario actual en ella. Todos los requerimientos para cambios de accesos y propiedad son enviados al propietario actual después del paso a través del nodo que es el administrador de la página. El administrador para una página entonces se da siempre cuando es el propietario de la página. Para determinar el propietario de la página, se usa el método que da solución al problema de la falla doble.

2.3.2. Algoritmo de falla doble

El algoritmo base sigue el protocolo previamente expuesto para determinar las fallas en lecturas y escrituras y usa algunos indicios para determinar quien es el propietario. Con este algoritmo, una localidad en una página inválida de DSM que le es leída y es escrita por dos referencias sucesivas obtendrá un par de fallas, una falla de lectura seguida de una falla de escritura. Nos referimos a este par de defectos (leer, escribir) para la misma página como par de falla doble. Si el algoritmo DSM pudiese predecir que la falla de lectura fuera rápidamente seguida por la falla de escritura, entonces la falla de escritura podría ser eliminada, tomando el equivalente de una falla de escritura cuando la falla de lectura ocurre. La información podría ser provista por la aplicación para señalar que una página a ser leída pronto estaría escrita (por un indicio para el sistema operativo). La predicción sin ayuda de la aplicación requeriría interpretación del flujo de instrucciones causando la falla, particularmente es difícil desde que los dos defectos pueden ser generados de instrucciones diferentes. En tal situación de falla doble, un nodo para transferir la página del nodo ocurrirá como el resultado de ambas fallas de lectura y escritura al usar el algoritmo base o el administrador centralizado perfeccionado; la diferencia en estos algoritmos

es que usan métodos diferentes determinar el dueño de una página dada: centralizado e indicio, respectivamente. Cuando los derechos de acceso para una página estén actualizados de sólo lectura para lectura y escritura (durante una falla de escritura), una transferencia innecesaria de la página puede ocurrir si la página transferida es la misma como la previa copia de sólo lectura mantenida en el nodo que falla.

Para dar un ejemplo de por que el algoritmo base realiza una transferencia de página durante todas las fallas de escritura, ya sea una copia de sólo lectura previa existe en el nodo o no, considere el siguiente escenario: Dos nodos han leído sólo las copias de una página que es poseída por un tercer nodo. Ambos nodos fallan en la escritura en esa página al mismo tiempo y envía los mensajes de petición al dueño de la página. Puede haber un camino larguísimo de estos nodos para el dueño actual de la página, y, asumiendo una red arbitraria entre los nodos (sólo se asume que todos los mensajes de un nodo para otro son recibidos en la orden en que fueron enviados), el retraso para cada uno de los mensajes es alcanzar al dueño que está indeterminado. Uno de los nodos recibirá el acceso de lectura y escritura (y la propiedad) para la primera parte de la página, invalidan otras copias de la página y modifican su propia copia. El otro nodo peticionario todavía puede esperar para obtener acceso de escritura para la página, pero, desconocido para el silencio de mensaje de petición sobresaliendo, el nodo peticionario ya no tiene una copia de actual de la página. Una transferencia de la página es completada en cada falla de escritura, considerando si el nodo peticionario tuvo uno en estado de marcha para actualizar la copia de sólo lectura de la página en el mismo tiempo el mensaje demanda acceso de escritura, este fue enviado desde que es posible que para la copia de sólo lectura sea deshabilitada durante el tiempo

la petición pues el acceso de escritura es prominente (conjuntamente con la propiedad).

Existen alternativas para el escenario de transferencia de doble página. Una alternativa es adaptar un esquema de *cache* en un bus simple de hardware en una implementación DSM. Un solo bus sirve como un excelente emisor y fabrica en serie todas las peticiones coherentemente. El propietario de un bloque del *cache* (página) puede estar resuelto por la emisión de un sólo un mensaje, con todo los nodos recibiendo el mensaje y decidiéndose si son del propietario. Una transferencia del bloque (página) puede ser eliminada al escribir a un bloque (página) de la cual hay una copia de sólo lectura, en la cual la aplicación tuvo la petición cuándo fue una escritura (una invalidación de todas las copias y la transferencia de posesión) aparece en la emisión y el bloque (página) no es deshabilitado antes de que la petición aparezca. Un esquema de emisión es mucho menos apropiado para una implementación DSM para un diseño coherente en un bus de hardware por las siguientes razones:

(i) Grandes cantidades de tiempo de procesador estarían obligadas a responder a todas las peticiones emitidas.
(ii) La transferencia de mensaje *Ethernet* no es tan confiable como un solo mensaje del bus.

Mientras los medios de comunicación no son esencialmente transmitidos, los diseños del hardware requieren transferencias en bloque dobles en lecturas y escrituras sucesivas a un bloque ya que no hay la serialización transmitida de la petición.

Otra alternativa potencial para el problema de la doble falla esta simplemente dado por el algoritmo de la patata caliente. Este algoritmo soluciona el problema tratando todas las fallas (dé lectura o de escritura) como si fueron una falla de escritura. La implicación de este algoritmo, sin embargo, es que exista una y sólo una copia de cada página en un el tiempo dado. Esta copia es almacenada y poseída por el nodo que ha dado lectura y de escritura, obtenido acceso a los derechos para la página. En caso de una falla doble, la falla inicial (lectura) logra obtener acceso de lectura y de escritura para la página. La segunda falla (escritura) no tiene lugar porque la página ha sido concedida con acceso de lectura y escritura después de que la primera falle. El algoritmo de la patata caliente, muestra cómo solucionar el problema de la falla doble si sé esta dispuesto a vivir con una sola copia DSM. La pregunta que entonces surge está en sí hay que dejar ir las ventajas de copias múltiples (sólo de lectura) DSM para solucionar el problema de la falla doble. El siguiente algoritmo indica que la transferencia doble de la página puede ser eliminada en múltiples copias DSM.

El algoritmo del administrador dinámico distribuido, llamado también algoritmo *Li*, llamado así por su diseñador, permite múltiples copias de solo lectura puedan existir. El algoritmo *Li* difiere del algoritmo base solo cuando la posesión de la página es transferida. Este algoritmo transfiere la posesión durante ambas fallas, de lectura y escritura, mientras que el algoritmo base solo transfiere la posesión sólo durante la falla de escritura. Cuando un nodo lee primero y posteriormente escribe en la misma página, la primera falla (lectura) obtendrá una copia y la posesión de la página. Cuando ocurre la falla de escritura inmediatamente después que se ejecuto la falla de lectura, una invalidación de otras páginas son necesarias para permitir accesos de lectura y escritura, ya que el nodo local es el propietario de la página.

El algoritmo *Li* es una mejora sobre el algoritmo base cuando se consideran pares de fallas dobles, en las cuales se elimina las transferencias necesarias cuando se presenta una falla de escritura seguida de una falla de lectura. Esta es una mejora sobre el algoritmo base al considerar pares dobles de fallas en que elimina la transferencia de la página necesaria para la falla de escritura cuando ocurre al poco tiempo de la falla de lectura. Desgraciadamente, no minimiza el número de transferencias de la página cuando la escritura no ocurre al poco tiempo de la lectura. Cuando una falla de escritura esté ocupada en un nodo que tiene una copia de sólo lectura de la página, pero, no es el propietario, la transferencia innecesaria de la página todavía puede resultar. Además, trasladar propiedad en una falla leída no es siempre la mejor decisión. Cuando un escritor simple / múltiples lectores comparte (productor - consumidor) el mismo lugar, la transferencia de la posesión es una opción pobre. En este caso, la propiedad es transferida para los lectores cuando obtienen copias de la página, mientras que, sería preferible para el solo para el proceso productor permaneciera propietario todo tiempo.

El algoritmo *Shrewd* elimina todas las transferencias innecesarias de la página con la ayuda de un número de secuencia por cada copia de una página. Cada vez que el acceso de lectura y de escritura es recién obtenido, el número de secuencia de la copia es el número de secuencia de la copia previa incrementada en uno. Cada vez que el acceso sólo de lectura para una página es obtenido, el número de secuencia de la copia nueva es el mismo número de secuencia de la vieja copia.

En cada falla de escritura en un nodo con una copia previamente de sólo lectura existente, el número de secuencia de la copia es enviado con la petición para el acceso lectura y de escritura. Al llegar al nodo que es propietario de la página, este compara el número de la secuencia de su copia de página con el número de secuencia en la petición entrante. Si son equivalentes, entonces el nodo peticionario puede obtener acceso de lectura y escritura para la página sin una transferencia de la página. La minimización de transferencia de la página está garantizada desde que el número de secuencia sea una indicación directa de la validez de la copia previamente existente de la página en el nodo de petición. En caso de una pareja doble de falla, la primera (lectura) falla gana una copia de la página, mientras la segunda (escritura) falla es probablemente para encontrarse con una transferencia de la página que es innecesaria desde que los números de secuencia son equivalentes.

Los algoritmos alternativos que proveen una página garantizada trasladan la minimización del algoritmo *Shrewd*, al transferir propiedad sólo en las fallas de escritura son también considerados. Una posibilidad particular es un algoritmo optimista en el cual está asumido que una transferencia de la página será innecesaria cuando un nodo demande acceso de escritura para una página de la que ya tiene una copia de sólo lectura. La transferencia de la propiedad ocurriría durante la falla de escritura. Si la copia estuviera todavía disponible (no invalidada) después de la petición de escritura regrese del propietario previo de la página para el nodo solicitante, entonces sería la copia actual de la página. Es posible que la copia en el nodo solicitante podría ser deshabilitada antes que la propiedad es transferida para el nodo solicitante.

En ese caso, el nodo solicitante rescataría la copia actualizada de la página del propietario previo. Este algoritmo optimista de este modo requeriría transferencia separada para la posesión y para la copia de la página en esa situación, mientras que en las transferencias de algoritmo *Shrewd* ambas se ejecutan al mismo tiempo. No se considera este algoritmo optimista más aún, porque sólo puede requerir más mensajes que el algoritmo *Shrewd*.

La diferencia entre el algoritmo *Shrewd* y algoritmo Base, es que el primero elimina la transferencia de la pagina cuando la falla del nodo tiene una copia actualizada de la pagina. La diferencia entre el algoritmo *Shrewd* y *Li*, es el método por el cual el problema la falla doble es solucionado, el primero lo soluciona por números de secuencia y en segundo lo soluciona por la transferencia de la propiedad.

3. IMPLEMENTACIONES DE HARDWARE

El problema de mantener la consistencia de un espacio virtual compartido global en sistemas DSM es muy similar para al problema de coherencia del *cache* en sistemas de multiproceso de memoria compartida. Además, las memorias *caches* privadas son inevitablemente implementadas en todos los sistemas de cómputo modernos, incluyendo a DSM, ya que es implementada como una memoria básica con técnicas que reducen latencia de la red. Por consiguiente, es muy natural que los avances del hardware DSM son, de hecho, protocolos extendidos de consistencia de *cache* que se adaptaron al ambiente DSM. Estas soluciones son predominantemente basadas en el diseño de directorio con potencial escalabilidad, lo cual es finalmente importante para la construcción de sistemas de gran escala. El principio *snooping* es ocasionalmente usado en los sistemas con un tipo apropiado de red (el bus, el anillo). Los avances del hardware DSM ofrecen ventajas significantes, en su estado actual completamente transparente para el usuario, proporcionando directamente soporte para el modelo compartido de programación de memoria. Evitando cualquier capa de software, estas soluciones pueden ser mucho más rápidas que los avances software DSM. Desde que las implementaciones del hardware típicamente usan unidades más pequeñas de uso compartido (por ejemplo, los bloques de *cache*), tienen éxito eliminando los efectos de *file sharing* y *trashing*. Esta estrategia es especialmente superior para aplicaciones que expresan un nivel de alto de uso compartido del granulo fino. Las ventajas anteriormente dichas obviamente son logradas a expensas de la complejidad aumentada del hardware.

3.1. La memoria como abstracción de red

3.1.1. Abstracción

Los sistemas de cómputo son sumamente complicados, y una técnica que ha probado ser efectiva en manejar esta complejidad es la modularización. Las actividades están descompuestas en una colección de módulos que cooperan entre sí, cada uno del cual provee una abstracción de los otros módulos a través de alguna interfaz. En el área de redes, la modularización está refinada a lo largo de líneas funcionales, llamadas capas (*layering*). Esta técnica intelectual permite que diversos comportamientos y complejidades de diferentes protocolos puedan estar descompuestas en pedazos, llamadas capas, donde cada uno de las cuales toma un paso hacia la traducción de datos de aplicación o realizar el proceso inverso de esa traducción. Estas capas son útiles para igualar un tipo de abstracción a otra. Entre otras cosas, las capas pueden:

(i) Hacer una operación del sistema de archivo hasta una petición de llamada de procedimiento remoto.

(ii) Una petición de llamada a un procedimiento remoto en un paquete independiente del servidor (esencialmente, convención de una serie de caracteres).

(iii) Convertir un paquete en uno o más mensajes.

(iv) Convertir un mensaje a uno o más paquetes de red, por medio de una interfaz multiplexada de entrada / salida (por ejemplo, una tarjeta del *ethernet*).

En cada capa, la abstracción cambia, con algún procesamiento de los datos en el paquete (en implementaciones actuales de protocolos, esta sobrecarga es un porcentaje alta del total de la sobrecarga total), copiándose o fragmentándose los datos en unidades más pequeñas, y encapsulando los datos en paquetes más grandes.

La cantidad sobre carga de las copias puede verse del hecho de que el protocolo actual esta mas restringido al ancho de banda de la memoria que al ancho de banda del procesador, dada la cantidad de copias que deben ser ejecutadas. Aún con procesadores rápidos, la comunicación de interprocesos sobre redes rápidas, a menudo ha logrado sólo una fracción pequeña del ancho de banda del cual la red es capaz.

Un problema con el *layering* es que puede esconder algún conjunto de suposiciones iniciales que puede ya no sea válida. Por ejemplo, en el caso del sistema de archivos distribuidos, mucho logros han minimizado la sobrecarga de la red; enfocando los esfuerzos en reservar copias locales de un archivo, o hacer la llamada de procedimiento remoto (RPC) más eficientes, o minimizar el largo de la ruta del código RPC para la interfaz del *ethernet*. La suposición básica de que una interfase de hardware basado en mensajes, es que es multiplexado y es accesado únicamente ya sea a nivel del sistema operativo, estos no han cambiado.

David Farber sugirió una diferencia fundamental para abordar el problema de IPC sobre redes rápidas.

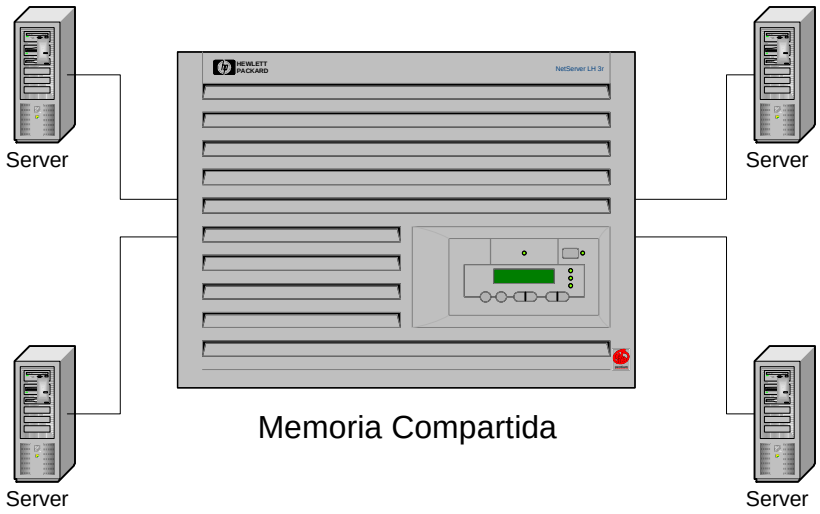
Él sostuvo la opinión que la esencia del problema fue la abstracción de la red, que es lo que modela la interfase de la red del hardware prevista para la computadora. Las interfaces actuales de la red se basan en la abstracción de entrada / salida, el cual requiere del procesador para dirigir la entrada y la salida de cada mensaje en la red. Ninguno de los datos puede fluir a través de la interfaz directamente en una aplicación, en cambio, el sistema operativo debe involucrarse en cada paso del proceso. La naturaleza de esta interfaz obliga a la computadora a realizar procesos costosos para transformar los datos en una forma aceptable para los cálculos. Farber sostuvo la opinión de que la red debería proveer la abstracción de la memoria de computadora, directamente para la aplicación, sin intervención del sistema operativo requerida para efectuar el movimiento de datos. Esta abstracción ha sido exitosamente aplicada para el almacenamiento secundario, tal como el disco, con la idea de la memoria virtual. La idea parecida madura para la exploración con el advenimiento de redes locales de alta velocidad y *switches* que facilitan realizar un intercambio rápido de paquetes en redes de área extendida (*WAN*).

3.1.2. Modelo de programación memnet

En el modelo de programación del sistema *memnet*, todos los procesadores tiene un acceso rápido y balanceado para una cantidad grande de memoria (figura 10). Mientras esta abstracción es conveniente para el programador, si el sistema fuera implementado en este modo sencillo, el tiempo de respuesta del sistema padecería de contención de memoria y de latencia de transmisión.

Mientras un servidor adicional es agregado a la red, el potencial requerido de ancho de banda de la memoria aumentará linealmente en el número de servidores existentes.

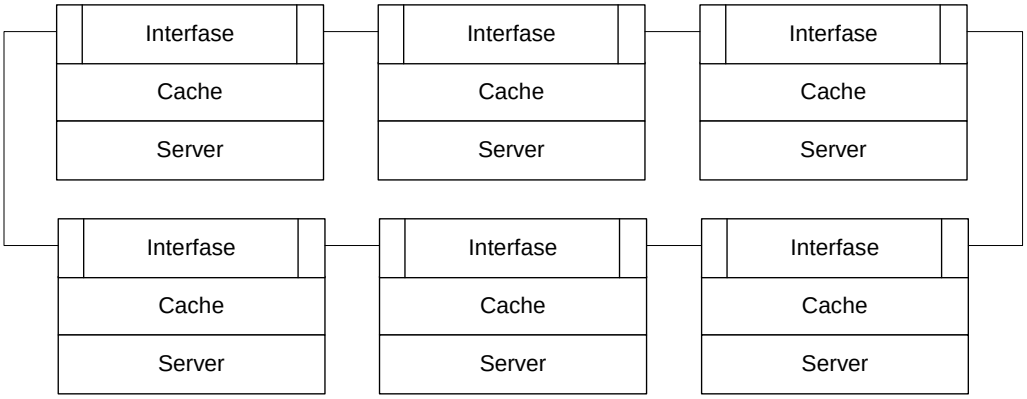
Figura 10. La perspectiva de memnet del programador



Una solución para la contención de la memoria que trabaja en la gran mayoría de los casos, es que cada servidor reserve en memoria *cache* la memoria que actualmente usa. Con una memoria *cache* del tamaño apropiado, el número de referencias que deben estar satisfechas por el recurso central puede acortarse dramáticamente. Reservar memoria *cache* es una tarea relativamente simple. Mantener copias múltiples de la misma localidad de memoria consistentemente respecto de otros, es una manera más difícil. La consistencia de los protocolos que acoplan fuertemente a los sistemas de multiprocesamiento, no tiene buena escalabilidad.

Los sistemas basados en bus sufren de bajó ancho de banda en la transmisión, mientras la longitud del bus es incrementada. Los sistemas *memnet* usan un sistema de interconexión en *token ring*, para evitar el descenso del ancho de banda proporcionalmente con la extensión del bus. La latencia de acceso de memoria, puede aumentar cuando la distribución física de servidores aumente, pero el ancho de banda de los medios de comunicación no disminuye, tal cual el caso con un sistema basado en bus que necesita permitir, control de acceso a las señales a propagar al final del bus y la devolución de los datos a su fuente posterior a cada *bit* (figura 11).

Figura 11. Vista macro del sistema memnet



En el sistema, cada nodo de cómputo tiene un dispositivo de *memnet* que es como un controlador de memoria en su bus local. Estos dispositivos están interconectados con una red *token ring* dedicada, paralela y de alta velocidad. La interfaz del nodo para del anillo es completamente una memoria con puertos duales. Cada nodo de cómputo establece referencias a esta memoria como la memoria real en su bus local, con el hardware de la interfaz del *memnet*, maneja la memoria *cache* con los datos recientemente usados. Las interfaces se comunican entre ellos mismos usando una red *token ring*.

Para conservar la contención del anillo delimitando la velocidad común de acceso a la memoria, la utilización de la red es reducida por el uso de memoria *cache* en cada nodo. Un canal distribuido multiplexado de interrupción es provisto para distribuir información de control.

La interfaz de anillo *memnet* usa hardware de máquinas de estado para manejar el tráfico del anillo. La consistencia de datos en cada interfaz, puede ser afectada sólo por el tráfico del anillo. El servidor no puede ganar acceso al anillo directamente. Todo tráfico del anillo es generado por la interfaz, no por el servidor local, y es interpretado por la interfaz con impacto mínimo en el bus local. El único impacto que el tráfico del anillo es que tiene la posibilidad de realizar algunas referencias a la memoria de *memnet*, existiendo retardas debido al acceso concurrente de la red y al acceso del nodo local para el dispositivo del *memnet*.

Un anillo es usado por dos razones. Primero, que el anillo provee un salto superior absoluto en el tiempo de acceso de la red. En segundo lugar, porque el paquete que atraviesa el anillo llega a cada interfaz en el mismo orden, las etiquetas usadas para manejar el protocolo de consistencia son sí mismas consistentes.

El anillo es un *token*, ya sea de inserción o modificación. La acción predeterminada de la interfaz es repetir lo que es recibido en su puerto de entrada en su puerto de salida. La latencia en cada interfaz es lo suficientemente larga para modificar la salida según el caso.

Cuándo una interfaz desea usar el anillo, espera a un *token*, ensambla y transmite un paquete del anillo, entonces suelta el *token*. Cada interfaz tiene una máquina de estado del hardware observando el puerto entrante. Como los paquetes llegan en la entrada, son examinados para efectos en la memoria local y requieren que esta interfaz puede satisfacer. Si el paquete entrante es un paquete que esta interfaz generó, entonces esta interfaz no repite este paquete para la salida, por consiguiente remueve el paquete del anillo. Si el paquete entrante es una petición que esta interfaz puede satisfacer, entonces el relleno del paquete entrante es reemplazado con datos suministrados por esa interfaz. Toda esta comunicación con el anillo tiene lugar sin cualquier intervención del servidor local.

El espacio de direcciones del ambiente *memnet*, está dividido en trozos (*chunks*) de 32 *bytes*. Este término es usado para describir esta división, para evitar algunas connotaciones entre los términos de página y marco de página. El espacio entero de la dirección del ambiente del *memnet* es localizado en el espacio local de la dirección de cada nodo. Cada interfaz de *memnet* tiene dos tablas de etiquetas que se usan para mantener información consistente para cada trozo en el espacio entero de *memnet* y manejar los recursos de la interfaz local. Estas etiquetas usadas en la primera parte de dos mensajes, incluyen a *VALID, EXCLUSIVE, RESERVED, CLEAN & INTERRPT*, y la localización de la memoria *cache*. Estas etiquetas son mantenidas en cada interfase para todo los trozos en el espacio completo de direcciones *memnet*. La segunda tabla de etiquetas esta asociada con los trozos del *cache* local, y contiene una entrada para cada localización en la cache. Estas etiquetas incluyen: *CLEAN, INUSE*, y contenedor de *chunk* (un puntero referenciado la localidad).

El protocolo de consistencia corre por las interfaces en un solo protocolo escritor / lector. La operación es similar a las caches *snooping*, sin embargo, hay dos diferencias principales. La primera, mientras las multicomputadoras usan las técnicas *snooping cache* tienen todos los procesadores en un solo bus, el esquema *memnet* permite sistemas con el rango físico de una Lan. La segunda, el *snooping* de los sistemas confían en alguna memoria principal separada, en la cual todos los caches tienen acceso. En el ambiente del memnet, la memoria principal es distribuida entre las interfaces.

El tráfico es mantenido a un mínimo en el anillo a través del uso de *chunks* de gran tamaño en el *cache* de cada interfaz. Con un *caching* efectivo, la mayoría de los accesos al espacio de direcciones de *memnet* se satisfacen los datos precargados en la interfaz local, y no generará tráfico en el anillo. Si el tráfico del anillo es minimizado, entonces el tiempo promedio que una interfaz deba esperar por un *token* para satisfacer un fallo del *cache* es también minimizado. El comportamiento del ambiente *memnet* bajo varios tamaños del *chunk* puede variar bajo diferentes razones de aciertos / fallas y del numero de nodos que han sido proyectados.

3.1.3. Consideraciones del sistema y recuperación de errores en memnet

El proceso *fork* de los sistemas de multiprocesamiento y las primitas de control de flujo, encuentran soporte en las operaciones de memoria atómica multiplexado porciones de *memnet*.

La extensión física de los sistemas *memnet*, sin embargo, añade un ancho de banda en el bus y la distribución física no es encontrada en solo bus, solo en sistemas de localización sencilla. El grado de riesgo de compartir recursos entre los nodos está más evidente con un sistema físicamente distribuido que con un solo sistema, pero las consideraciones de tamaño del sistema son similares. Las interfaces *memnet* están interconectadas con una red en anillo. Los errores en el anillo, en la mayoría de los casos, pueden ser detectados por la interfaz transmisora. Este protocolo de consistencia no es ídem potente, así es que si los paquetes son corrompidos, los datos en el trozo correspondiente no son recuperables probablemente. La confiabilidad del sistema de memoria es discutiblemente al mismo nivel como la confiabilidad del subsistema de disco. Existen esquemas para manipular los errores del bloque en disco, los esquemas para manipular los errores de memoria en la red serán desarrollados de igual forma. Para la aplicación que requiere un grado de confiabilidad que justifique el costo del sistema, fácilmente puede ser implementada en un sistema compatible *memnet*.

La conclusión que se alcanzan relacionando la confiabilidad de sistemas que no son absolutos, no importando cuanto, es importante recordar que la fiabilidad del anillo queda adecuadamente entendida. Las interfaces raramente pierden paquetes. De los paquetes perdidos, aun es más raro que estos son perdidos sin notificación. Dos de tres pasadas del protocolo de anillo estaba más cercano de ser idempotente podría ser proyectado, pero el desempeño en el caso general sufriría grandemente. La fiabilidad realista de los cálculos necesita ser realizada sobre la base de la experiencia, pero al parece ese este factor de fiabilidad, en cambio al factor

crudo de rendimiento, será el factor que limita el tamaño práctico de los sistemas de memoria simple. Esto no debe decir que el tamaño potencial del sistema virtual implementado en *memnet* es limitado, justo que el grado de riesgo compartido en un sistema distribuido debe ser contenido.

3.1.4. Modelo capnet

Una de las principales interrogantes que afronta el modelo memnet es la aplicabilidad de la abstracción del modelo en ambientes de redes extendidas (WAN). Hay un número alto de diferencias cruciales entre las redes locales (LAN) y extendidas (WAN), las cuales pueden afectar esta abstracción.

(i) Las redes extendidas tienen a generar latencias más altas (el tiempo que toma el envío de la información y el recibimiento de la respuesta).
(ii) Las redes extendidas no tienen una emisión segura.
(iii) El ancho de banda de las redes extendidas es limitado.

Los sistemas DSM residen fuertemente como una abstracción de los sistemas para la escritura de aplicaciones distribuidas que requieren compartir información.

El modelo *capnet* incluye desarrollos en cualquier sistema DSM, como por ejemplo, implementaciones del memoria consistente y en otras

ocasiones en desarrollos específicos de redes extendidas (WAN). En ambos casos, utilizan el uso protocolo de múltiples lectores y un solo escritor.

Los sistemas DSM sobre redes extendidas están construidos sobre redes punto a punto, en cuyo caso la transmisión es inherentemente costosa. Esto de debe a que el directorio de las páginas debe de ser controlado por algún administrador de páginas. Los requerimientos de las páginas al administrador de páginas, son quienes hacen las peticiones de la pagina al propietario. Un propietario es aquel quien en el servidor hace una modificación a la pagina, y posteriormente tiene la copia mas actualizada. El administrador de páginas designa el requerimiento al servidor como un nuevo propietario antes de enviarle un requerimiento de escritura al propietario actual. El propietario actual es el que en el servidor escribe el requerimiento que le ha sido enviado. Un propietario lee un requerimiento por el envío de la página y por la actualización de su juego de páginas. Para indicar que es el propietario de la página, se utiliza la información de invalidación cuando un requerimiento de escritura es solicitado al administrador de páginas. Este esquema requiere que se ejecute un viaje completo de un paquete entre el administrador de páginas y el propietario de la página antes que el requerimiento sea valido.

En las redes extendidas, la latencia es elevada y el número de requerimientos / respuestas pueden implicar tiempos de respuestas muy largos. Para elevar el nivel de rendimiento, se tiene que optimizar el protocolo utilizado o la arquitectura para eliminar estas latencias.

3.1.4.1. Esquema de localización de páginas de capnet

Este esquema integra la arquitectura de red en el sistema operativo. Esto se logra por medio del aumento de paquetes en los *switches*, los cuales contienen la información necesaria con la localización de las páginas. Por medio de una tabla de distribución en cada uno de los *switches* de la red, ella esta en capacidad de rutar el requerimiento de la página al propietario directamente. Bajo este esquema, un requerimiento de memoria es rutado de acuerdo a su dirección virtual, el cual es compartido por todos los participantes en el área de DSM. Cada *switch* de la red tiene su propia versión de la página en su tabla. Cada entrada de tabla de páginas, contiene un identificador saliente el cual indica quien es el propietario de la página (esta tabla es parecida a la tabla de rutas de una red). Los contenidos de estas tablas son actualizados por medio de trasferencias y actualizaciones que satisfacen un requerimiento de escritura. Por lo tanto, la localización actual exacta de una pagina es mantenida por la red. El requerimiento de un servidor por una pagina compartida, puede generar un requerimiento de página. Este requerimiento es enviado a la red, esta localiza la página, y la transfiere al solicitante. El proceso completo toma solo dos mensajes y no trasmite en toda la red el requerimiento. La información de las direcciones en la red se mantiene así misma. Esto es posible si hay mas de un requerimiento por una misma pagina. Estos requerimientos son atendidos en una cola del tipo primero en entrar, primero en salir (PEPS) sobre las páginas. Cuando la página es transferida, la cola de requerimientos es transferida conjuntamente.

Sí un requerimiento de una pagina es emitido cuando el requerimiento de la página esta en transmisión, el solicitante puede emitir el requerimiento al nuevo propietario o la dirección del *switch* del propietario previo y redireccionar su petición al nuevo propietario. Esto puede depender si la ruta del segundo requerimiento se intercepta con uno de los primeros requerimientos, y si es así, en la tabla de páginas del *switch* donde ocurre esta intersección ya haya sido alterada.

Colocando la página completa en la tabla de páginas del *switch*, puede provocar ciertos problemas, dado que el tamaño asignado por el espacio de direcciones puede ser invadido, dado el número de participantes en el *switch*. Las memorias rápidas, especialmente aquella del tipo *CAM* (*Content-Addressable Memory*) es muy costosa para este tipo de implementaciones, pero dada sus características, es la mejor. Se han implementado sistemas híbridos, los cuales preservan mucha de la reducción de la latencia del esquema original y lo hacen reduciendo los requerimientos de memoria. El sistema híbrido viene de la distribución de las entradas a través de todos los *switches* participantes. El grado en el cual se pueden localizar estas entradas en los *switches*, es que ellos contengan las localidades que frecuentemente utiliza y a la vez que sean exitosas. La localización de las páginas que son referenciadas infrecuentemente, pueden ser controladas por el administrador de páginas. Esto es posible gracias al particionamiento del espacio virtual de direcciones por algún otro esquema y la designación de servidores como administradores de paginas para esas particiones. Este esquema utiliza un administrador de paginas como parte del espacio virtual de direcciones.

El administrador de página de un grupo de páginas que expone la localidad geográfica puede ser migrado como cambios de referencia (las localidades), reduciendo la propagación promedio en comparación al uso de un administrado de páginas con localidades fijas.

3.2. Interfaz coherente escalable - escalabilidad a sistemas de alto rendimiento

Derivado de los estándar *ANSI/IEEE* 1596/1992, la interfase Coherente Escalable (*Scalable Coherent Interfase - SCI*), con un mínimo de procesadores, estos pueden compartir datos en su memoria *cache* de una forma coherente (su memoria *cache* es transparente para el software). Existen restricciones de escalabilidad en los protocolos, ya que los cuales se basan con protocolos de requerimiento / respuesta. La naturaleza lineal es la base del protocolo SCI, el cual limita su rendimiento cuando la actividad de la información es compartida por un gran número de procesadores.

3.2.1. Abstracción

3.2.1.1. Escalabilidad

Una arquitectura escalable permanece estable basándose en una serie de parámetros de diseño, tales como: el costo, rendimiento, distancia, inclusive, el tiempo. Infortunadamente, el término escalable se ha convertido en una palabra industrial, la cual esta incluida en literatura de mercadeo pero

representa un impacto mínimo en el diseño de un producto. Un sistema escalable elimina la necesidad de rediseñar los componentes para reducir o cambiar dramáticamente los requerimientos de un producto.

Gracias a la existencia del estándar *IEEE* 1596-1992, la interfaz coherente escalable posee un ente regularizador que da soporte y desarrolla sistemas de interconexión escalables.

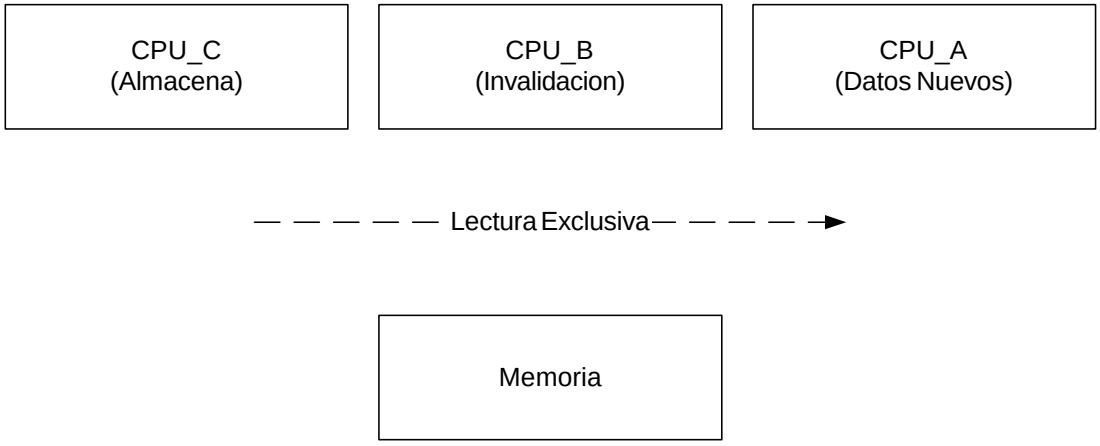
3.2.1.2. Requerimientos de las restricciones

En un diseño de cualquier sistema escalable, hay un objetivo el cual no puede ser cumplido al 100% de lo estimado. Dentro del estándar *SCI*, el objetivo primario es el alto rendimiento. En sistemas de alto rendimiento, los sistemas no poseen un solo procesador; estos están construidos por un gran número de procesadores compartiendo memoria distribuida por medio de memoria *cache* para ocultar su latencia. Por conveniencia del software, se asume que los procesadores están fuertemente acoplados: la memoria puede ser compartida y la memoria *cache* son transparentes al *software*. Los sistemas con multiprocesadores fuertemente acoplados mantienen el desarrollo de los protocolos coherentes en *cache* para sistemas con configuraciones masivas de procesamiento paralelo (*massively-parallel-processor, MPP*).

3.2.1.3. Protocolo eavesdrop

Los protocolos tradicionales de cache coherente se basan en *eavesdropping bus transactions*. Cuando un valor es leído de memoria, otro procesador *eavasdrop* (sí es necesario) provee la copia más actualizada de ese valor. Cuando un valor es almacenado en el *cache* del procesador CPU_C, la transacción exclusiva de lectura (una lectura con intento de modificación) es transmitida. Otros procesadores están requiriendo la invalidación de las copias antiguas y si la memoria tiene una copia antigua, puede requerir que se le provea la nueva información, como se ilustra en la figura 12.

Figura 12. Protocolo coherente eavesdrop

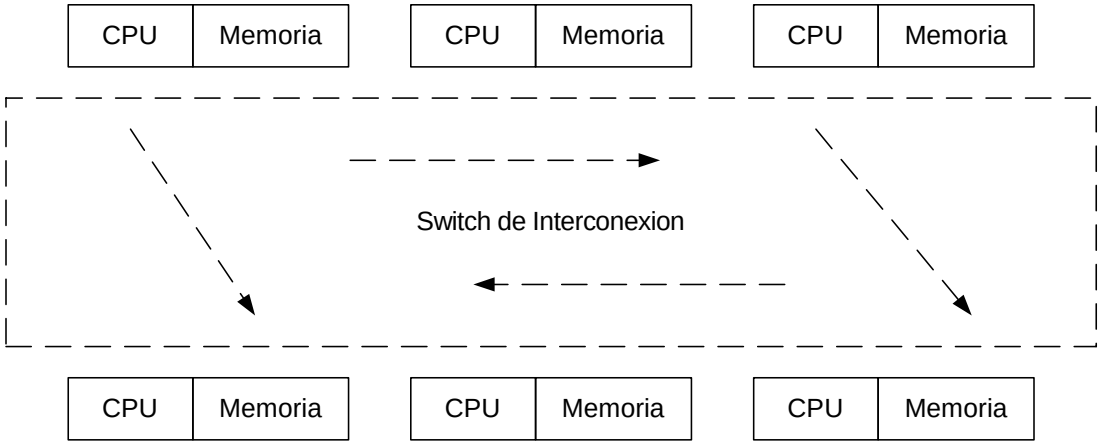


Los protocolos basados en *Eavesdrop* tienen un costo efectivo en buses de respuesta unificada, dado que la verificación de la coherencia puede ser ejecutada mientras la información es extraída de la memoria. El protocolo *eavesdrop* es difícil de implementar en buses de respuesta compartida de alta velocidad, donde razón de las transacciones es muy larga dado que no están restringidas a latencias de tiempo de acceso a memoria.

3.2.1.4. Protocolo de Directorio

Dado que cada procesador puede consumir una fracción significante de ancho de banda para lograr la interconexión, cualquier sistema MPP puede tener múltiples caminos activos concurrentes para el acceso de sus datos, como se ilustra en la figura 13.

Figura 13. Sistemas basados en switch



Las topologías totalmente conectas a través de un *cross-bar* son pobres, desde el punto de vista del costo, así que la interconexión se asumirá que es mas general si se basa en un *switch*. El protocolo *eavesdrop* no es posible dentro de un sistema basado en un *switch*, ya que las transacciones activas concurrentes no pueden ser observadas por todos.

Los protocolos coherentes de directorio central proveen una línea de memoria con una etiqueta, donde cada etiqueta identifica a todos los procesadores que tienen una copia en su memoria *cache*. La memoria y la memoria *cache* son típicamente del mismo tamaño, entre 32 y 256 *bytes*.

Cuando los datos son escritos, la memoria es responsable por la actualización o invalidación de las copias previamente compartidas.

El protocolo de directorio central tiene limitantes de escalabilidad, porque el tamaño total del sistema esta limitando al tamaño de la línea de la etiqueta. Algunos sistemas proponen el uso emisión de transacciones con sus directorios sobrecargados. Otros sistemas proponen el uso de software residente en memoria para manejar esta sobrecarga, eliminando la necesidad de soportar transacciones especiales para su manejo.

El esquema de directorio central efectúa las transacciones de lectura y escritura de forma serial. El esquema *SCI* evita el flujo de almacenamiento de las etiquetas y los cuellos de botella por la distribución de directorios: las etiquetas de cada memoria contienen el estado y un apuntador al primer procesador, y las etiquetas de la memoria *cache* contienen estados y apuntadores a otros procesadores.

3.2.2. Restricciones del juego de transacciones

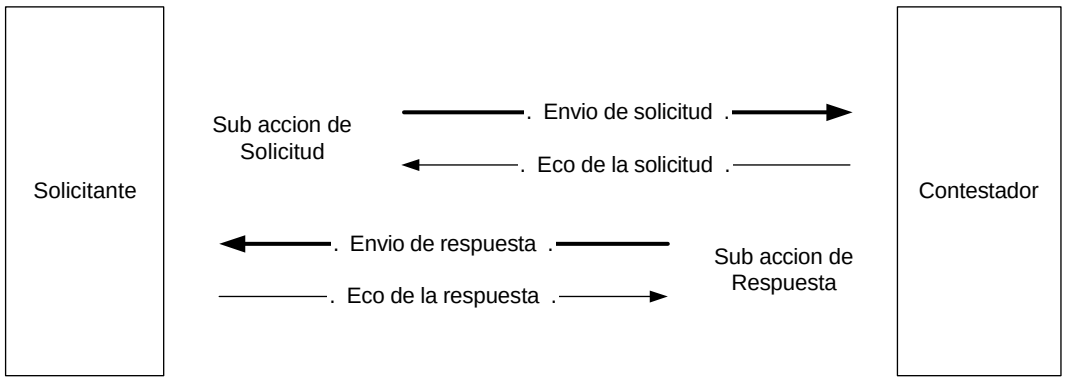
3.2.2.1. Transacciones básicas SCI

Para simplificar el diseño de *switches* de alto rendimiento, la comunicación entre los componentes SCI, es usando un protocolo simple de transacciones de requerimiento - respuesta. Una transacción consiste de un requerimiento y de una respuesta sub - acción. El requerimiento sub - acción

transfiere una dirección y un comando; la respuesta sub-acción retorna un estado. Para transacciones de escritura, los datos son incluidos dentro del paquete enviado. Para transacciones de lectura, los datos son incluidos dentro del paquete de respuesta. Para transacciones compuestas, los datos son incluidos dentro de sus respectivos paquetes enviados y respondidos.

Cada sub-acción consiste de un envío de un paquete y de un eco. La información transferida dentro del paquete (pudiendo ser comandos, datos y estados), el control del flujo de la información es retornado dentro del paquete de eco, como se observa en la figura 14.

Figura 14. Transacciones solicitud / respuesta



La mayoría de transacciones no-coherentes y todas las transacciones coherentes tienen estos componentes. Se evita el uso de transacciones coherentes especiales, ya que su implementaron complicarían el diseño de la interconexión, reduciendo el rendimiento.

3.2.2.2. Transacciones coherentes SCI

Para transacciones de lectura y escritura, el protocolo coherente adiciona unos cuantos bits en el paquete enviado y otros cuantos bits en el paquete que es devuelto. Dado que existe una restricción de alineación que arregla el tamaño del paquete enviado, el protocolo no presenta impacto alguno por el tamaño del mismo.

Sin embargo, un requerimiento extendido es necesario para soportar las transacciones *cache* a *cache*, dado que hay dos direcciones están implicadas (la dirección de la memoria de los datos y la dirección enrutada en el cache). La información adicional es contenida dentro una extensión del encabezado del paquete normal. Los bits del protocolo de coherencia son típicamente ignorados en la interconexión y no tienen impacto en el diseño de los *switches SCI*. Así, los protocolos de coherencia son escalables en el sentido que los componentes del mismo *switch* pueden ser usados por sistemas SCI coherentes y no-coherentes.

3.2.3. Estructuras de Directorio Distribuidas

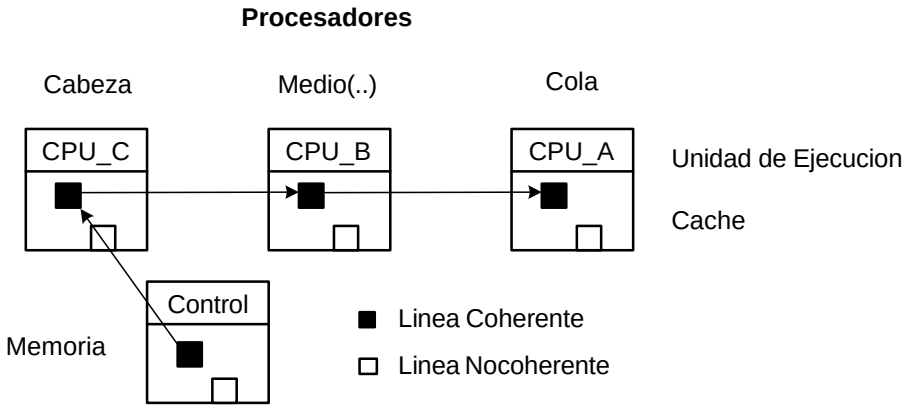
3.2.3.1. Listas Compartidas Lineales

Para soportar un número de procesadores arbitrariamente, el protocolo coherente *SCI* se basa en directorios distribuidos. Distribuyendo los directorios entre todas las memoria *cache* de los procesadores, la capacidad

potencial del directorio crece como un *cache* adicional más, y las actualizaciones al directorio no necesitan ser serializadas en la controladora de memoria. La base del protocolo coherente esta basado en listas lineales; un protocolo coherente extendido provee soporte compatible para árboles binarios. Las listas lineales son escalables, en el sentido que miles de procesadores pueden compartir la información en modo lectura. Así, las instrucciones y la información en modo lectura puede eficientemente ser compartida por un gran número de procesadores.

La memoria provee etiquetas, por lo que cada línea de la memoria tiene asociada un estado e identifica la memoria *cache* en la raíz de la lista compartida (sí la información es compartida). La memoria *cache* también tiene una etiqueta, el cual identifica el elemento previo y siguiente dentro de la lista compartida, como se ilustra en la figura 15. La estructura doblemente encadenada simplifica las eliminaciones en la lista compartida (la cual es causada por los reemplazos en la memoria *cache*).

Figura 15. Listas Compartidas Lineales



Sin embargo, un componente simple *SCI* puede contener procesador(es) y memoria. Desde la perspectiva de los protocolos coherentes, no existe una separación lógica de esos componentes. Nótese que cada direccionamiento de memoria, puede tener listas compartidas distintas, y esas listas compartidas cambian dinámicamente dependiendo del comportamiento de los estados del procesador y su carga.

3.2.3.2. Actualizaciones de las listas compartidas

Las actualizaciones de la lista compartida involucran nuevas adiciones (nuevas entradas son siempre incluidas por la raíz de la lista), supresiones (causadas por el reemplazo del encadenamiento en la lista) y eliminaciones (la lista compartida es colapsada a un solo elemento). Las adiciones ocurren cuando los datos son compartidos por un nuevo procesador, y las eliminaciones ocurren cuando los datos son modificados. Se presenta un caso especial de pares compartidos, donde todas las modificaciones son ejecutadas por la raíz de la lista compartida, que es donde también residen las entradas al resto de listas compartidas.

Los protocolos de actualización para la lista compartida, son similares a las estructuras de software doblemente encadenadas, sujetas a las siguientes restricciones:

(i) Actualizaciones concurrentes: Todas las adiciones, supresiones y eliminaciones son ejecutadas concurrentemente - sujetas a reglas de precedencia y listas

de integridad que utilizan comparaciones (no se utilizan semáforos)

(ii) Fallas recuperables: Las secuencias de actualización soporta recuperación después de un número arbitrario de transmisión de fallas - exclusivamente los datos son copiados antes que sean eliminados.

(iii) Envió de progreso: El progreso de los envíos es *ensured* (cada memoria *cache* eventualmente tiene una copia) - una nueva adición a la memoria *cache* es adicionada a la raíz y es eliminada en la cola de acuerdo a la precedencia.

3.2.3.3. Pares compartidos

Las estructuras de listas lineales son eficientes si los datos compartidos son escritos infrecuentemente o sí las escrituras de datos frecuentemente son procesados por el mínimo de procesados. Como una opción de los pares compartidos, es intentar de dar soporte a estos pares compartidos. La implementación de estos pares es de dar soporte a las aplicaciones, donde precisamente dos memorias *cache* están compartiendo datos compartidos. Los pares compartidos permiten a todos los procesadores hacer transparencias entre memorias *cache*, ejecutando una carga de los datos más recientemente modificados. Esto reduce las latencias de los accesos y elimina potenciales cuellos de botella en el dispositivo

3.2.3.4. Extensiones logarítmicas

Las bases del protocolo coherente escalable *SCI* son limitadas, en lo que adiciones y eliminaciones en la lista compartida, las cuales se efectúan de forma serializada. Este tipo de operaciones se efectúan perfectamente en un número pequeño de procesadores donde se comparte la información sin problema.

Sin embargo, una lista lineal es una solución efectiva de bajo costo para la mayoría de las aplicaciones, los objetivos de la escalabilidad son mantenidos como una solución arbitraria en sistemas de múltiples procesadores. Como consecuencia, se han desarrollado extensiones compatibles que dan valor agregado al protocolo SCI. Estas extensiones utilizan y combinan la utilización de árboles binarios para reducir la latencia, llevándola de una forma línea a una forma logarítmica. Las estructuras de listas compartidas son creadas dinámicamente y destruidas inmediatamente después de haber sido creadas.

3.2.3.5. Creación de la lista compartida

En algunos sistemas fuertemente acoplados de memoria compartida, los datos son accesados concurrentemente por un gran número de procesadores, típicamente cuando un punto de sincronización ha sido alcanzado. Sí estos requerimientos son serializados en una controladora de memoria, las latencias de tiempo de acceso presentarían un acceso lineal.

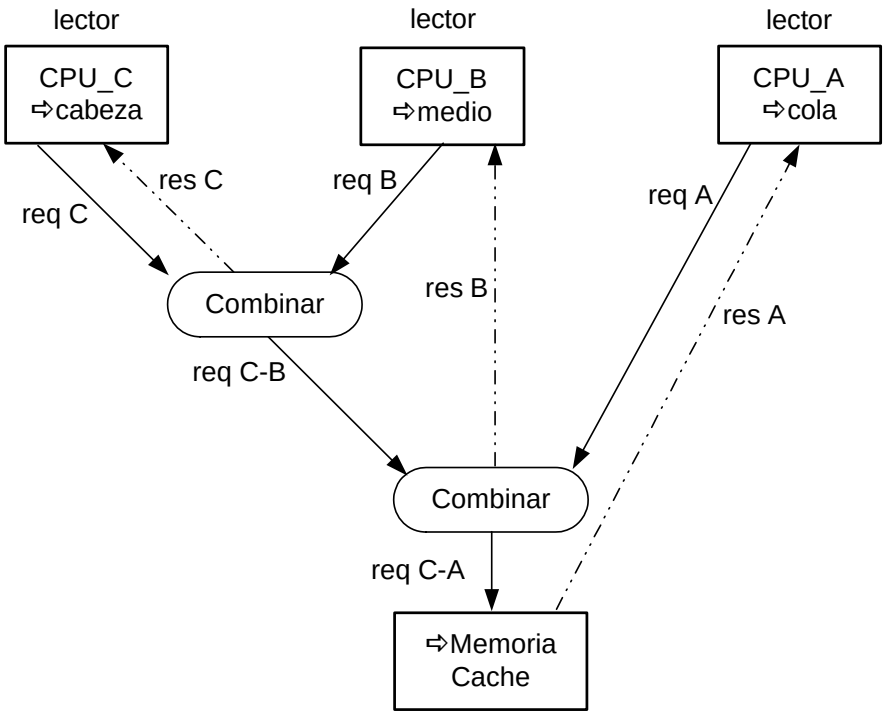
Para lograr latencias logarítmicas, los requerimientos de lectura deberían ser combinados cuando se ejecuta la interconexión (generando la opción de ser optativos). Se asume que este comportamiento solo ocurre cuando el sistema esta corriendo en condiciones de carga pesada, que es cuando se presentan retardos de tiempo comparados con los requerimientos de los paquetes de las transacciones cuando la cola del sistema esta esperando por un cambio de contexto.

Por ejemplo, consideremos tres requerimientos a la misma dirección de memoria, las cuales son combinadas durante su transito. Dos requerimientos (req B & req C) se pueden combinar en uno solo generando un requerimiento modificado (req B-C) emitiendo una respuesta inmediata (res C). De una forma similar, los requerimientos combinados (req B-C) puede ser combinado con otro requerimiento (req A) como se ilustra en la figura 16.

La utilización de esta combinación coherente para reducir la latencia al momento de la creación de la lista, retornara una lista combinada de apuntadores, la cual seria más simple para aprovechar los estados en la fase de interconexión, modificando la respuesta cuando estas son retornadas.

Dado que los nodos básicos del SCI son personalizados, en los cuales se reciben respuestas sin información, ninguna complejidad es introducida en la naturaleza del mecanismo de combinación.

Figura 16. Combinación con interconexión



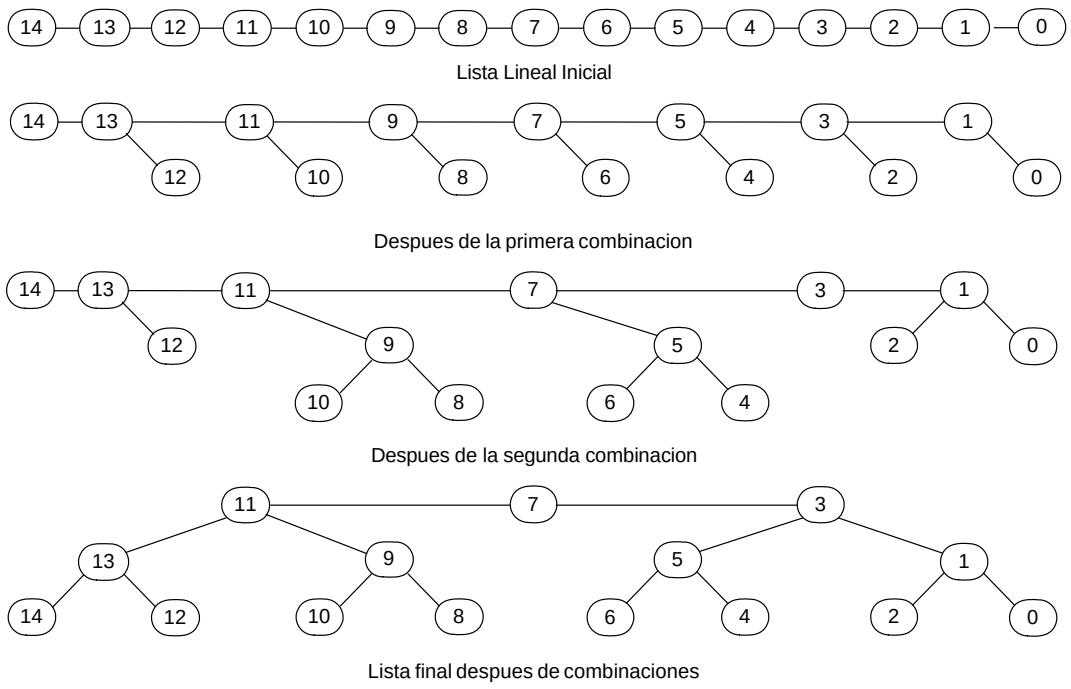
3.2.3.6. Conversión de listas compartidas en árboles compartidos

Después de que la lista lineal compartida esta disponible, se ejecutan los procesos de cambio que transforman esta lista lineal en una estructura de árbol binario. Las latencias para ejecutar esta transformación, tardan un tiempo promedio de O(log(n)), donde n es el numero de elementos en la lista compartida, siendo este un limite pragmático en la mayoría de protocolos escalables.

El proceso de cambio es distribuido y ejecutado concurrentemente a través de las memorias *cache* compartidas. La distribución de los datos es también ejecutada durante este proceso. Múltiples pasos están relacionados, uno para cada nivel en el árbol binario que es creado, siendo algunos de estos traslapados en el tiempo.

Bajo condiciones ideales, el primer paso en el proceso de transformación de la lista es la creación del primer nivel del árbol, el segundo paso transforma al anterior en un árbol con dos niveles, y así sucesivamente hasta que se llega a un estado estable, como se muestra en la figura 17.

Figura 17. Estructuras de árboles binarios



Cuando llega a un estado estable, la lista lineal se ha convertido en una lista de subárboles, donde la altura del árbol esta localizado cerca de la raíz. Las adiciones increméntales son soportadas, se pueden combinar árboles de igual altura en un solo árbol con el total de la altura de los árboles participantes hasta que se llegue de nuevo a un estado estable. Se necesita tres operaciones para realizar esta combinación.

Durante el proceso de combinación, un algoritmo de distribución selecciona cual será el par de subárboles que participaran en la combinación. Para eliminar los conflictos de actualización, las entradas 2-y-1 no pueden ser combinadas si la combinación 1-y-0 ya ha sido combinada. Para generar árboles balanceados, las entradas 3-y-2 deben ser combinadas y la combinación 1-y-0 ya ha sido combinada. Nótese que los algoritmos de *software* usuales para la creación de árboles balanceados no pueden ser utilizados, porque el proceso debe distribuir y solo localizar la información cuando es necesitada. El objetivo es alternar las etiquetas de los sub - árboles con etiquetas 0-y-1, permitiendo la combinación de este con otro sub - árbol. La asignación de las diferentes etiquetas es difícil, dado que los elementos de entrada son casi idénticos y las posiciones en la lista son desconocidas. Una asignación de una etiqueta doble soluciona la generación de etiquetas, usando pseudo-números aleatorios, generando así identificadores de 16 bits, los cuales son utilizados normalmente para el ruteo de los paquetes que usa el protocolo *SCI*.

3.2.3.7. Utilización de los árboles compartidos

Una vez formado, el árbol compartido puede ser usado para invalidar rápidamente otras entradas del mismo, y para la actualización distribuida de los datos eficientemente. Las instrucciones *Store* son ejecutadas en el procesador ubicado en la cabeza del árbol compartido, invalidando así otros elementos en los árboles compartidos que dependan de este. Sí no hay una cabeza en el árbol compartido, el procesador deja el árbol y se convierte en la cabeza de un nuevo árbol después de que la instrucción es ejecutada.

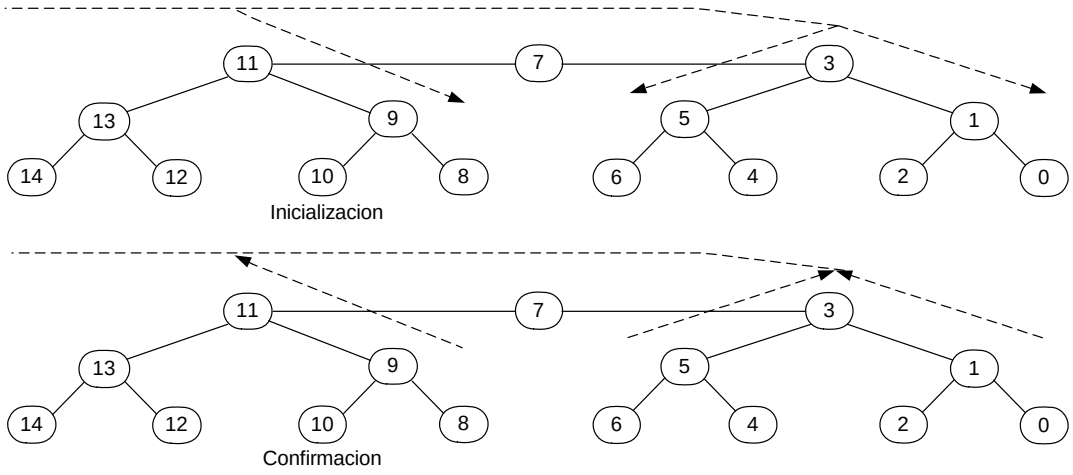
Como una optimización del rendimiento, una instrucción especial *StoreInto* también puede ser utilizada. Esta instrucción también actualiza otras copias y deja las estructuras del árbol compartido intactas. Sí no hay una cabeza en el árbol, el procesador actualiza la cabeza del árbol compartido y los datos son distribuidos a otras entradas de otros árboles compartidos.

Las actualizaciones de los datos y las invalidaciones, son ejecutadas de forma similar. La transacción inicial distribuye la información a otras entradas en otros árboles compartidos; una transacción de confirmación es retornada cuando la transacción inicial ha sido procesada por los niveles inferiores del árbol, como es ilustrado en la figura 18.

Para las instrucciones *Store*, el árbol compartido es eliminado y las entradas son eliminadas a sí mismas durante la fase de confirmación. Para

las instrucciones *StoreInto*, los datos son distribuidos en la fase de inicialización y el árbol compartido permanece intacto.

Figura 18. Actualización y confirmación



3.2.3.8. Evasión de puntos muertos

Las transacciones de inicialización y confirmación son especiales, ya que son enviadas a través de los árboles compartidos. El procesamiento de requerimientos entrantes genera una cola de requerimientos salientes. En una cola de requerimientos de salida saturada conlleva a que el procesamiento de requerimientos entrantes generen puntos muertos.

Para evadir estos, el tamaño de la cola de requerimientos saliente debe ser lo suficientemente grande para contener un gran número de entradas dentro del nodo. Las implementaciones no suponen que puedan duplicar el espacio utilizado dado el tamaño de las colas. En lugar de esto, cada nodo

almacena su propia implementación de la cola de requerimientos salientes como una lista encadenada de elementos de entrada.

3.2.3.9. Ordenamientos débiles

Los protocolos coherentes permiten que la raíz de la lista compartida proceda con las operaciones de escritura antes que el resto de la lista compartida haya sido invalidada. Se almacenan en la misma línea de memoria *cache* por otro procesador que requiere cambios en la estructura de la lista compartida y posteriormente son diferidos hasta que la fase de confirmación se completa. Esto implica que el protocolo *SCI* pueda explotar completamente el potencial del paralelismo de los modelos de memoria débiles. Se han eliminado detalles del modelo de consistencia de memoria del procesador del modelo de coherencia del protocolo *SCI*. Dependiendo del modelo de consistencia de memoria del procesador, algunas o todas las instrucciones de almacenamiento / recuperación pueden ser evadidas mientras las invalidaciones están siendo ejecutadas.

3.3. Arquitectura de memoria cache

3.3.1. Abstracción

Los sistemas multiprocesador proveen una vista de la memoria compartida al programador al momento de estar implementando alguna aplicación, utilizando memoria compartida. A continuación, se describe una arquitectura utilizando memoria *cache* para reducir la latencia y carga de la red. Partiendo del supuesto de que todos los sistemas de memoria residen en memoria *cache*, un mínimo de accesos de red es necesitado.

3.3.1.1. Bus simple

Los sistemas de memoria compartida basados en un bus simple tienen por ejemplo 10 procesadores, cada uno con una memoria *cache* local y típicamente sufriendo saturación en el bus. El protocolo coherencia de *cache* explora el tráfico en el bus común y previene inconsistencias en los contenidos de la memoria *cache*. Dado que provee un tiempo de acceso uniforme para la memoria compartida enteramente, es llamada arquitectura uniforme de memoria (*Uniform Memory Architecture - UMA*). La contención en la memoria común y del bus común, limita la escalabilidad de estos sistemas.

3.3.1.2. Bus distribuido

Cada nodo del procesador contiene una porción de la memoria compartida, así es que los tiempos de acceso para partes diferentes de la dirección compartida espacie pueden variar. Este tipo de arquitectura se llama arquitectura no uniforme de memoria (*Non Uniform Memory Architecture - NUMA*). Estos sistemas a menudo tienen redes fuera de un solo bus, y el retraso de la red puede cambiar para nodos diferentes. Los sistemas *NUMA's* iniciales no presentaban memoria caches coherentes y el problema de mantenimiento era delegado en el programador.

3.3.1.3. Memoria solamente de cache

En la arquitectura de solamente cache, la organización de la memoria es similar a *NUMA*, debido a que en cada procesador contiene una porción del espacio de direcciones. Sin embargo, la partición de los datos entre las memorias no tiene que ser estática, desde que todas las memorias distribuidas son organizadas como una memoria *cache* grande (segundo nivel). La tarea de tal memoria es dual. Además de ser una memoria *cache* grande para el procesador, también puede contener algunos datos del espacio compartido de la dirección que el procesador nunca tiene ganar acceso - en otras palabras - es una memoria *cache* y una parte virtual de la memoria compartida.

A la forma intermedia de memoria, se le conoce como memoria de atracción. Un protocolo de coherencia atrae los datos usados por un

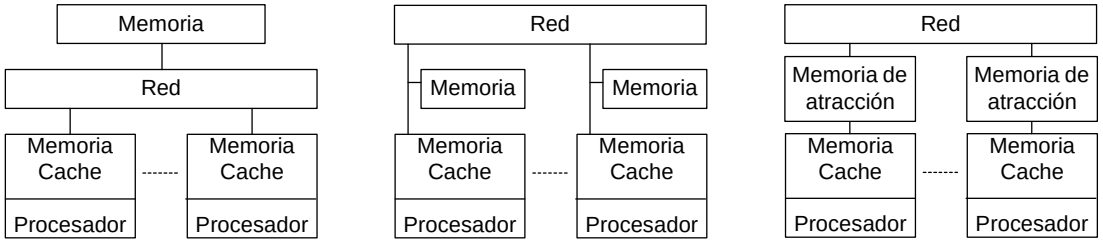
procesador para su memoria de atracción. Comparable para la línea de la memoria *cache*, la unidad coherente cambia de un lado hacia el otro por el protocolo es llamado *ítem*. En una referencia de memoria, una dirección virtual es traducida a un *ítem* identificador. El espacio del *ítem* identificador es lógicamente el mismo con el espacio físico de direcciones de máquinas típicas, pero no hay representación permanente de correspondencia entre un *ítem* identificador y una posición de memoria física. En lugar de eso, el ítem identificador corresponde a una localización en una memoria de atracción, de quién es la etiqueta que hace juego con él.

Un sistema *COMA* provee un modelo de programación idéntico a las arquitecturas de memoria compartida, pero no requiere distribución estática de ejecución ni de uso de memoria para poder funcionar eficazmente. Poner a funcionar un programa optimizado *NUMA* en una sistema *COMA*, resulta en un comportamiento análogo a *NUMA*, ya que los espacios de trabajo de procesadores diferentes, migran para sus memorias de atracción. Sin embargo, una versión *UMA* del mismo programa tendría un comportamiento similar, porque los datos son atraídos para el procesador utilizado, a pesar de tener las direcciones. Un sistema *COMA* también se adapta para ejecutar adecuadamente los programas con una planificación más dinámica o semi - dinámica. El espacio de trabajo migra, según su consumo durante todo el cálculo. Los programas pueden ser optimizados para un sistema *COMA*, para tener en cuenta esta propiedad para el registrar una mejor localidad.

Un sistema *COMA* permite uso dinámico de datos sin duplicar mucha memoria, comparada con una arquitectura en la cual un dato en memoria

reservada, también ocupa espacio en la memoria compartida. Para evitar aumentar el costo de memoria, las memorias de atracción deberían ser implementadas con componentes ordinarios de memoria. Por consiguiente, se observa que el método de *COMA* es un método de segundo nivel, o más alto. El tiempo de acceso de la memoria de atracción de los sistemas *COMA*, es comparable con la memoria de un sistema *NUMA* con memoria *cache* coherente. En la figura 19, se compara la arquitectura *COMA* con otras arquitecturas de memoria compartida.

Figura 19. Comparación método COMA



3.3.1.4. Máquina de difusión de datos

Esta nueva arquitectura, llamada Maquina de Difusión de Datos (*Data Difusión Machine – DDM*) tiene como base una estructura de red jerárquica. Con esta nueva arquitectura, se introducen nuevos diseños, los cuales describen a maquinas pequeñas y sus protocolos.

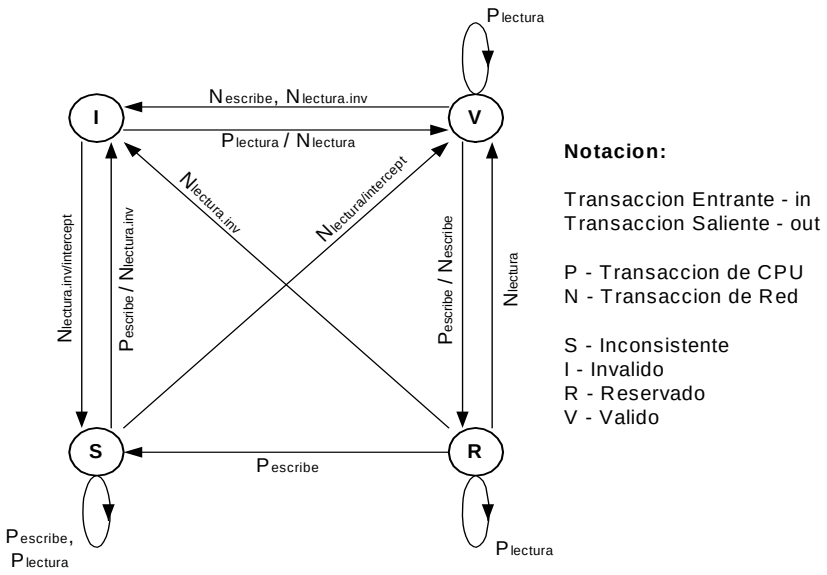
3.3.2. Estrategias de coherencia de cache

El problema de manutención de coherencia entre los datos compartidos de lectura / escritura por diferentes memorias *cache*, ha sido el obstáculo principal de este modelo. Ya sea el software o el hardware pueden mantener coherencia. Se asume que la coherencia del hardware es necesaria en un sistema *COMA* para mantener la eficiencia, dado que el ítem debe ser pequeño para prevenir degradación de rendimiento por el uso compartido falso (dos procesadores accedan a partes diferentes del mismo *ítem* y tienen conflicto uno con el otro, aunque entre ambos no compartan ningún *ítem*).

Los esquemas de base en *hardware* mantienen coherencia sin *software* envolvente y pueden ser implementados más eficazmente. Los ejemplos de protocolos bajo en hardware son: protocolo de *cache snooping* y protocolos bajo en directorio. Los protocolos espías tienen una implementación distribuida. Cada cache es responsable de espiar el tráfico en el bus y tomar acciones para evitar una incoherencia. Un ejemplo ese protocolo es el protocolo introducido por Goodman, llamado protocolo de escritura única (write-once protocol). Como se muestra en la figura 20, en este protocolo

cada línea de la memoria cache puede estar en uno de cuatro estados: invalido, valido, reservado o sucio. Varias memorias cache pueden tener el mismo estado valido al mismo tiempo y puede ser accesada localmente. Cuando se escribe a la memoria cache cuando esta valido, el estado cambia a reservado, y una escritura es enviada en el bus común a la memoria común. Todas las memorias caches espían los estados validos e invalidan sus contenidos. A este punto, solo existe una copia en cache con la información con el valor mas reciente. La memoria común ahora también contiene el nuevo valor. Si la memoria cache tiene su valor reservado, puede ejecutar una escritura local sin una transacción en el bus local. En este caso, este valor difiere de las otras memorias y en este estado cambia a inconsistente. Cualquier requerimiento de lectura de otras memorias cache a la memoria en estado inconsistente debe ser interceptado para que se le provea un nuevo valor.

Figura 20. Ejemplo de un protocolo de escritura única



Las memorias *cache* con protocolo espía se basan principalmente en la difusión y no satisfacen por general las interconexiones de redes; una difusión sin restricción puede reducir dramáticamente el ancho de banda, por esa misma razón las ventajas generales de la red. En lugar de eso, los diseños basados en directorios envían directamente entre los nodos. Un requerimiento de lectura es enviado a la memoria principal sin espiar. La memoria principal sabe si la memoria *cache* esta en la memoria *cache* y en cual memoria *cache* adicional, y como han sido modificados. Si esta ha sido modificada, el requerimiento de lectura es pasado al *cache* con una copia, el cual provee una copia del cache que efectúa el requerimiento. En la escritura de una memoria *cache* compartida, un requerimiento de escritura es enviado a la memoria principal causando un mensaje de invalidación a todos los caches que posean una copia. El *cache* responde con una confirmación. Para mantener una consistencia secuencial, todas las confirmaciones deben ser recibidas antes de que la escritura sea ejecutada.

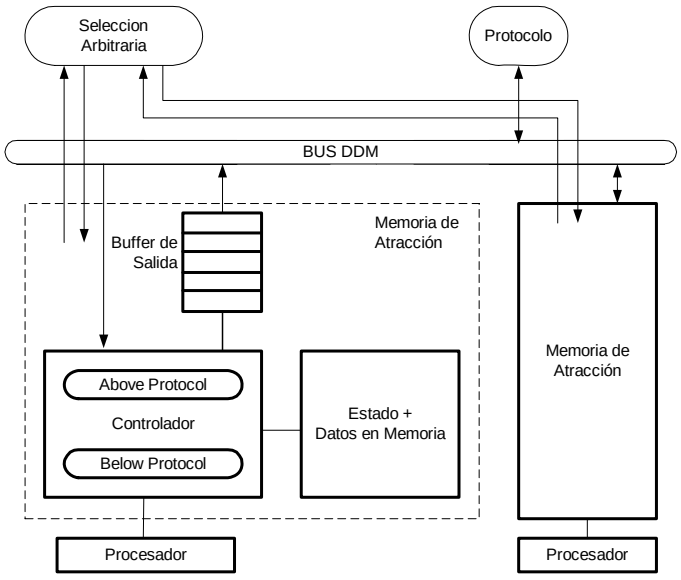
El protocolo de coherencia de *cache* para los sistemas *COMA*, puede adoptar técnicas usadas en otros protocolos de coherencia de *cache* y extender su funcionalidad para encontrar los datos dentro de una memoria *cache* perdida generando el manejo del reemplazo. Un protocolo basado en un directorio, puede tener una parte de la información del directorio, el directorio *home*, estáticamente distribuido en un modo *NUMA*, mientras que los datos son movidos libremente. La recuperación de los datos en una lectura perdida requiere un acceso indirecto extra al directorio *home* donde encontrara el *ítem* concurrentemente a la posición de memoria donde este último resida.

El tiempo de acceso incluye, el indireccionamiento extra, el cual identifica la lectura requerida a la memoria inconsistente que no esta en el *home* del sistema *NUMA*. El *home* del diseño de directorio puede estar seguro de que la copia mas reciente del elemento en cuestión no este perdido.

3.3.3. Arquitectura mínima

Un simple bus único conecta la memoria de atracción de una maquina de difusión de datos (*DDM*). La distribución y coherencia de la información a través de las memorias de atracción son controladas por el protocolo espía llamado *memory above*, y la interfase entre el procesador y la memoria de atracción esta definida por el protocolo *memory below*. El protocolo visualiza las memorias *cache*, como *items*, como una unidad. La memoria de atracción almacena un campo pequeño con el estado de cada *ítem*. La figura 21 muestra la arquitectura de un nodo utilizando un bus único *DDM*.

Figura 21. La arquitectura de bus único DDM



El diseño DDM utiliza un bus de transacciones divididas asincrónicas, el bus es liberado de dos formas: entre cada requerimiento de transacción y su respuesta, y entre cada requerimiento de lectura y su respuesta. El retardo entre cada requerimiento y su respuesta puede ser de una longitud arbitraria, logrando un gran número de requerimientos. Las transacciones de respuesta pueden aparecer eventualmente en el bus como una transacción diferente. Diferente al comportamiento de otros tipos de buses, el bus DDM tiene un mecanismo de selección más seguro tomando el nodo que servirá el requerimiento. Esto garantiza que cada transacción no producirá más de una nueva transacción en el bus, evitando así la generación de estancamientos.

3.3.3.1. Protocolo de bus simple DDM

Este protocolo es similar al protocolo de espía de *cache*, con la diferencia de que se limita la difusión a subsistemas más pequeños y adicionando soporte a los reemplazos. La parte de coherencia de escritura del protocolo es del tipo de invalidación de escritura: Para mantener la información coherente, todas las copias de un ítem con excepción de la que se esta actualizando, son borradas cuando se ejecuta la escritura. En un sistema *COMA* con un tamaño de ítem pequeño, la alternativa de la escritura con actualización es otra posibilidad a tomar presente: con una nueva escritura, el nuevo valor es multidifundido a todas las memorias *cache* que comparten una copia del *ítem*.

El protocolo maneja la memoria de atracción de los datos (lectura) y los reemplaza cuando un conjunto es atraído a la memoria y esta se llena. El protocolo espía define un nuevo estado y una nueva transacción es enviada como una función de la transacción en el bus, y el estado presente del ítem en la memoria de atracción cambia a otro estado:

Protocolo: estado anterior * transacción nuevo estado * nueva transacción

Un *ítem* puede estar en uno de siete posible estados (el subsistema en la memoria de atracción):

i) Invalido: Este subsistema no contiene el *ítem*.
ii) Exclusivo: Este subsistema y ningún otro contiene el *ítem*.
iii) Compartido: Este subsistema y posiblemente otro subsistema contienen el *ítem*.
iv) Lectura: Este subsistema esta esperando por un valor después que se emitió una lectura.
v) Espera: Este subsistema esta esperando por convertirse en exclusivo después de que se emitió una eliminación.
vi) Lectura y Espera: Este subsistema esta esperando por un valor, y después se convertirá en exclusivo.
vii) Respuesta: Este subsistema tiene comprometida una respuesta a un requerimiento de lectura.

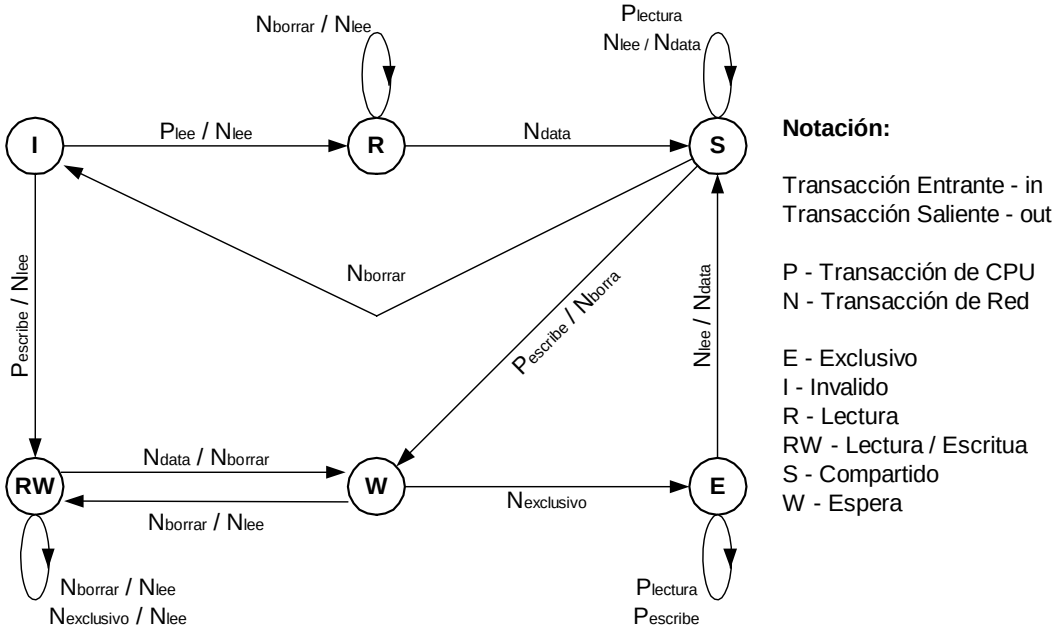
Los primeros tres estados – invalido, exclusivo y compartido – son equivalentes al protocolo de una escritura de *Goodman* descrito con anterioridad.

El estado inconsistente, en el protocolo de *Goodman* – con el significado que este es una copia *cache* y su valor difiere respecto del que esta en memoria – no tiene correspondencia en el modelo *COMA*. Los nuevos estados en el protocolo son estados trascendentes: lectura, espera, lectura y espera y respuesta. Estos estados trascendentes, son requeridos porque se ejecutara la división de una transacción en el bus y necesitan recordar el requerimiento. El bus soporta las siguientes transacciones:

i) Borrar: borra todas las copias del *ítem*.
ii) Exclusivo: Confirmación de un requerimiento de borrado.
iii) Leer: Lee una copia del *ítem*
iv) Data: Transporta los daos en la respuesta a un requerimiento de lectura.
v) Inyectar: Transporta solo la copia de un *ítem* y busca por los subsistemas para moverlo – esto siendo causado por un reemplazo.
vi) Expulsar: Transporta el *ítem* a ser expulsado del subsistema – esto siendo causado por un reemplazo. Este termina cuando otra copia del *ítem* es encontrada.

Una escritura del procesador de un *ítem* es un estado exclusivo o la lectura de un *ítem* en estado exclusivo o compartido procede sin interrupción. Como se muestra en la figura 22, una lectura intenta acceder un *ítem* invalido genera un requerimiento de lectura y un nuevo cambio de estado, lectura. El mecanismo de selección del bus seleccionara una transacción de memoria para servir el requerimiento, eventualmente colocando la transacción de datos en el bus. El requerimiento a la memoria de atracción, ahora esta en lectura, generando la grabación de la transacción de datos, cambiando a compartido y continua.

Figura 22. Representación simplificada del protocolo de memoria de atracción sin incluir reemplazo.



Los procesadores tienen permitido escribir solo en los times que están en estado exclusivo. Si el ítem esta en estado compartido, todas las otras copias tienen que ser borradas y recibir la confirmación antes de escribir el

nuevo valor. La memoria de atracción envía una transacción de borrado y espera por una confirmación exclusiva para el nuevo estado, quedando en espera. Varios intentos simultáneos de escritura al mismo *ítem* resultaran que varias memorias de atracción tengan que esperar, todas con una transacción de borrado en sus *buffers* de salida.

La primera transacción de borrado que llegue al bus es la que gana la competencia por la escritura. Todas las demás transacciones comprometen al mismo ítem a ser removido de los restantes *buffers* de salida. Por consiguiente, los *buffers* tienen transacciones espía. Los *buffers* de salida pueden ser limitados a una cantidad de tres, y los estancamientos pueden ser evitados con algoritmo especial de arbitraje. La memoria de atracción que fue perdida cuando se pasa del estado de espera al de lectura y espera, mientras uno de ellos coloca el requerimiento de lectura en el buffer de salida del otro. Eventualmente, el protocolo en el bus responde con una confirmación exclusiva indicando que la memoria de atracción ha salido del estado de espera y esta procediendo un cambio de estado. Escribiendo un *ítem* que resulta del estado inválido en un requerimiento de lectura genera un nuevo estado, lectura y espera. Adicionalmente a la respuesta de datos, el estado cambia a espera y un requerimiento de borrado es enviado.

3.3.3.2. Reemplazo

Como en memorias *cache* ordinarias, la memoria de atracción correrá fuera de su espacio, forzando a algunos *items* a posicionarse a la cola de los más recientemente accesados. Si el conjunto donde están los *items* esta

lleno, uno de los *items* del conjunto es seleccionado para ser reemplazado. Por ejemplo, el *ítem* más antiguo, el cual debe tener otras copias, es seleccionado. Reemplazar este *ítem* compartido genera una transacción de salida. El espacio utilizado por ese *ítem*, puede ser reclamado. Si una transacción de salida ve a la memoria de atracción en estado compartido, lectura, espera, escritura o lectura y espera, no se ejecutara acción alguna, en caso contrario es convertido en una transacción inyectada por el protocolo. En la implementación de bus simple, lo que procede primeramente es seleccionar un espacio vació, un estado invalido, seguidamente por el reemplazo de un ítem en un estado compartido. Por consiguiente, se presenta un decrecimiento del total de lo que esta compartido. Si el *ítem* identificador de espacio corresponde a una dirección física del espacio de direcciones de arquitecturas convencionales, no procede la suma de los espacios la memoria de atracción y es posible divisar un diseño simple que garantice la localización física de cada ítem.

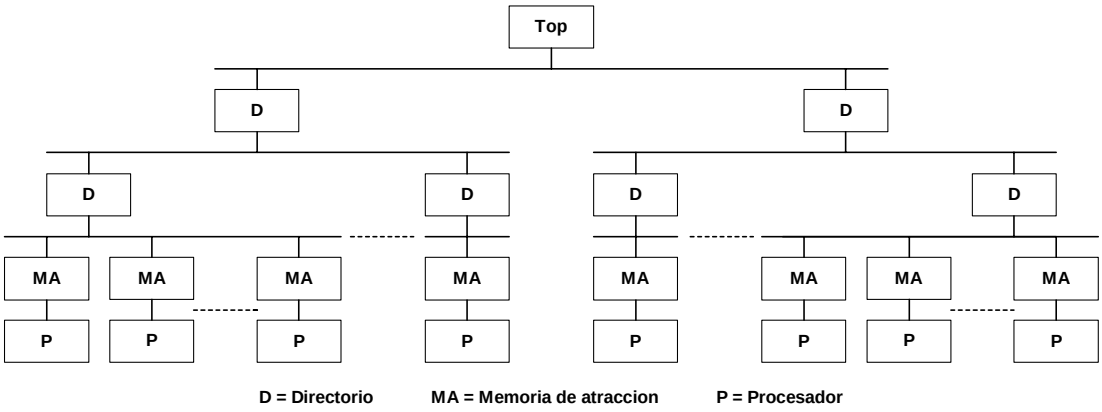
Frecuentemente, los programas utilizan únicamente una porción del espacio de direcciones físicas de una computadora. Esto es verdadero solamente si el sistema operativo facilita la recolección del espacio que ya no esta siendo utilizado. En los sistemas *DDM*, los *ítems* sin uso puede ser utilizados para incrementar la porción de lo que esta compartido por medio de la liberación de los *items* sin usar. El sistema operativo esta en posibilidad de cambiar esta porción de compartición dinámicamente.

3.3.4. La máquina de difusión de datos (DDM) jerárquica

Los recursos forman grandes memorias caches de segundo nivel llamadas memorias de atracción, minimizando el número de accesos al único recurso compartido que queda: el bus compartido. Los datos pueden residir en alguna de estas memorias de atracción. La información es automáticamente trasladada donde es necesitada.

Para convertir un subsistema de bus sencillo DDM en un subsistema jerárquico DDM se tiene que reemplazar la parte superior con un directorio, cuya interfase entre el bus y la parte alta del bus, sean del mismo tipo, tal como se muestra la jerarquía en la figura 23.

Figura 23. Sistema DDM jerárquico con tres niveles

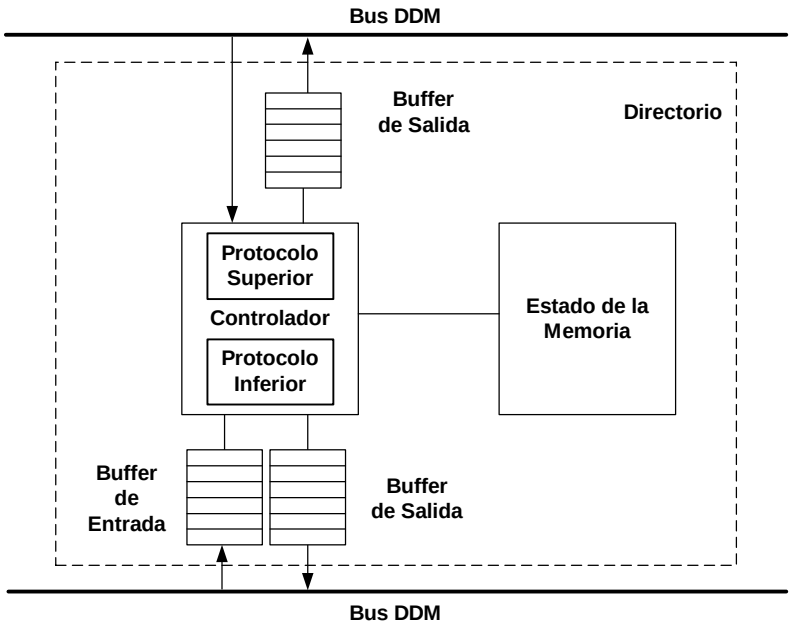


El directorio es un conjunto de estados de memoria asociados que almacenan información de todos los ítems contenidos en la memoria de atracción pero no contienen información en sí. El directorio puede responder

a las preguntas: ¿Esta este *ítem* bajo mi control? y ¿Está este *ítem* fuera de mi subsistema? Del bus superior, el protocolo espía de directorio se comporta muy similar al protocolo de memoria superior. Del bus inferior, el directorio inferior se comporta como el protocolo de los *items* en estado exclusivo. Esto hace operaciones en *ítems* locales hacia el bus idéntico, para ese bus sencillo *DDM*. El directorio pasa transacciones únicas directas provenientes de niveles inferiores que no son completados a su subsistema o sus transacciones de la parte superior que necesitan ser servidas por su subsistema. En ese caso, el directorio actúa como un filtro.

Como se muestra en la figura 24, el directorio tiene un pequeño buffer de salida en donde almacena la transacción que esta esperando ser enviada en el bus. Las transacciones para los buses inferiores son almacenadas en otro buffer de salida en la parte inferior, y las transacciones de los niveles inferiores son almacenadas en un buffer de entrada. Un directorio puede leer desde un buffer de entrada cuando tiene tiempo y espacio para hacer un chequeo de su estado en la memoria. Esta acción no forma parte de una transacción atómica espía en el bus.

Figura 24. La arquitectura de directorio



El diseño jerárquico *DDM* y su protocolo presentan varias similitudes con la arquitectura propuesta por *Wilson*, *Goodman* y *Woest*, anteriormente descrita. *DDM* es diferente en el uso de los estados transitorios en el protocolo, pero carece de memoria compartida física y de conectividad de red, ya que solo almacena los estados de la memoria y no de los datos en sí.

3.3.4.1. Lecturas multinivel

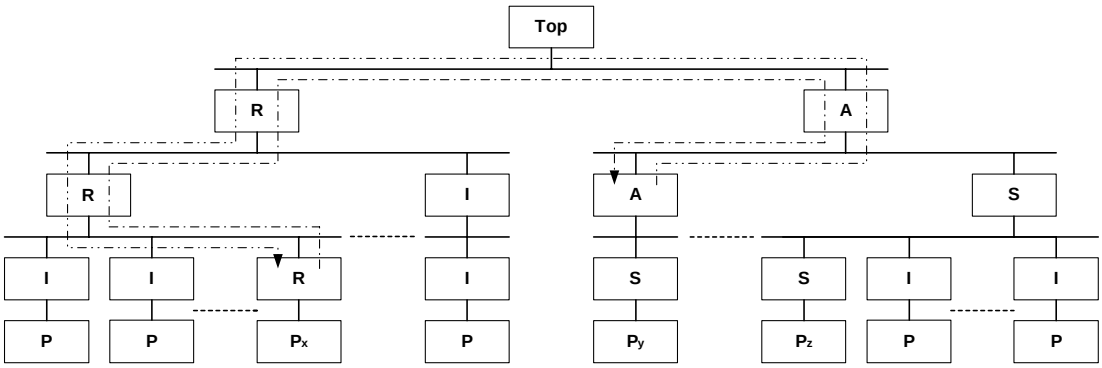
Si el subsistema conectado no puede satisfacer un requerimiento de lectura, el siguiente directorio de nivel superior retransmite el requerimiento al siguiente nivel del bus. El directorio también cambia el estado del *item* a lectura, generando un requerimiento. Eventualmente, el requerimiento alcanza un nivel en la jerarquía donde un directorio contiene una copia del

estado del *ítem* seleccionado y contestara el requerimiento. El directorio seleccionado cambia el estado del *ítem* a respuesta, generando un requerimiento de la parte más alta hacia la parte más baja del bus. Como se muestra en la figura 25, el estado trascendente de lectura y respuesta en el directorio marca el camino del requerimiento a lo largo de la jerarquía, desarrollando una especie de guía que lo lleva a través de un laberinto.

El mecanismo de flujo de control en el protocolo previene los estancamientos si muchos procesadores tratan de desenrollar una tarea de lectura en el mismo directorio. Cuando el requerimiento finalmente alcanza la memoria de atracción con una copia del ítem, su respuesta de datos simplemente sigue la traza leída de regreso al nodo que realizo la petición, cambiando todas las condiciones a lo largo del camino compartido.

Combinado lecturas y emisiones son simples de implementar en el diseño DDM. Sí una petición de lectura encuentra un proceso de lectura para el *ítem* requerido (estado de lectura o respuesta), eso simplemente termina y espera la respuesta de datos que eventualmente seguirá el camino en su camino de regreso.

Figura 25. Lectura multinivel en una jerarquía DDM



Un requerimiento de lectura del procesador P_x, encontro su camino a una copia del item en la memoria de atraccion del procesador P_y. Su camino es marcado con el estado Lectura (R) y respuesta (A), el cual guiara la respuesta de datos al procesador P_x. Los estados (I) indican que estan en un estado invalido, y la (S) indica que esta en estado compartido.

3.3.4.2. Escrituras multinivel

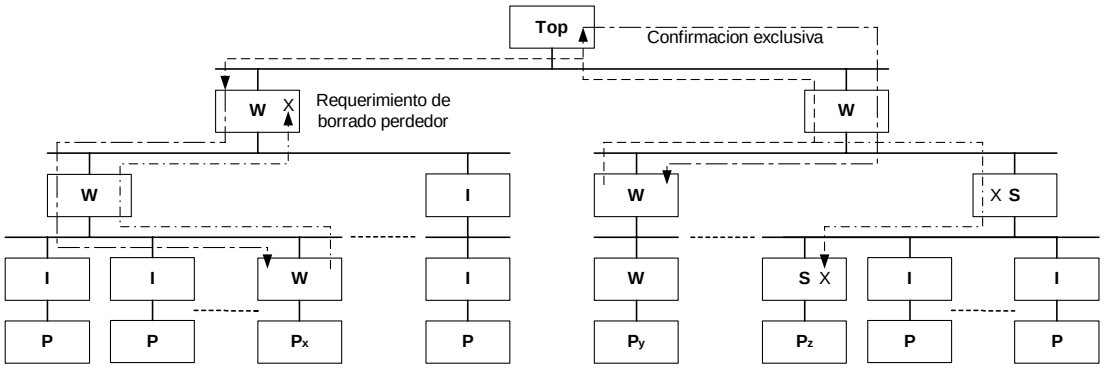
Un requerimiento de borrado emitido por el directorio inferior a un ítem en un estado exclusivo genera como resultado una confirmación exclusiva enviada al directorio que esta generando la petición. Un requerimiento de borrado que no pueda confirmar la operación al directorio que lo solicito, cambiara todos los estados de los directorios intermedios a un estado de espera hasta que se reciba el requerimiento en la jerarquía por la cual realizo la búsqueda del *ítem*. Todos los subsistemas en el bus llevan una copia de la transacción de borrado con el problema que esta marca la copia de cada directorio en el mismo estado.

La propagación de la transacción de borrado termina cuando el directorio que posee la información alcanza el estado exclusivo y la confirmación es retornada al directorio origen, cambiando todos los estados en espera a

estado exclusivo, a través de todo el camino que se utilizo para encontrar el estado del *ítem* requerido.

Una competencia entre dos procesadores cualesquiera en el bus jerárquico DDM ha sido la solución, con características similares a las del bus simple DDM. Cuando se ha requerido que se efectué dos transacciones de borrado son propagadas en su jerarquía correspondiente, la primera transacción que alcance el directorio con la información requerida, es el ganador, como se muestra en la figura 26, y la transacción que no gano la competencia, la cual queda en estado de lectura – espera, reiniciar su acción automáticamente cuando la confirmación de la transacción ganadora haya terciando su acción.

Figura 26. Escritura multinivel en una jerarquía DDM



Una competencia entre dos procesador P_x y P_y, es resuelto cuando el requerimiento originante de P_y cuando alcanza el top del bus (el bus mas bajo comun para ambos procesadores). El tope envia entonces una confirmacion exclusiva el cual sigue el camino marcado con W (el procesador en estado de espera) de regreso al procesador P_y, quien gano la competencia. El estado de espera cambia a exclusivo por la confirmacion. La transaccion de borrado borra los datos en los procesadores P_x y P_z, forzando a P_x, a reconstruir su intento

3.3.4.3. Reemplazos en un sistema jerárquico DDM

Los reemplazos de ítem compartidos en un sistema jerárquico *DDM* da como resultado una transacción saliente propagándose a lo largo de la jerarquía, terminando cuando encuentra un subsistema en alguno de lo siguientes estados: compartido, lectura, espera o respuesta. Si la última copia de un *ítem* marcado compartió es reemplazado, entonces una transacción saliente no puede terminar la relación alcanzará un directorio en la exclusividad y cambiará la transacción en una entrante. El reemplazo de un *ítem* exclusivo genera una transacción entrante que tratara de encontrar un espacio vació en la memoria de atracción. La transacción entrante primero tratara de encontrar un espacio en la memoria de atracción en el bus local *DDM*. A diferencia del bus simple *DDM*, una transacción entrante podrá fallar en la búsqueda de un espacio vació en el bus local *DDM*, el cual se convertirá en un bus especial, su bus padre, el cual es determinado por el *ítem* identificador. En el bus padre, la transacción entrante se forzará apropiadamente en una memoria de atracción, posiblemente desalojando a un *ítem* extranjero o compartido. El espacio del ítem padre es por igual a la copia de los *ítems* inferiores, por consiguiente el espacio esta garantizado en el bus padre.

La localización preferida en DDM es diferente a la posición de memoria en *NUMA* en que un ítem busca una sede sólo en reemplazo después de encontrar lugar a otro sitio. Cuándo el *ítem* no está en su lugar, otro *ítem* pueden usar su sitio. La sede también difiere de la estrategia *NUMA* en ser un bus: Cualquier memoria de atracción en ese bus lo ejecutará.

3.3.4.4. Reemplazos en el directorio

El reemplazo de multinivel tiene una implicación en el sistema: Un directorio en el nivel i + 1 tiene que estar en un mega conjunto de directorios o memorias de atracción en el nivel i. En otras palabras, el tamaño de un directorio y su asociatividad (numero de caminos) tiene que ser B_i veces el número de niveles abajo del nivel i, donde B_i es un factor de bifurcación del nivel i, y el tamaño denota el número de *items*, donde:

$$\text{Tamaño: } Dir_{i+1} = B_i * Dir_i$$
$$\text{Asociatividad: } Dir_{i+1} = B_i * Dir_i$$

Inclusive sí fuese implementable, los niveles más altos de memoria serian costosos y lentos si sus propiedades fueran cumplidos a lo largo de toda la jerarquía del sistema. Sin embargo, los efectos de esta propiedad de inclusión multinivel esta limitada en los sistemas DDM. Estos solo almacenan el estado de información del estado en sus directorios y no replica información en los niveles superiores. Existe otra forma de limitar los efectos usando directorios imperfectos con grupos pequeños de caminos que son requeridos por la inclusión multinivel y dar a los directorios la habilidad de ejecutar reemplazos, moviendo todas las copias a un *ítem* saliente en su subsistema. Se puede mantener una probabilidad de reemplazo razonable en los niveles incrementando moderadamente el grado de asociatividad en los niveles superiores en la jerarquía del sistema. Un alto grado de compartición también ayuda a mantener una probabilidad baja. Un *ítem* compartido ocupa espacio en varias atracciones de memoria, pero solo un espacio de direcciones en el directorio es utilizado. La implementación de

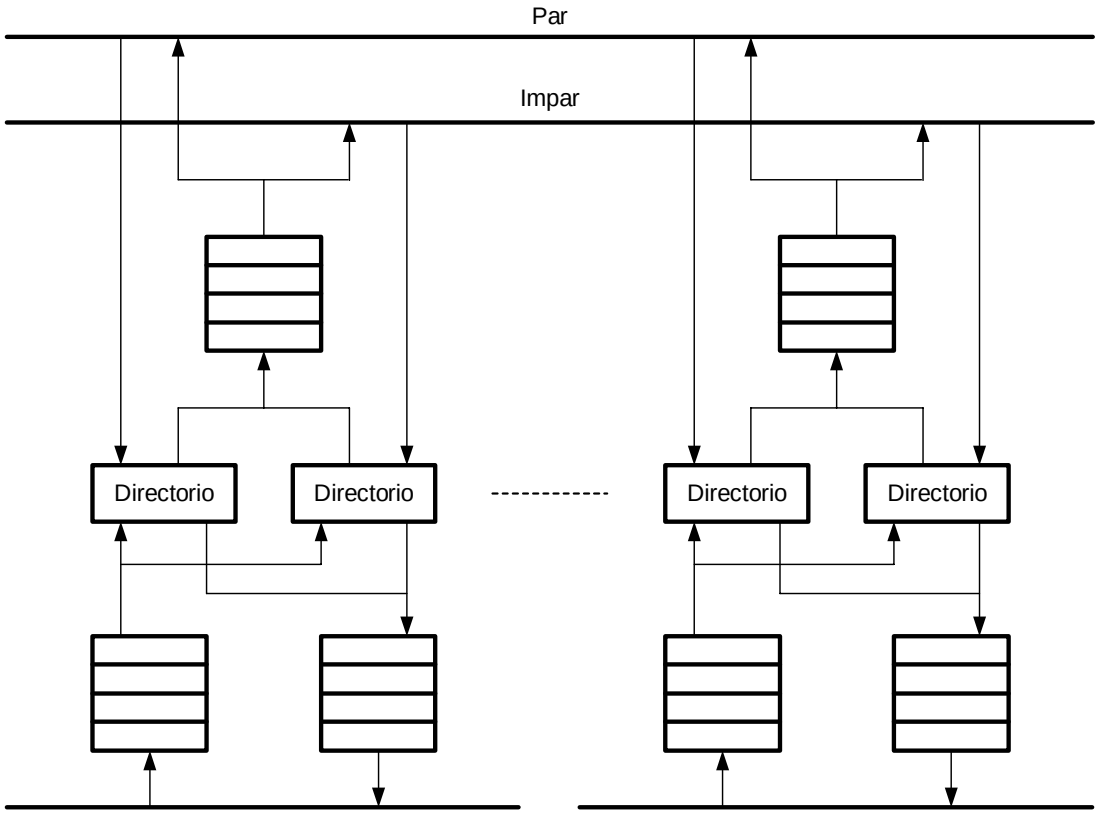
reemplazos en el directorio requiere un estado y dos transacciones adicionales.

3.3.5. Aumento del ancho de banda

A pesar de que la mayor parte de la memoria tiende a ser localizada en la maquina, los niveles altos en la jerarquía pueden a pesar de todo tener una demanda muy alta por los niveles inferiores, generando un cuello de botella. Pare reducir la carga de los niveles más altos, se utilizan pequeñas bifurcaciones hacia los niveles superiores de la jerarquía.

Esta solución, sin embargo, incrementa el número de niveles en la jerarquía, resultando en un acceso diferido remoto y en un incremento sobre la memoria. En lugar de lo antes descrito, se pueden ensanchar los niveles superiores de la jerarquía para generar un árbol ancho. Con esto se logra a partir de un directorio divido en dos partes iguales a partir del tamaño original. Los dos nuevos directorios contendrán direcciones de dominios diferentes (pares e impares). La comunicación con otros directorios también se parte de la misma forma, incrementando así al doble el ancho de banda. Se pueden ejecutar la cantidad de particiones necesarias en cualquier nivel de la jerarquía, tal como se muestra en la figura 27.

Figura 27. Incremento del ancho de banda a utilizando particionamiento de los buses



Otra solución, es la implementación de una red heterogénea; donde se utilizan todas sus ventajas hasta el punto donde le sea posible, tomando varias jerarquías juntas al mismo tiempo llevándolas al nivel superior de la red con un protocolo basado en directorio. Este diseño requiere algunos cambios en el protocolo para conseguir la misma consistencia del otro modelo antes descrito.

4. IMPLEMENTACIONES DE SOFTWARE

Los sistemas distribuidos tradicionales han desarrollado ampliamente dos paradigmas de comunicación: el paso de mensajes y las llamadas a procedimientos remotos (*RPC*). Ambos métodos presentan un inconveniente muy obvio: son muy complicados para el programador dado que son más complicados que el paradigma de la memoria compartida, e introducen problemas severos en el paso de construcciones complejas de estructura de datos, ya que el concepto de paso de apuntadores usados por sistemas uníprocesadores no tiene sentido cuando no tienen un espacio común de direcciones. Por consiguiente, la idea de construir un mecanismo que provea el paradigma de memoria compartida para el programador por encima del paso de mensajes parece razonable. La flexibilidad general de la metodología informática también facilitó a los investigadores la implementación de los mecanismos de consistencia para el comportamiento de aplicación, en sus esfuerzos para obtener mejor rendimiento.

El concepto de implementación de soluciones *DSM* en software es algo similar a las implantaciones de hardware en las arquitecturas *NUMA* (donde todos los datos pueden tener una localidad predeterminada) y *COMA* (donde los datos pueden migrar entre sitios sin restricciones). El tamaño del granulo (en el orden de los *kilobytes*) es típicamente para soluciones de software, porque los mecanismos de *DSM* y memoria virtual están frecuentemente integrados.

Ésta puede ser una ventaja para aplicaciones que expresan altos niveles de localidad, así como también una desventaja para esos los que se caracterizan por compartir una granularidad fina, a causa de los efectos adversos de hiperpaginación. El soporte del software para DSM es generalmente más flexible que el soporte del hardware, pero en muchos casos no puede competir con implementaciones de hardware en el ámbito de rendimiento. Con excepción de tratar de introducir aceleradores de hardware para solucionar el problema, los diseñadores se han concentrado en liberar el modelo de consistencia, generando una carga adicional para el programador. Algunos sistemas también tratan de integrar ambos paradigmas: el paso de mensajes y el DSM.

4.1. Programación distribuida con datos compartidos

4.1.1. Abstracción

Hasta hace poco, al menos una cosa fue clara acerca de la programación paralela: las máquinas fuertemente acopladas (memoria compartida) estaban programadas en un lenguaje basado en variables compartidas, y los sistemas débilmente acoplados (distribuidos) estaban programados usando paso de mensajes. Las primitivas del sistema operativo y de lenguajes para la programación de sistemas distribuidos, han sido propuestos para que soporten el paradigma de la variable compartida sin la presencia de memoria física compartida. Las computadoras paralelas de la clase *MIMD* (*Multiple Instruction Multiple Data* – Múltiples Instrucciones Múltiples Datos) tradicionalmente se dividen en dos grandes categorías: los fuertemente acoplados y los débilmente acoplados.

Los sistemas fuertemente acoplados, una parte de la memoria primaria es compartida; todos los procesadores tienen acceso directo a su memoria compartida en una instrucción de maquina. Los sistemas débilmente acoplados, estos son sistemas distribuidos donde el procesador solo tiene acceso a su memoria local, los procesadores se pueden comunicar entre sí por medio del envió de mensajes sobre un canal de comunicación, entre ellos: Comunicación punto a punto o una red de área local. El primer tipo de sistema tiene una ventaja significante respecto del segundo: comunicación rápida a través de la memoria compartida. En cambio, los sistemas distribuidos son mucho más fáciles de construir, especialmente si una gran cantidad de procesadores es requerida. Como consecuencia de esta clasificación, los diseñadores de sistemas operativos y lenguajes de programación generaron dos paradigmas de programación paralela: variables compartidas (para los sistemas fuertemente acoplados) y el paso de mensajes (para los sistemas débilmente acoplados – sistemas distribuidos).

4.1.2. Variables compartidas y paso de mensajes

La comunicación a través de variables compartidas probablemente es el paradigma más antiguo de programación paralela. Un gran número de sistemas operativos para sistemas uníprocesador están estructurados de colecciones de procesos, ejecutándose casi paralelamente y comunicándose entre ellos por medio de variables compartidas. Los accesos sincrónicos a los datos compartidos han sido objeto de investigación desde inicios de los años sesenta. Numerosos lenguajes de programación existentes hoy en día, todavía usan variables compartidas.

La semántica del modelo es medianamente simple, excepto por lo que ocurre cuando dos procesos simultáneamente tratan de escribir (o leer y escribir) la misma variable. En la semántica, cualquiera puede definir lecturas o escrituras simples indivisibles (los conflictos de lecturas y escrituras son serializados) o puede dejar el efecto de escrituras simultaneas en estado indefinido. La base para el paso de mensajes, como una interpretación elaborada de lenguaje de programación es un clásico de *Hoare* en *CSP*. Un mensaje en *CSP* es enviado de un proceso (el emisor) para un otro proceso (el receptor). El emisor espera hasta que el receptor ha aceptado el mensaje (paso de mensaje sincrónico).

Muchas variaciones del paso de menajes han sido propuestas. Con el paso de mensaje asincrónicos, el emisor continúa inmediatamente después de enviando el mensaje. La llamada de procedimiento remoto y *rendez.vous* son interacciones de dos formas entre dos procesos. La difusión y la multidifusión son interacciones entre un emisor y muchos receptores. Los puertos de comunicación o los buzones, pueden usarse para evitar direccionamiento explícito de procesos. A continuación, se describen las diferencias más importantes entre los dos extremos del espectro: Las variables compartidas y el paso de mensajes simples (sincrónico y asincrónico):

i) La transferencia de información en un mensaje entre dos procesadores, tiene que existir entre ambos cuando la interacción tome lugar. Al menos, el proceso emisor tiene que saber la identidad del proceso receptor. Los datos almacenados en una

variable compartida son accesados por varios procesos. La interacción de los procesos a través de las variables compartidas no tienen que traslaparse durante su tiempo de vida con otras variables existentes. Solo tienen que saber la dirección de la variable compartida.

ii) Una asignación a una variable compartida tiene un efecto inmediato. En cambio, existe un retardo que se puede medir entre él envió de un mensaje y su recepción. Para el paso de mensajes asincrónicos, por ejemplo, existe una ramificación para el orden en el cual los mensajes son recibidos. Usualmente, la semántica preserva el orden: los mensajes entre un par de procesos son recibidos en el mismo orden en el que fueron enviados. Cuando hay más de dos procesos, la demora todavía tiene que ser tomada en consideración. Por ejemplo, supongamos que el proceso P_1 envía un mensaje X a P_2 y posteriormente a P_3. Al recibir X, P_3 envía un mensaje Y a P_2. No hay garantía que P_2 reciba X antes que Y.

iii) El paso de mensaje intuitivo es más seguro que compartir variables. En este caso, la seguridad significa que un modulo del programa no puede afectar la corrección de otro modulo. Las variables compartidas pueden ser cargadas por cualquier proceso, así que, la seguridad es un problema potencial. Una solución a esto, es la utilización de monitores, los cuales encapsulan datos y serializan todas las operaciones en los datos.

iv) Un mensaje de intercambio de información, pero siendo un proceso sincrónico. El proceso receptor espera a que el mensaje arribe con el paso del mensaje sincrónicamente, el proceso emisor también espera a que el receptor este listo. Con variables

compartidas, los dos tipos de sincronización son útiles. La exclusión mutua previene escrituras simultáneas (o lecturas y escrituras) de la misma variable; la condición de sincronización permite al proceso esperar una cierta condición para ser verdadero. El proceso puede sincronizarse a través de variables compartidas quedando en estado de espera ocupada (busy-waiting), pero este comportamiento es indeseado ya que pierde muchos ciclos de procesador. Para un mejor mecanismo, se pueden utilizar semáforos, contadores de eventos y condicionantes.

El modelo del paso de menajes presenta algunos problemas adicionales en su implementación. El paso de una estructura de datos compleja a un proceso remoto se torna complicado. El proceso no puede fácilmente moverse hacia otro procesador, generando una administración de procesos eficiente más complicada. El modelo variable compartida no padece de estos problemas.

4.1.3. En medio de variables compartidas y paso de mensajes

Los paradigmas de variables compartidas y paso de mensajes, presentan cada uno sus propias ventajas y desventajas. Debería originarse como una sorpresa que el lenguaje y los diseñadores del sistema operativo han mirado primitivas que está en algún lado en medio de estos dos extremos, y que comparte las ventajas de ambos.

Una variable compartida simplemente puede estar simulada en un sistema distribuido almacenándola en un procesador y dejando otros procesadores leído y de escritura a eso con llamadas de procedimiento remoto. En la mayoría de sistemas distribuidos, sin embargo, la llamada de un procedimiento remoto es de dos a cuatro el orden de magnitud más lenta que dando la lectura a los datos locales. Esta diferencia hace una simulación poco atractiva.

La mayoría de sistemas que se describirán a continuación presentan el uso de primitivos de disponibilidad que tienen algunas propiedades de variables compartidas y una parte de paso de mensajes. La semántica está en algún lado adentro en medio variables compartidas y paso de mensajes. A menudo, los datos son sólo accesibles por algún de los procesos y sólo hasta el final algunas operaciones específicas. Estas restricciones hacen a los primitivas más seguros desde un cliente habitual con variables compartidas y hacen implementaciones eficientes que posiblemente son el reconocimiento de la memoria física ausente. Estos sistemas se enfocan basándose en los siguientes 4 tópicos:

i) ¿Cuál es la semántica de cada primitiva?
ii) ¿Cómo pueden ser los datos compartidos direccionables?
iii) ¿Cómo sé accesan los datos compartidos de una forma sincronizada?
iv) ¿Cómo puede ser una primitiva implementada eficientemente sin el uso de memoria física compartida?

4.1.3.1. Puertos de comunicación

En los lenguajes de programación tradicionales, la interacción entre los procesos debe ser explícitamente nombrada entre ellos. Para muchas aplicaciones, especialmente en las basadas en el modelo cliente / servidor, este es un inconveniente. Una solución para este modelo, es el envió de un mensaje indirectamente a través de un puerto de comunicación. Un puerto o un buzón de correo es una variable en donde el mensaje puede ser enviado o recibido. Un puerto puede ser considerado como una estructura de cola de datos compartida, con las siguientes operaciones definidas:

```
Send(msg,q);          # Se agrega un mensaje al final de la cola
Msg := receive(q);    # Espera mientras la cola no este vacía
                      # Extrae los mensajes por la cabeza de la cola.
```

La operación más reciente adicionalmente sincroniza los procesos. Los puertos pueden ser direccionados como variables normales. La implementación requiere de un buffer para almacenar los mensajes enviados pero no necesariamente cuando son recibidos. A pesar que la semántica de los puertos es esencial para el paso de los mensajes de forma asincrónica,

es importante notar que los puertos pueden ser descritos como estructuras de datos compartidos con operaciones de acceso especializadas.

4.1.3.2. Modelo de objetos

Los lenguajes orientados a objetos se están incrementando popularmente, no solo para implementar programas secuenciales, sino también para la escritura de programas paralelos. Los diferentes lenguaje de programación de hoy en día, usan diferentes definiciones del termino objeto, pero en general, un objeto encapsula comportamiento y datos. Entre estos lenguajes, se enumeran los siguientes: *ABCL/1, Aeolus, Concurrent-Smalltalk, Emerald, Raddle y Sloop.*

Un objeto en un lenguaje basado en objetos concurrentes puede ser considerado como un dato compartido que es accesible solo a través de un conjunto de operaciones definidas por el objeto. Estas operaciones son invocadas por el envió de un mensaje al objeto. La invocación de la operación puede ser asincrónica (el invocador puede continuar inmediatamente después del envió del mensaje) o sincrónica (el invocador espera hasta que la operación ha sido completada). Los objetos son usualmente direccionados por una referencia de objeto o por un nombre global de objeto. Para sincronizar los accesos a los objetos compartidos, existen varias metodologías, como por ejemplo: *Emerald* usa un monitor como constructor para sincronizar invocaciones de múltiples operaciones al mismo objeto; *Sloop* soporta objetos indivisibles, en el cual solo la invocación de una operación es permitida al ser ejecutada, en cuyo caso para casos de

condiciones sincrónicas, este lenguaje operaciones de suspensión a través de expresiones lógicas (boleanas), causando que la invocación del proceso se bloquee hasta que la expresión sea verdadera nuevamente.

Un asunto de importancia crucial en una implementación distribuida de objetos es encontrar el objeto en los procesadores que lo usan más frecuentemente. Alternamente, la colocación de objetos puede quedar enteramente desligada al sistema en tiempo de ejecución. El modelo de objetos ya presenta la ilusión de datos compartidos. El acceso para los datos compartidos está restringido para algunas operaciones bien definidas, haciendo al modelo más seguro que el modelo de variable compartida simple. La sincronización fácilmente puede ser integrada con las operaciones.

4.1.3.3. Problemas orientados a la memoria compartida

D.R. Cheriton propone un tipo de memoria compartida que puede ser ajustable a una aplicación específica, conocido como el problema orientado a la memoria compartida. La memoria compartida puede ser definida como un servicio de sistema distribuido, implementado en múltiples procesadores. Los datos son almacenados (replicados) en uno o más procesadores, y puede ser depositado en una estación de trabajo.

La semántica del problema orientado a la memoria compartida es reajustada a las necesidades de la aplicación que están haciendo uso de

ella. En general, la semántica es más flexible que el uso de variables compartidas. En particular, las copias inconsistentes de los mismos datos son permitidas coexistir, así es que una operación de lectura no necesariamente devuelve el valor almacenado por la más reciente escritura. Hay varias metodologías para ocuparse de estos datos echados a perder. El problema orientado a la memoria compartida es el direccionamiento, ya que de alguna forma afecta a una aplicación especifica. Las direcciones son emitidas para los procesadores del servidor.

No existe alguna provisión especial que sincronice los procesos (los procesos pueden sincronizarse utilizando el paso de mensajes). La implementación significativamente se aprovecha las semánticas flexibles. Lo más importante, es que no tiene que usarse en esquemas complicados donde las actualizaciones se ejecutan atómicamente a todas las copias que contengan la misma información.

4.1.3.4. Espacios pares (tuplas)

Los espacios pares (tuplas) es un mecanismo de sincronización novedoso, diseñado por *David Gelernter*. Los espacios pares están formados por memoria global conteniendo pares, similares a los registros que se usan en el lenguaje Pascal. Por ejemplo, el par "["Miami",305]" consiste de un campo alfanumérico y un campo entero, respectivamente. Los espacios pares son manipulados por tres operaciones atómicas: *add*, el cual agrega un nuevo par; *read*, el cual lee el par actual; e *in*, el cual lee y elimina un par. Nótese que no existe una operación que cambie el par

actual. En lugar de eso, primeramente se elimina el par del espacio asignado y posteriormente es regresado, ya sea con el mismo valor o llevando uno nuevo.

A diferencia de otros conceptos de memoria compartida, los espacios pares son direccionables asociativamente, por contenido. Un par es denotado suministrado por su contenido actual o a través de parámetros formales para cada campo. El par mencionado anteriormente, puede ser leído o borrado, como por ejemplo:

Ejemplo 1:	Ejemplo 2:
In (“Miami”, 305);	Integer areacode;
	In (“Miami”, var areacode);

En ambos ejemplos, el par debe existir. Sí dos procesos simultáneamente tratan de remover (*in*) el mismo par, solamente uno de ellos podrá accederlo y el otro se queda bloqueado. Un par debe ser removido antes de ser cambiado, dado que las actualizaciones simultáneas son sincronizadas.

A pesar de que esta semántica es significativamente de las variables compartidas. Este método da la impresión de que sé esta utilizando memoria compartida.

4.1.3.5. Memoria virtual compartida

Kai Li extiende el concepto de memoria virtual a los sistemas distribuidos, dando como resultado una memoria virtual compartida. Esta memoria es accesible por todos los procesos y es direccionable como la memoria virtual tradicional. El sistema descrito por *Li* garantiza coherencia en la memoria: el valor retornado por una lectura siempre será el último valor generado por la última escritura. El espacio de direcciones esta particionado en páginas numéricas de tamaño fijo. En cualquier punto en el tiempo, varios procesos pueden tener una copia en modalidad de lectura de la misma página; alternativamente, un procesador puede tener una copia en modalidad lectura y escritura de la misma página.

Sí un proceso trata de escribir en cierta página mientras el procesador no tiene la página en modalidad de lectura y escritura, se genera una falla de página. La rutina de control de fallas le indica a los otros procesadores que invaliden la copia de la página y la recarguen la copia de la página (si no han obtenido alguna), se pasa a modo protegido para lectura y escritura, y reinicia la instrucción de falla. Sí un proceso quiere leer una página, pero no ha tenido una copia de ella previamente, se genera una falla de lectura de página. Sí algún procesador tiene la página en modalidad de lectura y escritura, se le envía una señal al procesador para que cambie la protección a solo lectura. La copia de la página es generada y la instrucción de falla es reiniciada.

La memoria virtual compartida es direccionable como la memoria virtual normal. Una implementación puede soportar varios mecanismos de sincronización, tales como semáforos, contadores de eventos y monitores. La memoria virtual compartida puede ser utilizada para simular variables compartidas verdaderas, permitiendo una semántica exacta. La implementación utiliza la unidad de administración de memoria del hardware y puede beneficiarse de la disponibilidad del multidifusión (para la invalidación de todas las copias de una página). Existen varias estrategias para tratar el problema de las escrituras múltiples simultáneas y administrar cual procesador tendrá una copia de una página. El esquema completo se ejecutara pobremente si los procesos en los diferentes procesadores escriben a la misma página repetidas veces. Migrando todos los procesos al mismo procesador se puede resolver este problema.

4.1.3.6. Variables lógicas compartidas

La mayoría de lenguajes lógicos de programación concurrentes, por ejemplo: *prolog, parlog*, usan variables lógicas compartidas como los canales de comunicación. Las variables lógicas accionariales tienen la propiedad de la sola asignación: media vez son vinculados con un valor (o con otra variable) no pueden variar. La asignación simple no es una restricción severa, porque una variable lógica puede estar obligada con una estructura u otras más, las variables desligadas, lo cual puede servir para una comunicación futura. De hecho, muchos patrones de comunicaciones pueden ser expresados usando variables lógicas compartidas.

La sincronización en los lenguajes lógicos de programación concurrentes se parece a la sincronización de flujo de datos: El proceso (implícitamente) puede esperar una variable para ser vinculado. Las variables lógicas compartidas proveen un modelo semántico claro, asemejándose a las variables lógicas normales. El direccionamiento también es el mismo para ambos tipos de variables. La propiedad de la una sola asignación, permite al modelo ser implementado con una eficiencia razonable en un sistema distribuido. Si un proceso trata de leer una variable lógica almacenada en un procesador remoto, entonces el procesador remoto añade el proceso a una lista asociada con la variable. Tan pronto como la variable es vinculada (sí no se encuentra lista), su valor es enviado a todos los procesos en la lista. Estos procesos mantendrán el valor para utilización futura. De este modo, las variables están automáticamente replicadas en referencia.

4.1.4. El Modelo de objetos de datos compartidos

Recientemente, se ha desarrollado un nuevo modelo conceptual de datos compartidos, llamado modelo de objetos de datos compartidos. En este modelo, los datos compartidos están encapsulados en objetos pasivos. Los datos contenidos por el objeto son solamente accesibles a través de un conjunto de operaciones definidas por el tipo de objeto. Los objetos son instancias de tipo de dato abstracto. A diferencia de los lenguajes *Emerald* y *Sloop*, no se consideran los objetos como entidades activas ni se consideran todas las entidades en el sistema como objetos.

La actividad paralela origina la creación dinámica de procesos secuenciales múltiples (mono tarea). Cuando un proceso genera un proceso hijo, este puede pasar cualesquiera de sus objetos como parámetros compartidos al proceso hijo. Estos procesos hijos pueden pasar el objeto a sus procesos hijos, y así sucesivamente. De este modo, el objeto queda distribuido entre algún de los descendientes del proceso que declara el objeto. Todos estos procesos comparten el objeto y pueden realizar el mismo conjunto de operaciones en él, los cuales están definidos por el tipo del objeto. Los cambios hechos a un objeto por un proceso, son visibles para los otros procesos, así es que los objetos de datos compartidos son un canal de comunicación entre procesos.

El modelo de objetos de datos compartidos tiene muchas ventajas sobre las variables compartidas regulares. El acceso para los datos compartidos es sólo permitido a través de operaciones directas definidas por un tipo de dato abstracto. Todas estas operaciones son indivisibles. Las invocaciones simultáneas del mismo objeto tienen el mismo efecto como sí fuera ejecutado una por una, de tal manera obtienen acceso a los datos compartidos automáticamente sincronizados.

4.1.4.1. Implementación de objetos de datos compartidos en un sistema distribuido

El modelo de objetos de datos compartidos es sencillo, porque proporciona un mecanismo seguro para compartir datos y una sincronización al acceso de los datos. La implementación de la distribución se ha diseñado para basarse en una replicación selectiva y migración de objetos, bajo control completo del sistema en tiempo de ejecución. El compilador distingue dos clases de operaciones:

i) La operación "*read*" no modifica el objeto; es realizado en una copia local, sí uno existe
ii) La operación "*write*" puede leer y modificar los datos del objeto, este afecta a todas las copias de este.

Se usará una vista simplificada del modelo para describir su implementación. En particular, se asumirá que un objeto contendrá un solo entero y se consideran sólo dos operaciones:

```
Operación read (x: object): integer;    # retorna el valor actual
Operación write(x: object, val: integer); # almacena un nuevo valor
```

El sistema en tiempo de ejecución dinámicamente lleva control de cuánto tiempo los procesadores realizan operaciones lectura y de escritura remota de cada objeto. Sí un procesador frecuentemente lee un objeto remoto,

entonces es beneficioso para él mantener una copia local de los objetos. La sobrecarga de tiempo de ejecución de mantener las estadísticas es despreciable comparado con el tiempo necesitado para hacer referencias remotas. La sobrecarga de espacio no es una preocupación real tampoco, considerando la condición actual de tecnología de memoria.

Un aspecto principal al implementar réplica es que debe de considerar la velocidad de los cambios hechos a los datos. Dos estrategias son posibles: Invalidando todas las copias excepto una, o actualizando todas las copias. El modelo de memoria virtual compartida de *Kai Li* utiliza invalidación. En este modelo, la invalidación es ciertamente factible pero tiene algunas desventajas. Primero, si un objeto es grande es costoso la invalidación de todas copias, especialmente sí una operación cambia sólo una parte pequeña. En este caso, es mucho más eficiente ejercer la operación para todas las copias, por lo tanto actualiza todas las copias. En segundo lugar, sí un objeto es pequeño, enviar un valor nuevo es probablemente tan costoso como enviar mensajes de invalidación.

Un tema relacionado a esto, es como sincronizar las invocaciones simultáneas de una operación que trata de modificar el mismo objeto. Para serializar tales invocaciones, se consigna una copia del objeto como copia primaria, directo a este, todas las operaciones de escritura son efectuadas, y posteriormente se propagan a las copias secundarias.

Una metodología alternativa sería tratar a todas las copias por igual utilizando un protocolo distribuido que suministrase exclusión mutua. El método de la copia primaria, sin embargo, permite una optimización importante: La copia primaria puede ser migrada al procesador que más frecuentemente cambia el objeto, generando actualizaciones más eficientes.

4.1.4.2. Distribución con inconsistencia

La presencia de copias múltiples de los mismos datos introduce el problema de consistencia. Como no se quiere la confusión de la semántica del modelo, la implementación adecuadamente deberá solucionar este problema. Para ocuparse del problema de consistencia, primero se necesita una comprensión más profunda del problema mismo. Se asume que se implementara el modelo como sigue. Para actualizar un objeto x, su copia primaria será bloqueada y un mensaje conteniendo el valor nuevo de X se envía todos los procesadores que contienen una copia secundaria. Tal procesador actualiza su copia y entonces devuelve una aceptación. Cuando todos los mensajes hayan sido aceptados, la copia primaria está actualizada y desbloqueada.

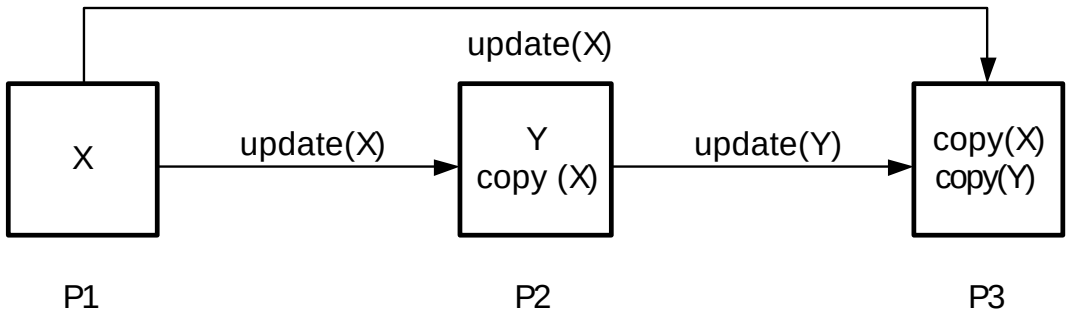
Durante la actualización del protocolo, algunos procesadores han recibido el nuevo de valor X, mientras los otros todavía usan su valor anterior. Éste es intuitivamente poco atractivo, solamente por si mismo no es el problema real. No todos los procesadores observarán las modificaciones de los diferentes objetos en el mismo orden. Consideremos el siguiente programa:

```
X, Y: shared object;          # inicialmente 0

Proceso P1:
        for i := 1 to ∞ do
                write (X,i);
Proceso P2:
        repeat
                y := read(Y); x := read(X);
                if x > y then
                        write (Y,x);
Proceso P3:
        repeat
                y := read(Y); x := read(X);
                assert x ≥ y;
```

P_2 trata mantener actualizado Y con el valor X; P_3 verifica que el valor de X sea mayor o igual a Y. Claramente, esta ultima condición siempre será verdadera. Ahora se va a suponer que X y Y son replicadas como se muestra en la figura 28.

Figura 28. Distribución de X y Y



P_1 contiene la copia primaria de X, P_2 y P_3 tienen una copia secundaria. P_2 tiene una copia primaria de Y, y P_3 tiene una copia secundaria. La secuencia de estos puede ocurrir:

i) X se incrementa y se convierte en 1; P_1 envía un mensaje de actualización a P_2 y P_3.
ii) P_2 recibe el mensaje de actualización; asigna 1 a la variable Y y envía un mensaje de actualización de Y a P_3.
iii) P_3 recibe el mensaje de actualización de P_2; asigna 1 a su copia de Y, este se sorprende que Y es ahora mayor que X (quien todavía tiene un valor 0).
iv) P_3 recibe el mensaje de P_1, y almacena el valor de 1 en su copia de X.

P_3 observa que el cambio de X y Y en el orden incorrecto. El problema se debe a la cantidad arbitraria de tiempo que los mensajes pueden tomar al viajar de la fuente al destino y por la incapacidad de transferir información simultáneamente de una fuente a varios destinos. Tal implementación básicamente provee transición de mensajes en la que la semántica disfrazó la sintaxis de variables compartidas. La solución para el problema de consistencia depende de la arquitectura del sistema distribuido subyacente.

4.1.4.3. Implementación con paso de mensajes punto a punto

Un modelo de un sistema distribuido se puede presentar como una colección de procesadores que se comunican entre sí, enviándose mensajes de un punto a otro punto, el uno al otro. Un camino de comunicación entre

cualquier par de procesadores es provisto, ya sea por el hardware o el software. Los mensajes son entregados confiadamente, en la misma orden en el que ellos fueron enviados. Para implementar el protocolo, usamos un proceso administrador para cada procesador. Se asume que el proceso administrador y el proceso del usuario están en el mismo procesador y pueden compartir parte de su espacio de direcciones. Las operaciones de escritura en objetos compartidos son almacenadas en este espacio compartido de direcciones. Las operaciones de escritura en objetos compartidos son dirigidas al administrador del procesador, conteniendo los procesos de la copia primaria, los procesos de usuario directamente pueden leer las copias locales, aunque pueden usar bloques temporales. Cada proceso administrador contiene hilos múltiples de control. Un hilo se comunica con el administrador remoto. Los hilos restantes son creados dinámicamente para manipular las operaciones de escritura. Así es que las operaciones de escritura múltiples para objetos diferentes pueden estar en marcha simultáneamente; las operaciones de escritura para el mismo objeto son serializados. En el momento de recibir una petición de un proceso del usuario W (posiblemente remoto) para realizar una operación "*write (X, val)*", el administrador de X crea un hilo nuevo de control para manipular la petición:

```
receive write-req (X, Val) from W →
        fork handle_write (X, Val, W);
```

El proceso “*handle_write*” se define a través del siguiente algoritmo:

```
process handle_write (X, Val, W);
begin
    set write-lock on X;
    store val in X;
    let S = set of processors having a copy of X;
    # first phase
    for all P ∈ S do
            send update-and lock (X, Val) to manager of P;
    for i := 1 to |S| do
            receive ack;
    # second phase
    for all P ∈ S do
            send unlock (X) to manager of p;
    unlock X;
    send ack to w;
end;
```

El proceso emite la petición de escritura y espera hasta que recibe una confirmación. Un administrador responde como sigue a los mensajes de administradores remotos:

```
    receive update-and-lock (X, Val) from P →
            set write-lock on local copy of X;
            store Val in local copy of X;
            send ack to P;
```

receive unlock(X) →

unlock local copy of X;

El protocolo de actualización de 2 fases garantiza que ningún proceso usara el valor nuevo de un objeto mientras otro proceso todavía usa el valor anterior. El valor nuevo no es usado hasta que haya iniciado la segunda fase. Cuando la segunda fase comienza, todas las copias contienen el nuevo valor. Las operaciones de escritura simultáneas en el mismo objeto son serializadas por medio del bloqueo de la copia primaria.

La siguiente operación de escritura puede comenzar antes que todas las copias secundarias estén desbloqueadas. Una petición nueva para la actualización y bloqueo de una copia secundaria no es servido hasta que el mensaje de desbloqueo generado por la escritura previa ha sido manejado (recordar que los mensajes punto a punto son recibidos en el mismo orden en el que fueron enviados).

Sí un objeto tiene N copias secundarias, entonces le lleva 3 * N mensajes actualizar todas estas copias. Leer un objeto remoto toma 2 mensajes (una petición, una respuesta). Así es que los objetos sólo deberían ser duplicados en procesadores que dan lectura a un objeto al menos dos veces antes que cambien de nuevo. Éste puede ser determinado (o estimado) dinámicamente. El protocolo fácilmente puede ser optimizado en un protocolo de actualización de 1 fase, sí un objeto tiene una única copia secundaria.

Para objetos pequeños (como un entero) que varían frecuentemente, puede ser más eficiente invalidar copias cuando el objeto varía y replicarlos por referencia. La primera operación de lectura después de uno de escritura, el objeto de un procesador remoto crea su una copia local. Subsiguientemente lee su copia local, hasta que esté sea invalidado por una modificación para el objeto.

4.1.4.4. Implementación de mensajes con multidifisión confiable

El protocolo de actualización de 2 fases adecuadamente soluciona el problema de consistencia, aunque al precio de alguna sobrecarga en la comunicación. La semántica prevista por la implementación estrechamente se parece a esos de variables compartidas. Sí una la operación de escritura completa en el tiempo T_w, entonces lee las operaciones que se emitieron en el tiempo $T_r > T_w$ las cuales devuelven el valor nuevo. El ordenamiento temporal estricto, sin embargo, no es un requisito necesario para programación de sistemas análogos a *MIMD*, en los cuáles los procesadores ejecutan asincrónicamente. Los procesadores en tales sistemas no están sincronizados por relojes físicos. Cada proceso secuencial en un sistema asincrónico, realiza una secuencia de pasos computacionales: C_0, C_1, ..., C_i,..... Dentro de un proceso simple, estos pasos son completamente ordenados; C_n ocurre después de C_m, si y sólo n > m. No hay un orden total entre los pasos computacionales entre diferentes procesos. Hay sólo un orden parcial, inducido por interacciones explícitas (como el envío de un mensaje o ajustar y probar variables compartidas).

Esta falta de orden total permite una implementación de los objeto-datos compartidos, ligeramente flexibiliza la semántica sin afectar al modelo programación subyacente. Suponga que el proceso P_1 ejecuta "*write (X, Val)*" y el proceso que P_2 ejecuta "*read (X)*". Sí no hay relación de precedencia entre estas dos acciones (por ejemplo, ni lo uno ni lo otro, uno de ellos antecede al otro en orden parcial), entonces el valor que lee P_2 puede ser ya sea el viejo valor de X o el valor nuevo. Aun si, físicamente, la escritura es ejecutada antes de ser leída, entonces la lectura todavía pueden devolver el valor antiguo. La diferencia principal con los sistemas que permiten las operaciones de lectura devolver datos antiguos arbitrarios (dañados), es que el modelo soporta un ordenamiento lógico coherente de acontecimientos, como definido implícitamente en el programa desde su inicio.

En un sistema distribuido que soporta sólo mensajes punto a punto, un ordenamiento lógico coherente es difícil de conseguir, porque el mensaje enviado para los diferentes destinos puede llegar con retrasos arbitrarios. Algunos sistemas distribuidos, dan soporte al hardware para enviar un mensaje simple a varios destinos simultáneamente. Más precisamente, en sistemas que soportan multidifusión de forma confiable e indivisible, presentan las siguientes propiedades:

i) Un mensaje es enviado confiadamente de una fuente a un conjunto de destinos.
ii) Sí dos procesadores simultáneamente difunden dos mensajes (por ejemplo, m1 y m2), entonces todos los destinos primero reciben m1 primeramente, o todos reciben m2 primeramente.

Con la facilidad de multidifusión, podemos implementar un solo protocolo de actualización. Una petición "*write (X, val)*" es manejada como sigue por el administrador de X:

receive write-req(X, val) from W $\rightarrow$

set write-lock on X;

store Val in X;

let S = set of processors having a copy of X;

multicast update(X, val) to manager of every P $\in$ S;

unlock X;

send write-ack(W) to manager of W;

Después de que el mensaje del requerimiento de escritura haya sido manipulado, el acuse se envía al administrador de W (el proceso que emitió la petición). El administrador se lo reenvía a W. Esto garantiza que la copia local de X en el procesador de W, ha sido actualizado cuando W reanuda su ejecución. El administrador puede ser un proceso de una sola tarea en esta implementación. Un administrador manipula la solicitud de escritura entrante, la actualización, y el mensaje del acuse de recepción de escritura, en la orden a ellos les fueron enviados. Un administrador conteniendo una copia secundaria responde como sigue al mensaje de los administradores remotos:

Receive update (X, val) $\rightarrow$

Set write-lock on local copy of X;

Store val in local copy of X;

Unlock local copy of X;

Receive write-ack (W) →

Send ack to W;

Sí un procesador P lee un valor nuevo de un objeto X, y actualiza el mensaje para X conteniendo ese valor, también se ha enviado a todos los otros procesadores. Los otros procesadores que no pudieron haber manipulado el mensaje todavía, solamente harán eso antes de que manipulen cualquier otro mensaje. Cualquier cambio en objetos compartidos iniciados por P, será observado por otros procesadores después de aceptar el valor nuevo de X. Los problemas como ésos en la figura 24, no ocurren.

4.1.4.5. Implementación de mensajes con multidifisión inconfiable

Una forma beneficiosa para construir un sistema distribuido es conectar una colección de micro computadoras por una red de área local. Tales sistemas son fáciles para construir y fácil de extenderse. Muchos sistemas operativos distribuidos han sido diseñados con este objetivo. Muchas LAN's tienen soporte del hardware para hacer multidifusión. Una red ethernet, por ejemplo, físicamente envía un paquete a cada computadora en la red, aunque usualmente sólo uno de ellos lee el paquete, no hay diferencia en el tiempo de transmisión entre un mensaje multidifusión y de punto a punto.

Infortunadamente, la multidifusión en una *LAN* no es completamente confiable. Ocasionalmente, el paquete de red se pierde. Peor aun, uno o más receptores pueden no tener espacio de *buffer* cuando el paquete llega,

así que un paquete puede ser entregado en una parte de los destinos. En la práctica, la multidifusión es altamente confiable, aunque menos del 100%. La multidifusión inconfiable puede ser confiable, agregando un protocolo extra. Tal protocolo tiene una alta sobrecarga de comunicación y puede resultar en multidifusión que no sea indivisible (como ya se indico).

El algoritmo básico es el mismo que multidifusión confiable. Cuando una X variable compartida sea actualizada, algunos (o todos) procesadores que contengan una copia secundaria de X pueden no poder recibir el mensaje de *update(X, val)*. Estos continuarán usando el valor anterior de X. Esto no es desastroso, mientras el ordenamiento parcial (lógico) de acontecimientos es obedecido. Para garantizar un ordenamiento consistente, los procesadores que fracasaron en la recepción del mensaje de *update (X, val)*, deben ser detectados antes que el manejador adquiera el siguiente mensaje de actualización que lógicamente debería llegar después del mensaje de X.

Esto se está realizado como sigue: El mensaje de actualización es multidifundido a todos los procesadores que participan en el programa, no sólo para los procesadores que contienen una copia secundaria. Cada procesador cuenta el número de mensajes de actualización que envía. Este número es llamado *mc-count*. Cada procesador registra el *mc-count* de todos los procesadores. Este número sea almacenado en un vector llamado *mc-vector* (inicializado con ceros). Para el procesador P, mc-vector[P] siempre contiene el valor correcto del *mc-count* de P; las entradas para otros procesadores pueden ligeramente desactualizadas.

En cualquier momento que un mensaje es multidifundido, este envía su propio *mc-vector* como parte del mensaje. Cuando un procesador Q recibe un mensaje multidifundido de P, este incrementa la entrada para P en su propio *mc-vector* y entonces compara este vector con el *mc-vector* contenido en el mensaje. Sí una entrada R en su propio vector es menor respecto de la entrada correspondiente en el mensaje, entonces Q ha perdido un mensaje multidifundido del procesador R. Q actualiza la entrada para R en su propio vector. Como Q no conoce cuál variable debería haber estado actualizada por el mensaje de R, Q temporalmente invalida las copias locales de todas las variables que tienen su copia primaria del procesador R. Este envía (confiadamente) el mensaje de punto a punto al administrador de R, solicitando los valores actuales de estas variables. El mensaje de respuesta de R también contiene *mc-vector*, y experimenta el mismo procedimiento que respecta al mensaje multidifundido. Hasta que las copias están al día de nuevo, las operaciones locales leídas de estas copias se bloquean.

Está dentro de lo posible que la actualización perdida del mensaje permanezca no-descubierta durante algún tiempo. Suponga que el procesador Q pierde un mensaje de actualización para una variable Y del procesador R y entonces recibe un mensaje de actualización para X del procesador P. Si P también perdiera el mensaje de R, entonces la entrada para R en el *mc-vector* de P y Q estará acorde (aunque ambas estén equivocadas) y la copia de X será actualizada. Sin embargo, como P contuvo el valor anterior de Y cuando actualizó a X, el valor nuevo de X no depende del valor nuevo de Y, así es que es coherente actualizar X.

Sí un proceso pierde un mensaje de actualización para X, entonces este fallo eventualmente será detectado mientras maneja subsiguientes mensajes. La suposición es que habrá subsiguientes mensajes. Esta suposición no necesita ser cierta. Por ejemplo, un proceso puede colocar una variable de bandera compartida y esperar que otro proceso responda. Si estos otros procesos perdieron el mensaje de actualización de la bandera, entonces el sistema muy bien puede venir y deshecharlo. Para prevenir esto, se emite una mensaje ficticio de actualización que es generado periódicamente, el cuál no actualiza cualquier copia, pero justamente causa que le *mc-vector* sea revisada. La implementación delineada citada anteriormente tiene una ventaja considerable: lleva un solo mensaje de actualización a cualquier número de copias, con la condición de que el mensaje sea entregado en todos los destinos. Ésta es una pena severa en la perdida de los mensajes. En una LAN moderna altamente confiable, se espera que no ocurra infrecuentemente. La implementación también tiene varias desventajas. El mensaje de actualización es enviado a cada procesador. Cada mensaje que contenga información adicional (el *mc-vector*), debe ser verificado por todos los procesadores receptores. Para un número limitado de procesadores, por ejemplo 32, se toma esta sobrecarga como aceptable. El protocolo puede ser integrado con el protocolo de actualización de 2 fases.

4.1.5. Un lenguaje de programación basado en dato-objetos compartidos

A continuación, se describe la programación de un lenguaje llamado *Orca*, el cual esta basado en dato-objetos compartidos. A diferencia de otros lenguajes de programación paralelos, *Orca* esta orientado para la

programación de aplicaciones que para sistemas de programación. El paralelismo de *Orca* esta basado en la creación dinámica de procesos secuenciales. Los procesos se pueden comunicar indirectamente, a través de los dato-objetos compartidos. Un objeto puede ser compartido por medio del paso de parámetros compartidos en la creación de un nuevo proceso.

4.1.5.1. Definiciones de tipo de objetos

Un objeto es una instancia de un tipo objeto, quien es esencialmente un tipo de dato abstracto. Una definición de tipo de objeto consiste de una parte de especificación y una parte de implementación. La parte de especificación define una o más operaciones en los objetos que son del mismo tipo. La declaración de un tipo de objeto puede ser la siguiente:

```
Object specification IntObject;
    Operation value (): integer;          # current value
    Operation assign(val: integer);       # assign new value
    Operation min(val: integer);
                    # set new to minimun of current value and “val”
    operation max(val: integer)           # idem for maximum
end;
```

La parte de implementación contiene los datos del objeto, el código de inicialización de los datos para las nuevas instancias (objetos) del tipo, y el código de implementación de las operaciones. La implementación del código

de las operaciones son las que dan acceso a los datos internos del objeto. Los objetos pueden ser creados y operados como sigue:

```
Myint: IntObject;              # create an object
…
Myint$assign(83);              # assign 83 to myint
…
x := Myint$value();            # read the value of myint
```

La implementación del tipo de objeto IntObject se muestra a continuación:

```
Object implementation IntObject;
    X: integer;            # local data
    Operation value(): integer;
    Begin
        Return X;
    End
    Operation assign(val: integer);
    Begin
        X := val;
    End

    Operation min(val: integer);
    Begin
        If val < X then X := val; fi;
    End
```

```
    Operation max(val: integer);
        Begin
                If val > X then X := val; fi;
        End
Begin
        X:= 0;          # inicialización interna de los datos
End;
```

4.1.5.2. Sincronización

El acceso para los datos compartidos está automáticamente sincronizado. Todas las operaciones definidas en la parte de especificación son indivisibles. Sí dos procesos simultáneamente invocan a los procedimientos x$min (UNO) y x$min (UNO), entonces el valor nuevo de X es el mínimo de A, B; y el viejo valor de X. Por otra parte, una secuencia de operaciones, como

```
    If A < X$value() then
        X$assign(A);
    fi
```

no es indivisible.

Esta regla define cuáles acciones son indivisibles y cuales no en ambos casos, es decir, fácil de entender y ser flexible: Las operaciones simples son indivisibles y las secuencias de operaciones no lo son. El conjunto de operaciones puede ser ajustado a la medida, de acuerdo a las necesidades de una aplicación específica, definiendo operaciones simples hasta las más complejas, según sea necesario.

Para la sincronización de condición, las operaciones que generan los bloqueos pueden estar definidas. Una operación de bloqueo consiste de uno o más comandos protegidos:

Operation name (parameters);
Begin
    Guard $expr_1$ do $steents_1$; od;
    Guard $expr_2$ do $steents_2$; od;
    ….
    Guard $expr_n$ do $steents_n$; od;
End;

Las expresiones deben ser libres de efectos secundarios en expresiones lógicas (booleanas). Las operaciones iniciales bloquean (suspenden) hasta que al menos uno de sus guardas es valuado a verdadero. Enseguida, un guarda en verdadero es seleccionado no deterministicamente, y su secuencia de sentencias es ejecutada. Por

ejemplo, un tipo *IntQueue* con una operación bloqueadora *remove-head* puede ser implementado como puede sigue:

```
Object implementatin IntQueue;
    Q: list of integer;              # internal representation
    Operation append(X: integer);
    Begin
            Add X to the end of Q;
    End
    Operation remove_head(): integer;
            R: integer;
    Begin
            # wait until queue not empty
            # then get head element
            guard Q not empty do
                    R:= first element of Q;
                    Remove R from Q;
                    Return R;
            Od
    End;
Begin
    Q := empty;
End;
```

Una invocación de *remove_head* suspende hasta que la cola no esté vacía. Si la cola es inicialmente vacía, entonces el proceso espera hasta que

otro proceso agrega un elemento para la cola. Si la cola contiene solo un elemento y varios procesos tratan de ejecutar la declaración simultáneamente, se suspenderán hasta más elementos sea agregados a la cola.

4.2. Distribución heterogenia de memoria compartida

4.2.1. Abstracción

La heterogeneidad en los sistemas distribuidos es progresivamente un hecho de la vida, debido a la especialización del equipo de cómputo. Es altamente deseable poder integrar computadoras heterogéneas en un ambiente coherente de computación para soportar aplicaciones distribuidas y paralelas, a fin de que las fuerzas individuales de las diferentes computadoras puedan ser explotadas conjuntamente. La memoria distribuida compartida, es un modelo de alto nivel, altamente transparente para comunicaciones de ínterprocesos en sistemas distribuidos, y lo convierte en un vehículo prometedor para lograr tal integración.

La memoria compartida distribuida es un modelo de comunicación de interprocesos en sistemas distribuidos. En el modelo *DSM*, el proceso se ejecuta en computadoras separadas ganando acceso a un espacio compartido de direcciones a través de operaciones normales de carga y almacenamiento, y otras instrucciones de acceso de memoria. El sistema fundamental *DSM* provee a sus clientes un espacio de direcciones de

memoria compartida coherente. Cada cliente puede acceder cualquier posición de memoria en el espacio compartido de direcciones en cualquier momento y puede ver el último valor escrito por cualquier cliente. La ventaja primaria de *DSM* es la abstracción simple que se provee para el programador de aplicaciones. La abstracción es algo que el programador ya entiende adecuadamente, desde que el protocolo de acceso está consistente con los datos de acceso de aplicaciones muy secuenciales. El mecanismo de comunicación está enteramente oculto para el programador de la aplicación, a fin de que él no tenga que estar consciente del movimiento de datos entre procesos, y las estructuras completas de datos pueden ser pasadas por referencia, requiriendo proceso de empacamiento y desempacamiento.

En principio, el rendimiento de las aplicaciones que usan *DSM* se espera que sea peor que el paso de mensajes directo de los usuarios, dado que el paso de mensajes es una extensión directa del mecanismo fundamental de comunicación, y adicionalmente *DSM* es implementado como una capa separada de la aplicación y del sistema de paso de mensajes. Sin embargo, varias implementaciones de algoritmos DSM han demostrado que DSM puede ser competitivo para el paso de mensajes en términos del rendimiento para muchas aplicaciones. Algunas aplicaciones han demostrado que DSM puede ser superior en rendimiento. Esto es posible por dos razones: Primero, para muchos algoritmos *DSM*, los datos son movidos entre servidores en bloques grandes. Por consiguiente, si la aplicación muestra un grado razonable de localidad en su acceso, entonces la sobrecarga de comunicación es amortizada sobre múltiples acceso de memoria, reduciendo completamente requisitos completos de comunicación. Segundo, muchas

aplicaciones (distribuidas) se ejecutan en fases, dónde cada fase de calculo es precedida por una fase de intercambio de datos. El tiempo necesitado para el intercambio de datos entre fases es a menudo dictada por el rendimiento específico de cuellos de botella existentes de comunicación.

En el contraste, los algoritmos *DSM* típicamente mueven los datos a petición en su estado actual siendo accesados, eliminando la fase de intercambio de datos, extendiendo la carga de comunicación sobre un período de tiempo más largo, y permitiendo un grado mayor de concurencia. Se podrían replicar los métodos citados anteriormente accesando los datos pudieron estar programados usando paso de mensajes, en el efecto imitando a DSM en las aplicaciones individuales. Tal programación para la comunicación, sin embargo, usualmente representa esfuerzo sustancial además para la implementación de la aplicación misma.

El algoritmo mas extensamente conocido para implementar a *DSM* es gracias a *Li*. En el algoritmo de *Li*, conocido como *SVM*, el espacio compartido de la dirección está subdividido en páginas, y las copias de estas páginas son distribuidas entre los servidores, seguido de un protocolo de múltiples lectores / un escritor simple (*MRSW*). Las páginas que están marcadas de sólo lectura pueden estar replicadas y puede residir en la memoria de varios servidores, pero una página que esta siendo escrita sólo puede residir en la memoria de un servidor. Una ventaja del algoritmo de *Li* es que fácilmente se integra con la memoria virtual del sistema operativo del servidor. Si una página de memoria compartida es contenida localmente en un servidor, esta puede ser mapeada en el espacio de direcciones virtuales

de la aplicación en ese servidor y por consiguiente puede ser accesada por instrucciones normales de acceso a la memoria. El acceso a la página no es contenida localmente si se esta ejecutando una falla de página, pasando el control al administrador de fallas.

El administrador de fallas se comunica con los servidores remotos para obtener una copia valida de la página antes de ser mapeada en el espacio de direcciones de la aplicación. Cuandoquiera, una página puede ser migrada de cualquier servidor, y es removida del espacio de direcciones local la cual es colocada en el mismo lugar. Similarmente, cuandoquiera un servidor intenta escribir en una página la cual no esta localmente marcada como escribible, se genera una falla de página y el administrador de fallas se comunica con otros servidores (después de obtener una copia de la página, sí es necesario) para invalidar todas las copias en el sistema pero antes la copia local es marcada como disponible y permitiendo que el proceso que falla continué. Este protocolo es similar al algoritmo de escritura inválida usado para la consistencia del *cache* en multiprocesadores de memoria compartida, exceptuando que la unidad básica en la cual la operación ocurre es en la página en lugar del *cache*. Los administradores de comunicación de *DSM* se comunican con los administradores *DSM* de memoria, el cual corre uno en cada servidor. Cada administrador de memoria *DSM* manipula una tabla de páginas virtuales local, de acuerdo al protocolo *MRSW*, manteniendo el rastro de la localidad de las copias de las de cada página que cada administrador *DSM* tiene; pasando las páginas por requerimiento de fallas.

Para la programación de aplicaciones distribuidas y paralelas, la memoria distribuida compartida puede ocultar la complejidad de las comunicaciones de las aplicaciones cuando es un conjunto homogéneo de servidores. Los sistemas *DSM* homogéneos logran una transparencia funcional completa, en el sentido que los programa escritos para sistemas multiprocesador de memoria compartida puedan correr en sistemas *DSM* sin sufrir cambio.

El hecho que no exista memoria física compartida pueda ser completamente oculto al programador de aplicaciones, así como el hecho, de la transferencia de los datos y el paso de mensajes entre servidores. La transparencia del rendimiento puede solo ser conseguida con cierto grado de limitantes, ya que la localización física de los datos que son accesados afectan el rendimiento de la aplicación, mientras que en un sistema multiprocesador con acceso de memoria uniforme (*UMA*), el costo de acceso de los datos no es afectado por la localidad en la memoria compartida. En el caso del protocolo MRSW, sí una página no esta disponible en el servidor local cuando se necesita accesar, esta tiene que traerse de otro servidor, causando un retraso adicional.

4.2.2. Heterogeneidad

Los componentes de una aplicación distribuida comparten memoria directamente, están más estrechamente acoplados que cuando los datos están compartidos a través de *RPC* o un sistema de archivos distribuidos. Por esta razón, es más difícil extender un sistema *DSM* en un ambiente heterogéneo.

4.2.2.1. Conversión de datos

Los datos pueden ser representados de forma diferente entre los servidores dependiendo de la arquitectura de las maquinas, los lenguajes de programación para las aplicaciones y sus compiladores. Para los tipos de datos simples como los enteros, el orden de los bytes puede ser diferente. Para los números de punto flotante, la longitud de la mantisa y los campos de los exponentes, pueden cambian de posición. Para estructuras más complejas, como registros y vectores, la alineación y el orden de los componentes en la estructura de datos puede diferir de un servidor a otro. Un ejemplo sencillo, presenta dos tipos de datos, un vector de cuatro caracteres y un entero, el primero en orden big-endian y el segundo en orden little-endian, como se muestra en la tabla III:

Tabla III. Ordenamiento big-endian y little-endian

byte	Big-Endian		Little-Endian	
	int	char array	int	char array
I	MSB	‘J’	LSB	‘J’
i + 1		‘O’		‘O’
i + 2		‘H’		‘H’
i + 3	LSB	‘N’	MSB	‘N’

MSB=Most Significant Byte LSB=Least Significant Byte

Este ejemplo ilustra los diferentes tipos de dependencia en la representación de los datos que pueden surgir entre los diferentes servidores.

Compartir datos entre servidores heterogéneos quiere decir que la representación física de los datos tendrá que convertirse cuando los datos sean transferidos entre servidores de tipos diferentes. En el caso más general, la conversión de datos sólo tendrá una sobrecarga en el tiempo de ejecución, sino que también puede ser imposible debido al contenido poco equivalente (los bits perdidos de precisión en los números de coma flotante, y la desigualdad en los rangos de representación).

Esto puede representar una limitación potencial para *HDSM* en algunos sistemas y aplicaciones. La pregunta apunta a las necesidades de ser direccionada para un conjunto específico de servidores y de lenguajes de programación, la conversión de datos puede ser ejecutada para todos o la mayoría de tipos de para datos para formar un sistema *HDSM* (un sistema que soporte un gran numero aplicaciones reales).

4.2.2.2. Administración de tareas

Para soportar un espacio de direcciones compartidas, la memoria compartida distribuida usualmente marcha de común acuerdo con el sistema de tareas, el cual permite la ejecución de tareas múltiples para compartir el mismo espacio de direcciones. Tal combinación hace la programación de las aplicaciones paralelas particularmente sencillas. En un ambiente heterogéneo, las instalaciones para la administración de tareas, incluyen las primitivas de creación de tareas, finalización, planificación y sincronización, todos pueden ser de diferente tipo en los servidores, si existen en todos.

La migración de tareas de un servidor para otro dentro un sistema homogéneo *DSM* es usualmente sencillo, dado que el contexto mínimo se conserva para las tareas. Típicamente, la pila de tareas es asignada en el espacio compartido de direcciones, así que la pila no necesita ser movida explícitamente. El descriptor, o bloque de control de tareas (*TCB*), constituye una pequeña cantidad de memoria que necesita ser movida en el tiempo de migración.

En un sistema heterogéneo *DSM*, sin embargo, la migración de tareas es mucho más difícil. Las imágenes binarias del programa son diferentes, así es más difícil de identificar que puntos son equivalentes en la ejecución en los binarios; por ejemplo, en diferentes partes del binario la ejecución puede ser suspendida y reiniciada posteriormente por otro servidor de diferente tipo, de tal forma que la ejecución no se ve afectada. Similarmente, los formatos de las pilas de la tarea son igualmente diferentes, dada la diferencia de arquitectura, lenguaje y compilador; por consiguiente, la conversión de la pila al momento de migrar se dificulta, e incluso puede llegar a ser imposible.

Mientras se tenga presente que la migración de tareas presenta todavía otra limitación para *HDSM*, su significado es debatible para dos razones: Primero, en *HDSM*, las tareas pueden ser creadas e iniciadas en servidores remotos de cualquier tipo, reduciendo así la necesidad de la migración dinámica de la tarea. Segundo, la migración entre servidores del mismo tipo es todavía sencilla de alcanzar en *HDSM*, y para muchas aplicaciones, esto puede ser todo lo que es requerido. Para una aplicación que se ejecuta en una estación de trabajo y con un conjunto de servidores (homogéneos) de

cómputo, por ejemplo, sus tareas pueden libremente migrar a los servidores de cómputo a balancear su carga.

4.2.2.3. Tamaño de página

La unidad de administración y transferencia de datos en *DSM* se llama bloque, al cual le llamaremos pagina *DSM*. En un sistema homogéneo *DSM*, una pagina *DSM* tiene usualmente el tamaño de la página nativa de la memoria virtual (*VM*), por lo que la unidad de administración de memoria del hardware (*MMU*) puede ser usada para disparar fallas de pagina *DSM*. En un sistema heterogéneo *DSM*, los servidores pueden tener diferentes tamaños de pagina de memoria virtual (*VM*), presentando ambos sistemas una alta complejidad en el algoritmo de coherencia de paginas y oportunamente el control de la granularidad de los datos compartidos.

4.2.2.4. Acceso uniforme en los archivos

Un sistema *DSM* que soporta una aplicación que puede ejecutarse en un número de servidores puede beneficiarse con la existencia de un sistema de archivos distribuido, el cual permite a las tareas abrir los archivos y ejecutar acceso de *I/O* de una manera uniforme. Mientras éste sea probablemente el caso en un sistema moderno y homogéneo; un sistema múltiple de archivos distribuidos incompatible puede existir en servidores heterogéneos, debido a la multiplicidad de los protocolos en el sistema de archivos distribuidos. Una interfaz uniforme de accesos de archivo, extendiéndose sobre nombres y operaciones de archivo, debería ser provisto para una aplicación *HDSM*.

Una posibilidad es escoger uno de los sistemas de archivo como el estándar para el sistema. Se logra también definir una estructura independiente del sistema de archivo, y hacer el sistema de archivos distribuidos nativo emulado.

4.2.2.5. Lenguaje de programación

El sistema que los lenguajes de programación usan en los servidores heterogéneos puede ser diferente. Esto significa que la implementación equivalente múltiple de un sistema *HDSM* puede tener que estar hecha en los lenguajes diversos. Sin embargo, la ejecución de aplicaciones en *HDSM* no debería ser afectado por el lenguaje usado para implementarlo, bastante con una interfaz de aplicación funcionalmente equivalente es soportado por *HDSM* en todos los servidores. Sí un lenguaje de programación común de aplicaciones está disponible en todos los servidores, entonces el mismo programa sería utilizable en los servidores (con recompilación). En caso contrario, las implementaciones múltiples de una aplicación tendrían que estar escritas, aumentando las dificultades del *HDSM* sustancialmente.

4.2.2.6. Comunicación interservidor

La realización de *HDSM* requiere la existencia de un protocolo común de comunicación entre los tipos diferentes de servidores involucrados. Este requisito no es particular para *HDMS*, sin embargo, algún protocolo común de transporte debe existir para el servidor que se comunica en todo caso. La

disponibilidad de los protocolos *OSI* y *TCP/IP* en la mayoría de sistemas operativos, la comunicación entre servidores es progresivamente factible.

4.3. Control de acceso puntual de granularidad fina para DSM

4.3.1. Abstracción

La computación paralela se está volviendo ampliamente disponible gracias al surgimiento de redes de estaciones de trabajo, convirtiéndose en las minicomputadoras paralelas del futuro. Desgraciadamente, los sistemas actuales directamente soportan sólo comunicación de paso de mensajes. La memoria compartida está limitada a los sistemas basados en páginas, lo cuál no es secuencialmente consistente por lo que puede funcionar pobremente en presencia de datos compartidos con cierto grado de granularidad. Estos sistemas carecen de control de acceso granular, una característica de importancia crucial de las maquinas de memoria compartida. El control de acceso es la habilidad para restringir selectivamente lecturas y escrituras a las regiones de memoria. En cada referencia de memoria, el sistema debe realizar una búsqueda para determinar si las referencias establecidas para datos en la memoria local, se encuentran en una condición apropiada. Sí los datos locales no satisfacen la referencia, entonces el sistema debe invocar una acción de protocolo para recuperar los datos deseados para el nodo local. Esto implica que la combinación de realizar una búsqueda en una referencia de memoria condicionalmente, invoca una acción de control de acceso. La granularidad de control de acceso se define como la cantidad menor de datos que pueden estar independientemente controlados, también

llamado como tamaño de bloque. El control de acceso es llamado granularidad fina sí su granularidad es similar a un bloque de memoria *cache* de hardware (de 32 a 128 *bytes*).

Las máquinas de memoria compartida actuales logran un alto rendimiento por implementaciones intensivas en hardware utilizando un control de acceso de granularidad fina. Sin embargo, este hardware adicional impondría una carga imposible en el mercado de la estación de trabajo y de la computadora personal. La memoria compartida eficiente en clusteres de estas máquinas requiere métodos con costos bajos o sin costo, para uso del control de acceso de granularidad fina.

4.3.2. Alternativas de control de acceso

El control de acceso de granularidad fina ejecuta una búsqueda en cada referencia de memoria, y basado en el resultado de la búsqueda, condicionalmente se ejecuta una acción. La localización referenciada puede estar en uno de los siguientes tres estados: *Read-Write, ReadOnly* o *Invalid*. El programa cargado y almacenado, tiene la siguiente semántica:

```
Load(address) =
    If (lookup(address) ∉ { ReadOnly, ReadWrite })
        Invoke-action (address)
    Perform-load (address)
```

```
Store(address) =
    If (lookup(address) ≠ ReadWrite)
        Invoke-action (address)
    Perform-store (address)
```

El control de acceso de granularidad fina puede ser implementado de varias formas. La búsqueda y la acción pueden ser ejecutadas ya sea por software, hardware o una combinación de ambas. Estas alternativas pueden presentar diferente rendimiento, costo y características de diseño. Esta clasificación se basa en la técnica de control de acceso en donde la búsqueda es ejecutada y en donde la acción es ejecutada. La tabla IV muestra el área del diseño y los lugares que administra.

Tabla IV. Taxonómia de los sistemas de memoria compartida

Búsqueda	Acción		
	Hardware Dedicado	Procesador Primario	Procesador Auxiliar
Software		Orca (object) Blizzard-S	
TLB		IVY (page)	
Cache	Alewife KSR-1	Alewife	
Memoria	S3.mp	Blizzard-E	Flash
Snoop	DASH		Typhoon

4.3.2.1. Donde se ejecuta la búsqueda

Ya sea por software o hardware, se puede ejecutar una revisión de acceso. El software de búsqueda evita el alto costo y el diseño del hardware incurriendo en una sobrecarga mínima en cada búsqueda. El hardware típicamente no incurre en sobrecargas cuando la búsqueda no requiere una acción. La búsqueda de hardware puede tomar lugar en cualquier nivel de la jerarquía de la memoria – TLB, controladora de *cache*, controlador de memoria o en una controladora espía adicional. Sin embargo, por razones económicas y de rendimiento, la mayor parte del hardware delega esta responsabilidad al microprocesador.

4.3.2.1.1. Por software

El código en una búsqueda por software verifica la estructura de datos de la memoria principal para determinar el estado de un bloque antes de ser referenciado. Una codificación cuidadosa y el uso liberal de memoria hacen esta búsqueda razonablemente rápida. Un análisis estático puede detectar y eliminar potenciales pruebas redundantes. Sin embargo, la falta de sincronización en programas paralelos hace más difícil predecir si un bloque de cache permanece accesible entre dos instrucciones.

4.3.2.1.2. TLB

El hardware estándar de traducción de direcciones provee control de acceso, solo que la granularidad se da al nivel de página.

4.3.2.1.3. Controlador de cache

Para detectar fallas en la memoria *cache* del *hardware*, se utilizan controladoras que determinan cuando invocar una acción de protocolo. En algunos sistemas, un directorio local es consultado en las fallas en las direcciones físicas locales para determinar sí una acción de protocolo es requerida. Los fallos en direcciones físicas remotas siempre invocan una acción. En algunas arquitecturas, cualquier referencia que falla en ambos niveles de memoria cache, requieren una acción de protocolo.

4.3.2.1.4. Controlador de memoria

Si el sistema puede garantizar que la memoria *cache* del hardware del procesador nunca contenga bloques inválidos, y que los bloques que son extraídos son almacenados en memoria *cache* en una condición de sólo lectura, el controlador de memoria puede realizar la búsqueda de las fallas en la memoria *cache*. Mientras sea efectivo, este método tiene varios defectos. El acceso sólo de lectura es implementado con protección a nivel de página, así es que el almacenamiento puede obtener una trampa innecesaria de protección. Esto conlleva la modificación de los valores *ECC* en una operación complicada y privilegiada.

4.3.2.1.5. Bus espía

Cuando un procesador soporta una coherencia basada en un esquema de bus, un agente que espía el bus por separado, puede realizar una búsqueda similar ejecuta por un controlador de memoria. Los fallos locales pueden requerir un protocolo de acción basado en la condición local del directorio, y las fallas remotas siempre invocan una acción.

4.3.2.2. Donde se ejecuta la acción

Cuando una búsqueda detecta un conflicto, se tiene que invocar una acción estipulada por el protocolo de coherencia para obtener una copia accesible del bloque. Al igual que con la misma búsqueda, el hardware o el software, o una combinación de los dos puede realizar esta acción. El software de acción de protocolo puede ejecutar ya sea en el mismo CPU como la aplicación (el procesador primario) o en un procesador separado auxiliar.

4.3.2.2.1. Por hardware

En algunos sistemas, las acciones son delegadas al hardware, el cual provee un alto rendimiento utilizando un protocolo simple. Mientras el hardware ejecuta acciones rápidas, la búsqueda despliega ese protocolo simple no óptimo para todas las aplicaciones o aún para todas las estructuras de datos dentro de una aplicación. El alto costo de diseñar y las restricciones de recurso también hacen que el hardware personalizado sea

poco atractivo. Los protocolos híbridos de software / hardware implementan los casos comunes esperados en hardware e implementan trampas en el sistema, para que el software pueda manejar eventos infrecuentes complejos.

4.3.2.2.2. Procesador primario

La ejecución de acciones en el CPU principal provee flexibilidad de protocolo y evita el costo adicional de hardware personalizado o un CPU adicional.

4.3.2.2.3. Procesador auxiliar

Algunos sistemas logran un alto rendimiento y una gran flexibilidad de protocolo, gracias a que la ejecución es llevada a cabo en un procesador auxiliar dedicado con ese propósito. Esta estrategia evita un cambio de contexto en el procesador primario, y puede ser crucial si el procesador primario no puede recuperarse de una excepción retrasada, causada por una búsqueda de control de acceso en los niveles inferiores de la jerarquía de la memoria. Además, un procesador auxiliar puede proveer invocación rápida de código de acción, acopiándose fuertemente de la interfaz de la red, los registros y operaciones especiales.

5. CASO DE ESTUDIO – HP *ALPHASERVER* SERIE GS

A continuación, se describe la arquitectura e implementación básica del *AlphaServer GS320*, un sistema multiprocesador de memoria de acceso no uniforme de *cache* coherente, desarrollado por *Hewlett-Packard* (*HP*). La arquitectura del *AlphaServer GS320* es específicamente dirigida a sectores de multiprocesamiento de la escala mediana, con capacidad de 32 a 64 procesadores. Cada nodo en el diseño consta de cuatro procesadores Alpha 21264, cerca de 32GB de memoria coherente, y un subsistema *I/O* optimizado. La implementación actual soporta un total de 8 nodos, para un total de 32 procesadores. Las características que se utilizaron en el diseño del *AlphaServer GS320* fueron diseñadas para implementar la coherencia y la consistencia eficazmente. El principio orientado para el protocolo basado en directorios es la exactitud de la localización que las cosas relacionadas con el protocolo que corre a alta velocidad sin sobrecargar el flujo de la transacción común. El protocolo exhibe ocupación y conteos inferiores de mensajes comparados a los diseños predecesores, y provee un manejo eficiente de transacciones de 3 saltos. Además, el diseño naturalmente presenta soluciones elegantes para bloqueos muertos, bloqueos vivos, inanición, y equidad. La arquitectura del *AlphaServer GS320* también incorpora un par de técnicas innovadoras que extienden avances previos eficazmente implementados en modelos de consistencia de memoria. Estas técnicas nos permiten generar acontecimientos comprometidos (los cuales se utilizan para propósitos de ordenamiento) con bastante anticipación para formular la respuesta de una transacción. Además, la separación de los eventos comprometidos permite respuestas críticas en tiempo, pasando por

encima de demandas múltiples de entrada sin propiedades de ordenamiento. Si bien es cierto, este diseño específicamente apunta a los servidores de escala mediana, muchas de las mismas técnicas pueden ser aplicadas a protocolos de directorio de escala mayor.

5.1. Abstracción

Los sistemas de multiproceso de memoria compartida han sido un foco principal de estudio y desarrollo por ambos, academia e industria, ocasionando mejoras significativas del diseño durante la última década. Los sistemas de multiproceso basados en espionaje, los cuales dependen de difundir las transacciones de coherencia para todos los procesadores y la memoria, se han movido adecuadamente más allá de los diseños iniciales basados en un solo bus. El *Sun Enterprise 10000*, por ejemplo, extiende esta estrategia hasta 64 procesadores usando buses de direcciones interpaginados de cuatro formas y una barra transversal de datos del 16x16. No obstante, las limitaciones de ancho de banda del protocolo espía, y la necesidad sobre la de actuar sobre todas las transacciones en cada procesador, conlleva diseños sumamente desafiantes especialmente a consecuencia de procesadores con múltiples requerimientos. Los sistemas de multiproceso basados en directorios, los cuales dependen de mantener la identidad de participantes (en el directorio) para evitar la necesidad de transmitirlo, es mas adecuado en diseños mayores. Un ejemplo de esta tecnología es el *SGI Origin 2000* que puede incrementarse hasta varios centenares de procesadores. Además, la naturaleza típica *NUMA* (memoria de acceso no uniforme) de diseños basados en directorios, considerado ser una responsabilidad por algunos, conserva la superioridad de hecho para el

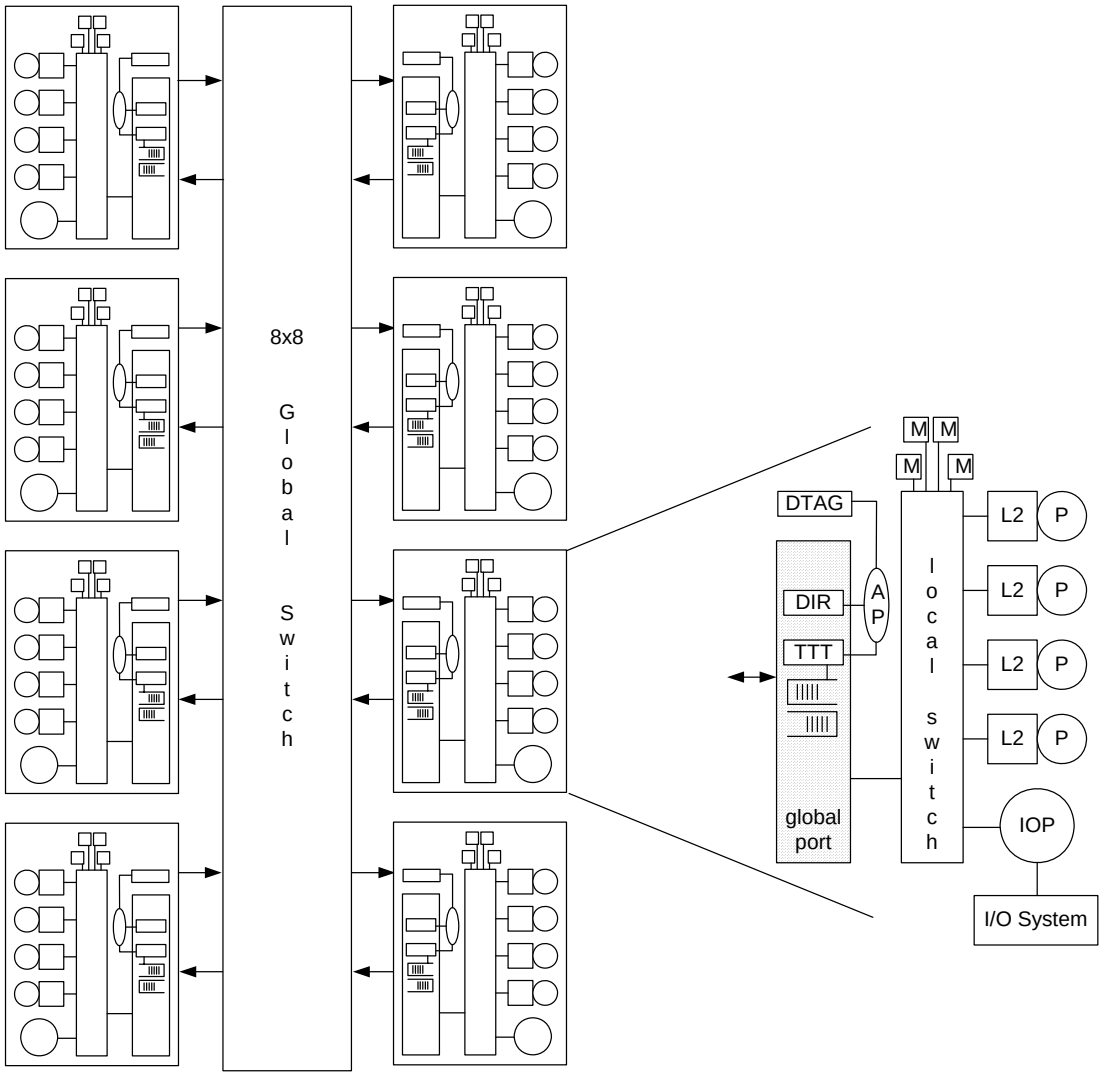
rendimiento mayor, aportando en diseños espías generando provecho de la más baja latencia y una memoria local con un ancho de banda más alto aliviando la necesidad de requerir alguna acción adicional. Las técnicas simples, como la replicación de la aplicación y del código del sistema operativo, pueden proveer mayores ganancias en las cargas de trabajo comercial. Más técnicas informáticas sofisticadas que transparentemente emigran y reproducen páginas también - se ha presentado - siendo realmente efectivas. No obstante, los protocolos existentes del directorio exhiben varias ineficiencias relativas a los protocolos entrometidos, en parte debido a su foco inalterable en los sistemas de gran escala. Por ejemplo, el uso de mensajes diversos de reconocimiento e invocaciones múltiples de protocolo en el nodo principal (en transacciones de 3 saltos), el cual ayuda a ocuparse de las carreras que se levantan debido a la naturaleza de los protocolos distribuidos y las redes dimensionables subyacentes, puede ocasionar de forma no deseada altas ocupaciones de recurso de protocolo.

5.2. Visión general de la arquitectura del *AlphaServer GS320*

Como se muestra en la figura 29, la arquitectura del *AlphaServer GS320* es un sistema multiprocesador de memoria compartida jerárquica que consiste de 8 nodos, a los cuales se les conoce como *quad-processor building block (QBB)*. Cada *QBB* consiste de 4 procesadores con un máximo de 32 GB con una interfaz de *I/O* en un *switch* local. Los *QBB* están conectados en un *switch* global de 8x8. Un sistema totalmente configurado soporta hasta 32 procesadores *Alpha* 21264, 256GB de memoria, 64 buses *PCI* (224 adaptadores *PCI*), con un ancho de banda de memoria de 51.2

GB/s, un ancho de banda global en el *switch* de datos en bisección de 12.8 G/s y un ancho de banda de *I/O* de 12.8 GB/s.

Figura 29. Arquitectura *AlphaServer GS320*



5.2.1. *Quad-Processor Building Block (QBB)*

La figura 29 describe la organización lógica de un bloque construido por cuatro procesadores. El *QBB* es creado alrededor de un *switch* local de 10 puertos. Cuatro puertos son ocupados por procesadores, cuatro por memoria, y uno para la interfase de *I/O* y uno para el puerto global.

El *switch* tiene un ancho de banda de datos de acumulado de 6.4 GB/s, en cada puerto (exceptúe puerto global) a 1.6 GB/s (ancho de banda de transferencia de datos, excluyendo los *bits* de direcciones y códigos de error). El puerto global es usado para conectar al *QBB* a otros nodos, y soporta 1.6 GB/s en cada dirección, para un ancho de banda total por puerto de 3.2 GB/s. El *switch* local no es simétrico; por ejemplo, ninguna de las conexiones es posible entre dos puertos de memoria.

El *QBB* soporta hasta cuatro procesadores *Alpha* 21264, actualmente corriendo en 731 MHz. El procesador *Alpha* 21264 tiene por separado 2 canales asociados de 64 KB en una sola instrucción de *chip* y un *cache* de datos de 64 *bytes* por línea, y un 4MB memoria *cache* externa. Cada procesador soporta hasta 8 requerimientos de memoria extraordinarias y adicionalmente, 7 sobre escrituras extraordinarias. Cada *QBB* también soporta hasta cuatro módulos de memoria, cada uno con capacidad de 1 a 8 GB de memoria *SDRAM*, con una interpolación máxima de 8 formas. Los cuatro módulos proveen una capacidad total de 32 GB y un ancho de banda de memoria total de 6.4 GB/s. La interfaz *I/O* soporta hasta 8 buses *PCI* (64 bits, 33MHz), con soporte para 28 ranuras *PCI*. Esta interfaz soporta una

memoria *cache* pequeña (de 64 entradas, totalmente asociadas) para sacar provecho de localidad espacial para operaciones de memoria generadas por dispositivos *I/O*, y permitiendo hasta 16 operaciones de memoria extraordinarias. Adicionalmente, la interfaz soporta un mecanismo precargado, para permitir acceso a la memoria simultánea aunque los dispositivos *I/O* requieren ordenamiento estricto entre operaciones de memoria.

El *QBB* utiliza un indicador de doble almacen (*DTAG*), para llevar control copias escondidas en reserva dentro del nodo. El *DTAG* es una estructura de datos lógicamente centralizada que mantiene una copia externa de cada uno de indicadores que se apoyan en la memoria *cache* de cuatro procesadores, y sirve de módulo primario para mantener coherencia dentro de un *QBB*. Para mantener coherencia a través de múltiples *QBB's* requiere dos módulos adicionales: El directorio (*DIR*) y la tabla transacciones en transito (*TTT*). El directorio mantiene una entrada de 14 bits por línea de memoria de 64 *bytes* (approx. 2.5% de sobrecarga), el cual incluye un campo de 6 *bits* que identifica uno de 41 propietarios posibles (32 procesadores, 8 interfaces *I/O*, y la memoria), y un campo de 8 bits que es usado como un vector de bits, para mantener la identidad de participantes en el granularidad del *QBB*. La identidad del propietario y los participantes son mantenidos simultáneamente, gracias a que el protocolo permite compartirlos. El uso compartido del vector de bits en el directorio (basándose en la granularidad del *QBB*) junto con el *DTAG* en los nodos, conjuntamente identifica la identidad exacta de la memoria *cache* del procesador de uso compartido. Finalmente, el *TTT* es una tabla asociativa de 48 entradas que mantiene la lista de transacciones pendientes de un nodo. Dos *QBB's* pueden estar

conectados directamente a través de sus puertos globales a formar una configuración de 8 procesadores. Para configuraciones superiores a esta, se requiere el uso del *switch* global.

5.2.2. El *switch* global

El *switch* global posee 8 puertos, cada uno de ellos soporta un ancho de banda de 3.2 Gb/s (1.6 Gb/s en cada dirección), con un ancho de banda global de bisección de datos de 12.8 Gb/s. El GS es implementado como un *buffer* central, y soporta sendas virtuales múltiples para aliviar puntos muertos del protocolo de coherencia. Todos los paquetes entrantes son lógicamente puestos en la cola del búfer central, y sacados de la cola independientemente por planificadores del puerto de salida. Este modelo permite implementar eficazmente una multidifución ordenada para sendas virtuales específicas, donde está es aprovechado al máximo.

5.3. Protocolo optimizado de *cache* coherente

El diseño del protocolo de coherencia de *cache AlphaServer GS320* tiene dos objetivos. El primer objetivo es reducir que las ineficiencias en los protocolos basados en directorios de avanzada tecnología que provienen de sobrecargar flujos de transacción comunes por las soluciones solieron ocuparse de carreras raras de protocolo. El segundo objetivo es explotar el tamaño limitado del sistema, y las propiedades adicionales de ordenamiento que se interconectan, para reducir el número de mensajes de protocolo y las ocupaciones correspondientes de recurso del mismo. Como se puede

observar, hay sinergia entre los diversos mecanismos que se utilizan en el protocolo, ocasionando una implementación simple y eficiente que minimiza las estructuras lógicas de las que se ocupa cuando se presentan carreras raras de protocolo.

El protocolo de coherencia de *cache* en el *AlphaServer GS320* utiliza un protocolo basado en invalidaciones del directorio con soporte para cuatro tipos de requerimientos: lectura, lectura exclusiva, exclusiva (requerimiento de un procesador para una copia compartida) y exclusiva sin datos. El protocolo da soporte al *uso compartido sucio*, lo cual permite compartir datos sin requerir el nodo donde se ubica originalmente, obteniendo una copia actualizada. Se mantiene el reenvío de respuestas a propietarios remotos y las contestaciones exclusivas (la pertenencia dada antes de todas las invalidaciones es completa). Se elimina la necesidad de las confirmaciones de invalidación, sacando provecho de las propiedades ordenadoras del switch.

El protocolo intra-nodo usa virtualmente los mismos mecanismos de flujo de transacciones, como el protocolo inter-nodo para mantener la coherencia. Para un sistema de un solo nodo, o nodo sede, el indicador duplicado (*DTAG*) lógicamente funciona como un mapa completo de directorio centralizado porque tal información se utiliza por los cuatro procesadores locales. Los accesos remotos de memoria son enviados directamente al nodo sede, y sin retrasos, tiene que comprobar si ellos pueden ser servidos por algún otro procesador local.

Una decisión crucial en el diseño del protocolo es manipular los casos sin dependencia en reconocimiento negativo (*NAK's*)/reintentos o el bloqueo en la sede del directorio: Los *NAK's* son típicamente usados en los protocolos dimensionables de coherencia para:

(i) Resolver dependencias de recursos que pueden resultar en punto muerto (cuando las sendas salientes de la red den marcha atrás).

(ii) Resolver carreras donde una petición no puede encontrar los datos en el nodo o el procesador al que es remitida (o, en algunos diseños, cuando el directorio sede esté en una condición "ocupada"). Análogamente, el bloqueo en el directorio sede se usa algunas veces para resolver tales carreras.

Eliminar los *NAK's*/reintentos y el bloqueo en la sede conlleva varias características importantes y deseables. Primero, garantizando que un nodo propietario (o el procesador) siempre puede dar servicio a una petición reenviada, todos los cambios de condición del directorio pueden ocurrir inmediatamente cuando el nodo de la sede es el primero visitado. Por consiguiente, todas las transacciones se completan con la mayoría con un solo mensaje para la sede (la petición original) y un acceso único para el directorio (y *DTAG*). Esto ocasiona menos mensajes y menos utilización de recursos, pues todas las transacciones de lectura y escritura de 3 saltos (involucrando a un propietario remoto) comparados con el protocolo, se envía un mensaje de confirmación adicional de retorno a la sede (cambio de propietario). Segundo, el controlador del directorio puede ser implementado como una máquina sencilla de condición segmentada en donde las transacciones inmediatamente actualizan el directorio, a pesar de otras

transacciones están corriendo en el mismo segmento. Consecuentemente, se evitan obstrucciones y utilización adicional del directorio, y en lugar de eso se resuelven las dependencias en la periferia de sistema.

Tercero, la consigna de optimización para implementar la consistencia de memoria los modelos también depende de la garantía que un propietario siempre puede dar servicio a una petición. Cuarto, intrínsecamente se eliminan los bloqueos activos, inanición, y problemas de equidad que se levantan debido a la presencia de *NAK's*.

5.3.1. Eliminación del protocolo de interbloqueos (*Deadlock*)

El protocolo usa tres vías virtuales (Q0, Q1, y Q2) para eliminar la posibilidad de interbloqueos del protocolo sin recurrir a los *NAK's* o reintentos. La primera parte que la vía (Q0) transporta las demandas de un procesador a una sede. Los mensajes de la sede, ya sea del directorio o memoria (las respuestas o los mensajes reenviados para nodos de terceros o procesadores) son siempre llevados en la segunda vía (Q1). Finalmente, la tercera vía (Q2) contesta de un tercer nodo o un procesador al solicitante. El protocolo requiere una vía virtual adicional (QIO, usado para transmitir demandas múltiples para los dispositivos IO) para soportar un bus PCI ordenado basado en reglas. El protocolo depende de un orden total de mensajes Q1 (entra en un conmutador de barras cruzadas) y un ordenamiento de mensajes punto a punto de QI0 y Q0 (en la misma dirección), sin requisitos ordenados en los mensajes de Q2.

5.3.2. Manejo y competencia de requerimientos

Hay dos competencias posibles cuando una petición es remitida a un nodo propietario o un procesador. La competencia retrasada de petición ocurre sí la petición llega al propietario, después de que el dueño ya ha escrito hacia atrás la línea. La competencia anticipada de petición ocurre si una petición llega al propietario antes del propietario tiene recibido su copia de los datos. Las soluciones para estas competencias garantizan que la petición reenviada es reparada sin cualquier intentando de nuevo o bloqueando en el directorio.

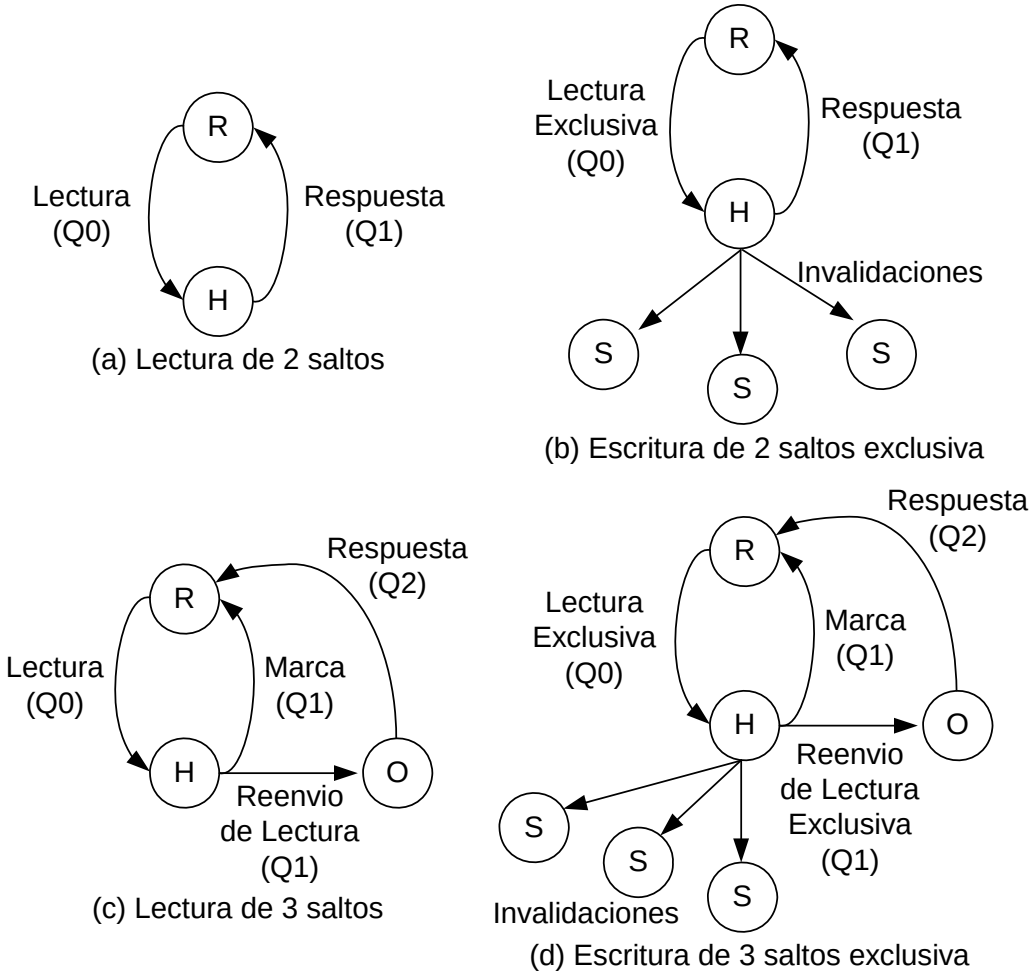
La solución para la competencia retrasada de petición involucra a mantener una copia válida de los datos en el dueño hasta que la sede reconoce al *writeback*, permitiéndonos satisfacer cualquier demanda múltiple reenviada en el ínterin. El protocolo usa un mecanismo de dos niveles. Primero, cuando el procesador *Alpha* 21264 sacrifica una línea, aguarda una liberación de la víctima emitiendo señales antes de descartar los datos del buffer de la víctima. La señal de liberación de la víctima es eficazmente demorada hasta que todas las demandas múltiples reenviadas pendientes del *DTAG* para un procesador dado estén satisfechos. La metodología citada anteriormente descarga la necesidad del emparejamiento complicado (utilizado en diseños entrometidos) de la dirección entre colas entrantes y salientes. Para los respaldos de escritura para sedes remotas, la responsabilidad de mantener los datos es completamente para la tabla de transacciones en transito. Esta copia es mantenida hasta que la sede confirme el *writeback*.

La solución para los requerimientos con anticipación requiere retardos de la petición reenviada (en Q1) hasta que los datos lleguen a su propietario (en Q2). Esta funcionalidad es soportada dentro el procesador *Alpha* 21264, por lo cual la dirección en la cabeza de la cola del sensor de entrada (Q1) la cual es comparada en contra el archivo de direcciones fallidas del procesador (rastro de los fallos pendientes) y es retardada en caso de ocurrencia. El almacenamiento intermedio con anticipación requiere de manera adicional en su totalidad, eliminar la posibilidad de respaldo de la línea Q1 la cual requeriría un *buffer* muy grande (256 accesos) debido a las propiedades compartidas del protocolo. Al quedar atascado en la cabeza de la senda Q1, el procesador objetivo provee un mecanismo de resolución extremadamente simple, y es relativamente eficiente evitando los atascos en la memoria intermedia de los nodos destinos eliminando el impacto progresivo en el *switch*. No obstante, el uso incorrecto de esta técnica potencialmente puede ocasionar ínterbloqueos. Finalmente, la naturaleza jerárquica del diseño permite transacciones ser atendidas dentro del nodo sin que necesariamente se involucre el *switch global*. Esta optimización es crítica para lograr un rendimiento óptimo del sistema, pero causa interacciones delicadas con el total de requerimientos ordenados para Q1. El siguiente esquema sirve para precisión. La tabla de transacciones en transito (*TTT*) en el nodo principal mantienen el registros de los mensajes a Q1, que son enviadas en representación de procesadores locales, pero todavía no han alcanzado el *switch* global debido al almacenamiento de la memoria intermedia. En el caso extremo en que (i) una petición subsiguiente para la misma dirección llega los pocos minutos tal como el mensaje Q1, y está en transito, y (ii) la requerimiento puede ser servido hacia la cola Q1 dentro del nodo, posteriormente, Q1 es enciclado en el *switch* global para asegurar el orden de los mensajes en Q1.

5.3.3. Protocolo eficiente de baja ocupación

La figura 30 muestra varias transacciones básicas del protocolo. Se usa la siguiente notación: La R es el solicitador, H la sede, O el propietario, y S un participante. La vía virtual (Q0, Q1, Q2) usada por un mensaje, es mostrada en paréntesis. Estos flujos de protocolo también tienen aplicación para el protocolo intra-node, con la etiqueta (*DTAG*) duplicada, comportándose como sede.

Figura 30. Flujo de transacciones del protocolo básico



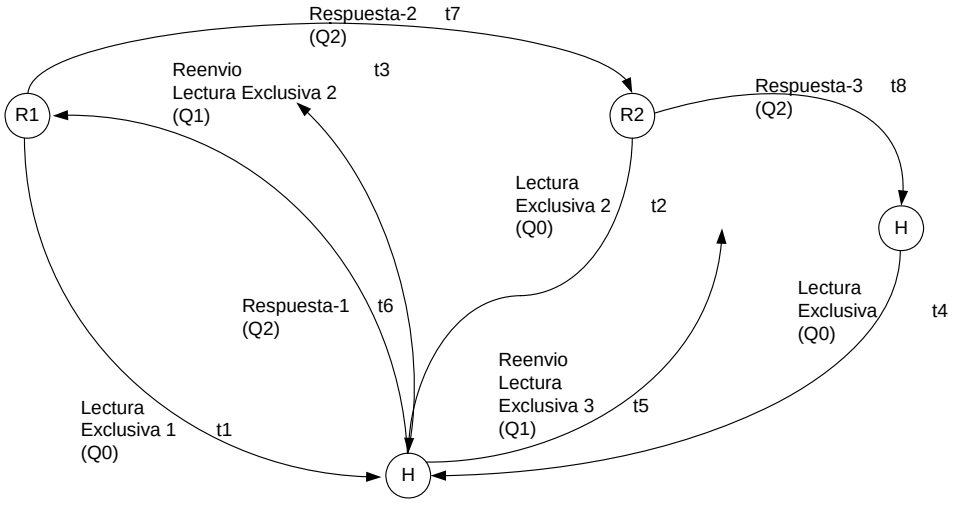
La figura 30 (a) muestra un caso básico de lectura de dos saltos. La figura 30 (b) muestra un caso básico de lectura de dos saltos con participantes múltiples. La respuesta de datos es enviada al solicitador, al mismo tiempo, se envían invalidaciones a los participantes. Este flujo ilustra dos propiedades interesantes de protocolo. Primero, el protocolo no usa mensajes del reconocimiento - invalidación, lo cual reduce el conteo de mensajes y ocupación del recurso. Dada la propiedad de orden total en Q1, una invalidación aparece al ser entregado a su nodo destino cuando está programada en el *switch*. En segundo lugar, como una optimización, utiliza la capacidad de multidifución del *switch* cuandoquiera que la sede necesite enviar mensajes múltiples Q1 a nodos diferentes como parte del servicio de una petición. En este ejemplo, se inyecta un solo mensaje en el *switch* que el automáticamente programa los mensajes de invalidación apropiados y envía la respuesta para el solicitador.

La figura 30 (c) muestra una transacción de lectura de tres saltos. La sede remite la petición al propietario e inmediatamente altera el directorio para reflejar el solicitador como un participante. Como se menciono anteriormente, el cambio inmediato en el directorio es posible porque el propietario garantiza el servicio a la petición reenviada. El propietario directamente responde para el solicitante, y la propiedad impura que comparte el protocolo evita la necesidad de un mensaje participativo de reescritura para la sede. El mensaje marcado enviado de la sede para los solicitantes tiene varios propósitos en el protocolo. Primero, el marcador es usado en el solicitador para no enviar una orden ambigua en la cual otras demandas múltiples para la misma línea fueron vistas previamente por el directorio.

Por ejemplo, el nodo solicitante limpia con un filtro a cualquier mensaje de invalidación que llegue antes del marcador, mientras uno invalida el mensaje que llega después de que el marcador, se envía al procesador solicitante. En segundo lugar, el marcador sirve como confirmación del acontecimiento leído, el cual sirve para la memoria en el ordenamiento de las acciones. Finalmente, la figura 30 (d) muestra una transacción de escritura de 3 saltos. Dado la naturaleza impura que comparte el protocolo, es posible que una línea tenga a un propietario y unos participantes múltiples como se muestra en este escenario. Como en el caso de lectura de 3 saltos, el directorio ha cambiado inmediatamente, y no hay más mensajes enviados a la sede para completar esta transacción. El marcador sirve para el mismo propósito como en el caso de lectura, y se usa también para accionar invalidaciones a otros procesadores (compartiendo la línea) en el nodo del solicitador.

La figura 31 muestra un efecto interesante de estado cambiante del directorio inmediato y nuestra una solución rápida a las peticiones. El escenario mostrado involucra nodos múltiples escribiendo en la misma línea (inicialmente el protocolo esta en un estado impuro en un procesador del nodo sede; los mensajes de marcador no son demostrados por simplicidad). Las escrituras son serializadas en el directorio, cada uno es remitido al propietario actual e inmediatamente cambia en el directorio para reflejar al nuevo propietario. La primera petición retraza el mecanismo de reenvió de demandas múltiples (en Q1) que alcanza sus destinos en tiempo. Como cada solicitante tiene su respuesta (en Q2), las respuestas de datos de un propietario al siguiente no involucran ninguna acción adicional del directorio. En tales casos patológicos, el protocolo es mucho más eficiente que otros protocolos.

Figura 31. Comportamiento del protocolo con múltiples escritores a la misma línea



5.4. Implementación eficiente de modelos de consistencia

A continuación, se describen algunas técnicas innovadoras usadas en el *AlphaServer GS320*, las cuales extienden los avances previos en la implementación eficientemente de modelos de consistencia en memoria.

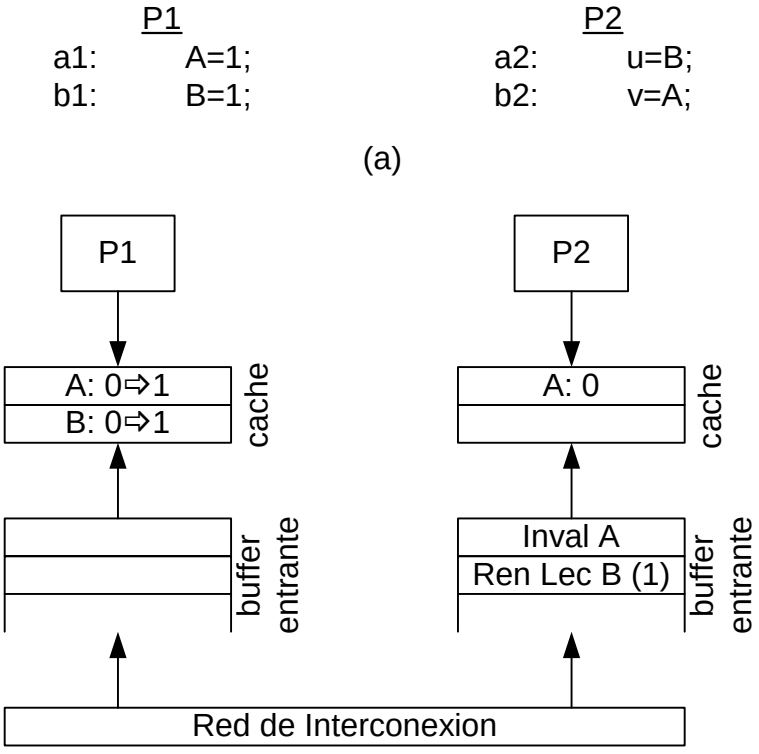
5.4.1. Reconocimiento con anticipación de peticiones de invalidación

Para reducir la latencia de las invalidaciones, una optimización común debe aceptar una petición de invalidación tan pronto como la solicitud es situado en la cola entrante destino (por ejemplo: una jerarquía de memoria *cache*) antes de todas las copias no actualizadas sean eliminadas.

Sin embargo, los usos sencillos de esta optimización pueden ocasionar un comportamiento incorrecto desde que la aceptación ya no signifique la finalización de una escritura con relación al procesador objetivo. Considere una operación de escritura con eventos múltiples de finalización con relación a cada procesador en el sistema. Puesto que cada uno escribe, también definimos un evento que confirma la relación en cada procesador. El evento de confirmación corresponde al tiempo cuando las invalidaciones causadas por la escritura, ya sea aceptado explícitamente o implícitamente, y precede el evento de finalización con relación a un procesador en los que se presento una conformación anticipada. En diseños que sacan provecho a las aceptaciones anticipadas, el orden del programa entre una escritura W y una siguiente operación Y, esta obligada a esperar confirmación de W con respecto a cada procesador esperando la emisión de Y (no hay mensajes explícitos que señalen la finalización de la escritura).

La figura 32 muestra un ejemplo para ilustrar los temas relacionados con los reconocimientos anticipados. Por simplicidad, se asume un protocolo secuencialmente basado en invalidaciones consistentes (SC). Considere el segmento del programa en figura 32 (A) con todas sus posiciones inicializadas en cero. El resultado (u, v) = (1,0) es denegado bajo SC. Se asume que P1 inicialmente obtiene ambas localizaciones y P2 obtiene la localización A. Sin reconocimientos anticipados, P1 emite una escritura a A, y espera para poder completar, y procede la escritura para B. Por consiguiente, la copia atrasada de A en P2 es eliminada antes que P1 aun emite su segunda escritura. Mientras P2 asegura su lectura completa de acuerdo al orden del programa, el resultado (u, v) = (1,0) ciertamente será denegado.

Figura 32. Ejemplo ilustrativo de reconocimientos anticipados de invalidación



Ahora considere el escenario con reconocimientos de invalidación anticipada. P1 escribe en A, y envía una invalidación a P2. Esta invalidación es registrada en una cola en P2 y una respuesta de confirmación es generada. Al este punto, la escritura de A le está comprometida pero le falta completar con relación a P2 (o sea, P2 todavía puede leer el valor anterior de A). Mientras la invalidación se queda en cola, P1 puede proceder a emitir su escritura a B, y P2 puede emitir su lectura a B.

La figura 32 (b) capta la condición del *buffer* entrante de P2, con ambas peticiones de invalidación para A y la respuesta de lectura por B (el valor de regreso de 1) es encolada. El asunto principal está en que permitiendo la respuesta de lectura sea ignorada por petición de invalidación en el *buffer*, lo cual es deseable por razones de rendimiento, violará a SC porque P2 puede proceder a leer el valor de atrasado antes de obtener un nuevo valor para B.

Hay dos soluciones conocidas para el problema citado anteriormente. La primera solución impone ordenar las restricciones entre los mensajes entrantes con relación a invalidaciones previamente comprometidas. Regresando al ejemplo, esta solución prohibiría la respuesta leída ignorando la petición de invalidación, lo cual obliga a comprometer la escritura a A para completar con relación a P2, antes que la lectura de B se complete. En el ordenamiento primero en entrar-primero en salir, todos los mensajes entrantes a partir del punto comprometido trabajaran, esto es suficiente para sólo mantener en orden la cola entrante de respuesta (los datos o confirmaciones) para alguna invalidación entrante previa (permitiendo bastante reordenaciones en la ruta de entrada por razones de rendimiento). La segunda solución no impone ninguna restricción ordenada entre los mensajes entrantes. En lugar de eso, requiere invalidaciones previamente cometidas para ser atendido, siempre que el programa los ejecute de forma ordenada. En el ejemplo, esta última solución permitiría la leer la respuesta para ignorar la invalidación entrante, pero forzaría a la petición de invalidación a ser atendida (vaciando la cola entrante) como parte de imponer el orden en el programa tanto para la lectura de B como la lectura de A. En ambas soluciones se prohíbe el resultado correctamente (u, v) = (1,0).

La eficiencia relativa de las dos soluciones citadas anteriormente depende fuertemente del modelo subyacente del modelo de memoria. La primera solución está más apropiada para modelos estrictos, como la serie *AlphaServer SC*, donde se implementan las órdenes del programa con ocurrencia mucho más frecuentemente que en la memoria *cache*; la segunda solución es más apropiada para modelos más desocupados donde el programa implementa las órdenes con menor frecuencia. Adicionalmente, para modelos más desahogados, la segunda solución puede proveer un servicio más rápido a las respuestas entrantes (datos o confirmaciones) permitiendo ignorar los requerimientos de invalidación previos. Sin embargo, la segunda solución puede llegar a ser ineficiente en diseños con colas de entrada intensa (debido a una jerarquía del *cache*); aun así, vaciado las colas entrantes puede ser infrecuente, y la sobrecarga de hacerlo puede ser realmente alta. La única solución parcial conocida para el intercambio citado anteriormente, es un diseño híbrido que utiliza la primera solución en los niveles inferiores y la segunda solución en los niveles más altos de la cola de entrada lógica (o la jerarquía del *cache*).

5.4.2. Separación de respuestas entrantes en componentes de respuesta y datos comprometidos

El *AlphaServer GS320* soporta el modelo de memoria *Alpha* el cual consiste en del uso de instrucciones explícitas de memoria para imponer orden. Además, el objetivo en el diseño está en el punto de arbitraje dentro de un nodo, para accesos satisfechos localmente o en el punto de arbitraje para en *switch* global para accesos que involucran transacciones remotas. En ambos casos, los dos conducen largas rutas de entrada para el procesador. La segunda solución descrita en anteriormente es impráctica ya

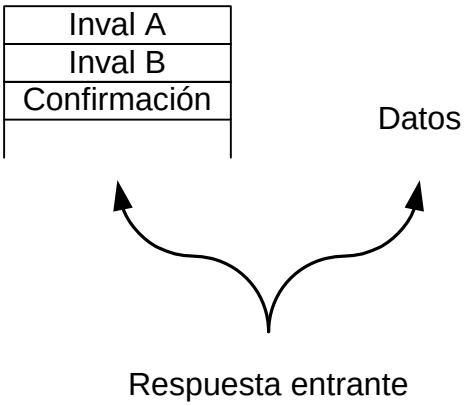
que produce una sobrecarga de evacuación en la ruta de entrada en la memoria. Al mismo tiempo, el uso de la primera solución puede causar un atraso el tiempo las respuestas críticas de entrada las cuales invalidan las peticiones en ruta de entrada al procesador. A continuación, se describe una técnica simple pero poderosa que aliviana el intercambio indeseable descrito anteriormente.

En la mayoría de protocolos de consistencia, las peticiones del procesador son satisfechas a través de un solo mensaje de respuesta. Por ejemplo, una petición de lectura o lectura exclusiva, la petición de lectura recibe una respuesta de datos, mientras una petición de lectura exclusiva (causada por una escritura de una copia actualizada) puede recibir una respuesta de éxito o de fracaso. Esta estrategia separa el mensaje de respuesta en sus dos componentes lógicos cuando el mensaje llega a la ruta de entrada: (i) el componente de datos o de respuesta que se necesitó para el satisfacer el requerimiento, y (ii) uno componente que confirma, el cual sirve para propósitos de ordenamiento. Esta separación esta ilustrada en Figura 33. Para dejar el componente crítico de datos / tiempo de respuesta para ignorar otros mensajes de entrada (Q1) en su ruta al procesador (usando una vía separada como Q2). Para lograr exactitud, el componente que confirma es utilizado como un marcador de orden colocándolo en la misma ruta como otros mensajes de entrada e imponiendo un orden parcial el cual es requerido con relación a los otros mensajes.

Por ejemplo, dada la respuesta de invalidación anticipada optimizada descrita anteriormente, el componente que confirma no puede ignorar

cualquier invalidación de entrada previamente efectuada. Este método es superior respecto de las dos soluciones descritas anteriormente; para permitir respuestas críticas en tiempo e ignorar otros mensajes de entrada, no se requiere una evacuación explícita de la ruta de entrada en la memoria.

Figura 33. Separación de una respuesta de entrada en sus componentes de datos y confirmación



Para que el modelo descrito anteriormente opere correcta y eficazmente, el soporte del procesador es necesario: (i) esperar dos componentes de respuesta en lugar de uno solo, y (ii) llevar apropiadamente las cuentas de los componentes de confirmación. El *Alpha* 21264 mantiene una cuenta de solicitudes pendientes. Este conteo es incrementado en cada solicitud emitida al sistema, y decrementado cada vez que un evento de confirmación es recibido. Recibir el componente de datos /respuesta no afecta el conteo, y de hecho no hay requisito para que el componente de datos / respuesta llegue antes de que la confirmación, para lograr ordenamiento correcto.

En memoria, el procesador espera por el contador alcance el valor cero, antes de proseguir con otras demandas de memoria. El diseño viene orientado a que los componentes de confirmación de otros mensajes de entrada en Q1 con un campo de 1 *bit* adicional sean aceptados; un mensaje nulo es insertado si ninguno de los mensajes está disponible por procesarse.

5.4.3. Confirmaciones anticipadas para solicitudes de lecturas y lecturas exclusivas

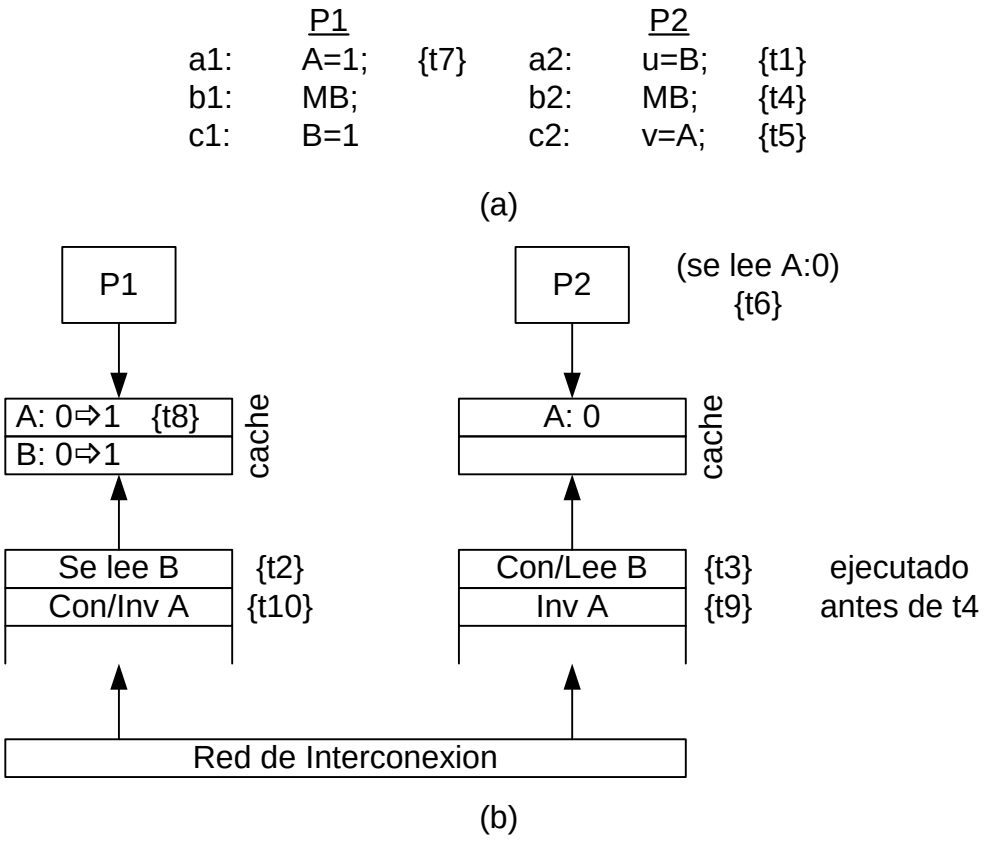
La arquitectura de *AlphaServer GS320* extiende la idea de comprometer con anticipación todos los tipos de demandas múltiples del procesador, en lugar de las demandas múltiples de lectura exclusiva de todos los participantes. Esta optimización puede reducir el atraso cuandoquiera que un procesador deba esperar sus demandas múltiples pendientes para completar sus propósitos. El impacto de esta técnica puede ser de gran amplitud ya que está agrega una optimización fundamental para los diseñadores en la implementación de modelos de consistencia de memoria de una forma correcta y eficaz.

En el diseño, las confirmaciones anticipadas son generadas por cualquier solicitud de lectura o lectura exclusiva que es reenviada para ser atendida por una copia impura o una copia impura compartida (Incluye un reenvió a una copia de *cache* dentro del mismo nodo). Este método puede estar sencillamente generalizado para requerimientos múltiples en memoria. Sin embargo, esto no tiene buenas consecuencias en el diseño porque la confirmación no puede ser generada antes que la respuesta de datos de la memoria. Similar a las confirmaciones anticipadas por invalidación, el

mensaje de confirmación anticipado es generado cuando se reenvía un requerimiento de lectura o lectura exclusiva en el punto de confirmación. Un mensaje de respuesta de datos por separado es devuelto una vez que la petición es servida en el *cache* destino. Al igual que las confirmaciones anticipadas de invalidación, un procesador tiene permitido ir más allá de un punto ordenado en el tiempo mientras que todas las demandas múltiples anteriores han recibido su respuesta de confirmación, aún sí las respuestas reales de datos todavía no se han recibido. Teóricamente, la optimización descrita anteriormente permite a un procesador proceder más allá de sus propósitos antes que retorne su valor pendiente de validación. Como sería de esperar, el uso incorrecto de esta optimización ocasionaría un comportamiento incorrecto.

La figura 34 muestra un ejemplo ilustrando los tópicos relacionados con las confirmaciones anticipadas por requerimientos de lectura y lectura exclusiva. Considere el segmento de programa en la figura 30 (a) con todas las localizaciones inicializadas con cero (“MB” es una memoria Alpha). El resultado (u, v) = (1,0) es prohibido bajo el modelo de memoria Alpha (y también bajo consistencia secuencial). Asuma que P1 inicialmente reserva en *casche* ambas localizaciones (con copia impura de B) y P2 reserva en cache la localidad A. La figura muestra un orden dado de sucesos puntuales los cuales son representados por [t1.. t10]. Suponga que P2 emite una lectura en B, con un requerimiento en cola para P1.

Figura 34. Ejemplo que Ilustra la confirmación anticipada por requerimientos de lectura



El mensaje de confirmación por lectura es generado una vez que la petición hizo cola en P1, y es devuelta a P2 (como se muestra en la cola en P2 en el tiempo t3). Media ves el evento de confirmación es recibida por P2, P2 puede ir después al punto ordenado representado por MB y puede leer el valor 0 de A (v = 0).

Note que se permitió a P2 completar su lectura de A antes de que el valor de regreso de la lectura de B (actualmente esperando ser atendido en la cola

entrante de P1) que está aun comprometido. Ahora asuma que P1 emite su escritura a A, lo cual genera una invalidación para P2 y le corresponde aceptación invalida (una confirmación anticipada por invalidación) a P1. La figura 34 (b) muestra la condición de las colas entrantes en este punto (con la confirmación de lectura de B ya ejecutada).

Ahora, se puede ilustrar un potencial comportamiento incorrecto en el escenario mostrado en la Figura 34. Considere el evento de confirmación de invalidación para A (confirmación / Invalidación A) ignorando la solicitud de lectura a B en la ruta de entrada para P1. Éste reordenando es permitido bajo requerimientos suficientes por invalidaciones anticipadas por reconocimientos de invalidación desde el único orden que está obligado a ser mantenido de acuerdo al requerimiento de invalidación previo. Por lo tanto, P1 puede recibir su confirmación, aunque no se mantenga en memoria, y emita una escritura a B quien cambiaría la copia de su *cache* para obtener el valor de 1. Al llegar a este punto, cuando la lectura de B (todavía en la ruta de entrada) llega a ser servida, devolverá el valor de 1 (u = 1). El escenario citado anteriormente viola el modelo de memoria Alpha desde ha permitido (u, v) = (1,0).

Al igual que las aceptaciones de invalidación anticipadas, estás dos soluciones se pueden usar para garantizar exactitud. La primera solución implica imponer orden entre los mensajes de entrada: un mensaje de confirmación no puede ignorar cualquier solicitud previa (lectura, lectura exclusiva o invalidación). Esta solución prohíbe los mensajes de confirmación / invalidación de A, tratando de ignorar la petición de lectura de

B en el escenario mostrado en la Figura 34. Por consiguiente, la solicitud de lectura de B se garantiza para que sea atendida antes de que P1 tenga permiso de cambiar el valor de B (P1 no puede pasar su valor en MB antes que reciba la confirmación / invalidación de A), por lo tanto asegura que la lectura de B devuelve el valor 0 para P2 (u = 0 genera (u, v) = (0,0), el cuál es un resultado correcto). La dinámica de cómo es asegura la exactitud con la optimización de conformación anticipada es realmente importante, ya que efectivamente forzamos el valor de regreso que leemos de B para ser obligados antes de que P1 obtenga permiso de cambiar el valor para B. La segunda solución no impone un ordenamiento entre los mensajes de entrada, sino requiere que cualquier mensaje de petición en la ruta de entrada (lectura, lectura exclusiva, o invalidación) sean atendido en cualquier momento que un procesador requiera un ordenamiento en la programación (en memoria). Nuevamente, antes de que P1 obtenga permiso de completar a su MB, está obligado a atender la lectura de entrada de B, lo cual producirá el comportamiento correcto. El diseño *AlphaServer GS320* usa la primera solución citada anteriormente porque naturalmente y sinergícamente se fusiona con la optimización: la confirmación anticipada es usada por propósitos de ordenamiento, la respuesta es crítica en tiempo de los datos que típicamente llega más tarde y está permitida para pasar por encima de otros mensajes de entrada, y no hay requisito para evacuar la ruta de entrada en memoria.

La optimización de confirmación anticipada que se describe aquí depende de la garantía de la lectura o la petición de lectura exclusiva que será atendido por el nodo destino o por el procesador que genero una respuesta de confirmación. Por lo tanto, los protocolos que no puede generar garantía

tal que el requerimiento (debido a *NAK's* / reintentos) no puede usar esta optimización. Hay algunos otros temas relacionados que surgen del diseño del *AlphaServer GS320* (aún para los mensajes de invalidación anticipado) debido a la presencia de un punto de confirmación dentro de un nodo (punto de arbitraje para el *switch* local) y un punto de confirmación externo del nodo (punto de arbitraje para *switch* global), y el hecho que algunas demandas son satisfechas solamente por los puntos de confirmación locales. Por ejemplo, debido al hecho que soporta respuestas exclusivas, es posible que una petición generada en el nodo sede, estén localmente satisfechas mientras las invalidaciones remotas causadas por una operación previa no la tienen todavía. Para eliminar temas de exactitud, las transacciones en la tabla de tránsito (*TTT*) detectan tales casos y fuerzan un evento de confirmación para la operación más reciente evite los ciclos en el *switch* externo e imponga un orden para dejar la operación previa mediante un evento de confirmación y tire de cualquier requerimiento en la ruta de entrada.

5.4.4. Aplicabilidad de las técnicas del modelo de consistencia en otros diseños

Las dos técnicas de optimización descritas anteriormente, pueden servir para implementar cualquier modelo de consistencia de memoria, pudiendo extenderse desde modelos de consistencia secuencial a los modelos agresivos flexibles en memoria. La optimización de la separación de los componentes de datos / respuestas, es primordialmente útil para implementación de modelos flexibles donde se permite al procesador alcanzar los datos / respuestas anticipadas (antes que un punto de ordenamiento de memoria sea encontrado) genera un beneficio. Además, la

técnica se puede aplicar a cualquier implementación que explota las confirmaciones anticipadas, incluyendo las aceptaciones de invalidación anticipadas, siendo este superior comparado con otras soluciones que implementan exactitud.

Los beneficios del rendimiento de la segunda optimización son superiores en modelos más estrictos de memoria, ya que un procesador puede continuar después de que la memoria ordene frecuentemente sus emisiones para atender las nuevas demandas múltiples tan pronto como recibe los requerimientos de anticipación de confirmación. Sin embargo, los beneficios pueden ser significativos en modelos flexibles, debido a la reducción de retrasos de ordenamiento en memoria; por ejemplo, las latencias de la barrera de memoria en multiprocesadores Alpha constituyen un fragmento significativo del tiempo de ejecución para las rutinas críticas importantes de la sección en aplicaciones de base de datos.

Con relación a la aplicabilidad de las diferentes implementaciones, la optimización de la confirmación anticipada es mejor ajustado a diseños donde la separación del tiempo entre generar la confirmación anticipada y la respuesta actual es lo suficientemente significativa para justificar la generación de dos mensajes.

Además en diseños escalables con redes ordenadas (buses, *cross-bars*, anillos), este método también puede tener generar buenos beneficios en diseños escalables jerárquicos donde el ordenamiento no es mantenido en el

parte exterior de la interconexión pero puede ser implementado dentro de un nodo (consigne punto puede estar colocado en la entrada de nodo).

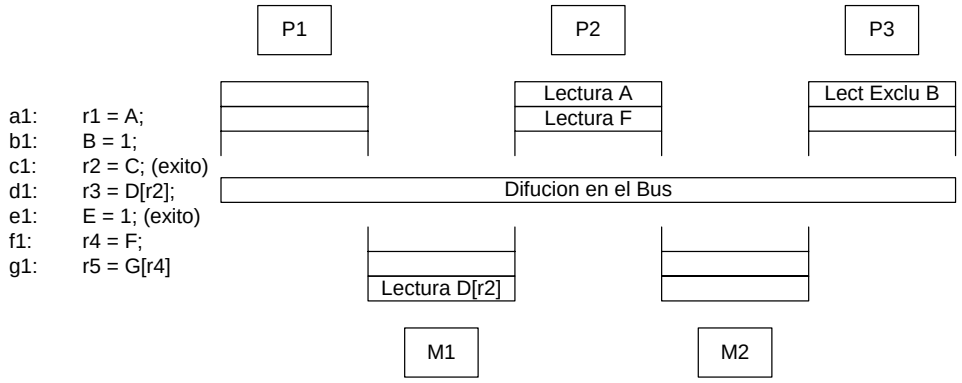
Para ilustrar el verdadero potencial de la optimización de confirmación anticipada, la figura 35 muestra a un ejemplo de un diseño curioso dando soporte a la consistencia secuencial. En el diseño, el punto de confirmación de la memoria cache está cuando es programada en el bus, y un procesador sólo debe aguardar un evento de confirmación (no necesita esperar por la respuesta actual) antes de emitir la siguiente referencia en orden de satisfacer la consistencia secuencial.

Dado el ejemplo en la figura, la primera parte P1 emite la lectura de A (queda en cola en P2). Dada que esta lectura es confirmada, P1 con toda seguridad puede continuar emitiendo la escritura a B en el bus. Media ves la escritura es programada en el bus (hace cola en P3), P1 con toda seguridad completa la lectura a C generando un acierto de la memoria *cache*. El valor de esta lectura se usa para calcular la dirección física de la siguiente lectura, la cual es también un acierto en el bus. Después, P1 con toda seguridad puede completar su de escritura para E que es un acierto de la memoria *cache*. Finalmente, P1 emite la lectura a F, y no puede proceder a emitir la última lectura (G [r4]) ya que su dirección no es todavía conocida.

Nótese que en este punto, P1 tiene 4 operaciones de memoria sobresalientes (uno de ellas en lectura exclusiva) que esperan sus respuestas de datos sin violar la consistencia secuencial. Además, P1 tuvo

permiso de consumir el valor de una lectura (C) y complete una de escritura (E) en la mitad de su flujo fallido. Del mismo modo, las respuestas pendientes de datos pueden regresar y pueden ser consumidos en cualquier orden.

Figura 35. Confirmaciones anticipadas en un diseño curioso



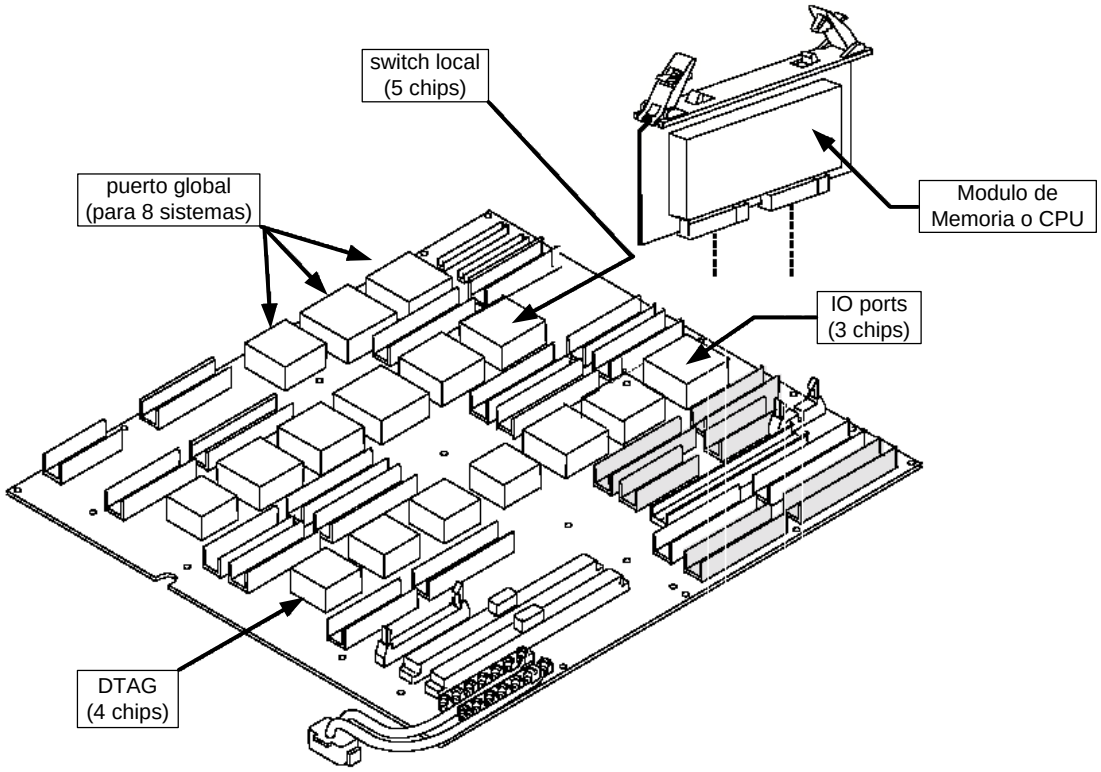
Esto es perfectamente seguro para P1, ejecutando su escritura (a E) visible a otros procesadores aun cuando aguarda las respuestas de datos por operaciones previas. Finalmente, a diferencia de otras técnicas propuestas para implementar modelos de consistencia, la metodología anteriormente citada no depende de cualquier forma de reversa; una vez que una operación es confirmada, se considera completa hasta donde el ordenamiento sea correspondido.

5.5. Implementación del *AlphaServer GS320*

El *AlphaServer GS320* esta diseñado y empacado para proporcionar modularidad y expansibilidad de 4 a 32 procesadores. Los bloques básicos son: ranuras simples para *CPU*, ranuras para memoria, las interfaces para

bus *PCI*, y dos tipos de *backplanes*. Un *QBB* esta construido para soportar 4 procesadores, ocupando solamente un estante; tal como lo hace el subsistema *PCI I/O*. Cada *QBB* soporta hasta 2 subsistemas *I/O*. Un gabinete soporta hasta 4 *QBB's* o 16 procesadores. Así, un sistema de 32 procesadores consta de dos gabinetes para procesadores y gabinetes adicionales para *I/O*.

Figura 36. Tarjeta madre para un *QBB* (4 *CPU*)



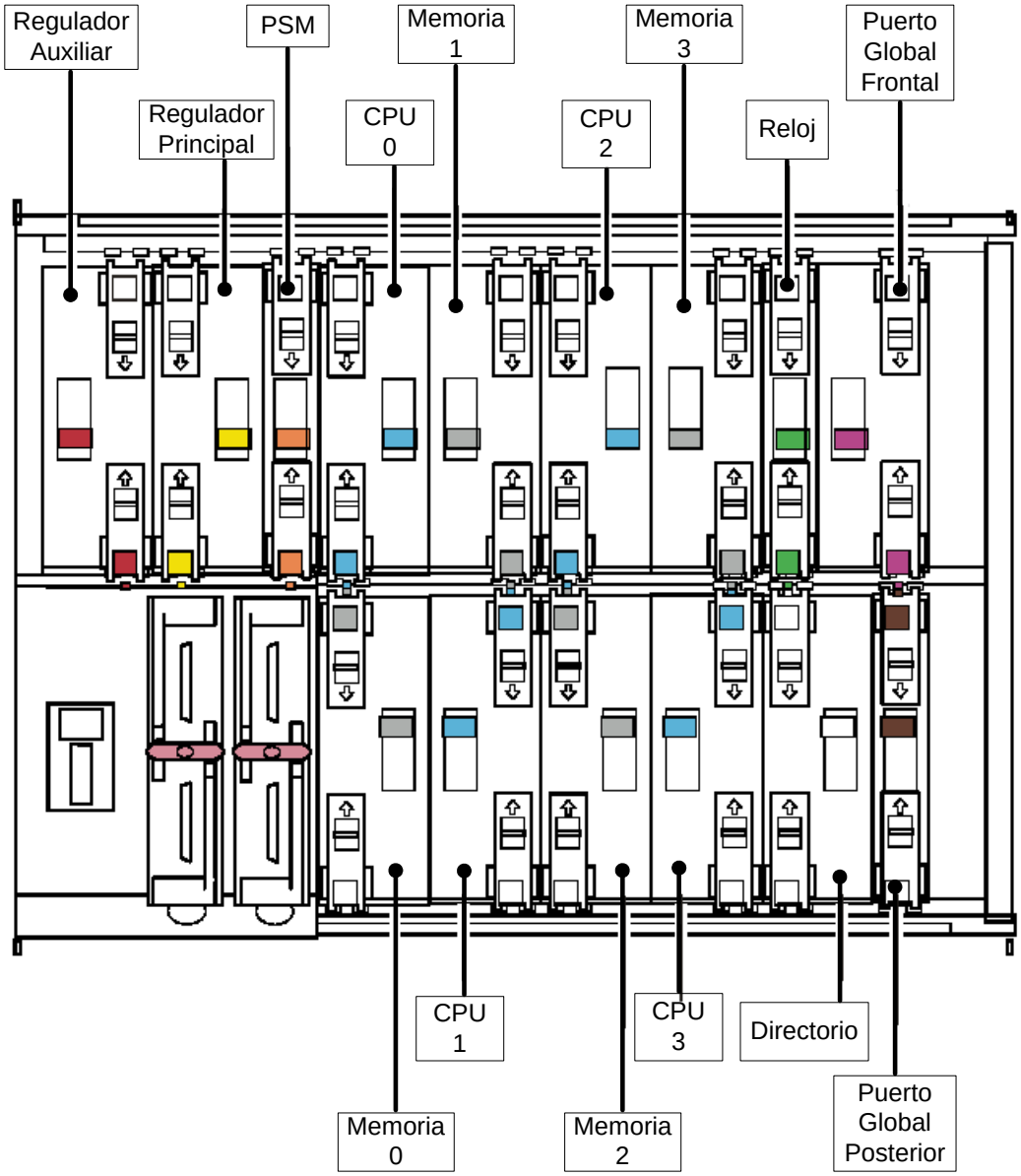
La figura 36 muestra la tarjeta madre para un *QBB* de 4 procesadores, con el switch local, *DTAG*, e interfaz *I/O* en el tablero. Los chips globales del puerto residen en la tarjeta madre para los sistemas de 8 procesadores (dos sistemas de 4 procesadores conectados espalda con espalda), sino es de un módulo enchufable para los sistemas mayores. Otros componentes están

también adjuntos como módulos enchufables para la tarjeta madre. Para los sistemas mayores que 8 procesadores, la tarjeta madre *QBB* está montada verticalmente, con un estante de 4 procesadores mirando hacia el frente y otro mirando hacia la parte posterior.

La figura 37 muestra la colocación de los módulos enchufables en el lado trasero. La figura 38 describe la parte posterior de los dos gabinetes usados para un sistema de 32 procesadores. Cada cuadrante es un estante de 8 procesadores. El *switch* global es montado en un panel plegable como se muestra. Los cables se conectan los puertos globales al *switch* global.

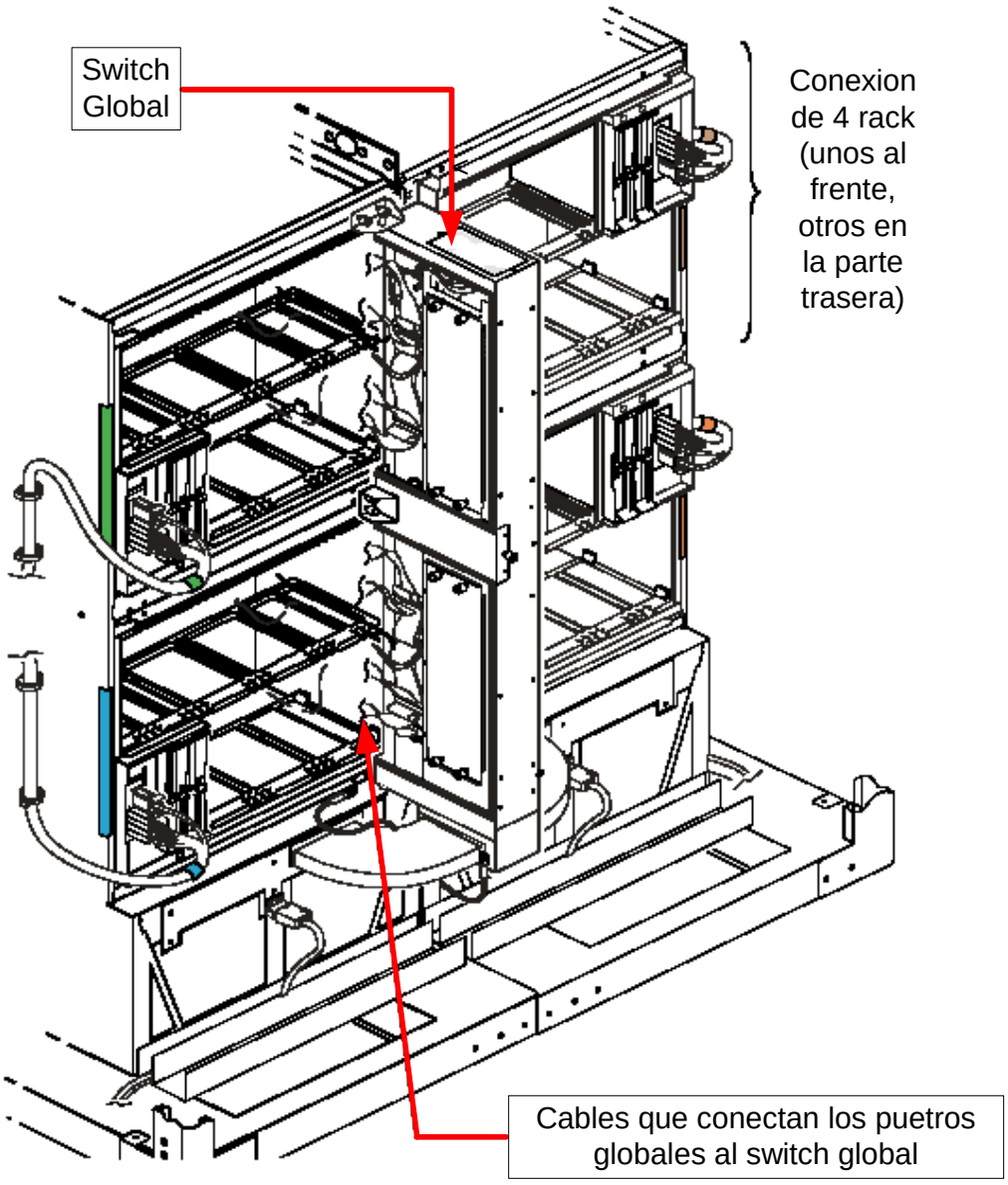
El sistema completo consta de 16 diseños únicos de circuitos integrados para aplicaciones específicas (*asic*) para un total de 7 millones de entradas. Hay 5 *asics* principales para enrutar las direcciones, las cuales constituyen la funcionabilidad fundamental del sistema, y 4 *asics* para enrutar datos los cuales son significativamente más sencillos que los primeros. Los *asics* restantes realizan las tareas más sencillas. La tecnología utilizada es *circa* 1997-98 con tiempo de ciclo 9.6ns, con acerca de 500k entradas utilizables *asics* de gran capacidad.

Figura 37. Parte posterior de un sistema de 8 procesadores



El diseño *AlphaServer GS320* soporta un número de características: fiabilidad, disponibilidad y funcionalidad. El sistema soporta hasta 8 particiones del hardware de acuerdo a la granularidad de los *QBB*, con hardware *firewalls* entre particiones.

Figura 38. Sistema de 32 procesadores conectados en el *switch* global



Ambos sistemas operativos, *Unix TruClusters* y *VMS Galaxy*, dan soporte a tal segmentación, con *Galaxy* sacando mayor provecho de software dinámico dividiendo en partes adecuadamente.

El sistema da soporte a la extracción en línea, la reparación, y actualización de los *QBB's*, el subsistema de *I/O*, y *CPU's* individuales. Este método permite probar el hardware y medirlo, antes de volver a insertarlo al sistema en operación. Finalmente, el sistema operativo *Tru64 Unix* y *Open VMS* se han extendido para administrar la naturaleza *NUMA* del *AlphaServer GS320*, para propósitos de planificación y de administración de memoria.

5.6. Mediciones de rendimiento en el *AlphaServer GS320*

A continuación se presentan los resultados que caracterizan los parámetros de latencia básica y de ancho de banda del *AlphaServer GS320* y la evaluación en el impacto de algunas de las optimizaciones. Además, el reporte resume en unos cuantos estándares de comparación estándar en la industria para el desempeño competitivo de *AlphaServer GS320* en ambas cargas de trabajo, especializadas y comerciales.

La tabla V presenta la medición de latencias por servicio a una petición de lectura en niveles diversos en la jerarquía de memoria. El informe muestra dos conjuntos de latencias en cada caso: (i) la latencia de lectura dependiente, y (ii) la latencia efectiva para las lecturas independientes. El caso de 3 brincos, muestra un rango de latencias porque incluye casos de 2.5 brincos donde el propietario o el procesador están en el nodo sede. Una de las razones principales por las cuales las latencias son relativamente altas, es el uso de una tecnología antigua de *asic* el cual esta implementado en el diseño actual.

Tabla V. Latencias efectivas de los diferentes tipos de memoria en el *AlphaServer GS320* con procesadores 21264 de 731 MHz

Caso	Back-to-Back Dependent Reads	Pipelined Independent Reads
L2 Cache Hit	23ns	7ns
Local, Clean	327ns	61ns
Local, Dirty	564ns	75ns
2-hop, Clean	960ns	136ns
3-hop, Dirty	1324-1708ns	196-310ns

Los resultados en la tabla muestran que alguna de las latencias remotas es alta, el *Alpha* 21264 tiene la habilidad de procesar fuera de orden y soportar demandas múltiples excepcionales múltiples sustancialmente puede reducir las latencias efectivas (aproximadamente de 5-7 veces) en caso de las lecturas fallidas independientes. Estas latencias también pueden usarse para calcular anchos de banda continuos (64 *bytes* por 61 ns generan un ancho de banda de 1.05 GB/s para la memoria local).

A continuación se compara al *AlphaServer GS320* dos otros diseños basados *AlphaServer* usando procesadores 21264. El *GS140* es una generación previa basada en bus que soporta hasta 14 procesadores. El *ES40* es un diseño basado en interruptores que soporta hasta 4 procesadores. Se usa una configuración de 4 procesadores en los tres sistemas (las frecuencias del procesador son diferentes). La tabla VI muestra la latencia de lectura dependiente de éxitos del *cache* L2 medido en un procesador.

Tabla VI. Impacto de carga en sistema con latencia de éxito en la memoria cache L2 en tres sistemas basados en Alpha 21264

	525 Mhz GS140	500 Mhz ES40	731 Mhz GS160
L2 Hit Latency (P1-P3 idle)	35ns	41ns	23ns
L2 Hit Latency (P1-P3 active)	68ns	113ns	23ns

Se consideran dos casos: (i) tres procesadores están ociosos, y (ii) tres procesadores están activos generando fallas. Como se observa en las tablas, la actividad de tres procesadores puede ocasionar una degradación mayor en la latencia de éxito del *cache* L2 (2-3 veces más largo) como observada el cuarto procesador en diseños basados en espionaje. La razón primaria en esto es que todas las transacciones de sistema deben ser espiadas por las memorias caches L2, ocasionándole a esté estar ocupado más a menudo (especialmente en el ES40 que no usa etiquetas duplicadas). El *AlphaServer GS320* no sucede esto, porque usa un protocolo basado en directorios que significativamente reduce el número de demandas múltiples que sé redirigen a cada L2. Dado la importancia de latencias de acierto del L2 en las cargas de trabajo comerciales como el proceso de transacciones, el uso de protocolos basados en directorios eficientes (en lugar de espiar) puede proveer aun más beneficios para los sistemas de un número pequeño de procesadores.

La tabla VII presenta latencias efectivas para operaciones de escritura exclusiva al variar la ubicación de la sede: local o remota y si hay más participantes.

Tabla VII. Latencia efectiva en operaciones de escritura

Case	Pipelined Writes	Writes Separated by Memory Barriers
Local Home, No Sharers	58ns	387ns
Local Home, Remote Sharers	66ns	851ns
Remote Home, No Sharers	135ns	1192ns
Remote Home, Remote Sharers	148ns	1257ns

En la tabla VII se encuentran dos conjuntos de latencias en cada caso: (i) La latencia efectiva para cada escritura en la línea privada de comunicación (*Pipelined Writes*) y (ii) la latencia de escritura ordenada a través de barreras de memoria (*Writes Separated by Memory Barriers* - 24ns mínimo). Las latencias de la línea privada de comunicación para la sede local y la sede remota están próximas a las latencias de las líneas privadas de comunicación locales y de 2 saltos. El impacto de latencia de envió de invalidaciones (debido a la presencia de participantes) es pequeña, ya que el protocolo no usa aceptaciones de invalidación.

Adicionalmente, estas latencias son independientes del número de participantes ya que el protocolo usa multidifusión. Las latencias por escritura están separadas de las barreras de memoria son ligeramente mayores que el dependiente local y de 2 saltos, en parte debido al costo de la barrera de memoria incluido en esta latencia. La sede local escribe con los participantes remotos sustancialmente más larga que sin participantes porque la barrera de memoria debe esperar para lo comete acontecimiento para volver de nuevo del interruptor global.

Finalmente, el envío de invalidaciones a la sede remota provoca el aumento de la latencia (otra vez debido a la falta de reconocimientos de invalidación que obtendrían latencias de 3 saltos).

Ahora, considere el impacto del diseño del protocolo de baja ocupación para manipular los conflictos de escritura en la misma línea. La tabla VIII muestra la latencia de serialización para dos escenarios. En cada experimento cada procesador tienen un ciclo ejecutando una escritura seguido por una barrera de memoria (las latencias incluyen una sobrecarga en la barrera de la memoria). En estado estable, el protocolo continuamente reenvía escrituras al propietario actual sin bloquearse y reintentando en el directorio, y reenviado escrituras atrasadas a su *cache* objetivo hasta que el requerimiento anticipado es resuelto.

Por consiguiente, se espera que la serialización de las escrituras sea aproximadamente igual a la latencia de un salto en el sistema. El primer escenario implica 4 procesadores en un solo nodo cuádruple o *QBB*. En este caso, 4 procesadores son insuficientes para generar suficiente rendimiento en condición estable para generar problemas en la línea del propietario anterior tan pronto como lo recibe. Consecuentemente, la latencia medida de serialización incluye algunos éxitos de cache, haciéndolo más pequeña que la latencia intra-nodo (approx 180-200ns) de 1 brinco. El segundo escenario consta de 8 procesadores en dos *QBBs* separados. Los 8 procesadores ocasionan el comportamiento esperado de estado estable y la latencia medida de serialización de 564ns es con mayor razón aproximadamente la mitad de la latencia de escritura de 2 brincos (de tabla

V). En comparación, los protocolos que dependen del uso de NAKs/reintentos para resolver tales comportamientos tendrían que mejorar la latencia de serialización en el caso de 2 saltos (con posibles latencias superiores basadas en el cronometraje de reintentos), y ocasionarían sustancialmente más tráfico y ocupación del recurso debido al gran número de NAK's y reintento de mensajes.

Tabla VIII. Latencia de serialización para conflictos de escritura en la misma línea

Case	Serialization Latency
1 QBB, 4 procs	138ns
8 QBBs, 1 proc/QBB	564ns

Finalmente, la tabla IX presenta algunas medidas para ilustrar el impacto de separar el componente de confirmación y la generación anticipada de confirmaciones. La latencia de lectura dependiente mide el tiempo que lleva al componente de respuesta de datos llegar al procesador, mientras la latencia medida con barreras de memoria mide el tiempo para el componente de confirmación. La sobrecarga intrínseca de una barrera de memoria es de 25-50 ns y son incluidos en la medida más reciente. Las medidas de 2 saltos claramente ilustran el beneficio de separar al componente de confirmación del componente de respuesta de datos. La diferencia en la latencia para los dos componentes es 205-230 ns (ajustados en la sobrecarga de la barrera de memoria), y mantenimiento los datos y las confirmaciones conjuntamente, causarían esta latencia adicional para ser incurridos por todos los fallos de lectura.

Tabla IX. Impacto en la separación del componente de confirmación y la generación de confirmaciones anticipadas

Case	Dependent Reads	Reads Separated by Memory Barriers
2-hop, Clean	960ns	1215ns
3-hop, Dirty	1478ns	1529ns

Además de esto, esta latencia adicional podría aumentar sí la ruta de entrada de la confirmación se ocupó por otras demandas múltiples reenviadas. Las medidas de 3 saltos ilustran el beneficio de generar con anticipación las confirmaciones en el diseño (en el caso de 2 saltos no genera una anticipación de confirmación). Se puede ver la diferencia en la latencia en las respuestas de datos y en las confirmaciones, en el caso de 3 saltos es aproximadamente 0-25ns (ajustada en la sobrecarga en la barrera de memoria), lo cual es muy más pequeña que la separación de 205-230ns (del caso de 2 saltos) que probablemente se observaría sin confirmaciones anticipadas. De hecho, las confirmaciones anticipadas probablemente entran en el procesador antes que el componente de respuesta de datos en el caso de 3 saltos, pero el *Alpha* 21264 espera ambos componentes, para llegar antes de proceder al paso de una barrera de memoria.

CONCLUSIONES

Después de haber presentado la investigación documental, se han llegado a las siguientes conclusiones.

1. La complejidad y el costo de desarrollo del software representan un costo creciente en los sistemas de cómputo, actualmente. Las nuevas topologías; cliente/servidor, los sistemas multi-capas y los sistemas distribuidos de software, aceleran la marcha de esta tendencia. A pesar de esto, los sistemas de objetos distribuidos ofrecen reducir el desarrollo del software aprovechando las ventajas de reutilización de código, brindando mejor escalabilidad para la simplificación del desarrollo de aplicaciones distribuidas.

2. Los sistemas de memoria distribuida compartida implementan la abstracción de memoria compartida a través de la arquitectura de la computadora, combinando la escalabilidad de las arquitecturas basadas en red, enfocándola en la programación de la memoria compartida.

3. El rendimiento de DSM esta, altamente, afectado por los patrones de acceso de memoria y los algoritmos de replicación de datos compartidos. Las implementaciones del hardware han generado reducciones enormes en la latencia de comunicación, conjuntamente con las ventajas de utilizar una unidad más pequeña de intercambio para compartir la información.

4. Los algoritmos son sensitivos al comportamiento de acceso de las aplicaciones y logra mejorar su rendimiento, significativamente, por medio de una afinación del uso de la memoria de la aplicación, o afinando el algoritmo que administra el acceso de la memoria compartida para una aplicación en particular, eliminando las ventajas del acceso transparente compartido de la memoria.

5. Se han clasificado varias primitivas de comunicación para la programación distribuida, los cuales soportan el paradigma de variables compartidas, sin la presencia de memoria física compartida. La programación de los muchos sistemas distribuidos que existen hoy en día, varios presentan un modelo computacional basado en el modelo de datos compartidos.

RECOMENDACIONES

Después de haber presentado las conclusiones de la investigación documental, se recomienda lo siguiente.

1. Ampliar el campo de acción de la carrera de Ingeniería en Ciencias y Sistemas, introduciendo seminarios que incluyan la participación de las empresas líderes en tecnología; los cuales muestren sus tendencias y expectativas.

2. Al momento de realizar una compra de servidores comerciales, solicitar a los fabricantes literatura técnica relaciona con la arquitectura e implementación de la máquina en cuestión.

3. Para la toma de decisiones, obtener información de todas las plataformas para analizar las ventajas y desventajas que cada una de ellas ofrecen y hacer un cuadro comparativo en el que muestre su rendimiento en términos numéricos basados de alguna medida.

4. Consultar diferentes fuentes en Internet que muestren los *benchmarks* más recientes, relacionados con el equipo seleccionado para evaluación y/o compra.

BIBLIOGRAFÍA

Libros

1. Gharachorlooy, Kourosh; Scales, Daniel J. **Software Distributed Shared Memory**. Estados Unidos: Western Research Laboratory Digital Equipment Corporation. 2000

2. Gharachorlooy, Kourosh; Sharma, Madhu; Steely, Simon; Van Doren, Stephen. **Architecture and Design of AlphaServer GS320**. Estados Unidos: High Performance Servers Division Compaq Computer Corporation. 2001

3. Hewlett-Packard**.** **New Compaq AlphaServer GS Series: Architecture White Paper**. Estados Unidos: Hewlett-Packard. 2001

4. Kühn, Eva. **Virtual Shared Memory for Distributed Architectures**. Estados Unidos: Editorial Nova Science Publishers Inc. 2001

5. Milutinovic, Veljko; Protic, Jelica; Tomasevic, Milo. **Distributed Shared Memory: Concepts and Systems**. Estados Unidos: Editorial IEEE Computer Society. 1998

6. Scales, Daniel J.; Lam, Monica S. **The Design and Evaluation of a Shared Object System for Distributed Memory Machines**. Estados Unidos: Computer Systems Laboratory Stanford University. 2000

Referencia electrónica

7. ACM. http://www.acm.org Julio 2006

8. Computer dictionary online. http://www.computer-dictionary-online.org Julio 2006

9. Elook. http://www.elook.org Julio 2006

10. Wikipedia. http://www.wikipedia.org Julio 2006