MÁSTER EN COMPUTACIÓN PARALELA Y DISTRIBUIDA, UPV

Interconexión de sistemas distribuidos

Tesis de Máster

Mª Remedios Pallardó Lozoya

Director: Francisco D. Muñoz Escoí

Diciembre de 2010

Índice de contenidos

1. Introducción

1.1 Descripción del problema.

En la era digital cada vez es más habitual el uso de grandes sistemas distribuidos. Por ejemplo, hoy en día están en auge los servicios online que nos ofrecen los gigantes de Internet (*cloud computing*), siendo también habitual el uso de las redes P2P. Ambos tipos de sistemas son sistemas distribuidos que necesitan ser escalables, ya que constantemente añaden y eliminan nodos (servidores y clientes) de dichos sistemas y no sería admisible que, en el caso de que se añada una gran cantidad de nodos, el sistema se sature y deniegue el servicio al resto de nodos que se intentan unir a él. Así pues, la escalabilidad de los sistemas distribuidos es un aspecto crítico que debe ser estudiado.

Si el problema consiste en que un nodo servidor está saturado porque ha llegado al máximo de su potencia de cálculo, siempre se podrá añadir otro nodo servidor más para repartir la carga. No obstante, el problema viene dado cuando el sistema ya no admite más nodos, no sólo porque la escalabilidad del sistema ya no mejora, sino porque empeora. Por ejemplo, imaginemos un sistema en el que uno de los nodos necesita difundir un mensaje a todos los nodos del sistema. Como hoy en día es habitual utilizar un sistema de comunicación a grupos para gestionar las entradas y salidas de los nodos en el sistema que además nos permita gestionar las difusiones, vamos a asumir en este ejemplo la existencia de uno de estos sistemas en el que existirá un único grupo que englobe a todos los nodos de un sistema. Cuando se realiza una difusión a n nodos, el sistema de comunicación a grupos convierte este mensaje en n mensajes (o $n-1$ si asumimos que el emisor recibe el mensaje en $t=0$), debiendo entregar un mensaje a cada uno de los destinatarios. Los nodos que estén ubicados en una misma LAN recibirán su mensaje rápidamente, mientras que, para el resto de nodos, los mensajes atravesarán varias redes físicas hasta llegar a ellos. Si al hecho de que en una difusión se generan muchos mensajes que atravesarán la red para llegar a sus destinos, se le añade que normalmente los nodos de un sistema distribuido están ubicados en LANs con velocidades de transmisión ultrarrápidas comparadas con las redes que interconectan dichas LANs, y que puede haber muchos nodos realizando difusiones al mismo tiempo, pronto será evidente que no podemos tratar el sistema como si se tratase de un solo grupo, ya que el gran número de mensajes transmitidos en redes con velocidades bajas (en comparación con las LANs) se convertirá en el gran cuello de botella del sistema.

El otro aspecto del problema es la necesidad de interconectar sistemas de un mismo tipo, que por separado pueden funcionar de forma autónoma, con el objetivo de conseguir un sistema global de mayor tamaño.

En el siguiente apartado se verá que la solución a ambos problemas consistirá en utilizar algoritmos de interconexión cuya estrategia consiste en interconectar grupos de procesos o sistemas más pequeños.

1.2 Descripción de la solución general.

Como se ha comentado, la solución al problema de la escalabilidad planteado en el apartado anterior consistirá en utilizar algoritmos de interconexión.

Por un lado, veremos que será conveniente dividir en grupos más manejables los nodos de los grandes sistemas de acuerdo a la topología de red y/o la frecuencia con la que se comuniquen. De esta forma, cuando se difunda un mensaje, el mensaje se difundirá localmente en el grupo emisor, se enviará al resto de grupos destinatarios a través de los enlaces de interconexión y se difundirá localmente en los grupos destinatarios. Nótese que para interconectar dos grupos solamente será necesario enviar un único mensaje, reduciendo drásticamente el número de mensajes que circularán por la red.

Por otro lado, esta misma técnica podrá aplicarse en los algoritmos de interconexión que interconecten sistemas autónomos, posibilitando la interconexión de sistemas que ejecuten distintas implementaciones de un mismo algoritmo.

También veremos que una característica deseable en los algoritmos de interconexión será que éstos puedan aplicarse en un sistema o en la interconexión de varios sistemas sin necesidad de modificar el protocolo que se ejecute en dicho sistema (algoritmo no intrusivo), aunque comprobaremos que esto no siempre será posible.

Así pues, el objetivo del presente trabajo es realizar un estudio bibliográfico de las principales publicaciones que plantean arquitecturas, protocolos y soluciones de interconexión de sistemas en el estado del arte, para resolver el problema de la escalabilidad en sistemas distribuidos.

Es importante destacar que a día de hoy no existe ningún trabajo que compile y revise las principales soluciones utilizadas en la interconexión de sistemas distribuidos. Así pues, la contribución principal de este trabajo radica en realizar dicha revisión, no aportando ningún nuevo diseño de algoritmo ni ningún estudio de rendimiento comparativo.

Cabe señalar que el contexto en el que se ha desarrollado este trabajo (tesis de máster), no permite realizar un estudio extensivo a toda la bibliografía existente en la materia, restringiéndose al análisis de algunas de las publicaciones y aportaciones más relevantes en este ámbito. Un análisis profundo de esta materia sería, posiblemente, objeto de una tesis doctoral. Es por ello que se han dejado fuera del análisis trabajos tales como el de *Chockler, Keidar y Vitenberg* sobre sistemas de comunicación a grupos [18], cuyo contenido se explicó detenidamente en las correspondientes asignaturas del Máster de Computación paralela y Distribuida y, por tanto, se asume que son conocidos por el lector.

1.3 Estructura de la tesis de máster.

En este apartado se describe la estructura que se seguirá en el resto de este documento.

En el apartado 2 se definirán los conceptos básicos necesarios para la interconexión de sistemas. En él se describirán las características que definen el modelo de sistema sobre el que se ejecutarán los sistemas distribuidos que analizaremos, se presentarán otros problemas de escalabilidad y se describirán las características principales de los algoritmos de interconexión. Además, también se dará un amplio repaso a los conceptos fundamentales relativos a los modelos de memoria (tipos de acceso y modelos de consistencia) y a los diferentes tipos de difusiones fiables.

A continuación, en los apartados 3 y 4 se revisarán por orden cronológico diversas publicaciones relativas a la interconexión de sistemas distribuidos. Concretamente, en el apartado 3, se estudiará la posibilidad de interconectar sistemas distribuidos de memoria compartida, donde veremos que no todos los modelos de memoria podrán ser interconectados apropiadamente.

Posteriormente, en el apartado 4, se revisarán un total de 5 publicaciones en las que se mostrará la evolución de los protocolos de interconexión de paso de mensajes, teniendo en cuenta que las soluciones presentadas también podrán utilizarse en los sistemas de memoria compartida que estén implementados sobre un sistema de paso de mensajes. En este punto comenzaremos hablando de la importancia de reducir el tamaño de la información de control detectando los *routers* que actuarán como separadores entre zonas de procesos (grupos), para posteriormente analizar diferentes arquitecturas de interconexión. Además, también veremos que todos los grupos/sistemas podrán interconectarse mediante protocolos no intrusivos excepto los sistemas de orden total, donde la coordinación entre los nodos de un grupo será necesaria para mantener la semántica de entrega de orden total en el sistema.

Por último, en el apartado 5 se comentarán las conclusiones generales del estudio realizado comentando los detalles y conclusiones más destacables de los algoritmos de interconexión estudiados.

2. Conceptos básicos.

2.1 Introducción.

Antes de comenzar a describir las distintas estrategias de interconexión de modelos de memoria y de difusión estudiados en este trabajo, es importante conocer los pilares básicos relativos a los modelos de memoria y de los distintos tipos de difusiones fiables.

En este apartado comenzaremos definiendo las características fundamentales de los modelos de sistema sobre los que se ejecutarán los algoritmos de interconexión que estudiaremos más adelante.

Además, para entender mejor los algoritmos de interconexión de sistemas de memoria, se recordarán los diversos modelos de consistencia de memoria existentes, utilizando para ello las definiciones y descripciones proporcionadas por el autor *David Mosberger* en [38].

En cuanto a la interconexión de sistemas de paso de mensajes, se describirán los conceptos relativos a las difusiones fiables. En este caso se tomarán como fuentes las definiciones proporcionadas por los autores *V. Hadzilacos y S.Toueg* en [25].

Por último, se describirá el problema de la escalabilidad y los tipos de algoritmos de interconexión.

El conjunto de todos estos conceptos nos será de gran utilidad cuando se definan los modelos de interconexión que se estudiarán en los próximos apartados.

2.2 Modelo de sistema.

Cuando se implementa una aplicación distribuida es importante definir y conocer el entorno en el que la aplicación funcionará correctamente. Diremos que una aplicación distribuida se comporta correctamente cuando presenta el mismo comportamiento y resultados que cuando se ejecuta en un único nodo. Para ello se deberá tener en cuenta si la aplicación requiere de algún elemento de sincronización en el sistema o no, si soporta algún tipo de fallo y en tal caso de qué tipos, el modelo de consistencia que se va a emplear, semántica de orden de entrega del sistema, etc.

Así pues, el objetivo de este apartado es presentar y definir los conceptos relativos a la especificación de los modelos de sistema que irán apareciendo a lo largo de este trabajo.

2.2.1 Sistemas distribuidos y tipos de nodos de ejecución

Un sistema distribuido puede definirse como el conjunto formado por nodos independientes que proporcionarán la imagen de un sistema único y coherente:

- Independiente porque debe ser independiente de los fallos y de la asignación de recursos.

- Único porque la imagen que debe ofrecer al usuario/otras aplicaciones debe ser transparente, como si de un nodo único se tratara.

- Coherente porque un usuario/otra aplicación no tiene por qué saber donde están ubicados los datos y los debe de poder acceder en cualquier ubicación obteniendo siempre los mismo resultados.

Por otra parte, el concepto de nodo es bastante abstracto. La palabra "nodo" proviene de la teoría de grafos ya que las interacciones entre los nodos de un sistema podrán representarse mediante un grafo. Esta palabra podrá emplearse indistintamente para definir procesos, procesadores u ordenadores.

En las soluciones de interconexión de sistemas que se analizarán en los apartados 3 y 4 para mejorar la escalabilidad, como norma general, diremos que el sistema distribuido estará compuesto por nodos interconectados a través de una red. Dentro de cada uno de estos nodos es donde se ejecutarán los procesos de aplicación. Estos procesos podrán estar agrupados o no en grupos del nivel de aplicación y en uno o varios grupos del sistema de comunicación a grupos (asumiendo que se usará alguno de estos sistemas).

2.2.2 Sistemas síncronos y asíncronos

Definir el grado de sincronía de un sistema es muy importante ya que la aplicación dependerá por completo de esta propiedad. Por ejemplo, en sistemas asíncronos es muy difícil discernir si un mensaje se ha perdido o si simplemente se está retrasando porque la red está congestionada.

En los sistemas síncronos, todos los componentes avanzan siguiendo fases bien definidas, debiendo cumplir con las siguientes propiedades, necesarias para la detección de fallos por caída mediante *timeouts*:

- Existe una cota de tiempo máxima para que un proceso realice una acción.

- Cada proceso tiene un reloj local en el que la desviación con respecto al tiempo real es conocida y acotada.

- Existe una cota superior para el retraso (*delay*) de un mensaje (tiempo de envío, transporte y recepción sobre un enlace).

Diseñar una aplicación distribuida en un sistema síncrono es, en cierto modo, algo peligroso. El diseño de la aplicación será mucho más sencillo, pero hay que tener en cuenta que lo relojes no son exactos, al mismo tiempo que no se puede definir el tiempo de entrega de un canal y que es muy difícil acotar correctamente cuánto costaría enviar un mensaje a otro nodo.

Por el contrario, en los sistemas asíncronos no se asumirá ningún tipo de sincronía, es decir, no se asumirá un tiempo máximo de entrega (*delay*) por envío, ni se asumirá la existencia de relojes de ningún tipo y los procesos podrán tardar un tiempo ilimitado en ejecutar el siguiente paso.

Los sistemas asíncronos plantean más problemas a la hora de diseñar algoritmos, pero tiene una semántica más simple y permite portar con mayor facilidad una aplicación de un sistema a otro.

2.2.3 Tipos de redes de comunicaciones

El tipo de red determinará la forma en la que se comunicarán los procesos. Básicamente distinguiremos entre dos tipos: redes punto a punto y los canales de difusión.

En las redes punto a punto los procesos estarán conectados mediante enlaces que comunicarán pares de procesos. Este tipo de redes pueden modelarse mediante grafos y veremos que en los apartados 3 y 4 se suelen utilizar para demostrar el caso base en interconexión de sistemas, que será interconectar dos sistemas/grupos.

En cuanto a las redes de difusión, la comunicación entre procesos se realiza a través de un único canal que comparten todos los procesos. Ejemplos de este tipo de red son las redes Ethernet, Token Bus, Token Ring y las redes FDDI.

2.2.4 Tipos de fallo

A groso modo, los tipos de fallo de un sistema pueden clasificarse en dos tipos que abarcan tanto los procesos como a los enlaces:

- Fallos de omisión: fallos por caída, por omisión en el envío por omisión en el receptor, tanto en los procesos como en los enlaces.
- Fallos de tiempo: fallos de omisión y en los relojes.

La mayor parte de los sistemas que se describirán en este trabajo admitirán fallos por caída y por omisión, siendo tolerantes a estos tipos de fallos. A continuación se definen detalladamente estos tipos de fallo, teniendo en cuenta las definiciones de [45]:

- Fallo por caída: un proceso tiene un fallo de tipo caída si el proceso falla al ejecutar una acción. Este fallo puede no ser detectable en sistemas asíncronos, puesto que no existe ningún *timeout* que nos indique si el proceso no va o si el canal va lento.
- Fallo por omisión en un enlace: este fallo se produce cuando en un enlace l que une dos procesos p y q, un mensaje m se inserta en el buffer de salida de p pero nunca se recibe en el buffer de entrada de q.
- Fallo por omisión en el envío: un proceso comete un fallo en el envío de un mensaje m si él ejecuta el envío pero m no se inserta en el buffer de envío de mensajes.
- Fallo por omisión en la recepción: un proceso comete un fallo por omisión en la recepción de un mensaje m si m se inserta en el buffer de mensajes recibidos pero nunca se entrega al proceso.

- Fallos por omisión general: un proceso comete un fallo por omisión general cuando los enlaces funcionan correctamente pero los mensajes enviados nunca se insertan en el buffer de salida y los mensajes recibidos se insertan en el buffer de entrada pero nunca se entregan.

2.2.5 Estructura de los procesos

La Figura 1 muestra la estructura que se asumirá en los procesos de las soluciones que se presentarán en los apartados 3 y 4. Como puede observarse, en la cima de la estructura estará el nivel de aplicación, donde se estará ejecutando la propia aplicación distribuida, y por debajo se encontrará el gestor de entregas que normalmente viene proporcionado por el sistema de comunicación a grupos.

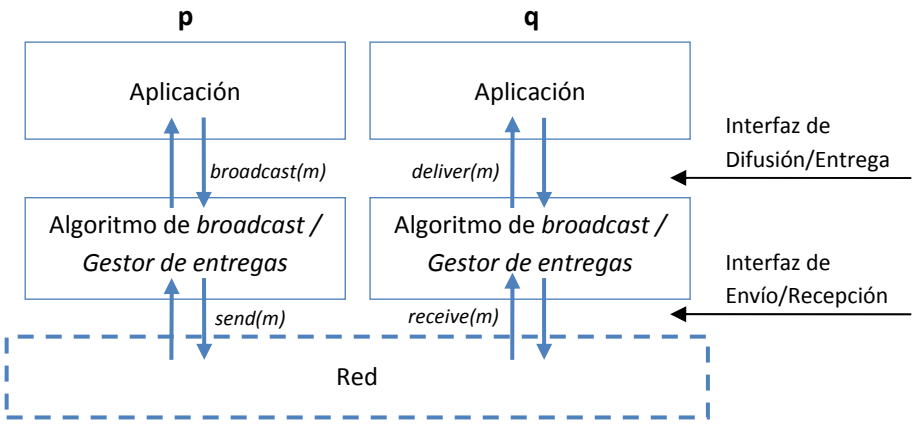


Figura 1. Estructura de comunicaciones en un sistema distribuido

El nivel de aplicación generará un evento de envío cuando desee enviar un mensaje a otros procesos. Este evento ejecutará una operación de *broadcast* de la que se encargará el gestor de entregas. El gestor enviará el mensaje a todos los procesos destinatarios a través de la red (*send*), encolándolos previamente en un buffer de salida.

Del mismo modo, cuando un proceso reciba un mensaje, éste será gestionado por el gestor de entregas, almacenándolo en un buffer de entrada de mensajes. Cuando un mensaje del buffer cumpla con las reglas de entrega, el gestor de entregas entregará el mensaje al nivel de aplicación.

El motivo por el que se diferencian ambas partes es porque de esta forma liberaremos al nivel de aplicación de tener que ordenar los mensajes recibidos para mantener las restricciones de la semántica de entrega. Además, con esta estructura tendremos bien diferenciadas las partes en las que se puede producir un fallo por omisión en el proceso e incluso se puede asumir la existencia de un evento de caída enviado por el gestor de entregas al nivel de aplicación que provocaría que el proceso se parara, siendo éste el último evento que recibiría el nivel de aplicación.

2.3 Modelos de memoria.

2.3.1 Clasificación de los tipos de accesos a memoria

En este apartado se van a describir los "*Modelos de Consistencia de Memoria*" (MCM) según [38], tal y como se ha comentado en la introducción.

En sistemas de computación paralela, cuando se quiere mantener la consistencia en la memoria compartida como si se trabajase con un solo procesador, debe emplearse el modelo secuencial. Sin embargo, este modelo de consistencia es muy estricto, de forma que el compilador no puede realizar optimizaciones para mejorar el rendimiento.

Por este motivo en ocasiones conviene trabajar con modelos de consistencia de memoria más relajados, teniendo en cuenta que el modelo de programación será más complejo y restrictivo.

La memoria compartida podrá implementarse a nivel de hardware o a nivel de software, denominándose en este último caso "*Memoria Compartida Distribuida*" (DSM).

Los MCM imponen restricciones en el orden de accesos a memoria, en base a los siguientes atributos:

- Ubicación de Acceso (*Location of Access*).
- Dirección de Acceso: lectura, escritura o lectura y escritura.
- El valor transmitido en el acceso.
- Relación de causalidad en el acceso: se utilizará el concepto de causalidad tal y como lo definió Lamport [33] para saber si dos accesos a memoria tienen relación de causalidad.
- Categoría del acceso: se distinguirán 10 categorías diferentes (ver Figura 2):

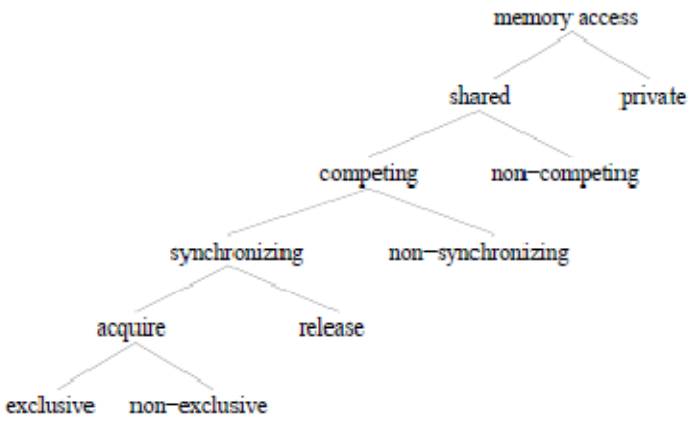


Figura 2. Categorías de Acceso. [38]

- Accesos a una variable privada: estos accesos no generan ningún conflicto, puesto que una posición de memoria de este tipo sólo podrá ser accedida por el proceso al que pertenece.

- Accesos a variable compartida: este tipo de acceso se da cuando dos accesos a memoria acceden a una misma variable compartida.

- Accesos competitivos y no competitivos: un par de accesos son competitivos y pueden generar problemas de consistencia cuando los procesos acceden a una misma variable compartida y al menos uno de los accesos implique una escritura. Si los dos fueran de lectura, no se generaría ningún conflicto.

- Accesos de sincronización o de no sincronización: los accesos de sincronización se emplearán para establecer puntos de ordenación. Por ejemplo, se pueden utilizar para paralizar los accesos nuevos hasta que no hayan terminado de ejecutarse accesos anteriores.

- Accesos de adquisición o de liberación: los accesos de sincronización se dividirán en estas nuevas categorías de acceso, en función de si estamos hablando de una operación de escritura (liberación) o de lectura (adquisición).

- Accesos exclusivos o no exclusivos: Las operaciones de adquisición podrán comportarse de forma exclusiva, si para obtener el acceso a la variable debe esperar a que se liberen el resto de accesos, o no exclusiva si puede acceder a la variable sin necesidad de esperar.

Cuando en un modelo de consistencia se distingue entre estos tipos de accesos, se estará hablando de un modelo "*híbrido*". Por el contrario, los modelos donde no se distingue el tipo de accesos, se denominarán modelos "*uniformes*", que serán los modelos con los que se van a trabajar en este documento.

2.3.2 Clasificación de los modelos de consistencia de memoria

Según [38], dentro de los MCM uniformes, existen 7 tipos diferentes de modelos de memoria. En la Figura 3 se pueden observar dichos modelos. Las flechas de un modelo A a otro B nos indican que A es más estricto que B.

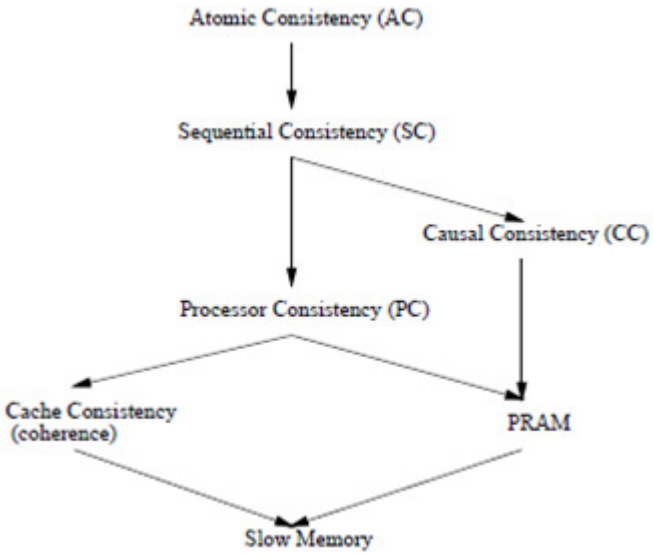


Figura 3. Estructura de Modelos Uniformes. [38]

A continuación se detallarán estos modelos de consistencia.

Consistencia Atómica (AC)

Este es el modelo más estricto de todos los propuestos y suele emplearse como modelo base pare evaluar un rendimiento de un MCM. En este modelo, una operación de acceso a memoria se llevará a cabo en un determinado intervalo o ciclo. Los accesos múltiples están permitidos, pudiéndose ocasionar conflictos con los accesos de escritura en una misma ubicación (*location*). Para resolver este problema se proponen dos soluciones:

- Consistencia atómica estática: Las operaciones de lectura se llevan a cabo al principio del intervalo y las de escritura al final.
- Consistencia atómica dinámica: En este caso las operaciones pueden llevarse a cabo en cualquier punto del intervalo, siempre y cuando el histórico de operaciones sea equivalente a una ejecución secuencial.

Consistencia Secuencial (SC)

El modelo de consistencia secuencial, fue definido por Lamport en 1979 [34]:

> *"El resultado de cualquier ejecución es el mismo que si las operaciones de todos los procesos fueran ejecutadas en algún orden secuencial, y las operaciones de cada proceso individual aparecen en esta secuencia en el orden especificado por su programa."*

La principal característica de este modelo es que todos los procesadores deben estar de acuerdo con el orden de ejecución de las operaciones de acceso a memoria. En la Figura 4 se muestra un ejemplo de diagrama de ejecución de consistencia secuencial:

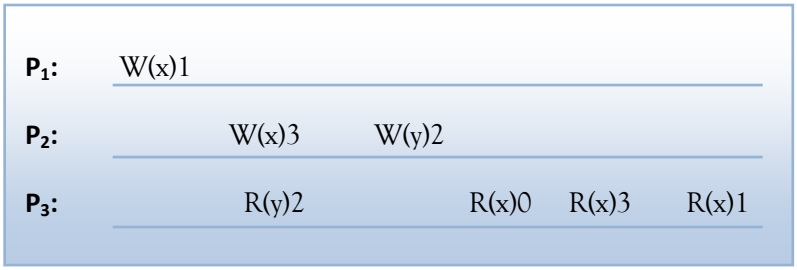


Figura 4. Ejemplo de ejecución en consistencia secuencial

En este ejemplo se destacan características particulares del modelo secuencial que hacen que se diferencie de la consistencia de memoria atómica:

- Se permite leer un valor que aún no ha sido escrito: cuando P_3 ejecuta *R(y)2*, P_2 aún no ha escrito el valor *2* en la variable *y* (operación *W(y)2*). Este caso no puede darse en un sistema físico real, pero demuestra que el modelo secuencial es más flexible de lo que puede parecer a priori.

- Se admite que P_3 acceda al valor de *x* en este orden: *R(x)0*, *R(x)3*, *R(x)1*. Aunque este orden contradice el orden físico de escritura (el orden utilizado por el modelo atómico), tal orden puede darse en el modelo secuencial, obligando a que todos los procesos que lleguen a obtener esos tres valores de la variable *x* lo hagan en dicho orden (*R(x)0*, *R(x)3*, *R(x)1*).

- Las operaciones *R(x)0* y *R(x)1* de P_3 mantienen el orden secuencial con respecto a la operación *W(x)1* de P_1.

Consistencia Causal (CC)

En 1990, *Hutto & Ahamad* [27] introdujeron por primera vez el concepto de consistencia de memoria causal. Sin embargo fue *Lamport* quien en 1978 [33] define el concepto de "relación de causalidad", también llamado relación "ocurre antes" (*happens before*), para conocer el flujo de información entre procesadores de un sistema distribuido. Formalmente, definiremos la relación *happens before* de la siguiente forma:

> En un sistema distribuido en el que la información sólo se intercambia mediante paso de mensajes, un mensaje m_1 precederá causalmente a otro mensaje m_2 ($m_1 \rightarrow m2$) sólo si se cumple alguna de las siguientes condiciones:
>
> - m_1 y m_2 son enviados por el mismo proceso y m_2 es enviado después de m_1.
> - m_1 es entregado (*delivered*) en el proceso emisor de m_2 antes de que m_2 sea enviado.
> - Existe un mensaje *x* tal que $m_1 < x$ y $x < m_2$.

Para aplicarlo en un sistema de memoria, una escritura sería el equivalente a un evento de mensaje de envío, mientras que una lectura se comportaría como un evento de recepción de mensaje.

La memoria será causalmente consistente cuando todos los procesadores estén de acuerdo en el orden causal de los eventos dependientes. En cambio, los eventos independientes o concurrentes podrán observarse en órdenes diferentes.

En el siguiente ejemplo (Figura 5) se muestra un ejemplo de ejecución válido en consistencia causal, pero no en consistencia secuencial:

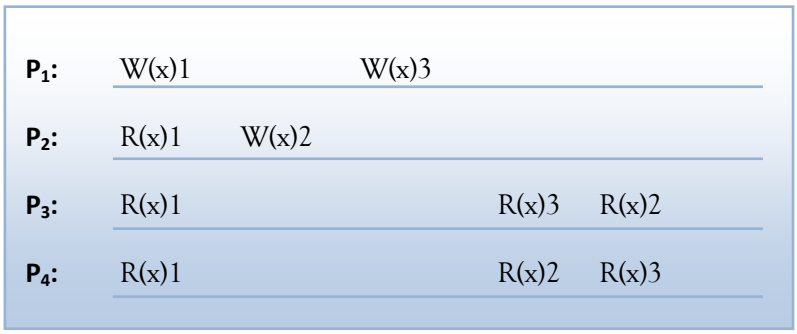


Figura 5. Ejemplo de ejecución en consistencia causal

En este ejemplo se observa que:

- Las operaciones *W(x)1* y *W(x)2* están relacionadas causalmente puesto que P_2 ejecuta *R(x)1*, accediendo al valor escrito previamente en P_1.
- Además, P_3 y P_4 observan con distinto orden las escrituras generadas por P_1 (*W(x)3*) y P_2 (*W(x)2*), siendo esto un comportamiento no válido en la consistencia secuencial.

El modelo de consistencia causal es probablemente el modelo más difícil de implementar en hardware. Esto es debido a que el resto de modelos de consistencia uniformes han sido diseñados teniendo en mente una implementación física previa. Sin embargo, esto no significa que tenga que tener peor rendimiento que el resto de modelos.

Consistencia Pipelined RAM (PRAM)

El modelo de consistencia PRAM se definió en 1988 por *Lipton & Sandberg* [37]. Este modelo se basa en un sistema multiprocesador en el que cada procesador contiene una copia local de la memoria compartida. Para fomentar la escalabilidad, un acceso debe ser independiente del tiempo que tarde en acceder a las memorias del resto de procesadores. Teniendo en cuenta esto, los autores definen las siguientes reglas para el modelo PRAM:

- Ante una petición de lectura, el procesador devolverá el último valor almacenado en su copia de la memoria.

- Ante una operación de escritura, el procesador primero actualizará su copia local y después realizará un *broadcast* al resto de copias.

En términos de orden, esto se traduce en que todos los procesadores observan las operaciones de escritura generadas por un mismo procesador en el mismo orden, mientras que las escrituras generadas por procesadores distintos podrán ser percibidas en distinto orden.

El diagrama de ejecución mostrado en la Figura 6 muestra una ejecución legal en el modelo PRAM pero ilegal en el de consistencia causal porque:

- Los procesadores P_3 y P_4 observan las escrituras de P_1 y P_2 en distinto orden, a pesar de que estas escrituras están relacionadas causalmente debido a la operación *R(x)1* que se ejecuta en P_2 (imposible en el modelo CC).
- P_4 mantiene el orden de las escrituras de un mismo procesador (en este caso, P_2).

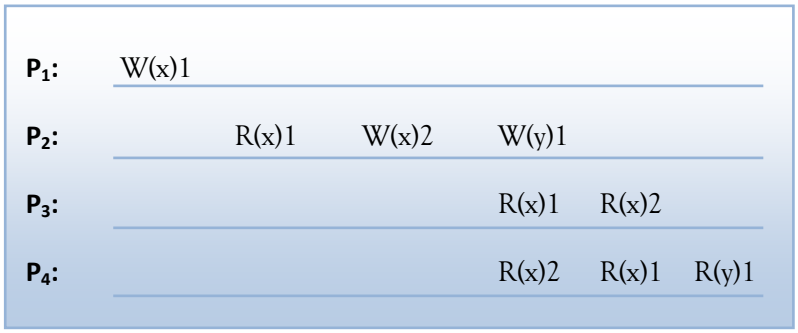


Figura 6. Ejemplo de ejecución en consistencia PRAM

Consistencia de Caché o Coherencia

La consistencia de caché o coherencia es un modelo estrictamente más débil que el modelo secuencial. La filosofía de este modelo es eliminar las restricciones de orden que el orden de programa impone para acceder a distintas posiciones de memoria. Para ello, los accesos mantendrán la consistencia secuencial siempre y cuando accedan a la misma posición de memoria.

Este modelo de consistencia fue definido por *Goodman* en 1989 como consistencia de caché [24] y bautizado como coherencia en 1990 [23].

Un modelo de memoria con consistencia secuencial siempre cumplirá con el modelo de consistencia de coherencia, pero esta relación no se cumple al revés. En la Figura 7 se muestra un diagrama de ejecución que cumple con la consistencia coherente, pero no con la secuencial ya que se puede asumir que la variable *x* puede ser accedida con el orden de ejecución *R(x)0,W(x)1* y que la variable *y* puede ser accedida con el orden *R(y)0,W(y)1*.

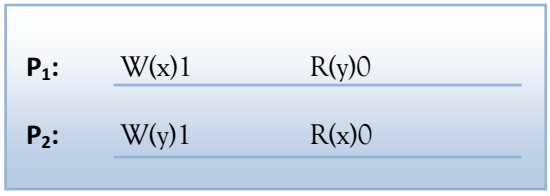


Figura 7. Ejemplo de ejecución en consistencia de caché o coherencia

Consistencia de Procesador (PC)

Durante mucho tiempo el modelo de consistencia secuencial se ha utilizado como modelo de referencia. Sin embargo, con la llegada de las máquinas multiprocesador, el modelo de consistencia de procesador (PC) es empleado cada vez más, puesto que es un modelo algo más relajado que el secuencial.

Históricamente, la consistencia de procesador es definida por primera vez en 1989 por *Goodman* en [24], aunque de una manera informal, provocando cierta confusión. Más tarde, en el año 1992, *Ahamad et al.* [4] definieron formalmente este modelo de consistencia.

Otros autores (*DASH group* [23] han expuesto su propia definición del modelo de consistencia de procesador, pero dista mucho de la definición propuesta por *Goodman*. En este caso, *Mosberger* decide utilizar la versión de *Goodman*.

Según *Goodman*, la consistencia de procesador es más fuerte que la consistencia de coherencia, y como ya hemos dicho antes, más débil que la consistencia secuencial. Además, un histórico de ejecuciones que cumpla con el modelo PC, también será coherente y PRAM. Sin embargo, no siempre que se cumpla el modelo de coherencia y el PRAM, se cumple el modelo PC.

En el modelo PC, los procesadores deben de estar de acuerdo con el orden de escritura de cada procesador, pero pueden diferir en el orden de escritura de procesos diferentes, siempre y cuando las escrituras se realicen en distintas posiciones de memoria.

Como ejemplo de diagrama de ejecución, el diagrama visto en el caso de la consistencia de coherencia también cumple con la consistencia de procesador. En la Figura 8 se muestra un ejemplo de diagrama en el que no se cumple la consistencia de procesador:

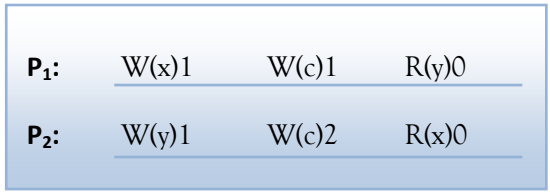


Figura 8. Ejemplo de ejecución sin consistencia de procesador

El orden de los accesos para cada procesador es:

- P_1: W(x)1, **W(c)1**, R(y)0, W(y)1, **W(c)2**

- P2: W(y)1, **W(c)2**, R(x)0, W(x)1, **W(c)1**

En el orden de los accesos se observa que el orden de las escrituras en la variable *c* sigue un orden distinto en cada procesador, con lo que no se cumpliría con las especificaciones del modelo PC.

Memoria lenta (Slow memory)

El nombre de este modelo de consistencia de memoria se debe a que las operaciones de escritura se propagan muy lentamente por el sistema.

En el modelo *Slow-Memory*, todos los procesadores deberán estar de acuerdo en el orden de acceso de las operaciones de escritura en cada posición de memoria por cada procesador. Además, las escrituras locales serán visibles inmediatamente, tal y como se estipula en el modelo PRAM.

Este modelo es menos estricto que el modelo PRAM y no se usa habitualmente, ya que, aunque con el algoritmo de exclusión mutua diseñado por *Hutto & Ahamad* [27] puede utilizarse para la comunicación entre procesos, este algoritmo sólo garantiza exclusión a nivel físico.

2.4 Difusiones fiables.

2.4.1 Tipos de difusiones (*Broadcast*)

Para mantener la consistencia de memoria entre los distintos componentes (subsistemas o grupos) de un mismo sistema distribuido, hay que tener en cuenta que las operaciones de escritura y de lectura sobre una variable en la memoria local deben replicarse en la memoria de dichos componentes. Para ello, las operaciones necesarias para mantener la consistencia de memoria deberán difundirse de un subsistema a otro manteniendo un orden adecuado, teniendo en cuenta factores como el modelo de consistencia que se esté utilizando o el número de grupos con los que se esté trabajando.

Así pues, las difusiones fiables nos permitirán mantener un orden adecuado en el paso de mensajes y, por tanto, cumplir con el modelo de consistencia de memoria que se haya definido.

A continuación se listan los tipos de *broadcast* siguiendo las definiciones proporcionadas por *Hadzilacos & Toueg* en 1993 [25], donde se asume que las definiciones son correctas si sólo se trabaja con fallos benignos (sin fallos bizantinos).

Difusión Fiable (Reliable Broadcast)

Este tipo de difusión es la más débil de todas, o lo que es lo mismo, el resto de tipos de *broadcast* se basan en el *reliable broadcast* y, por lo tanto, todos los tipos cumplirán las siguientes 3 propiedades:

- Acuerdo (*Agreement*): Todos los procesos estarán de acuerdo con el conjunto de mensajes que han recibido, o lo que es lo mismo, todos los procesos reciben exactamente los mismos mensajes.
    - Definición formal: Si a un proceso correcto se le entrega un mensaje *m,* entonces al resto de procesos correctos también se le entrega *m*.

- Validez (*Validity*): Todos los mensajes enviados por un proceso correcto se entregarán al resto de procesos, incluido el proceso emisor.
    - Definición formal: Si un proceso difunde el mensaje *m*, a este proceso también se le entrega *m*.

- Integridad (*Integrity*): No se entregarán mensajes falsos.
    - Definición formal: Para cualquier mensaje correcto *m*, cualquier proceso correcto recibe (*delivers) m* como máximo una vez, y sólo si previamente ha sido difundido (*broadcast)* por un emisor *sender(m)*.

Para difundir y entregar mensajes se utilizarán dos primitivas básicas:

- *Broadcast:* Un proceso *p* difundirá un mensaje *m*, siendo *m* un mensaje sacado de la cola de espera de mensajes *M*.

- *Deliver:* Esta operación se da cuando a un proceso *q* se le entrega el mensaje *m.*

Dado que un mensaje se puede perder o simplemente porque se quiere evitar la recepción de mensajes duplicados, es importante que dichos mensajes contengan un identificador único. Así pues, cada mensaje contendrá los siguientes campos que, unidos, lo identificarán del resto de mensajes:

- Identidad del proceso que lo envía (*sender(m)*).

- Un número de secuencia que permita distinguirlo del resto de mensajes de un mismo proceso (*seq(m)*).

Es importante darse cuenta de que en el caso de que el emisor de una difusión falle, todos los procesos mantendrán la consistencia, puesto que a todos los procesos correctos se les habrá entregado el mensaje difundido o a ninguno, produciéndose esto último en el caso de que el proceso que realiza el broadcast falle tras ejecutar una operación de broadcast que nunca llegó a procesarse.

Difusión FIFO (FIFO Broadcast)

En las difusiones fiables, los mensajes se reciben sin seguir ningún tipo de orden. Sin embargo, en ocasiones el orden en el que se reciben los mensajes sí que es relevante, y por eso es importante observar y conocer el contexto del mensaje antes de entregar un mensaje (*deliver*), ya que los procesos que desconozcan el contexto de un mensaje deberán ignorarlo (recibirlo pero no entregarlo).

En el caso del broadcast de tipo FIFO, el contexto de un mensaje *m* se refiere a los mensajes que previamente ha enviado el emisor de ese mensaje (*sender(m)*). Un ejemplo que ilustra este comportamiento sería un sistema de reservas. Si un usuario quiere pagar una reserva que aún no ha sido pagada o simplemente cancelar una reserva, previamente tiene que haber hecho la reserva. Por tanto, al realizar la petición de cancelación y teniendo en cuenta las propiedades de la difusión fiable y el contexto del mensaje, el usuario jamás conectará con un *site* que desconozca que previamente se ha hecho una reserva.

Para convertir una difusión fiable en una difusión fiable con orden FIFO, además de las propiedades definidas para las difusiones fiables, hay que añadir la siguiente nueva propiedad de orden:

- Orden FIFO: Si un proceso *p* difunde (*broadcast*) un mensaje m_1 antes de difundir otro mensaje m_2, entonces a ningún proceso correcto se le entrega m_2 a no ser que se le haya entregado m_1 previamente.

Nótese que esta definición del orden FIFO es muy completa ya que no sólo dice que los mensajes deben entregarse en el mismo orden en el que los difunde el proceso *p,* sino que además no permite que se entregue un mensaje *m* si previamente no se ha entregado el mensaje anterior tal y como indica el contexto de *m*. Esto se ve más claro en el siguiente ejemplo:

- El proceso *p* difunde tres mensajes en el siguiente orden: m_1, m_2 y m_3.
- El proceso *p* falla transitoriamente al difundir el mensaje m_2.
- Si se cumple con la especificación de forma estricta, a todos los procesos les será entregado el mensaje m_1. Sin embargo, m_3 se recibirá (*receive*) pero jamás se entregará (*deliver*), puesto que su contexto nos indica que se entregará tras entregar m_2, y m_2 no se ha podido enviar por parte del proceso *p*, con lo que jamás se recibirá. No obstante, este comportamiento es poco práctico si el proceso *p* todavía funciona (tras el fallo no ha caído o se ha parado), ya que lo normal es que *p* todavía mantenga m_2 en sus buffers de envío y que cuando algún proceso reciba m_3 le envíe un reconocimiento negativo (solicitando m_2) y m_2 sea redifundido. Es decir, m_3 no se entregaría hasta que m_2 se entregase, pero no sería descartado.

Difusión causal (Causal Broadcast)

El orden FIFO no es suficiente cuando un mensaje m_1 no sólo depende de un mensaje m_2 difundido previamente por un mismo proceso *p*. Puede darse el caso de que el contexto de m_1 sea más amplio, dependiendo también de los mensajes que se han entregado (*delivered*) a *p* antes de difundirse m_1 (*broadcast*), o lo que es lo mismo, generando una relación causal entre las operaciones de difusión y de entrega.

Este comportamiento se corresponde con la difusión causal y es más estricto que la difusión de orden FIFO. Así pues, una difusión causal se considerará una difusión fiable que además cumple con la siguiente propiedad de orden:

- Orden causal: Si la difusión de un mensaje m_1 precede causalmente a la difusión de un mensaje m_2, entonces a ningún proceso correcto se le entregará m_2, a no ser que previamente se le haya entregado m_1.

La definición de orden causal es más general que la del orden FIFO. De hecho, el orden causal es el equivalente a la unión de la restricción de orden FIFO junto con la restricción del orden Local, definido a continuación:

- Orden local: Si un proceso difunde un mensaje m_1, y a un proceso se le entrega m_1 antes de difundir m_2, entonces a ningún proceso correcto se le entregará m_2, a no ser que previamente se le haya entregado m_1.

Por último, destacar que comprobar que se cumple el orden causal considerando el orden causal como la unión del orden FIFO junto con el orden Local, es mucho más fácil que comprobar que se cumple el orden causal directamente.

Difusión Atómica (Atomic Broadcast)

Si dos mensajes difundidos no tienen relación causal, estos mensajes pueden entregarse en dos procesos en distinto orden. Sin embargo, en determinadas aplicaciones es conveniente que todos los procesos correctos estén de acuerdo con el orden de entrega de los mensajes. En este caso, el orden causal no es suficiente, siendo necesario el orden total:

- Orden total: Si dos procesos correctos p y q reciben (*deliver*) dos mensajes m_1 y m_2, entonces m_1 se entrega a p antes que m_2 sólo si a q se le entrega m_1 antes que m_2.

El orden total junto con la propiedad de acuerdo (*Agreement*), garantizan que a todos los procesos correctos se les entrega la misma secuencia de mensajes.

Difusión Atómica FIFO (FIFO Atomic Broadcast)

Este tipo de difusión consiste en añadir el orden FIFO a las difusiones atómicas, siendo por tanto más fuerte (restrictiva) que la difusión atómica y la difusión fiable con orden FIFO.

Utilizando la difusión atómica FIFO se evita el caso que se muestra a continuación, que sí que se podría presentar en un sistema con difusiones atómicas:

- Un proceso p sufre un fallo transitorio mientras realiza la difusión del mensaje m_1, y a continuación envía m_2.

- A todos los procesos correctos se les entregará m_2, a pesar de que nunca recibirán m_1.

Difusión Atómica causal (Causal Atomic Broadcast)

Por el mismo motivo que surge la difusión atómica FIFO, existe la difusión atómica causal. Este tipo de difusión consiste en unir el orden total junto con el orden causal en las difusiones fiables. Con esto se consigue mantener la relación de orden causal entre las operaciones de difusión (*broadcast*) y de entrega (*deliver*) mientras que todos los procesos están de acuerdo en el orden de la secuencia de mensajes que se les entregará.

La difusión atómica causal es más fuerte que la difusión atómica FIFO y que las difusiones causales.

Otras difusiones

Aparte de los tipos de difusión explicados anteriormente, existen las difusiones acotadas (Δ-*timeliness*) por un determinado intervalo de tiempo (*Timed Broadcast*). En los sistemas que emplean esta clase de difusiones, cuando un proceso difunde un mensaje, el mensaje debe de ser entregado en un determinado periodo de tiempo, siendo descartado si no llega a tiempo.

En los modelos de interconexión que se van a analizar en este trabajo no se utilizan este tipo de difusiones y, por tanto, tampoco se van a dar más detalles.

2.4.2 Difusiones uniformes (*Uniform Broadcast*)

Las propiedades de acuerdo, integridad y orden vistas en el punto anterior sólo hacen referencia a los procesos correctos del sistema, no teniendo en cuenta a los posibles procesos fallidos. Este comportamiento puede no ser deseable en determinadas aplicaciones en las que estas propiedades tengan que ser más robustas para que los procesos fallidos también las cumplan. Diremos que una propiedad es uniforme cuando se cumple tanto en los procesos correctos como en los procesos con fallos benignos.

A continuación se reformulan las propiedades de acuerdo e integridad para hacerlas uniformes:

- Acuerdo uniforme (*Uniform Agreement*): Si a un proceso (correcto o fallido) se le entrega un mensaje *m,* entonces al resto de procesos correctos también se le entrega *m*.

- Integridad uniforme (*Uniform Integrity*): Para cualquier mensaje *m*, cualquier proceso (correcto o fallido) recibe (*delivers) m* como máximo una vez, y sólo si ha sido difundido (*broadcast)* por un emisor *sender(m)*.

Observar que cuando se cumple la propiedad de “acuerdo uniforme” se garantiza la entrega del mensaje *m* a los procesos correctos, sin mencionar a los fallidos. Esto es

debido a que son los únicos que siguen activos una vez ha ocurrido un fallo, ya que los fallidos estarán caídos.

Así mismo, las propiedades de orden que caracterizan a los distintos tipos de difusiones fiables también puede reformularse para ser uniformes:

- Orden FIFO uniforme (*Uniform FIFO Order*): Si un proceso p difunde (*broadcast*) un mensaje m_1 antes de difundir otro mensaje m_2, entonces a ningún proceso (correcto o fallido) se le entrega m_2 a no ser que se le haya entregado m_1 previamente.

- Orden local uniforme (*Uniform Local Order*): Si un proceso difunde un mensaje m_1, y un proceso recibe (*deliver*) m_1 antes de difundir m_2, entonces a ningún proceso (correcto o fallido) se le entregará m_2, a no ser que previamente se le haya entregado m_1.

- Orden causal uniforme (*Uniform causal Order*): Si la difusión de un mensaje m_1 precede causalmente a la difusión de un mensaje m_2, entonces a ningún proceso (correcto o fallido) se le entregará m_2, a no ser que previamente se le haya entregado m_1.

- Orden total uniforme (*Uniform total Order*): Si dos procesos (correctos o fallidos) p y q reciben (*deliver*) dos mensajes m_1 y m_2, entonces m_1 se entrega a p antes que m_2 sólo si a q se le entrega m_1 antes que m_2.

Por último, destacar que la propiedad *Δ-timeliness* también puede reformularse para hacerla uniforme, pero las difusiones con cotas temporales no se van a tratar en este trabajo, con lo que se omite su definición.

2.4.3 Equivalencias entre difusiones

En este apartado se va a mostrar cómo están relacionados los distintos tipos de difusiones. Todas ellas se basan en el tipo de difusión fiable, que es el tipo de difusión más simple, con lo que todas cumplen con las propiedades de validez, acuerdo e integridad.

La única propiedad que diferencia un tipo de difusión de otra es el tipo de orden que se utiliza. Así pues:

- Difusión Fiable = Validez + Acuerdo + Integridad

- Difusión FIFO = Difusión Fiable + orden FIFO

- Difusión causal = Difusión Fiable + orden causal

Además, las anteriores difusiones se pueden convertir en Atómicas:

- Difusión Atómica = Difusión Fiable + orden total

- Difusión FIFO Atómica = Difusión FIFO + orden total
- Difusión causal Atómica = Difusión causal + orden total

En la Figura 9 se muestran las relaciones entre las distintas difusiones y los tipos de orden vistos.

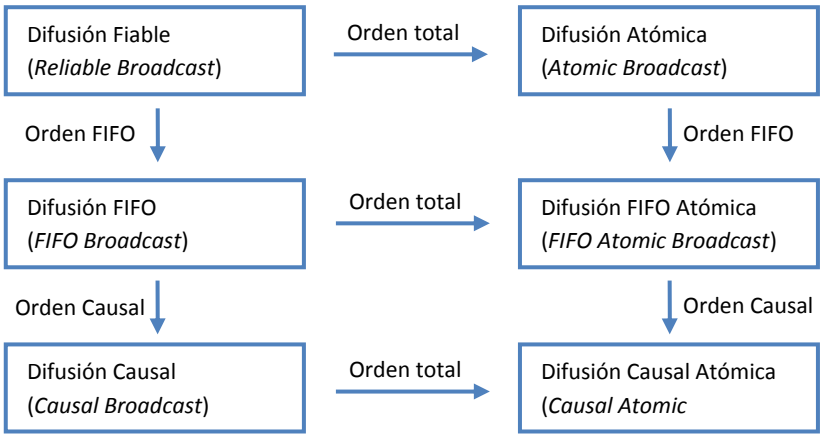


Figura 9. Diagrama de relaciones entre tipos de *broadcast*

Además, en el punto anterior se han reformulado todas las propiedades relativas a las difusiones (excepto la propiedad de Validez). Así pues, una difusión fiable será uniforme siempre que todas sus propiedades sean uniformes, excepto la propiedad de Validez.

2.4.4 Difusiones a grupos (*Multicast*)

En las difusiones *broadcast*, el mensaje que se difunde va dirigido a todos los procesos del sistema. Sin embargo, en ocasiones es preciso realizar una difusión a un conjunto de procesos (grupo), no siendo necesario difundirlo al resto de procesos del sistema. Por este motivo surgen las difusiones a grupos o *multicasts*.

Formalmente, un grupo G es la representación de un subconjunto de procesos del sistema. Un proceso p es miembro de G ($p \in G$), cuando p está en el subconjunto de procesos llamado G. Para que un mensaje m se difunda (*multicast*) a un determinado grupo G, deberá contener un campo que indique el nombre del grupo (*group(m)*).

Cómo se añade o se elimina un proceso de un grupo será un problema del que no nos tendremos que preocupar a la hora de hacer una difusión *multicast*, puesto que de este problema se hará cargo el protocolo de pertenencia a grupos que se esté empleando ([20],[43]). Además, gracias a este protocolo, cada proceso sabrá a qué grupos pertenece (un proceso puede pertenecer a varios grupos), y quienes son los procesos restantes que pertenecen al grupo.

Como en el caso del *broadcast*, con *multicast* se trabajará con dos primitivas:

- *Multicast*: Cuando el proceso *p* realiza un *multicast* de un mensaje *m*, diremos que *p multicasts m* a *grupo(m)*, siendo *m* un mensaje obtenido de una cola de mensajes *M*.

- *Deliver*: Un proceso *q* recibe un mensaje *m* en *group(m)* tras ejecutar la operación *deliver(m)*.

Estas definiciones son válidas si se asume que se trabaja con fallos benignos (fallos no arbitrarios).

Del mismo modo que existen distintos tipos de difusiones *broadcast*, también existen diversos tipos de difusiones *multicast*. A continuación se definen estos tipos de *multicast*.

Difusión Fiable a grupos (Reliable Multicast)

La difusión fiable *multicast* es el equivalente a la difusión fiable *broadcast* aplicada a grupos. Este tipo de difusión a grupos requiere que a todos los procesos correctos de un grupo *G* se les entregue el mismo conjunto de mensajes. Este conjunto de mensajes incluirá todos los mensajes difundidos (*multicast*) en *G*. Además, ningún mensaje engañoso/modificado malintencionadamente será entregado (*delivered*).

Basándonos en las primitivas de difusión a grupos *(multicast*) y entrega (*deliver*), la difusión fiable a grupos cumple con las siguientes propiedades:

- Acuerdo (*Agreement*): Si a un proceso correcto se le entrega *m,* entonces *m* será entregado a todos los procesos correctos del grupo *group(m)*.

- Validez (*Validity*): Si un proceso correcto difunde un mensaje *m* a un grupo, entonces, a algún proceso correcto del grupo *group (m)* se le entregará *m* o, de lo contrario, ningún proceso en este grupo será correcto.

- Integridad (*Integrity*): Para cualquier mensaje *m*, cualquier proceso correcto *p* recibirá (*delivers*) *m* como máximo una vez, sólo si *p* pertenece a *group(m)* y *m* ha sido difundido (*multicast*) previamente por *sender(m)*.

La difusión a grupos o *multicast* será el equivalente a hacer una difusión a todos los procesos del sistema (*broadcast*), si se asume que todos los procesos de un sistema conforman un único grupo *G*.

Un detalle importante de la difusión fiable a grupos es que en ocasiones un proceso tiene que enviar un mensaje a un grupo al que no pertenece. Un ejemplo claro de este caso se daría en una aplicación que sigue la arquitectura cliente-servidor. Por este motivo, la difusión fiable a grupos no requerirá que el emisor de un mensaje *m* sea un miembro de *group(m)*.

Difusión FIFO a grupos (FIFO Multicast)

La difusión FIFO a grupos es una difusión *multicast* teniendo en cuenta el orden FIFO, es decir, a un proceso *p* sólo se le entrega un mensaje *m* si previamente se le han entregado todos los mensajes difundidos por el emisor *sender(m)*.

En las difusiones a grupos con orden FIFO, existen dos tipos de orden FIFO:

- Orden global FIFO: Si un proceso difunde (*multicast*) un mensaje m_1 antes de difundir (*multicast*) un mensaje m_2, entonces en ningún proceso correcto en $grupo(m_1)$ se entregará m_2, a no ser que se haya entregado previamente m_1.

- Orden local FIFO: Si un proceso difunde (*multicast*) un mensaje m_1 antes de difundir (*multicast*) un mensaje m_2 tal que $group(m_1)=group(m_2)$, entonces en ningún proceso correcto se entregará m_2, a no ser que se haya entregado previamente m_1.

El orden local FIFO es menos restrictivo que el orden global FIFO, puesto que sólo afectará a los mensajes dentro de un mismo grupo.

Difusión causal a grupos (causal Multicast)

Este tipo de difusion a grupos surge cuando hay que tener en cuenta la precedencia causal de los mensajes. Por tanto, la difusión causal a grupos consiste en añadir el orden causal a la difusión fiable a grupos, de forma similar a como se hacía en el caso de los *broadcasts*. Como en el caso del *FIFO Multicast*, se distingue entre orden causal global y orden causal local:

- Orden causal global: Si el *multicast* de un mensaje m_1 precede causalmente el *multicast* de un mensaje m_2, entonces a ningún proceso correcto en $group(m_1)$ se le entregará m_2, a no ser que se haya entregado previamente m_1.

- Orden causal local: Si el *multicast* de un mensaje m_1 precede causalmente en $group(m_1)$ el *multicast* de un mensaje m_2, entonces a ningún proceso correcto se le entregará m_2, a no ser que se haya entregado previamente m_1.

Tal y como ocurría en las difusiones *FIFO Multicast*, el orden local causal se considera un orden más débil que el orden Global causal.

Difusión Atómica a grupos (Atomic Multicast)

En la difusión atómica a grupos también se distingue entre distintos tipos de orden de entrega de mensajes:

- *Local Atomic Multicast*: Es una difusión fiable a grupos que cumple con la propiedad de orden total local. Las difusiones atómicas a grupos en el sistema *Isis* utilizan este tipo de orden ([12],[13]):

- *Orden total local:* Si a dos procesos correctos p y q se les entregan dos mensajes m_1 y m_2 y $grupo(m_1)=grupo(m_2)$, entonces a p se le entrega m_1 antes que m_2 sí y sólo sí a q se le entrega m_1 antes que m_2.

- *Pairwise Atomic Multicast*: El orden total local permite que los procesos de un mismo grupo se pongan de acuerdo con el orden de entrega de los mensajes. Por ejemplo, suponemos que existen dos grupos $G_1=\{p,q,r\}$ y $G_2=\{p,q,s\}$ y que r difunde m_1 en G_1 y s difunde m_2 *en* G_2. El orden total local permitiría que los procesos del grupo G recibiesen m_1 y m_2 en orden distinto con respecto a G_2: p y q en G_1 podrían recibir m_1 antes que m_2, mientras que p y q en G_2 podrían recibir primero m_2 y luego m_1.
  Sin embargo, en ocasiones necesitamos que el orden en la entrega entre procesos pertenecientes a 2 grupos sea el mismo, y es por este motivo que surge la difusión atómica a grupos entre pares de procesos. Esta difusión es una difusión fiable a grupos que además cumple con la siguiente propiedad de orden:

  - *Pairwise total order*: Si a dos procesos correctos p y q se les entrega m_1 y m_2, entonces a p se le entregará m_1 antes que m_2 si y sólo si a q se le entrega m_1 antes que m_2.

- *Global Atomic Multicast*: El orden total entre pares de grupos no es suficiente, ya que en ocasiones puede provocar ciclos. Por ejemplo, vamos a asumir que tenemos 3 grupos de procesos cuya intersección entre pares de grupos consiste en un solo proceso: $G_1=\{p,q\}$, $G_2=\{q,r\}$ y $G_3=\{r,p\}$. En cada grupo se lanza m_1, m_2 y m_3 respectivamente. El orden total entre pares permite que al proceso p se le entregue m_3 antes que m_1, que a q se le entregue m_1 antes que m_2 y que a r se le entregue m_2 antes que m_3.

  Para evitar estos ciclos, surge este tipo de difusión *multicast*. Si consideramos el conjunto de mensajes entregados a procesos correctos, se define la relación de precedencia "$\rightarrow$" en este conjunto de la siguiente forma: $m_1 \rightarrow m_2$ si y sólo si a cualquier proceso correcto se le entrega m_1 antes que m_2, en este orden.

  De nuevo, la difusión a grupos atómica global es una difusión fiable a la que se le aplica el siguiente orden:

  - *Orden total global:* la relación "$\rightarrow$" es acíclica.

  Al evitar los ciclos con la relación "$\rightarrow$", se permite que los mensajes puedan ser globalmente ordenados en todos los procesos correctos del sistema.

Por último, nótese que el orden total global es estrictamente más fuerte que el orden total entre pares de grupos que, a su vez, es estrictamente más fuerte que el orden total local. Sin embargo, ninguno de estos órdenes garantiza el orden FIFO. Así pues, combinando las dos modalidades de orden FIFO con las tres modalidades de orden total, se obtendrían seis nuevos tipos de orden (*FIFO Atomic Multicasts*), de los cuales

sólo tendrían sentido el orden FIFO local con el orden total local y el orden FIFO global con el orden total global.

Del mismo modo se pueden definir seis tipos de difusiones causales atómicas a grupos (*Causal Atomic Multicasts*) y como en el caso anterior, solamente tendrían sentido combinar las versiones locales de ambos tipos de orden o combinar las versiones globales.

Otras difusiones a grupos

Como en el caso de las difusiones *broadcasts*, también se pueden poner cotas temporales (*Δ-timeliness*) a la hora de difundir un mensaje en un grupo (*multicast*).

Además, también se puede definir la versión uniforme de las propiedades *Agreement, Integrity, Order y Δ-timeliness* en las difusiones a grupos de forma similar a como se hacía en las difusiones *broadcast*.

2.4.5 Consenso (Consensus)

El problema del consenso consiste en que todos los procesos de un mismo grupo se pongan de acuerdo en un valor propuesto. Un ejemplo altamente conocido es el problema de la elección de líder, en el que cada proceso de un mismo grupo propone un proceso líder ante el resto de procesos y entre todos seleccionan el proceso líder.

En términos más formales, se puede definir el problema del consenso a través de dos primitivas:

- Propuesta (*propose*): Si un proceso *p* invoca *propose(v),* estaremos indicando que *p* propone el valor *v*. Este valor se tomará de algún conjunto *V*.
- Decisión (*decide*): Cuando un proceso *q* retorna de la ejecución de la primitiva *decide* con el valor *v*, diremos que *q* decide *v* (*decide(v)*).

En el problema del consenso, cuando un proceso propone un valor, se deben cumplir las siguientes condiciones:

- Terminación (*termination*): cualquier proceso correcto, normalmente decide un solo valor.
- Acuerdo (*agreement*): si un proceso correcto decide *v,* entonces todos los procesos correctos deciden *v*.
- Integridad (*integrity*): si un proceso correcto decide *v,* entonces *v* fue propuesto anteriormente por algún proceso.

En el problema del consenso también se pueden fortalecer las propiedades de acuerdo e integridad exigiendo uniformidad:

- Acuerdo uniforme (*uniform agreement*): si un proceso (correcto o fallido) decide *v,* entonces todos los procesos correctos deciden *v.*

- Integridad uniforme (*uniform integrity*): si un proceso (correcto o fallido) decide *v,* entonces *v* fue propuesto anteriormente por algún proceso.

La característica más destacable del problema del consenso es que este problema, bajo ciertas condiciones, es equivalente a la difusión atómica (*Atomic Broadcast*):

- Transformar de Difusión Atómica a Consenso:

    - Se tolera cualquier tipo de fallos benignos.

- Transformar de Consenso a Difusión Atómica:

    - Se asumen difusiones fiables.

    - Sólo pueden ocurrir fallos de tipo parada. En este punto los autores comentan que actualmente, con una transformación más compleja, se admiten cualquier tipo de fallos [16].

Estas transformaciones no asumen ningún tipo de sincronía en la red, pero se debe tener en cuenta que, en redes asíncronas punto a punto con fallos de tipo parada, la difusión atómica no se puede resolver, pero sí se puede resolver empleando detectores de fallos.

2.5 El problema de la escalabilidad.

El problema de la escalabilidad viene dado originalmente por los sistemas causales. La implantación de este tipo de sistemas se ha ido incrementando a lo largo del tiempo, sobre todo en sistemas asíncronos donde la relación de causalidad entre mensajes al menos proporciona un mínimo de garantía de entrega de un mensaje. Por ejemplo, si un proceso p envía un mensaje m_1 a un proceso q, p no puede saber si m_1 se ha entregado, se ha perdido o si todavía está viajando por la red porque en un sistema asíncrono el tiempo no está acotado. Sin embargo, si a q se le entrega $m1$ y esto provoca que q envíe un mensaje m_2 a p, p sabrá que m_1 se ha entregado en q porque m_2 estaba relacionado causalmente con m_1 ($m_1 \rightarrow m_2$).

El motivo por el que en el ejemplo anterior p sabe que $m_1 \rightarrow m_2$ es que para saber qué mensajes están relacionados causalmente se debe almacenar una determinada información de control que irá aumentando de tamaño conforme aumente el número de procesos y el número de mensajes entre ellos.

Así pues, en grandes sistemas donde el número de procesos es muy grande y se genera una gran cantidad de mensajes, la información de control crece descontroladamente en los procesos, incrementando también el tamaño de los mensajes, llegando un punto en el que el sistema deja de ser escalable. Cuando un sistema deja de ser escalable, el hecho de añadir más procesos no mejorará la productividad del sistema, sino que incluso la empeorará debido a que las difusiones se vuelven muy costosas.

Este problema también ocurre en sistemas no causales donde, a pesar de que no se necesita almacenar información de control causal, difundir un mensaje a una gran cantidad de procesos genera un gran tráfico de red, además de que los mensajes deberán atravesar distintas redes con velocidades de transmisión muy dispares.

En estos casos, para aumentar la escalabilidad del sistema de forma que sea posible que el sistema siga creciendo, existen dos posibles soluciones: por una parte, se puede intentar disminuir la cantidad de información de control causal en los sistemas causales realizando optimizaciones en las implementaciones habituales; por otra parte, se puede dividir el sistema global en grupos o subsistemas más pequeños (de acuerdo a la topología de red o a la frecuencia de comunicación) e interconectar dichos grupos con un algoritmo de interconexión que reducirá el tráfico de mensajes que circulará por la red entre los distintos grupos.

Esta última solución será la que estudiaremos en los siguientes apartados ya que, además de dividir un gran sistema en grupos para que sea más escalable, también nos permitirá aumentar la escalabilidad de un sistema interconectando sistemas distribuidos ya existentes para formar un sistema global más grande.

2.6 Algoritmos de interconexión.

En este trabajo se analizarán diversos algoritmos de interconexión para Sistemas Distribuidos de Memoria compartida (DSM) y para sistemas distribuidos de paso de mensajes.

El objetivo de los algoritmos de interconexión en los sistemas DSM será mantener la coherencia de un determinado modelo de consistencia en la memoria global distribuida. En este tipo de algoritmos, lo importante será analizar qué restricciones deben cumplir los sistemas para poder ser interconectados según un determinado modelo de consistencia.

En cuanto a los sistemas de paso de mensajes, veremos que lo importante en este caso será mantener la semántica de entrega global entre los sistemas o grupos interconectados. Esto también ocurrirá en los DSM que se implementen utilizando sistemas de paso de mensajes.

En los siguientes apartados veremos que para poder interconectar grupos/sistemas será necesario elegir un representante en cada grupo/sistema que actúe de puente para poder transmitir la información de un sistema a otro. Estos representantes serán procesos de aplicación que asumirán el rol de *routers*[1] y deberán escogerse de forma determinista.

También veremos que una característica deseable en ambos tipos de algoritmos de interconexión será que dichos algoritmos permitan la interconexión de grupos/sistemas

[1] Notación de [30].

sin necesidad de alterar el protocolo local de estos grupos/sistemas. Así pues, diremos que un algoritmo de interconexión no es intrusivo cuando permite interconectar grupos/sistemas sin necesidad de modificar el algoritmo local de los grupos/sistemas. Por el contrario, diremos que un algoritmo de interconexión es intrusivo cuando el algoritmo local de un grupo/sistema debe modificarse para mantener una determinada semántica de entrega en el sistema global.

Como norma general, se debe intentar utilizar algoritmos de interconexión no intrusivos, ya que permitirían interconectar sistemas ya existentes donde es posible que cada sistema utilice su propia implementación del algoritmo local que se deba ejecutar en el sistema. Sin embargo veremos que ante determinadas semánticas de entrega (orden total), el uso de algoritmos intrusivos será necesario.

3. Interconexión de modelos de memoria.

3.1 Introducción.

En este apartado se va a analizar la posibilidad de interconectar apropiadamente sistemas de memoria compartida distribuida de forma no intrusiva. Diremos que dos sistemas de memoria se han interconectado de forma apropiada si el sistema resultante mantiene la consistencia de memoria de los sistemas interconectados. Interconectar sistemas de memoria mejorará la escalabilidad, ya que más adelante veremos que reduce de forma importante el tráfico de red entre dichos sistemas.

Para ello, se van a analizar dos artículos. En el primero de ellos [21], el objetivo es detallar cómo sería el protocolo de interconexión de sistemas distribuidos de memoria compartida basados en sistemas de consistencia causales. En este artículo se presentará la arquitectura de sistema que se utilizará para poder interconectar sistemas de consistencia causal, poniendo especial atención en que el sistema resultante tras realizar la interconexión mantenga la consistencia causal.

El objetivo del segundo artículo [19] es analizar los sistemas de consistencia de memoria que permiten ser interconectados. Veremos que será necesario definir el concepto de rapidez en el sistema (*fastness*) para saber, como norma general, si los sistemas con un determinado modelo de consistencia pueden interconectarse. Concretamente, se analizará la posibilidad de interconectar sistemas rápidos, con consistencia de memoria Caché, PRAM y Casual, siendo los sistemas Caché los únicos que podrán interconectarse manteniendo la consistencia en todo el sistema, sin necesidad de añadir restricciones adicionales en el sistema.

3.2 Interconexión de sistemas de memoria causales

En *On the Interconnection of causal memory systems*, los autores proponen interconectar Sistemas de Memoria Distribuida (DSM) causales basados en propagación, de forma que el sistema resultante siga siendo causal.

Este trabajo presenta como novedad (en el momento de su publicación) un protocolo que permite interconectar dos o más DSM causales sin importar los algoritmos de consistencia de memoria con los que han sido implementados. Este hecho es una propiedad muy destacable, ya que permite aumentar la escalabilidad del sistema sin tener que rediseñar los sistemas existentes que se quieren interconectar (protocolo no intrusivo). En esta línea, hasta la publicación del artículo, no se había publicado ningún otro trabajo que propusiese una solución a la interconexión de sistemas DSM causales. Hasta 2004, otros autores ([44],[2],[11],[30]) habían propuesto protocolos para la interconexión de sistemas de orden causal de paso de mensajes para formar grandes sistemas de paso de mensajes de orden causal. Estos sistemas pueden utilizarse para implementar sistemas DSM causales que incluso pueden ser interconectados para hacer sistemas más grandes. Sin embargo, si los procesos ya están agrupados en sistemas DSM causales, esta idea ya no es viable, ya que, para construir un sistema DSM causal más grande, habría que construir por encima de los sistemas DSM causales, sistemas de paso de mensajes con orden causal.

De este artículo existen dos versiones: la versión publicada en el año 2000 y la revisión publicada en 2004 [21], que corrige y extiende la primera.

3.2.1 Introducción al problema

La memoria compartida distribuida es el mecanismo empleado en la comunicación entre procesos. Si en vez de trabajar con una memoria compartida centralizada se distribuye la memoria compartida entre varios procesos, el mecanismo para leer y escribir variables mediante las correspondientes operaciones de memoria se complica. Además, esto se agrava si en lugar de usar el modelo de consistencia de memoria secuencial se emplea uno en el que se permitan accesos concurrentes a variables compartidas.

Para evitar la rigidez del modelo de consistencia secuencial, muchos autores han implementado numerosos protocolos ([5],[28],[42]) que implementan el modelo de memoria causal. Este modelo ha sido bien valorado por los investigadores, puesto que es fácil de programar, se considera potente y además no necesita de implementaciones costosas.

Varios de estos protocolos permiten el uso de réplicas de datos para permitir el acceso a las variables de forma simultánea a los procesos. Sin embargo, la replicación tiene un inconveniente obvio, y es que a mayor grado de replicación, más difícil se hace mantener la consistencia en el sistema. Por este motivo, el sistema debe controlar las actualizaciones de las variables, bien invalidando las réplicas antiguas, o bien

propagando los nuevos valores de las variables para actualizarlos en cada réplica, siendo esta última opción la utilizada en este caso.

Los autores definen en este trabajo dos protocolos de interconexión de sistemas DSM causales no intrusivos que, a su vez, generarán sistemas DSM causales. Para ello, comenzarán explicando cómo se interconectarían dos sistemas, para posteriormente demostrar que si se pueden interconectar dos sistemas manteniendo la causalidad en la consistencia de memoria, también se podrán interconectar *N* sistemas.

Interconectar varios sistemas en vez de trabajar como si de un sistema DSM causal global se tratara, tendrá como ventaja que se generará menos tráfico de mensajes entre sistemas. Además, los sistemas DSM causales interconectados no tendrán que hacer cambios en los protocolos de propagación que ya utilizaban para poder ser interconectados.

3.2.2 Descripción del sistema

Es este punto se van a describir las condiciones que se darán en los sistemas DSM causales que se van a interconectar. Basándonos en la arquitectura de margarita propuesta en [11] (ver apartado 4.3), un sistema distribuido estará formado por un conjunto de nodos interconectados a través de una red, donde cada nodo de este sistema contendrá uno o varios procesos de aplicación.

Estos sistemas utilizarán replicación de variables y propagarán el nuevo valor de una variable al resto de copias tras ejecutar una operación de escritura. Cada sistema DSM causal podrá utilizar el protocolo de propagación que desee, ya que no interferirá en la interconexión de los sistemas, con lo que los sistemas causales existentes no tendrán que verse alterados.

En cuanto a los canales que interconectarán los sistemas, estos serán FIFO confiables. Con el orden FIFO de estos canales se conseguirá mantener el orden causal en todo el sistema global. Esto es debido a que los sistemas se interconectarán de dos en dos, formando una topología de árbol jerárquica que evitará posibles ciclos. Así pues, como las difusiones se harán entre pares de sistemas, una difusión con orden FIFO será equivalente a realizar una difusión de orden causal.

Además, para poder interconectar dos sistemas DSM causales con el protocolo de Interconexión de Sistemas (protocolo-IS), en cada sistema será necesaria la existencia de un proceso que se encargue de encaminar la información de un sistema a otro para mantener la consistencia en la memoria. Este proceso es denominado por los autores proceso-IS (*IS-process*) o *isp*, pero dado que su función es encaminar la información a otros sistemas y recibir mensajes de otros sistemas a través de los enlaces FIFO confiables anteriormente mencionados, de ahora en adelante vamos a denominar a este proceso *router*, siguiendo la nomenclatura de [30].

Nótese que con los protocolos-IS que se van a definir no sólo podrán interconectarse sistemas DSM causales, sino que podrán interconectarse sistemas secuenciales, puesto

que un sistema secuencial también es un sistema causal. Sin embargo, el sistema resultante tras realizar la interconexión no será secuencial, sino causal[2].

En cuanto al acceso a las variables por parte de los procesos de aplicación, se utilizarán operaciones de lectura y operaciones de escritura, asumiendo que:

- Un valor se escribe como máximo una vez: si una operación de escritura ya se ha ejecutado, no se volverá a ejecutar.
- Los valores iniciales de una variable se establecerán mediante una operación de escritura.

En el siguiente apartado veremos como la interconexión de los sistemas DSM consistirá en interconectar los sistemas de consistencia de memoria subyacentes en cada sistema.

3.2.3 Descripción del modelo MCS

La arquitectura en la que se van a basar los sistemas de memoria distribuidos causales será la propuesta en [7] por *Attiya & Welch*. En esta arquitectura, cada DSM está implementado por un Sistema de Consistencia de Memoria (MCS), que estará compuesto por un conjunto de procesos denominados procesos-MCS que se ubicarán dentro de los nodos del sistema DSM. Formalmente, definiremos un modelo de consistencia de la siguiente forma:

> *"Un modelo de consistencia M es un conjunto formado por todas las ejecuciones de tipo M.*[3]*"*

O lo que es lo mismo, todas las ejecuciones de memoria estarán restringidas en base a las restricciones que impone cada modelo de consistencia.

La función de los procesos-MCS será mantener la consistencia de la memoria mediante un algoritmo distribuido denominado protocolo-MCS. Formalmente, definiremos un sistema MCS de la siguiente forma:

> *"Un sistema MCS es de tipo M si todas las ejecuciones son del tipo M."*

Dentro de un sistema DSM, cada proceso de aplicación *p* tendrá asociado un único proceso-MCS (*mcs(p)*), que asumiremos que estará ubicado dentro del mismo nodo, mientras que un proceso-MCS podrá tener asociados varios procesos de aplicación. En un mismo nodo del sistema pueden convivir varios procesos-MCS, en cuyo caso podrán

---

[2] En el apartado 3.3.3 se introduce el concepto de rapidez en el sistema. Veremos que el modelo de consistencia secuencial no es rápido y, por tanto, sólo podrá interconectarse empleando algoritmos intrusivos.

[3] Para comprender correctamente el concepto de ejecución y de modelo de consistencia es conveniente leerse las definiciones de ejecución, vista, etc. que se proporcionan en [21] y [19].

comunicarse a través de operaciones locales. Sin embargo, cuando los procesos-MCS estén en nodos distintos, utilizarán la red para mantenerse actualizados. Este comportamiento queda reflejado en la Figura 10 donde se muestra la arquitectura sobre la que se implementará el protocolo-IS.

En cuanto a los accesos a memoria, los procesos de aplicación accederán a las variables mediante llamadas (*calls*) de lectura/escritura a su proceso-MCS y se bloquearán hasta recibir la respuesta a dichas llamadas. El comportamiento de estas llamadas será el siguiente:

- Una operación de escritura llevada a cabo por el proceso *i* en el sistema S^k sobre una variable *x* que actualice su valor al valor *v* se denotará de la siguiente forma: $w^k_i(x)v$. Este tipo de llamadas se realizarán enviando el par <*x,v*> (nombre de la variable y valor de la variable) y responderán con un *ACK*.

- Una operación de lectura llevada a cabo por el proceso *i* en el sistema S^k sobre la variable *x,* obtendrá como valor *v* y se denotará de la siguiente forma: $r^k_i(x)v$. Las llamadas de operaciones de lectura contienen el nombre de la variable cuyo valor se quiere obtener, y responden con el valor de la variable que contiene el proceso-MCS.

Así pues, interconectando los sistemas MCS de todos los sistemas DSM utilizando el protocolo-IS, se habrán interconectado apropiadamente los sistemas DSM. Esto se realizará con los *routers* de cada sistema a interconectar. Como ya sabemos, cada sistema tendrá un *router* al que vamos a asumir que se le asignará un proceso-MCS exclusivo que deberá mantener una réplica local de cada variable de la memoria compartida. De este modo, cuando un proceso de aplicación realice una operación de escritura, también se actualizará la copia local de la variable del proceso-MCS asociado al *router*.

Cuando la copia local de una variable se actualice, el proceso-MCS de un *router* se comunicará con su *router*. Estas comunicaciones se realizarán a través de una interfaz compuesta de dos llamadas adicionales o tareas denominadas *upcalls*. Si la actualización la lleva a cabo un *router*, la operación no generará otras llamadas (*upcalls*). En cualquier otro caso, el procedimiento es el siguiente:

- El proceso-MCS ejecuta el *upcall pre_update(x)* justo antes de actualizar la copia local de la variable *x* con valor *v*. Esta operación no siempre será necesaria, y su uso podrá configurarse en el *router* de un sistema.

- A continuación, inmediatamente después de haber actualizado el valor de la variable *x*, el proceso-MCS ejecutará *post_update(x,v)*.

- Cuando el proceso-MCS genere un *upcall*, se bloqueará hasta que el *router* le responda[4].

- El valor previo de la variable *x* se mantendrá, y no se modificará hasta que la actualización con el nuevo valor esté terminada, y el nuevo valor no se modificará hasta que obtenga respuesta del *upcall post_update*.

- Paralelamente, el protocolo-IS debe permitir las operaciones de lectura mientras se ejecutan estos *upcalls*, garantizando que estas operaciones de lectura terminarán (para prevenir posibles interbloqueos) y que deberán obtener como valor de *x* el valor antiguo de la variable o el nuevo, dependiendo de qué parte de la actualización de una variable se esté ejecutando (*pre_update* o *post_update*).

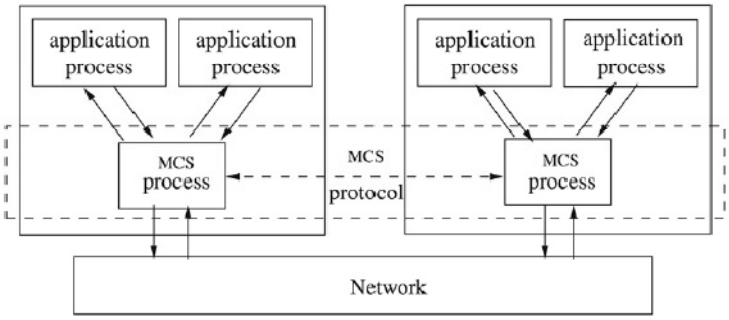


Figura 10. Arquitectura de un sistema DSM implementado con sistemas MCS. [19]

3.2.4 Protocolos de interconexión de sistemas de memoria causales

En este apartado se van a definir dos protocolos-IS para interconectar sistemas de memoria causales. Para explicar los detalles de cada protocolo, se va a trabajar con dos sistemas (S^0 y S^1) que, tras ser interconectados mediante uno de estos protocolos, formarán el sistema S^T. Otra forma de denominar los sistemas serán S^k y $S^{k'}$, siendo S^k el sistema que no sea $S^{k'}$. De esta forma, denominaremos $router^k$ al único proceso de interconexión asociado al sistema S^k.

En cuanto a los protocolos-IS propuestos, estos deben cumplir con el objetivo principal de interconectar dos sistemas causales: mantener el orden causal entre las operaciones de escritura causalmente relacionadas que se propagan de un sistema S^k a un sistema $S^{k'}$ de una forma no intrusiva. Sin embargo, tal y como se muestra en el siguiente ejemplo, si no se preservan las dependencias causales en las propagaciones, el sistema resultante no podrá considerarse como causal:

- Supongamos que el proceso *i* ejecuta la operación de escritura $w^k_i(x)v$ en S^k.

- A continuación el $router^k$ propaga la operación de escritura.

[4] Este hecho cambia en la revisión del protocolo de interconexión que se realiza en [19].

- En un proceso j de $S^{k'}$, se ejecutan las operaciones $r^{k'}{}_j(x)v$ y $w^{k'}{}_j(x)u$ en este orden, generando una dependencia causal al leer y actualizar el valor que ha propagado S^k.

- Tras esto y sin violar la causalidad de S^k, un proceso l de S^k puede ejecutar $r^k{}_l(x)u$ y $r^k{}_l(x)v$ en este orden, violando la causalidad del sistema S^T.

Para solucionar esto, los *routers* deberán ejecutar obligatoriamente operaciones de lectura para cada valor propagado entre sistemas, creando una relación causal entre las operaciones de escritura propagadas en ambas direcciones.

La forma en la que se crean estas relaciones causales es lo que distingue a los dos protocolos que se expondrán a continuación. Es decir, estos protocolos sólo difieren en el código que se ejecuta en los *router*, aunque tienen la misma interfaz, con lo que son totalmente compatibles y se pueden interconectar sin ningún problema. Los *routers* ejecutarán el código de un protocolo u otro en función de si el protocolo causal MCS cumple con la propiedad de *causal updating* que se define a continuación:

> *"En cualquier computación α^k de un sistema S^k, si los procesos de aplicación i y j llevan a cabo las operaciones de escritura $w^k{}_i(x)v$ y $w^k{}_j(y)u$, y $w^k{}_i(x)v$ precede causalmente a $w^k{}_j(y)u$, entonces el proceso-MCS asociado al router^k actualizará su réplica de x con el valor v antes de actualizar la réplica de y con el valor u."*

A continuación se definen los protocolos teniendo en cuenta la propiedad de *causal updating*.

Protocolo-IS con causal updating

Este protocolo se basa en que el protocolo causal MCS del sistema cumple con la propiedad de *causal updating* (la mayoría de los protocolos-MCS causales la cumplen).

En este protocolo-IS, cada *router* ejecutará dos tareas atómicas:

- *Propagate*$^k{}_{out}$*:* Transmite las operaciones de escritura del sistema S^k al sistema $S^{k'}$. Las operaciones de escritura que mantengan una relación causal deberán mantener el orden causal al ser transmitidas de un sistema a otro. Para ello se utilizará un canal confiable FIFO ordenado. Al estar los sistemas interconectados de dos en dos manteniendo una estructura de árbol, los mensajes se irán enviando manteniendo el orden causal y se procesarán cuando se reciban en $S^{k'}$ mediante *Propagate*$^{k'}{}_{in}$ conforme se vayan recibiendo. Esta tarea se ejecutará inmediatamente después de que se actualice la copia local de la variable x con el valor v (*post_update(x,v)*). Como resultado, lee el valor v de x y le envía el par $<x,v>$ al *router*$^{k'}$.

- *Propagate*$^k{}_{in}$*:* Recibe las operaciones de escritura transmitidas por el sistema $S^{k'}$ mediante la tarea *Propagate*$^{k'}{}_{out}$ y las ejecuta exactamente en el orden en el que las recibe (en orden FIFO). Esta tarea se ejecutará cuando se reciba un par $<x,v>$

enviado por $router^{k'}$. Como resultado, el proceso-MCS asociado al $router^{k}$ ejecutará una operación de escritura causal, propagando en orden causal el valor v en todas las réplicas de la variable x en el sistema S^k. En este punto hay que recordar que las operaciones de escritura generadas por $router^{k}$ no generan *upcalls*, es decir, un par recibido por $router^{k'}$ no se volverá a enviar a $S^{k'}$.

En la Figura 11 se define el pseudocódigo que deben ejecutar estas tareas.

$Propagate^k_{out}(x,v)$ :: task which is activated when the $post\_update(x,v)$ upcall is received from the MCS-process.
begin
  $r^k_{isp^k}(x)v$
  send $\langle x,v\rangle$ to $isp^{\overline{k}}$
  send *response* to the MCS-process
end

$Propagate^k_{in}(x,v)$ :: task which is activated when a pair $\langle x,v\rangle$ is received from $isp^{\overline{k}}$.
begin
  $w^k_{isp^k}(x)v$
end

Figura 11. Pseudocódigo de las tareas *Propagate*$^k_{out}$ y *Propagate*$^k_{in}$. [21]

Protocolo-IS sin causal updating

En este protocolo se trata un caso más general en el que no se cumple la propiedad *causal updating* en el protocolo-MCS causal del sistema S^k.

Las tareas que ejecutarán los *routers* en este proceso serán las siguientes:

- *Propagate*$^k_{out}$ y *Propagate*$^k_{in}$ se comportarán como en el caso anterior.
- *Pre-Propagate*$^k_{out}(x)$: se ejecutará inmediatamente antes de que la copia de la variable x del proceso-MCS asociado al $router^k$ actualice su valor con el nuevo valor v para mantener el orden causal cuando se propaguen las operaciones de escritura. Ejecutará una operación de lectura $r^k_{router^k}(x)s$, leyendo el valor antiguo de x y asegurando que dos operaciones de escritura causalmente ordenadas se propagan a $S^{k'}$ mediante *Propagate*$^k_{out}$ manteniendo el orden causal. El código de esta tarea se muestra en la Figura 12.

$Pre\_Propagate^k_{out}(x)$ :: task which is activated when the $pre\_update(x)$ upcall is received from the MCS-process.
begin
  $r^k_{isp^k}(x)s$
  send *response* to the MCS-process
end

Figura 12. Pseudocódigo de la tarea *Pre-Propagate*$^k_{out}(x)$. [21]

Todas estas tareas atómicas quedan reflejadas en la Figura 13, donde se muestran las tareas del protocolo-IS en el sistema S^k y las interacciones que se generan con los procesos-MCS y el sistema $S^{k'}$, mientras que la Figura 14 muestra cómo quedaría un sistema formado por dos subsistemas interconectados mediante el protocolo-IS.

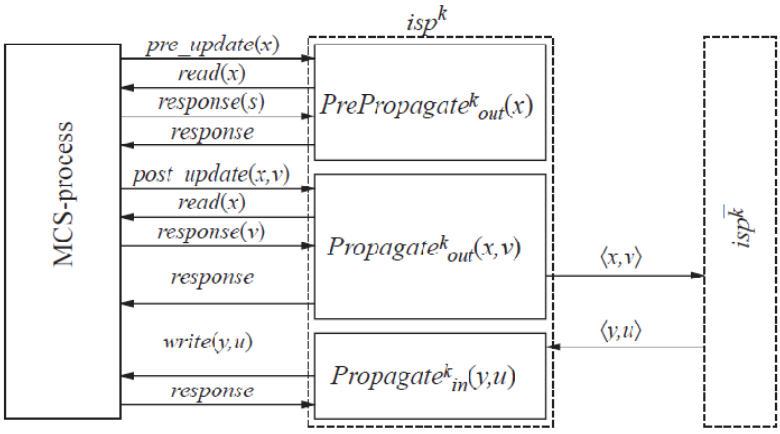


Figura 13. Esquema de tareas del protocolo-IS causal. [21]

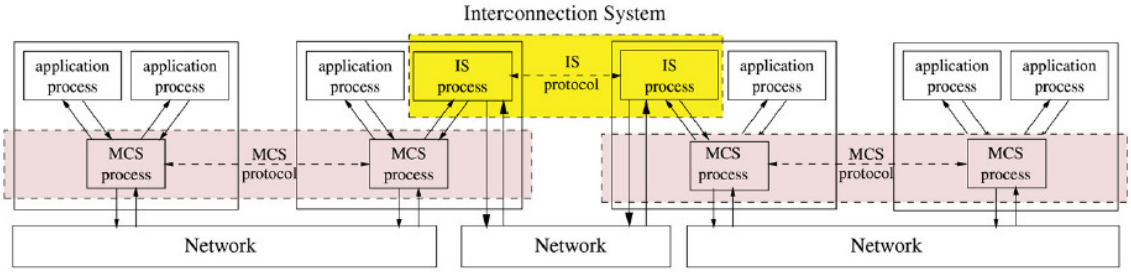


Figura 14. Interconexión de dos sistemas DSM mediante el protocolo-IS. [19]

Por último, los autores demuestran[5] que, del mismo modo que pueden interconectarse 2 sistemas DSM, pueden interconectarse apropiadamente *N* sistemas DSM con sus correspondientes sistemas MCS, manteniendo en todo momento la consistencia causal, siempre y cuando los sistemas se interconecten mediante pares de sistemas, formando una estructura de árbol jerarquizada que evitará posibles ciclos.

3.2.5 Análisis de rendimiento y conclusiones

En este apartado se comparará el rendimiento obtenido al utilizar el protocolo-IS frente a emplear un protocolo-MCS causal que conecte todos los procesos del sistema en un DSM global. Para ello se asumirá que el protocolo-MCS empleado, tanto en los múltiples sistemas interconectados con el protocolo-IS como en el caso del sistema DSM global, será el mismo, y sólo implementará consistencia causal.

En primer lugar, se observa que el tiempo de respuesta (*response time*) al ejecutar una operación de memoria no se ve afectado por el protocolo-IS. Esto es debido a que sus procesos-MCS no se ven afectados por la interconexión de los sistemas.

En cuanto al tráfico de red (*network traffic*), se asume que el protocolo-MCS genera *n-1* mensajes por cada operación de escritura, siendo *n* el número de procesos-MCS existentes en el sistema. Las operaciones de lectura no generan ningún mensaje

[5] La demostración formal por inducción está muy bien detallada en la sección 5 de [21], con lo que aquí se obvia.

adicional. Así pues, el número de mensajes total que se generarán al llevarse a cabo una operación de escritura en un sistema son:

- *n-1* mensajes en un sistema DSM global con *n* procesos-MCS.
- *n+1* mensajes si se interconectan 2 sistemas (*N=2*) empleando el protocolo-IS. Se generarán 2 mensajes más, puesto que se añaden 2 procesos-MCS adicionales (uno para cada *router* de los sistemas a interconectar) y un mensaje será enviado de un *router* a otro.
- *n+N-1* mensajes si se trabaja con *N* sistemas.

A priori, en el sistema DSM global, se utilizan menos mensajes. Sin embargo, vamos a suponer que tenemos dos sistemas, cada uno con $\frac{n}{2}$ procesos en redes diferentes. Normalmente la red que interconecta los procesos de un mismo sistema es mucho más rápida que la red que interconecta dos sistemas. Así pues, teniendo en cuenta este detalle:

- En el sistema DSM Global, $\frac{n}{2}$ mensajes atravesarán la red, pudiendo generar un posible cuello de botella. Además, este problema se agravaría al aumentar el número de sistemas y el número de procesos dentro de cada sistema.
- Con el protocolo de interconexión de sistemas (protocolo-IS), solamente atravesará la red un solo mensaje.

Empleando el protocolo-IS se obtiene la ventaja de que no se sobrecarga tanto la red que interconecta ambos sistemas, puesto que sólo se envía un mensaje.

Por último, sólo queda comentar la latencia (*latency*), o lo que es lo mismo, el tiempo que transcurre desde que un valor escrito es visto por el resto de procesos. Si se desprecia el cómputo local en los *routers* y, teniendo en cuenta que *l* es la latencia de un sistema que ejecuta el protocolo causal y que *d* es el retraso que sufre un mensaje entre 2 *routers*:

- Para 2 sistemas, la latencia será igual a *2l+d*.
- Para *N* sistemas, la latencia en el peor caso considerando una topología en estrella será *3l+2d*. Este caso se da cuando la escritura la realiza cualquier nodo que no es el central. Si la escritura la realiza el nodo central, entonces el coste será *2l+d*, el mismo que se obtiene cuando se interconectan dos sistemas.

Así pues, hemos visto que es posible interconectar sistemas distribuidos de memoria compartida causales mediante el protocolo de interconexión no intrusivo protocolo-IS, disminuyendo significativamente el número de mensajes necesarios para mantener la consistencia de memoria sin necesidad de realizar modificaciones en los sistemas ya existentes.

3.3 Interconexión de modelos de memoria distribuidos

En este apartado se analizará el reciente estudio que hicieron *Vicente Cholvi, Ernesto Jiménez y Antonio Fernández Anta* en el artículo *Interconnection of Distributed Memory Models* [19] de 2009 sobre la posibilidad de interconectar sistemas distribuidos de memoria compartida que empleasen como modelo de consistencia de memoria modelos rápidos (*fast*).

En el apartado 3.2 se proporcionó un protocolo de interconexión (protocolo-IS) de sistemas de memoria causales. En este caso se va a revisar el protocolo de interconexión para que también pueda utilizarse para interconectar sistemas con modelos de consistencia PRAM, caché o causales, debiendo contemplar nuevas asunciones en el modelo de sistema. Veremos cómo los sistemas DSM causales y los DSM PRAM sólo podrán interconectarse apropiadamente bajo ciertas condiciones en el sistema, mientras que los DSM caché podrán interconectarse sin ningún tipo de restricciones.

Este artículo es una revisión de una publicación realizada en 2003 [29].

3.3.1 Introducción al problema

En [21] ya se demostró que la principal ventaja de interconectar Sistemas de Memoria Distribuidos (DSM) es evitar el tráfico masivo de mensajes en la red cuando hay que realizar un broadcast a los procesos ubicados en otros sistemas. Este problema se acrecienta cuando los sistemas están ubicados en redes físicas diferentes, ya que, usualmente, la red que interconecte los sistemas será más lenta que la red dentro del propio sistema. Además, los sistemas podían ser interconectados sin necesidad de hacer modificaciones en ellos utilizando protocolos no intrusivos (cada uno podía usar el protocolo de consistencia que quisiera a nivel local).

Por todo esto, es conveniente interconectar los sistemas mediante un protocolo de interconexión similar al visto en el apartado 3.2 siempre y cuando el modelo de consistencia de memoria utilizado en los sistemas a interconectar lo permita. Sin embargo, veremos que no todos los modelos de consistencia de memoria pueden ser interconectados. Esto dependerá, en parte, de la rapidez (*fastness*) del modelo.

3.3.2 Descripción del sistema

En este apartado se van a presentar las características del modelo de sistema, que básicamente será el mismo que ya se describió en el apartado 3.2.2, pero con algunas modificaciones.

Consideraremos a los sistemas distribuidos de memoria compartida como sistemas formados por una colección de procesos de aplicación que interactúan mediante

operaciones de lectura/escritura[6] con variables de la memoria compartida manteniendo en todo momento un cierto tipo de consistencia. Asumiremos que un valor de una variable será escrito como máximo una vez e inicialmente ésta se inicializará mediante una operación de escritura. Aparte de este tipo de operaciones, también se utilizarán operaciones de sincronización para importar información (conseguir un cerrojo) o para exportar información (liberar un cerrojo).

Además, los autores asumirán como propiedad de viveza (*liveness property*) la siguiente restricción que hoy en día se cumple en todos los sistemas: al menos el valor de la última escritura de una variable será finalmente visible en todos los procesos del sistema. Formalmente, definimos esta propiedad de viveza de la siguiente forma:

> *"En toda ejecución α de un sistema S, si sólo hay un proceso escribiendo en una variable x y su última operación sobre x es w(x)u, todas las respuestas a la llamada r(x) hechas por cualquier proceso de aplicación contendrán el valor u."*

En cuanto a la implementación del sistema de consistencia de memoria, en el apartado 2.3.2 se definieron los modelos de consistencia de memoria de más renombre y vimos que escoger un determinado modelo de consistencia a veces no es fácil debido a que para poder permitir una mayor rapidez al ejecutar las operaciones de memoria (posibles optimizaciones en el compilador, etc.) a veces hay que sacrificar la consistencia de las variables, al mismo tiempo que se debe escoger un modelo compatible con la arquitectura de sistema que se va a utilizar. En este artículo estudiaremos cómo interconectar sistemas de consistencia de memoria causales, PRAM y caché. Para implementar la consistencia de memoria en estos sistemas se utilizará el Sistema de Consistencia de Memoria (MCS) descrito en el apartado 3.2.3 (ver Figura 10). Como ya sabemos, en este modelo, un proceso de aplicación *p* tendrá asociado un único proceso-MCS (*mcs(p)*), que asumiremos que estará ubicado dentro del mismo nodo[7].

Tal y como se estudió en el apartado 3.2, si se consigue interconectar apropiadamente los sistemas MCS subyacentes, se habrá conseguido interconectar también los sistemas DSM. Para ello se hará uso de un Sistema de Interconexión (IS) que ejecutará un protocolo de interconexión (protocolo-IS) cuya función será mantener la consistencia de memoria en todo el sistema distribuido, tal y como se muestra en la siguiente definición:

> *"Un modelo de consistencia podrá ser interconectado para cualquier conjunto de sistemas que implementen este modelo de consistencia siempre y cuando exista un algoritmo de interconexión (protocolo-IS) que los interconecte."*

El sistema IS estará compuesto de unos procesos especiales denominados procesos-IS o *routers* que ejecutarán dicho protocolo de interconexión de sistemas (protocolo-IS).

---

[6] Las definiciones de operación de lectura y operación de escritura pueden consultarse en la sección 3.2.3 de este documento.

[7] Ver "Asunción 1" en [19].

Para simplificar el protocolo asumiremos, de nuevo, que cada proceso-MCS a interconectar tendrá asociado un único *router*. Además, tal y como ocurría en los procesos de aplicación, el proceso-MCS de un *router* y el propio *router* deberán estar en el mismo nodo del sistema.

El *router* podrá ejecutar operaciones de lectura y escritura sobre las variables manejadas por el proceso-MCS que serían visibles en todos los nodos del sistema (se considerarían operaciones locales sobre la memoria compartida). Sin embargo, debe observarse que para que otro sistema se "entere" de los cambios realizados en este sistema, el *router* deberá informarle de los cambios (operaciones de escritura). Para ello, el *router* del sistema en el que se han realizado cambios conectará con el *router* del otro sistema a través de un canal FIFO confiable, enviándole los cambios realizados sobre las variables modificadas. De esta forma se conseguirá tener un sistema MCS global a partir de los sistemas MCS de cada sistema distribuido de memoria.

Nótese que se ha indicado que es el *router* de un sistema quien se encarga de notificar los cambios en la memoria local del sistema al resto de sistemas, pero no se ha especificado de qué forma se da cuenta el *router* de un sistema de que los procesos de aplicación han hecho modificaciones en la memoria. Para ello, existen tres posibilidades, en las que el *router* interactúa de diferentes maneras con su proceso-MCS:

- El *router* comprueba de forma periódica (leyendo la memoria por completo) que se hayan hecho modificaciones sobre variables. Esta forma de proceder no requiere intervención alguna por parte del proceso-MCS.

- El proceso-MCS comunica explícitamente a su *router* cualquier modificación sobre una variable de la memoria. Esta manera de proceder es más eficiente que la forma anterior, puesto que no hace falta recorrer toda la memoria. Sin embargo, el proceso-MCS tiene que estar capacitado para poder comunicarse con su *router* en cuanto se produce algún cambio en la memoria.

- Comprobaciones periódicas por parte del *router* junto con avisos explícitos por parte del proceso-MCS.

Los autores de este artículo proponen una interfaz de comunicación genérica entre el *router* y su proceso-MCS donde no se asume ninguna de estas opciones. Sin embargo, para desacoplar lo máximo posible el sistema original del protocolo de interconexión, el *router* sólo podrá ejecutar operaciones de escritura y lectura sobre el proceso-MCS, no pudiendo bloquearlo nunca.[8] Lo que sí se asume es que, cuando se ejecute una operación de escritura en la memoria, el *router* será notificado del cambio de forma asíncrona.

[8] Este hecho presenta una diferencia importante con [21], donde un *router* sí que bloqueaba su proceso-MCS.

En cuanto a la notación utilizada, diremos que N será el número de sistemas a interconectar. Así pues, denominaremos a los sistemas como S^0, S^1, ...S^{N-1}, y al sistema resultante tras realizar la interconexión lo denominaremos S^T. En S^T se incluirán todos los procesos excepto los procesos *router* de los sistemas interconectados.

3.3.3 Modelos de consistencia rápidos y no rápidos (*fastness*)

Para saber si un sistema DSM global puede ser interconectado, deberemos observar si el modelo de consistencia de memoria que utiliza es rápido (*fast*) o no, o lo que es lo mismo, si cumple con la propiedad de rapidez (*fastness*). Formalmente, diremos que un modelo de consistencia es rápido si:

> *"Existe un protocolo-MCS que implemente el modelo de consistencia de tal forma que las operaciones de memoria sólo requerirán de cómputo local antes de retornar el control, incluso en sistemas con varios nodos."*

Numerosos modelos de consistencia no cumplen esta propiedad, con lo que nunca podrán ser interconectados, tal y como han demostrado diversos autores: el modelo atómico [35], el modelo secuencial [7], el modelo de procesador ([8],[24],[4]), etc. Además, *Attiya & Friedman* demostraron en [8] que, cualquier modelo de memoria sincronizada que proporcione acceso exclusivo, tampoco podrá ser interconectado.

Por el contrario, los modelos de consistencia PRAM, causal y caché sí que son rápidos, con lo que a priori podrán ser interconectados. Sin embargo, veremos en los apartados siguientes que solamente los DSM caché podrán ser interconectados sin imponer restricción alguna.

En los protocolos de interconexión que se definirán en las siguientes secciones, se asumirá que dichos sistemas rápidos (*fast*) actualizarán las réplicas mediante propagación de los nuevos valores.

3.3.4 Interconexión de sistemas PRAM

En este apartado se va a estudiar la interconexión de los sistemas de memoria compartida distribuida cuyo modelo de consistencia es el PRAM [37]. Este modelo de consistencia ya se explicó en el apartado 2.3.2 y, en términos de orden, consiste en que todos los procesadores observan las operaciones de escritura de un mismo procesador en el mismo orden, mientras que las escrituras generadas por procesadores distintos pueden percibirse en orden distinto.

En este modelo, las escrituras locales se propagarán de un proceso a otro mediante canales FIFO dentro de un mismo sistema, teniendo en cuenta que estas actualizaciones se llevarán a cabo de forma asíncrona en los nodos remotos posteriormente.

Como norma general, un sistema con este modelo de consistencia no podrá ser interconectado, o lo que es lo mismo, no existe un protocolo-IS que interconecte cada

par de sistemas PRAM. El motivo es que algunos sistemas PRAM no tienen por qué ser FIFO ordenados (*FIFO ordered*):

> *"Diremos que un sistema es FIFO ordenado si para cada proceso p en S^k, si p realiza dos operaciones de escritura w(x)v y w(y)u, tal que w(x)v precede a w(y)u, el proceso mcs(router^k) actualiza su réplica de la variable x con el valor v antes de actualizar la réplica y con el valor u."*

Por ejemplo, en el caso de que fuera el *router* de un sistema el encargado de comprobar periódicamente si ha habido cambios en las variables de *mcs(router)*, el *router* sería capaz de detectar que ha habido cambios en las variables modificadas, pero no sabría en qué orden se han realizado. Así pues, debe ser el proceso-MCS quien avise explícitamente a su *router* de los cambios que se han llevado a cabo en la memoria local. Sin embargo, esto no sería suficiente para que el sistema sea FIFO ordenado, ya que el orden en el que se han recibido las operaciones no tiene por qué ser el orden en el que finalmente se han llevado a cabo (un ejemplo de esto es el algoritmo propuesto en [28] para sistemas causales y, por tanto, PRAM), con lo que el orden de las operaciones que se envían al *router* del otro sistema puede no ser el correcto.

Los autores demuestran formalmente que los sistemas PRAM que no son FIFO ordenados no pueden interconectarse, pero sí los sistemas PRAM FIFO ordenados[9]. Así pues, se propone un protocolo de interconexión de sistemas PRAM FIFO ordenados constituido por dos tareas concurrentes (ver Figura 15)[10]:

- *Propagate$^k_{out}$*: transfiere las operaciones de escritura ejecutadas en S^k a $S^{k'}$, tal que k≠k'. Esta tarea se activa inmediatamente después de que al proceso *router*^{k} se le notifique que la variable *x* ha sido actualizada con el valor *v*, enviando el par <*x,v*> al sistema $S^{k'}$.

- *Propagate$^k_{in}$*: ejecuta las operaciones de escritura que se reciben del sistema $S^{k'}$. Esta tarea se inicia cuando *router*^{k} recibe un par de valores <*x,v*> de *router*$^{k'}$, actualizando el valor de todas las réplicas de la variable *x* en el sistema S^k con el valor *v*.

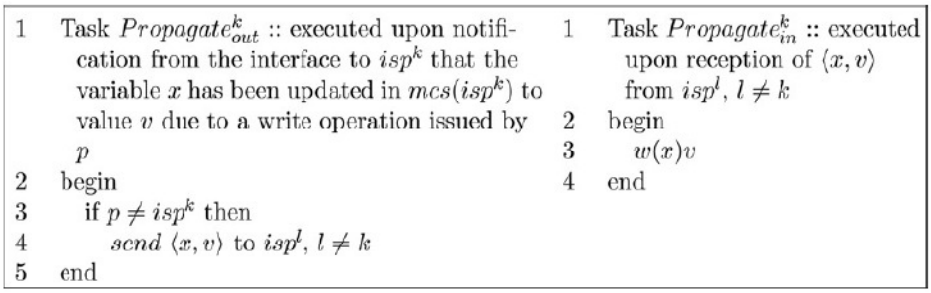
Para evitar que las actualizaciones de escritura recibidas por otro sistema se propaguen, se comprobará en *router*^{k} si la operación de escritura se ha originado en S^k.

En resumen, este protocolo de interconexión permitirá interconectar de 2 a *N* sistemas, tal y como se demuestra en [19], siempre y cuando los sistemas sean FIFO ordenados.

---

[9] Ver Apéndice A.1. en [19].

[10] Los autores presentan los protocolos de interconexión para los modelos de consistencia PRAM, causal y caché asumiendo que sólo interconectan dos sistemas, para posteriormente demostrar que si se pueden interconectar 2 sistemas, se pueden interconectar *N*.



1 Task $Propagate^k_{out}$ :: executed upon notification from the interface to isp^k that the variable x has been updated in $mcs(isp^k)$ to value v due to a write operation issued by p
2 begin
3 if $p \neq isp^k$ then
4 *send* $\langle x, v \rangle$ to isp^l, $l \neq k$
5 end

1 Task $Propagate^k_{in}$ :: executed upon reception of $\langle x, v \rangle$ from isp^l, $l \neq k$
2 begin
3 $w(x)v$
4 end

Figura 15. El protocolo-IS PRAM para cada $router^k$, $\forall k \in [0,1], / k'=l \wedge k \neq k'$. [19]

3.3.5 Interconexión de sistemas causales

En este apartado se vuelve a estudiar la interconexión de los sistemas de consistencia de memoria causales [5], ampliando el estudio realizado en [21].

En los sistemas causales, además de cumplir las condiciones del modelo PRAM, las operaciones de lectura deben devolver el último valor de la última operación que generó una relación de causalidad. Formalmente, un sistema causal se define de la siguiente forma:

> *"Un sistema es causal si para cualquier ejecución α y cada proceso p, existe una vista legal β_p de α_p que preserva el orden causal."*

Dado que el modelo de consistencia PRAM es estrictamente más débil que el modelo de consistencia causal y, si tenemos en cuenta que al igual que en el caso anterior los sistemas actualizan sus variables mediante propagación, el resultado que se ha obtenido sobre la interconexión de los sistemas PRAM también se aplicará en los sistemas causales. Así pues, los sistemas causales que no sean FIFO ordenados tampoco podrán interconectarse en este caso, o lo que es lo mismo, debemos realizar el análisis de interconexión de sistemas causales comprobando si los sistemas causales FIFO ordenados pueden interconectarse.

A pesar de que los sistemas PRAM FIFO ordenados sí que podían interconectarse, no ocurre lo mismo con los sistemas causales.[11] Esto es debido a que las operaciones de lectura ahora generan relaciones de orden causal, con lo que es importante que la restricción impuesta por el orden causal se cumpla en todos los nodos de la memoria. Si nos fijamos en la definición de sistema FIFO ordenado, la restricción de orden sólo se aplica en el proceso *mcs(router^k)*, siendo insuficiente en el modelo de consistencia causal ya que ahora necesitamos que el sistema mantenga el orden globalmente. Por tanto, sólo se podrán interconectar los sistemas FIFO globalmente ordenados:

> *"Diremos que un sistema es globalmente ordenado si dadas dos operaciones de escritura causalmente relacionadas w(x)v y w(y)u, ejecutadas en dos procesos*

[11] Las demostraciones formales de por qué un sistema FIFO ordenado causal no puede ser interconectado y la de por qué un sistema causal FIFO globalmente ordenado sí puede ser interconectado, pueden encontrarse en [19].

distintos o en el mismo proceso de un sistema S^k, cada mcs(p) con p en S^k actualiza su réplica local de x con v antes de que actualice la réplica de y con u."

Teniendo en cuenta esta restricción, los autores proponen un protocolo de interconexión de sistemas formado por dos tareas (ver Figura 16), tal y como ocurría en el caso de la interconexión de los sistemas PRAM:

- *Propagate*$^k_{out}$: esta tarea es prácticamente igual a la del protocolo-IS en sistemas PRAM, sólo que el par <x,v> no se envía al resto de sistemas hasta que se actualicen todas las réplicas de *x* en todos los procesos-MCS del sistema S^k.
- *Propagate*$^k_{in}$: esta tarea realiza la misma funcionalidad que en el protocolo-IS en los sistemas PRAM.

Tal y como ocurría en la interconexión de sistemas PRAM, insertar en el algoritmo de interconexión de sistemas la restricción de mantener el orden global no ha supuesto muchos cambios en el protocolo de interconexión, siendo esto un hecho ventajoso.

```
1  Task Propagate^k_out :: executed upon
     notification from the interface to
     isp^k that the variable x has been
     updated in mcs(q) for all q in S^k
     to value v due to a write operation
     issued by p
2  begin
3    if p ≠ isp^k then
4      send ⟨x, v⟩ to isp^l, l ≠ k
5  end
```

```
1  Task Propagate^k_in :: executed
     upon reception of ⟨x, v⟩
     from isp^l, l ≠ k
2  begin
3    w(x)v
4  end
```

Figura 16. El protocolo-IS causal para cada $router^k$, $\forall k \in [0,1], / k'=l \wedge k \neq k'$. [19]

3.3.6 Interconexión de sistemas caché

La interconexión de sistemas de consistencia caché [24] es el último de los modelos de consistencia que se estudian en [19].

Si refrescamos la memoria, en el modelo de consistencia caché se mantenían la restricciones que impone el modelo de consistencia secuencial siempre y cuando se acceda a la misma posición de memoria. Formalmente, definiremos un sistema de consistencia caché de la siguiente forma:

"Un sistema S es caché si para cada ejecución α y cada variable x existe una vista legal β_x de α_x que preserva la relación de precedencia ($\rightarrow$), tal que α_x representa al conjunto de operaciones que se llevan a cabo sobre la variable x en α."

Al contrario de lo que ocurría con el resto de modelos de consistencia analizados, el modelo caché sí que puede ser interconectado sin necesidad de imponer restricciones en el sistema.

En este tipo de sistemas, el protocolo-IS sólo tendrá una tarea (ver pseudocódigo en la Figura 17):

- *Propagate*^{k}: en esta tarea cada *router* mantiene una copia del último valor propagado para cada variable *x* (*last(x)*), cuyo valor inicial será *NoData*.

Además, hay que tener en cuenta que, para poder realizar la interconexión, uno de los procesos *router* (por ejemplo, *router*^{0}) debe enviar el par de valores <*x,NoData*> al otro *router* para cada variable *x*, generando mucho tráfico de red.

```
1   Task Propagate^k(x) :: executed upon reception of ⟨x, v⟩
      from isp^l, l ≠ k
2   begin
3       if v ≠ NoData then
4           w(x)v
5           last(x) := v
6       r(x)u
7       if u = last(x) then
8           u := NoData
9       send ⟨x, u⟩ to isp^l, l ≠ k
10  end
```

Figura 17. El protocolo-IS caché para cada *router*^{k}, $\forall k \in [0,1], / k'=l \wedge k \neq k'$. [19]

El hecho de que para cada objeto de la memoria se genere mucho tráfico de red es una desventaja, aunque este protocolo interconecta los sistemas de consistencia cache de forma que el sistema interconectado también es cache, cumpliendo con los objetivos de interconexión que se perseguían. Si además de interconectar sistemas caché se desea mejorar la eficiencia de este protocolo-IS, se pueden realizar algunas optimizaciones que reducirán el tráfico de red a costa de aumentar la latencia:

- La primera optimización consiste en acumular *n* pares <*x,v*> antes de ejecutar la tarea, teniendo en cuenta que deben ser de diferentes variables.

- Además, para reducir el tráfico de mensajes entre *routers*, la transferencia de pares entre sistemas puede realizarse cada cierto intervalo de tiempo o cuando la variable *x* se actualiza en el sistema emisor.

3.3.7 Análisis de rendimiento y conclusiones

Con los protocolos que se han presentado en [19] no se ha pretendido diseñar protocolos de interconexión eficientes, sino comprobar que ciertos sistemas de consistencia de memoria permiten ser interconectados. Teniendo en cuenta esto, en este análisis se va a

comparar el rendimiento en un sistema global con el rendimiento en un sistema de interconexión, asumiendo que en ambos casos se utiliza el mismo protocolo-MCS[12].

En primer lugar, se observa que el tiempo de respuesta (*response time*) al ejecutar una operación de memoria no se ve afectado por el protocolo-IS. Esto es debido a que sus procesos-MCS no se bloquean, y dado que los modelos de consistencia estudiados son *fast,* sólo se computa el tiempo de ejecución de las operaciones locales.

En cuanto a la latencia (*latency*), o tiempo máximo que pasa desde que un valor escrito se hace visible en el resto de procesos, sí que se aprecia diferencias:

- En el sistema global con n procesos, la latencia sólo depende del protocolo-MCS utilizado.
    - En los sistemas PRAM y en los sistemas causales, descartando el cómputo local, la latencia depende del tiempo que se tarda en difundir una actualización en el sistema ($T_B(n)$), con lo que la latencia será lineal con n.
    - En un sistema caché, según el algoritmo de [28], la latencia será $\Theta(nT_B(n))$.
- En la interconexión de 2 sistemas también distinguiremos entre los distintos modelos de consistencia:
    - En el modelo PRAM y causal el protocolo-IS propaga una nueva actualización en cuanto se entera de ella. Así pues, la latencia será $T_B(n_0) + T_B(n_1) + d$, siendo d el de una comunicación punto a punto (entre los *routers* de dos sistemas) y siendo $n=n_0+n_1$ el número total de procesos-MCS.
    - En el caso del modelo caché a lo mejor tiene que esperar a que el otro *router* le envíe un mensaje antes propagar una actualización. En este caso la latencia será $\Theta(n_0T_B(n_0)) + \Theta(n_1T_B(n_1)) + d$ (tomando como referencia el algoritmo de [28]).
- En un sistema de interconexión compuesto por N sistemas, la latencia dependerá de la topología, siendo $2N-1$ en el peor caso (bus).

En el caso del sistema global, la latencia es mejor, pero en todos los casos la latencia será lineal, puesto que $T_B(n)$ lo es.

Por otra parte, analizando el tráfico de red (*network traffic*), en el sistema global el tráfico será $M_B(n)$, siendo $M_B(n)$ el número de mensajes que se generarán al realizar una

[12] El análisis al completo puede consultarse en la sección 7 de [19].

difusión. En los sistemas PRAM y causales cada operación de escritura generará una difusión. Así pues:

- Si se interconectan 2 sistemas S^0 y S^1: el tráfico será $M_B(n_0) + M_B(n_1) + 1$ mensajes por operación de escritura, o lo que es lo mismo, será la suma de los mensajes generados al realizar una difusión en cada sistema más el mensaje que se enviará de un *router* a otro para mantener la consistencia entre ambos sistemas.

Sin embargo, en los sistemas caché no se puede evaluar en función de las operaciones de escritura, puesto que tanto en el protocolo-MCS global como en el protocolo-IS se envían mensajes contantemente. La diferencia en este caso estriba en que en el protocolo-IS únicamente se enviará un mensaje por variable:

- Si se interconectan 2 sistemas S^0 y S^1, el tráfico será $M_B(n_0) + M_B(n_1) + V$, siendo V el número de variables.

En los tres casos, el tráfico de red es mayor que si se tiene un sistema global, aunque la diferencia no es excesiva). Sin embargo, en los sistemas de consistencia caché, si el número de variables es muy elevado, el tráfico de red es mucho mayor que en el sistema global, con lo que sería conveniente realizar las optimizaciones que ya se han comentado. Por ejemplo, si todos los cambios en la memoria se almacenan en un solo mensaje, el tráfico de red se reduciría a $M_B(n_0) + M_B(n_1) + 1$ mensajes.

Sin embargo, pese a que en los protocolos de interconexión vistos el tráfico de red es ligeramente mayor que en el caso del sistema global, no hay que olvidarse de la ventaja que suponen los protocolos de interconexión frente al sistema global, puesto que, como ya se analizó en el apartado 3.2.5, evitan el tráfico de mensajes entre sistemas de distintas redes, reduciendo todos estos mensajes a uno sólo.

3.4 Conclusiones.

Con el análisis de los artículos [21] y [19], hemos visto que es posible interconectar sistemas distribuidos de memoria compartida (DSM) mediante el protocolo de interconexión de sistemas (protocolo-IS). Estos sistemas DSM estarán basados en sistemas de consistencia de memoria (MCS), y la interconexión de los sistemas DSM consistirá en interconectar dichos sistemas MCS, sin necesidad de realizar grandes cambios en los sistemas ya existentes.

Para ello, simplemente se añadirá un nuevo proceso en cada sistema que será el encargado de realizar las comunicaciones necesarias para mantener la consistencia de memoria entre pares de sistemas. Así pues, cada sistema MCS puede utilizar el algoritmo que desee para implementar el modelo de consistencia que se haya definido en el sistema, siempre y cuando la actualización de las variables se realice mediante propagación. Este hecho resulta muy ventajoso, ya que permite interconectar sistemas MCS existentes sin necesidad de ser modificados.

Además, también hemos visto que sólo en los sistemas *fast* puede considerarse la opción de la interconexión. En el caso de los sistemas caché, por el simple hecho de ser *fast* ya se pueden interconectar. Sin embargo, en los sistemas con modelo de consistencia PRAM o causal, no bastaba con asumir que se pueden interconectar por el hecho de ser *fast*, ya que sólo se pueden interconectar en el caso de ser FIFO ordenados y globalmente ordenados, respectivamente (ver Tabla 1).

Modelo de Memoria	Sistemas Globalmente Ordenados	Sistemas FIFO Ordenados	Otros Sistemas
Modelos no-rápidos	*No*	*No*	*No*
Causal	*Sí*	*No*	*No*
PRAM	*Sí*	*Sí*	*No*
Caché	*Sí*	*Sí*	*Sí*

Tabla 1. Posibilidades de interconexión en distintos tipos de Sistemas DSM.

Por último, no hay que olvidar por qué es interesante interconectar sistemas más pequeños en vez de trabajar con un solo sistema de memoria compartida global. A pesar de que el tiempo de respuesta no se ve afectado y de que la latencia es un poco mejor en el caso de trabajar con un sistema DSM global, la principal diferencia aparece a la hora de analizar el tráfico de red, que normalmente se verá reducido en la interconexión de sistemas. Esto es debido a que, como norma general, los nodos de un sistema global estarán distribuidos en redes distintas. Si se tiene en cuenta que la red que interconecta los sistemas suele ser más lenta que la red interna del sistema, en un sistema con una gran cantidad de nodos, cuando en el sistema global se envíen *broadcasts* a nodos de otra red para mantener la consistencia en la memoria, el enlace que une ambas redes podría colapsarse. Este hecho se reducirá con los protocolos-IS, ya que para mantener la consistencia en la memoria solamente será necesario enviar un mensaje por el enlace que interconectará un par de sistemas.

4. Interconexión de sistemas de difusión.

4.1 Introducción.

Los sistemas de paso de mensajes causales son los más utilizados hoy en día en sistemas distribuidos asíncronos ya que proporcionan un mínimo de garantía de entrega. Esto es debido a que, en un sistema en el que no podemos discernir si un mensaje se ha perdido o si sólo se está retrasando, la relación de causalidad nos garantiza que si se ha recibido un mensaje m_2 que depende causalmente de un mensaje m_1 enviado previamente, sabremos que m_1 se ha entregado correctamente.

Sin embargo, la información de control necesaria para mantener la causalidad del sistema debe mantenerse tanto en los mensajes como en los propios procesos y, en sistemas muy grandes, almacenar esta información y difundirla por la red provoca que el sistema deje de ser escalable. Para almacenar la información de control causal, originalmente se utilizaban los relojes lógicos [33]. Estos relojes ordenan más mensajes de los que realmente están causalmente relacionados, introduciendo siempre un retraso en la fase de entrega de un mensaje [46]. Para solventar este problema, surgieron nuevas soluciones basadas en los históricos causales y en los relojes vectoriales ([12], [40], [32]). No obstante en grandes sistemas distribuidos, aun utilizando estas soluciones, la información de control que hay que almacenar tanto en procesos como en mensajes crece de forma insostenible con el número de procesos. Además, si en vez de difundir un mensaje en todo el sistema (*broadcast*) se desea difundirlo a determinados procesos destinatarios (*multicast*), el tamaño de la información de control aumenta y el tiempo de proceso también.

A lo largo de la sección 4 veremos que, agrupando los procesos de acuerdo a la frecuencia con la que se comunican y/o su topología, se conseguirá reducir la información de control en los sistemas causales, ya que un proceso sólo tendrá que mantener la información del grupo al que pertenece. También estudiaremos la interconexión de grupos con otros tipos de semántica de entrega (sistemas de orden FIFO y sistemas de orden total), donde el objetivo será reducir el número de mensajes que viajarán por la red para interconectar dichos grupos, manteniendo siempre la semántica de entrega global. Para ello se utilizarán algoritmos no intrusivos siempre que sea posible. Por último veremos cómo paralelizar la interconexión de sistemas distribuidos FIFO y la interconexión de sistemas causales para evitar que los enlaces lentos que suelen interconectar los grupos de un gran sistema distribuido global se transformen en el cuello de botella del sistema.

Todas las propuestas se estudiarán en orden cronológico para observar la evolución de éstas a partir de las publicaciones de los diveros autores.

4.2 Separadores causales en comunicaciones multicast a gran escala

El artículo que vamos a comentar en este apartado es *Causal separators for large-scale multicast communications* de *Rodrigues & Veríssimo*, publicado en 1995 [44].

Este artículo es una de las primeras soluciones que se proponen para intentar reducir la cantidad de información que es necesaria almacenar para mantener la relación de causalidad, también llamada "ocurre antes" (*happens before*), en un gran sistema causal aprovechando la topología de red.

Para ello los autores proponen una versión propia de los denominados históricos causales (*causal histories*) cuya principal característica será optimizar la información que deberá mantener cada proceso de acuerdo a la topología de red del sistema global.

4.2.1 Introducción al problema

En un sistema distribuido los procesos intercambian información continuamente con el resto de procesos del sistema. De forma inmediata (asumiendo que viene indicado en el propio mensaje), un proceso puede saber quién es el emisor de un mensaje recibido, del mismo modo que puede saber quién o quienes reciben un mensaje enviado por él mismo. Sin embargo, en ocasiones esta información no es suficiente para que un proceso sepa por sí mismo si un mensaje está relacionado casualmente con otro (→). Por ejemplo:

- Si un proceso p envía un mensaje m_2 tras enviar un mensaje m_1, p sabe que $m_1 \rightarrow m_2$.
- Si un proceso p envía un mensaje m_2 tras recibir un mensaje m_1, p sabe que $m_1 \rightarrow m_2$.
- Si un proceso p' envía un mensaje m_2 tras recibir/enviar un mensaje m_1 ($m_1 \rightarrow m_2$) y otro proceso p envía un mensaje m_3 tras recibir un mensaje m_2, p sabe que $m_2 \rightarrow m_3$ pero no sabe que m_1 también precede causalmente a m_3 ($m_1 \rightarrow m_3$).

Por este motivo, los sistemas distribuidos causales deben de implementar mecanismos de control adicionales que permita a los procesos saber si un mensaje precede causalmente a otro.

Un primer paso que facilitará el diseño de las aplicaciones distribuidas será dividir en dos partes la entrega de un mensaje al proceso de aplicación (ver apartado 2.2.5):

- En primer lugar se realizará la recepción (*receive*) que consistirá en recibir el mensaje que proviene de la red. El orden causal en este caso podría obtenerse empleando difusiones de orden causal.

- En segundo lugar se realizará la entrega (*delivery*) que consistirá en que el servicio de *broadcast* entregue el mensaje correspondiente (respetando el orden causal de los mensajes) al proceso de aplicación para que pueda procesarlo.

Sea cual sea el método que se emplee para conocer la precedencia causal de los mensajes (relojes lógicos [33], *piggybaking*, etc), se necesitará almacenar una gran cantidad de información adicional para mantener la causalidad. Almacenar esta información y enviarla por la red supone un gran coste, especialmente cuando se trabaja en sistemas grandes que realizan habitualmente difusiones a grupos (*multicasts*).

En las siguientes secciones se propone una nueva estructura de datos basada en la técnica de los históricos causales que, junto con la identificación de los denominados separadores causales, permitirá minimizar la cantidad de información necesaria para mantener la causalidad.

4.2.2 Notación y descripción del sistema

El sistema con el que vamos a trabajar será un sistema causal global formado por un conjunto de procesos $P=\{p_0, p_1, p_2, \ldots p_{n-1}\}$ con memoria distribuida no compartida. Cada uno de estos procesos tendrá asignado un identificador que se utilizará como parte del identificador de los mensajes enviados por un determinado proceso. En concreto, el identificador (uid_m) de un mensaje m estará compuesto por 3 campos:

- El identificador del proceso emisor (*sender*): $s_m \in P$.
- Un identificador de mensaje c_m que se generará a partir de un contador local de mensajes en el proceso y que servirá para conocer el orden de los mensajes enviados por un mismo proceso.
- Un conjunto de identificadores de procesos, correspondientes al conjunto de destinatarios: $A_m \subseteq P$.

El identificador de los mensajes uid_m se utilizará en los históricos causales en vez de utilizar el propio mensaje, consiguiendo ahorrar un espacio considerable.

En cuanto al protocolo de *multicast* causal que proponen los autores, éste estará compuesto por 3 eventos: enviar, recibir y entregar. De esta forma, al proceso de aplicación se le entregarán (*deliver*) los mensajes ordenados, facilitando el diseño de las aplicaciones distribuidas. Para ello, se asumirá que la capa de transporte será fiable, es decir, que todos los mensajes llegarán a los procesos destinatarios. Sin embargo, no se asume ningún tipo de orden, con lo que los mensajes pueden recibirse (*receive*) desordenados.

4.2.3 Ampliación del histórico causal (*Extended causal history*)

El objetivo de este apartado es adaptar la técnica de los históricos causales al algoritmo de *multicast* que posteriormente se describirá en el apartado 4.2.4.

La nueva representación se denominará *extended causal history* y guardará la información causal en tres partes bien diferenciadas dentro de cada proceso:

- Histórico causal H_p (*causal history*): contiene una lista con los *uids* de todos los mensajes que preceden al siguiente mensaje enviado por el proceso *p*.

- Histórico de entrega D_p (*delivery history*): contiene una lista con los identificadores de todos los mensajes que ya se han entregado al proceso *p*.

- Histórico de copias C_p (*carbon copy history*): almacena donde se incluye la información causal que falta de un mensaje.

Nótese que el histórico causal contiene todos los mensajes del histórico de entrega. La razón por la que se mantienen las dos listas es porque a cada una se le va a aplicar un método de compresión distinto. En concreto, estas listas se utilizarán de la siguiente forma:

- Cuando un proceso *p* inicia su ejecución, las tres listas estarán vacías y el contador de mensajes se inicializará a 0: $H_p=D_p=C_p=\{\}$, $c_p=0$.

- Cada vez que un proceso *p* vaya a enviar un mensaje *m*, se generará un nuevo identificador de mensaje y se incrementará el contador de mensajes c_p. A continuación a *m* se le añadirá (*timestamp*) el histórico causal H_p, de manera que $H_m=H_p$. De esta forma todos los receptores de *m* recibirán el histórico causal de *p*. Tras hacer el *timestamp,* el uid_m se añadirá a H_p y D_p. Esta información también se almacenará en C_p.

- Cuando un proceso *q* tal que $q \in A_m$, recibe el mensaje *m* enviado por *p*, comparará el *timestamp* del mensaje H_m con su histórico de entregas D_q y comprobará que todos los mensajes precedentes se han entregado. Si esto es así, entrega el mensaje y actualiza las listas de históricos; si no, esperará a recibir todos los mensajes de H_m dirigidos a *q* que aún no se hayan recibido.

- Cuando a un proceso *q* se le entrega un mensaje *m* tal que $q \not\prec s_m$, H_m se añade a H_q. Además, el identificador de *m* se añadirá a H_q y a D_q.

Si nos fijamos, con este procedimiento y sin necesidad de usar el histórico de copia se consigue la entrega de mensajes causal. Sin embargo, a medida que se hacen nuevas difusiones en el sistema, estás listas crecen continuamente, pudiendo llegar a ser inmanejables. Por este motivo es necesario un mecanismo de recolección de basura que vaya eliminando la información que ya nos resulta innecesaria.

La primera regla de compresión es muy intuitiva y se aplica sobre el histórico de entregas:

- Entrega FIFO: La entrega causal impone que cuando se entrega un mensaje *m* a un proceso *q* enviado por un proceso *p* es porque todos los mensajes que preceden causalmente a *m* enviados por *p* ya se han entregado en *q*. Por tanto, cuando un mensaje *m* enviado por *p* se añada a D_q, *m* reemplazará a todos los mensajes entregados anteriormente a *q*.

La segunda regla de compresión se aplicará sobre el histórico causal, basándose en la técnica del último envío (*last send*) y última actualización (*last update*) de *Singhal & Kshemkalyani* [47]. A grandes rasgos, la compresión consistirá en eliminar los *timestamps* de mensajes anteriores en el *timestamp* de un mensaje posterior, dentro de un mismo proceso destinatario: cualquier mensaje n tal que $A_n \subseteq A_m$ y lleve el uid_m en su *timestamp*, no necesita llevar H_m, ya que n se entregará después de m y, por tanto, después de todos los mensajes que hay en H_m. Es en este momento donde entra en juego el histórico de copia: el histórico de copia contendrá un campo para cada mensaje del histórico causal, guardando la lista de procesos a los que se les ha informado de que el *timestamp* del mensaje asociado se encuentra incluido en el *timestamp* de otro mensaje.

A continuación se indica más detalladamente el proceso de actualización del histórico de copias:

- Cada proceso p almacena un histórico de copia C_p, que contiene un elemento $C_p(m)$ por cada mensaje m del histórico causal H_p.

- Timestamping extendido: opcionalmente, podrá adjuntarse el histórico de copia C_p, además de adjuntar H_p, con lo que un mensaje m viajaría junto con las listas H_m y C_m.

- Actualización de envío: después de que p envíe un mensaje m y antes de insertar m en H_p, se actualizarán todos los campos del histórico de copia C_p de la siguiente forma:

  $$\forall (i \in H_p),\ C_p(i)=C_p(i) \cup A_m$$

  Después, se añade m a H_p y si inicializa $C_p(m)=\{\}$. Todas estas operaciones serán atómicas.

- Actualización de entrega: cuando un proceso q recibe un mensaje m tal que $q \neq s_m$, q actualizará el histórico de copia de los mensajes previos cuyo emisor sea s_m de la siguiente manera:

  $$\forall (n \in H_q) / s_n = s_m \wedge c_n < c_m,\ C_q(n)=C_q(n) \cup A_m$$

  A continuación añade todos los elementos n de H_m a H_q e inicializa C_q de la siguiente forma:

  $$C_q(n)=C_m(n) \cup A_m \cup \{ s_m \} \cup \bigcup_{i \in H_q / s_i = s_n \wedge c_i > c_n} Ai,$$

  teniendo en cuenta que $C_m(n)=\{\}$ si no se adjunta C_p en m. Si n ya existe en H_q, simplemente se une $C_q(n)$ con el resultado de la expresión anterior.
  Por último, q inserta m en H_q e inicializa $C_q(m)=\{s_m,q\}$. Como en el caso del envío, estas operaciones se realizarán atómicamente.

Añadir el histórico de copia al mensaje define mejor el contenido de todos los históricos de copia del sistema. Sin embargo, esto tiene como desventaja el aumento del tamaño de los mensajes.

En sistemas en los que no es posible incrementar el tamaño de los mensajes añadiendo el *timestamp* del histórico de copias, el histórico de copias se puede mantener utilizando

solamente la información de los mensajes enviados y recibidos localmente en el proceso:

- Redundancia de *timestamps*: un mensaje no tiene por qué incluirse en el *timestamp* si ya se ha incluido en el *timestamp* de otro mensaje enviado al mismo destinatario. Formalmente diremos que, cuando se realice el *timestamp* de un mensaje *m*, un proceso *p* sólo incluirá en H_m los elementos de su histórico causal $i \in H_p$ que no hayan sido notificados a A_m desde el punto de vista de *p*: $H_m = \cup\, i \in H_p \,/ A_m \nsubseteq C_p(i)$.

- Redundancia en el histórico: cuando el campo del histórico de copias relativo a un mensaje incluye por completo los destinatarios del mensaje, este mensaje puede eliminarse del histórico causal, puesto que todos los procesos ya están informados. Formalmente diremos que, en el histórico causal H_p, si existe un mensaje *m* tal que $A_m \subseteq C_p\,(m)$, *m* puede eliminarse de H_p y C_p.

La suma de todas estas optimizaciones permitirá comunicaciones más fluidas y procesos más ligeros, puesto que se elimina la información redundante. En [44] se puede encontrar un ejemplo de 3 procesos a los que se les aplica el *extended causal history*.

En el siguiente apartado nos centraremos en cómo se aplica el *extended causal history* en sistemas con separadores causales, definiendo para ello, qué es un separador causal.

4.2.4 *Timestamping* en función de la topología del sistema causal

En un sistema distribuido, como norma general, un proceso sólo podrá comunicarse directamente con un subconjunto de procesos del sistema. El resto de comunicaciones se realizarán a través de cadenas de paquetes, según indique el algoritmo de *routing* que se esté utilizando. Teniendo en cuenta esto, se puede representar el sistema como si fuera un grafo *G=(P,A)*, donde los vértices serían los procesos y las aristas serían los enlaces directos entre pares de procesos. En la Figura 18 se muestra un ejemplo de sistema representado como un grafo.

Entre estos vértices, distinguiremos un conjunto de vértices especial, denominado separador de vértices (*vertex separator*) y, en el caso de los sistemas causales, separador causal. Formalmente, diremos que un conjunto de procesos *S* será un separador de vértices (F_S, B_S), siendo F_S el conjunto de procesos anteriores (*forward set*) y B_S el conjunto de procesos posteriores (*backward set*), si y sólo si $F_S \cap B_S = \{\} \wedge F_S \cup B_S \cup S = P$ y $\forall f \in F_S,\ \forall b \in B_S$ cada camino (*path*) que conecte b y f pasa por al menos un vértice contenido en *S*. Es decir, los separadores causales no son más que procesos sobre los que se encaminarán los mensajes, ejerciendo en cierto modo de *routers* lógicos. Por este motivo y, aprovechando la notación de [30] (ver apartado 4.4), de ahora en adelante los denominaremos *routers*.

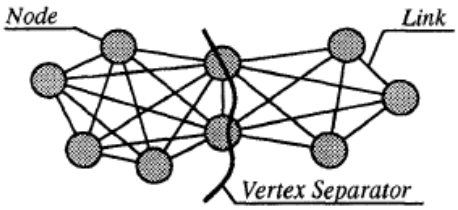


Figura 18. Grafo de un sistema con vértices separadores o *routers*. [44]

Si aplicamos el *extended causal history* en el sistema, todos los nodos se tratarán por igual. Sin embargo, este protocolo se puede optimizar aún más si se tiene en cuenta la separación entre nodos de un mismo sistema que ofrecen los *routers*. En concreto, cuando un *router* del conjunto *S* adjunte el *timestamp* a un mensaje cuyos destinatarios formen parte por completo del conjunto F_S, éste puede omitir en el *timestamp* los elementos del histórico causal tal que todos los destinatarios al completo sean miembros de B_S y que hayan sido notificados al resto de miembros de *S*. Esta acción se conocerá como el *timestamp* topológico y formalmente se definirá de la siguiente forma:

- *Timestamp* topológico: cuando un proceso p esté enviando un mensaje m y siempre que exista un *router* $S=(F_S,B_S)$ tal que $p \in S \wedge A_m \subseteq F_S \wedge A_n \subseteq B_S \wedge S \subseteq C_p(n)$, todos los mensajes $n \in H_p$ no necesitan insertarse en H_m.

Pese al ahorro en el tamaño del mensaje que supone utilizar el *timestamp* topológico, existe una serie de inconvenientes que se detallan a continuación:

- En redes con topología muy variable y/o arbitraria pueden existir muchos *routers*. En este caso, se podría utilizar esta optimización en todos ellos o sólo en un subconjunto de ellos.

- Existen numerosos algoritmos capaces de identificar los vértices separadores de un sistema. Sin embargo, estos algoritmos son costosos y, por tanto, es conveniente evitar su uso en aplicaciones en las que la topología sea muy dinámica.

- Cualquier cambio, por mínimo que sea, puede alterar la pertenencia de los procesos en los grupos F_S y B_S.

Por este motivo, deberá asumirse una topología estática en las aplicaciones que se ejecuten en el sistema causal. De esta forma será posible conocer a priori qué procesos compondrán los conjuntos F_S y B_S.

Por último, nótese que siempre que se ha hablado de topología se ha hablado de la topología de la aplicación distribuida. En un sistema físico, varios procesos de una misma aplicación pueden estar ejecutándose en un mismo nodo, con lo que la estructura lógica de la aplicación no coincide con la estructura física. Además, los mensajes de procesos ubicados en un mismo nodo realmente se enviarán por la red de forma secuencial. Por este motivo es conveniente intentar mapear la topología lógica con la topología física, de manera que ambas sean lo más parecido posible. En [44] los autores

tratan este tema de forma más profunda, proponiendo varias soluciones para realizar este mapeo, como por ejemplo que los *routers* físicos que interconectan dos redes se mapeen automáticamente como *routers* lógicos (separadores causales).

4.2.5 Análisis de rendimiento y conclusiones

Para analizar el rendimiento de su protocolo, los autores *Rodrigues & Veríssimo* optan por realizar simulaciones. Para ello, simulan un sistema compuesto por 4 grupos, tal y como se muestra en la Figura 19.

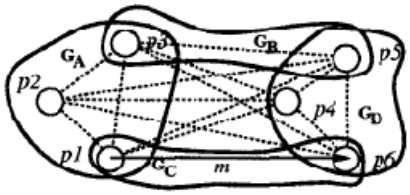


Figura 19. Grafo con la topología lógica que se va a simular. [44]

En este sistema, los procesos pueden enviar mensajes a los grupos a los que pertenecen, mapeándose en una cadena de mensajes en función del grafo extendido (ver Figura 20).

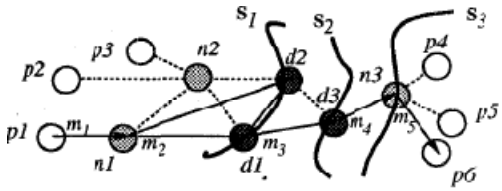


Figura 20. Grafo de comunicaciones extendido. [44]

En el análisis, los autores optan por comparar el *extended causal history* con otras metodologías utilizadas para mantener la información causal, como son los relojes vectoriales y los relojes matriciales. Además, con el protocolo del histórico causal extendido se realizarán 3 pruebas: una sin utilizar la optimización del *timestamping* topológico, una segunda prueba utilizando el *timestamping* topológico en S_2 y una tercera prueba utilizando *timestamping* topológico en S_1, S_2 y S_3.

Puesto que el sistema está compuesto de 6 procesos, con los relojes matriciales se utilizarían 36 elementos para mantener el histórico, mientras que en el caso de los relojes vectoriales, asumiendo que se utiliza un reloj por grupo, el tamaño del *timestamp* sería de 10 elementos (3+3+2+2). Asumiendo un ratio de 10 mensajes por segundo en cada proceso y un retraso de la red de 50 ms, se ha analizado el tamaño medio del *timestamp* de los mensajes, cuyo resultado se muestra en la Tabla 2.

En los resultados se observa que incluso sin usar el *timestamp* topológico, el protocolo extendido de los históricos causales reduce considerablemente el tamaño del mensaje. Además, también se han obtenido resultados para un sistema formado por 10 procesos, añadiendo al sistema de 6 procesos dos nuevos procesos en G_A asociados a n_1 y otros dos procesos en G_D asociados a n_3. En este nuevo sistema, la solución de los relojes matriciales dispara su tamaño, mientras que en el caso de los históricos causales apenas

se aprecian diferencias. Sin embargo, la diferencia entre utilizar *timestamp* topológico o no ha disminuido al aumentar el número de nodos.

Escenario	N=6	N=10
Relojes matriciales	36	100
Relojes vectoriales	10	14
Extendend causal history sin *timestamp* topológico	3.55	3.46
Extendend causal history con *timestamp* topológico en S2	2.70	3.09
Extendend causal history con *timestamp* topológico	2.10	2.75

Tabla 2. Tamaño medio del *timestamp* de los mensajes.

En mi opinión, 10 nodos sigue siendo un sistema muy pequeño, con lo que habría que ver cómo se comporta el protocolo de históricos causales extendido en sistemas mucho más grandes con y sin *timestamping* topológico. Además, existen optimizaciones conocidas para los relojes vectoriales y los relojes matriciales que no se aplican en este análisis.

4.3 La arquitectura jerárquica de margarita en la entrega causal

En este apartado se va a analizar el artículo *The Hierarchical Daisy Architecture for causal Delivery*, publicado en 1997 por *Baldoni, Friedman & van Renesse* [11].

El objetivo de este artículo es mantener controlado el tamaño de la información causal mientras se proporciona tolerancia a fallos. A diferencia del protocolo propuesto en el apartado 4.2, esto se hará de forma independiente a la topología, aunque obviamente, cuanto más se asemeje la topología lógica de los procesos a cómo están distribuidos físicamente (topología física) mejor resultado se obtendrá.

Para ello los autores proponen una arquitectura en la que los procesos que más se relacionen estén ubicados en un mismo grupo y que sea un representante de cada grupo el que se encargue de interconectar estos grupos a través de sus representantes, formando todos los representantes un nuevo grupo. A esta arquitectura se le conocerá como la arquitectura de margarita.

Para conseguir la tolerancia a fallos los autores asumirán la existencia de un sistema de comunicación a grupos. Este sistema facilitará mucho el diseño de la estructura que proponen, porque no habrá que preocuparse de qué ocurre si un proceso cambia de grupo, falla (fallo parada), o si se añade un nuevo proceso a un grupo.

En los siguientes apartados se desarrollarán ampliamente todos estos aspectos.

4.3.1 Introducción al problema

En sistemas asíncronos donde no existe mecanismo alguno que nos permita saber si un mensaje ha sido omitido o si simplemente se retrasa debido a que la red está saturada, el orden causal proporciona una garantía de entrega. Esto es debido a que cuando se entrega un mensaje *m*, automáticamente sabemos que todos los mensajes que preceden causalmente a *m* ya han sido entregados. Por este motivo se utiliza la entrega causal en muchos sistemas asíncronos.

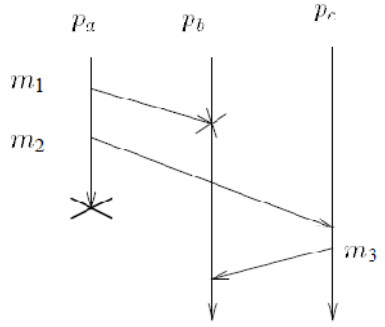
Existen numerosos protocolos cuya función es implementar el orden casual, como por ejemplo el protocolo de los históricos causales extendidos [44] en el que se optimizaba la cantidad de información causal que se almacenaba en los procesos junto con la información causal que viajaba en el mensaje. Sin embargo, otros protocolos abogan por optimizar el tiempo de entrega. En esta clase de protocolos, son bien conocidas las siguientes cotas:

- Si un proceso sólo realiza difusiones a un grupo de tamaño n, la información adicional que se añadirá en cada mensaje será $\Theta(n)$ [14].
- Si se soportan difusiones (*multicast*) a un número pequeño de grupos g, cada uno de ellos de tamaño n, la cantidad de información de control requerida será $\Theta(n \cdot g)$ [14].

- Si se permite a un proceso enviar mensajes a un subconjunto arbitrario de destinatarios escogidos de un total de *n* procesos, la cantidad de información de control requerida será de $\Theta(n^2)$ [41].

La cota $\Theta(n^2)$ es prohibitiva, sobre todo en sistemas donde *n* puede ser muy grande. Por eso se han propuesto diversas soluciones, como enviar sólo las diferencias entre matrices (matriz dispersa en vez de una matriz densa), hacer *piggybacking* de la información causal necesaria en todos los mensajes enviados, e incluso en el sistema de comunicación a grupos *Horus* se permite traducir cualquier envío por un broadcast que llegaría a todos los procesos, consiguiendo reducir el tamaño de la información de control a un vector de dimensión *n* a costa de sobrecargar la red con difusiones. Es decir, para reducir considerablemente la cantidad de información de control debemos introducir más tráfico en la red. Sin embargo, la estrategia de difundir un mensaje a todos los procesos permitiría restablecer el sistema en caso de que un proceso fallase y, de este modo, evitar que el sistema llegara a bloquearse.

El siguiente ejemplo (Figura 21) muestra como un solo fallo de un proceso, combinado con la omisión de un solo mensaje en un sistema sin entrega fiable en el que se asume que se libera un mensaje en la red tan pronto se recibe uno relacionado causalmente, puede provocar que el sistema se bloquee impidiendo la entrega de futuros mensajes:



- *P_a envía m_1 a P_b.* P_b comete un fallo de omisión y m_1 se pierde.
- La aplicación genera m_2 que es enviado por P_a a P_c antes de saber que m_1 se ha perdido. Nótese que m_1 y m_2 están relacionados causalmente.
- Cuando P_c recibe m_2, genera m_3 y lo envía a P_b.
- P_a cae sin darle tiempo a saber que m_1 nunca fue entregado.
- P_b no puede recibir m_3 porque aún no ha recibido m_1 (y nunca lo hará). En consecuencia, P_b se bloquea.

Figura 21. Ejemplo de escenario de fallo. [11]

El problema de la tolerancia a fallos en protocolos de orden causal en que los mensajes no se difunden a todos los destinatarios, puede ser resuelto de tres formas posibles:

- Esperar a la estabilidad de los mensajes enviados y de los mensajes que se hayan recibido anteriormente cada vez que cambie el subconjunto de destinatarios. Esta solución es la adoptada por el sistema ISIS y tiene la desventaja de que introduce mucho tiempo de retardo arbitrario entre mensajes enviados.

- Difundir cada mensaje a todos los procesos (difusiones uniformes) de forma que siempre sea posible recuperar el mensaje extraviado. Esta solución es la utilizada por Horus, pero tiene como desventaja la sobrecarga que produce en la red, no pudiendo utilizarse en grandes sistemas.

- Adjuntar en un mensaje (*piggybacking*) todos los mensajes previos que aún se consideren inestables. Esta fue la solución adoptada por el sistema ISIS en sus

versiones iniciales, pero el principal problema es que el tamaño de los mensajes puede ser extremadamente largo.

Con la configuración de la margarita se trabajará con una arquitectura jerárquica que reducirá la información de control y solucionará el problema de la tolerancia a fallos puesto que la división de los procesos en grupos hará que se reduzca la información de control añadida a los mensajes, al mismo tiempo que se permitirán difusiones uniformes dentro de cada grupo. Además, al tratarse de una arquitectura jerárquica, la solución será escalable incluso en un sistema con muchos procesos.

4.3.2 Notación y descripción del sistema

El sistema distribuido asíncrono con el que vamos a trabajar estará formado por un conjunto *P* de *n* procesos. Estos procesos estarán interconectados a través de una red fiable que utilizarán para enviar y recibir mensajes entre ellos. Que la red sea fiable implicará que los mensajes tienen muchas probabilidades de llegar a sus destinatarios, aunque muchos de ellos pueden retrasarse, llegar desordenados e incluso perderse. Así pues, se admiten en el sistema los fallos por omisión y también se tendrán en cuenta los fallos de tipo caída.

En cuanto el tipo de sistema, se trabajará con un sistema asíncrono, con lo que no existirá ningún reloj global que permitirá discernir si un mensaje se ha retrasado o se ha perdido. Por este motivo se realizarán entregas causales, dado que al menos se garantiza que los mensajes que preceden causalmente a un mensaje entregado, ya habrán sido entregados.

Para manejar la entrega de los mensajes a los procesos se asumirá que un proceso estará dividido en dos partes: el nivel de aplicación y el gestor de entregas (*Delivery Manager* o DM). Esta arquitectura ya se vio en el apartado 2.2.

Tal y como se ha comentado en la introducción, los procesos se dividirán en grupos locales (*local groups*), de forma que cada proceso sólo formará parte de uno de estos grupos locales. Dentro de cada grupo local se elegirá de forma determinista un miembro representante del grupo, denominado servidor causal (*causal server*). Estos representantes formarán, además, parte de otro grupo (el grupo de servidores causales) que tendrá como función difundir los mensajes de un grupo local a otro grupo local en el caso de que esto sea necesario (ver Figura 22). Nótese la gran semejanza de los servidores causales con los separadores causales vistos en el apartado 4.2.4. Ambos procesos ejercerán de *routers*[13] que se encargarán de encaminar los mensajes de un grupo a otro. Por este motivo y con el objetivo de unificar la notación, de ahora en adelante llamaremos *routers* a los servidores causales. Así pues, en una composición de margarita tendremos grupos locales y el grupo de *routers*.

[13] Notación de [30]. Ver apartado 4.4.

Los procesos se agruparán de acuerdo a la frecuencia con la que se comuniquen con otros procesos y serán gestionados por un sistema de comunicación a grupos, como por ejemplo Horus, que realizará las siguientes funciones:

- Detectará fallos en los procesos y los reportará al resto de miembros del grupo.
- Permitirá la inclusión de nuevos miembros al grupo y la exclusión de miembros debido a fallos por caída (reconfiguración automática del grupo).
- Dentro de un mismo grupo, el sistema interconectará los procesos mediante enlaces punto a punto fiables FIFO.
- Los mensajes entregados dentro de un grupo serán almacenados por el sistema de comunicación a grupos hasta que el sistema sepa que se han recibido en todos los destinatarios. De este modo, en caso de fallo, el sistema podrá reenviar los mensajes enviados por procesos fallidos a los procesos a los que aún no se han entregado.
- Proporcionará un mecanismo de estabilidad dentro de un grupo, considerando que un mensaje es estable cuando se ha entregado a todos los procesos destinatarios. Este mecanismo permitirá eliminar la información innecesaria de la información de control. Para ello, se va a asumir que cuando se entregue un mensaje al nivel de aplicación de un proceso, éste informará al sistema de comunicación a grupos de que se ha efectuado la entrega.

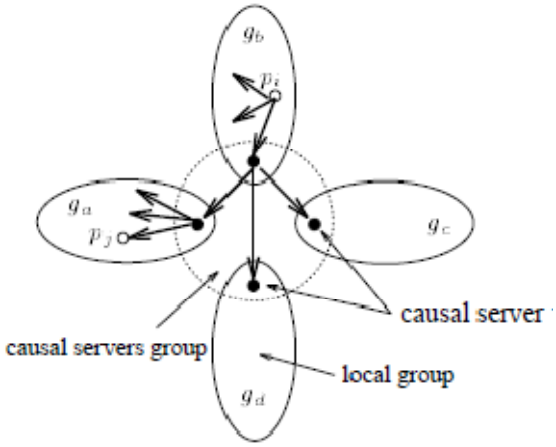


Figura 22. Composición con forma de margarita. [11]

4.3.3 La arquitectura de margarita

En este apartado se explicará cuál es el procedimiento para enviar/recibir mensajes tanto a/de procesos de un mismo grupo como a/de procesos de distintos grupos empleando la composición de la margarita. Para ello se comenzará describiendo el caso base en el que se tiene varios grupos locales asignados a un solo grupo de *routers* (ver Figura 22):

- Recepción y entrega de un mensaje *m*: cuando un proceso *q* reciba un mensaje *m*, *q* lo almacenará en un buffer de mensajes. Si *q* es destinatario de *m*, cuando

todos los mensajes que precedan causalmente a *m* se hayan recibido y entregado al nivel de aplicación, *m* será entregado.

- Envío y difusión de un mensaje *m* a destinatarios del mismo grupo local: cuando un proceso *p* envíe un mensaje *m*, el proceso *p* realizará una difusión a todos los procesos de su grupo local (incluido el *router* del grupo), utilizando para ello el protocolo de *broadcast* causal convencional, que requiere sólo un vector de enteros (un entero por cada proceso del grupo) [46].

- Envío y difusión de un mensaje *m* a destinatarios de otros grupos locales: en el caso de que alguno de los destinatarios sea un proceso que no se encuentre en el grupo local, cuando al *router* del grupo local de *p* se le entregue *m*, difundirá *m* a todos los miembros del grupo de *routers*. Cuando a los miembros del grupo de *routers* se les entregue *m* (después de que se les entregue todos los mensajes que preceden causalmente a *m*), los *routers* comprobarán si alguno de los procesos de su grupo local forma parte de los destinatarios de *m* y en tal caso difundirán el mensaje en su grupo local.

- Estabilidad de un mensaje *m*: un *router* no reportará al grupo de *routers* que *m* es estable hasta que sea estable en el grupo local al que pertenece y reportará inmediatamente un mensaje *m* como estable si *m* no está dirigido a ningún miembro de su grupo local. Además, el *router* del grupo local que ha originado *m* no reportará a su grupo local que *m* es estable hasta que todos los *routers* del grupo de *routers* indiquen que *m* es estable. Finalmente, cuando se reporte que *m* es estable en el grupo local emisor, *m* podrá ser descartado y eliminado de los buffers de recepción.

De esta forma, con este protocolo de difusión, todos los destinatarios de un mensaje recibirán dicho mensaje, manteniendo acotado en cada grupo el tamaño de la información de control causal, al mismo tiempo que se mantiene un nivel alto de tolerancia a fallos ya que todos los miembros de un grupo recibirán los mismos mensajes (difusiones uniformes), con lo que en caso de pérdida de un mensaje, éste podrá recuperarse rápidamente a partir de los buffers de recepción de mensajes de cualquier proceso.

No obstante, en este protocolo debe contemplarse explícitamente el caso en el que un *router* falla. Este caso es más problemático porque el *router* pertenece a dos grupos y los mensajes de un grupo no han sido propagados al otro grupo. Así pues, cuando un *router* cae, su grupo local lo detecta y escogerá un nuevo *router*, siempre de forma determinista. Cuando el nuevo *router* se una al grupo de *routers*, se le enviará un mensaje en el que se incluirá una copia de todos los mensajes que aún no han sido marcados como estables, que éste a su vez reenviará a los miembros de su grupo local. Los miembros del grupo local, basándose en el identificador de los mensajes, descartarán los mensajes duplicados. Para saber qué mensajes han recibido, cada proceso almacenará en memoria un vector con una entrada para cada proceso del

sistema que en ningún momento afectará a la escalabilidad, ya que nunca se adjuntará a los mensajes.

Nótese que, a diferencia del protocolo visto en el apartado 4.2 en el que se recomendaba un sistema estático en el que los miembros del sistema no estuviesen variando continuamente, en esta arquitectura se permite un sistema dinámico, ya que de la inserción o del abandono de un grupo se encargará el sistema de comunicación a grupos. Por tanto, la composición de margarita variará conforme se añadan o se eliminen procesos al sistema. La Figura 23 muestra un ejemplo de cómo surgen nuevos grupos en el sistema conforme se van añadiendo nuevos miembros a los grupos existentes (g_a), siendo demasiado costoso realizar difusiones dentro de ese grupo. Como puede observarse en g_b, cuando se cree un nuevo grupo, se elegirá un representante de ese grupo que realizará las funciones de *router*, integrándose de este modo en el grupo de *routers*. El sistema podría seguir creciendo hasta llegar a la situación mostrada en la Figura 22.

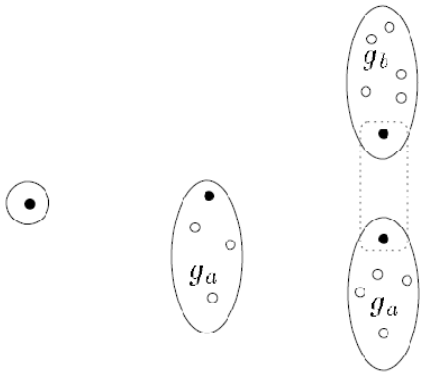


Figura 23. Evolución de la arquitectura de margarita. [11]

Sin embargo, cuando en la composición de margarita existen tantos grupos locales de forma que no es eficiente realizar difusiones dentro del grupo de *routers*, la arquitectura base en la que sólo existe una composición de margarita es insuficiente. Por este motivo, es conveniente añadir otra composición de margarita en el sistema e interconectar ambas composiciones, repartiendo los procesos existentes entre ambas composiciones, de acuerdo a la frecuencia con la que se comunican y/o de acuerdo a la topología física de la red.

Para unir las dos margaritas se formará un nuevo grupo, al que denominaremos grupo de *hiper-routers* (*hyper-server's group* en la nomenclatura de los autores), que estará compuesto por los dos *routers* que cada composición haya escogido como representante. A estos representantes se les denominará *hiper-routers* (*hyper-server*). En la Figura 24 se muestra un sistema compuesto por tres margaritas.

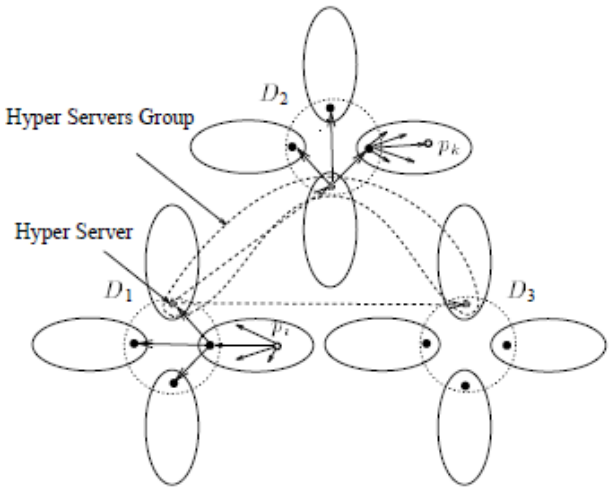


Figura 24. Sistema con 3 margaritas donde p_i envía un mensaje a p_k. [11]

Cuando se trabaja con varias composiciones de margarita en el sistema, el protocolo de envío de mensajes se modifica para adaptarse a la nueva arquitectura. De este modo, cuando al *router* de una margarita, que actúa además como *hiper-router,* se le entregue un mensaje *m* cuyos destinatarios pertenezcan a otra margarita, dicho *hiper-router* difundirá *m* en el grupo de *hiper-routers*. El resto de *hiper-routers* difundirá el mensaje en su margarita sólo si el mensaje va destinado a alguno de sus procesos. En el caso de que así sea, el *hiper-router* difundirá *m* en el grupo de *routers*, que, a su vez, difundirá *m* en los grupos locales correspondientes. En todo este proceso debe tenerse en cuenta que un mensaje sólo se entregará una vez se hayan entregado aquellos mensajes que lo precedan causalmente.

En caso de que el sistema continúe creciendo siempre se podrá jerarquizar el sistema un nivel más, siendo suficiente en cualquier aplicación utilizar de 2 a 4 niveles ya que el número de niveles crece de forma logarítmica con el número total de procesos.

Como resultado de este protocolo, con la arquitectura de margarita se consigue mantener la semántica de entrega causal en todo el sistema (propiedad de seguridad) y que todos los mensajes se entreguen a sus destinatarios (propiedad de viveza)[14].

4.3.4 Análisis de rendimiento y conclusiones

En lo referente al análisis de rendimiento, los autores del artículo no aportan muchos resultados, pero sí presentan posibles optimizaciones.

Sabemos que las veces que se reenvía un mensaje para que sea entregado a todos los destinatarios dependerá de en cuantos grupos estén repartidos dichos destinatarios. Siempre se realizará una difusión en el grupo local del proceso emisor, una difusión en cada grupo local en el que exista un destinatario y una difusión en cada grupo de *routers* que haya que atravesar.

[14] Las demostraciones de estas propiedades pueden consultarse en el apartado 4 de[11].

En cuanto a la información de control necesaria para mantener el orden causal, como ya se ha comentado antes, cada mensaje incluirá un vector de tamaño *n*, siendo *n* el número de componentes del grupo sobre el que se va a difundir el mensaje.

Sin embargo, tanto el número de mensajes, como el tamaño de la información de control y los retrasos pueden reducirse, teniendo en cuenta que reducir una de estas cosas penaliza otra:

- Se puede modificar el protocolo para hacer que un *router* difunda un mensaje al resto de miembros del grupo de *routers* tan pronto como lo reciba, sin esperar a que se le entregue el mensaje. Del mismo modo, un *router* puede difundir un mensaje en su grupo local en cuanto reciba un mensaje. Sin embargo, esto requiere que cada mensaje lleve adjunto un vector por cada grupo que atraviese y además complica las reglas de entrega causal a los destinatarios.

- Otra posible optimización es enviar un mensaje dentro de un grupo sólo a los destinatarios en vez de hacer un *broadcast*. No obstante, esto incrementa el tamaño de la información de control dentro de un mensaje, ya que ahora sería necesaria una matriz de tamaño n^2en vez de un vector de tamaño n. Además, para que el sistema siga siendo tolerante a fallos, en un sistema en el que se soportan *k* fallos, un mensaje no podría entregarse hasta que se supiera que se ha recibido en al menos *k* procesos más.

- Por último, los mensajes dentro de un grupo local pueden enviarse punto a punto al *router* para que éste lo reenvíe punto a punto a los destinatarios del mensaje dentro de ese grupo en vez de difundirlo directamente en todo el grupo. En los grupos de servidores, el reenvío de los mensajes se realizaría mediante el tradicional protocolo de *broadcast* causal.
  Con esta técnica se reduce considerablemente el número de mensajes para enviar un mensaje y la información de control, pero el *router* realizará más trabajo ya que todos los mensajes deberán pasar por él, realizando siempre un paso más de lo necesario.

Todas estas optimizaciones con sus consecuencias también podrán realizarse en niveles más elevados de la jerarquía de margarita.

4.4 Comunicación entre grupos en composiciones de grupos escalables

Tras haber analizado la solución del protocolo de los separadores causales [44] donde se pretendía reducir la información de control basándonos en la topología de red del sistema, y tras ver cómo agrupar los procesos independientemente de la topología para reducir la información de control y proporcionar tolerancia a fallos en la arquitectura de margarita [11], siempre manteniendo la semántica de entrega causal en ambas soluciones, en este apartado se introducirá un nuevo enfoque en cuanto a agrupación de procesos se refiere. Para ello nos vamos a basar en el artículo *The Inter-group Router Approach to Scalable Group Composition*, que fue publicado por *Johnson, Jahanian y Shah* en 1999 [30].

En este nuevo enfoque, para mantener la escalabilidad en el sistema, los procesos se agruparán de acuerdo a la frecuencia con la que se comuniquen, debiendo estar ubicados los procesos de un mismo grupo en la misma red (LAN), poniendo especial énfasis en mantener la semántica de entrega global del sistema, sea cual sea (FIFO, causal o total). Además, cada grupo escogerá un proceso representante que será el encargado de comunicarse con el resto de grupos, diferenciando entre la comunicación dentro de un mismo grupo (*intragroup*) de la comunicación entre distintos grupos (*intergroup*).

Nótese las semejanzas con la arquitectura de margarita donde, pese a diferenciarse en que no depende de la topología, la arquitectura de margarita no será más que un caso particular de los que se estudiarán en [30], en el que existen múltiples emisores en un sistema con entrega causal.

4.4.1 Introducción al problema

Es habitual en los sistemas distribuidos utilizar un sistema de comunicación a grupos. Esto es debido a que, de este modo, la aplicación que se ejecute en dicho sistema no tiene que responsabilizarse de la pertenencia a grupos ni de la tolerancia a fallos a la hora de realizar las difusiones al resto de procesos.

Sin embargo es bien conocido que los sistemas de comunicación a grupos no escalan bien cuando se usan con grandes aplicaciones distribuidas. Por ejemplo, un gran número de protocolos de comunicación a grupos utilizan la arquitectura lógica de *token ring* ([1],[39],[17]). Con esta topología la latencia del mensaje crece linealmente con el tamaño del grupo y el número de mensajes, ya que cada nuevo proceso incrementa el tiempo de rotación del *token*. Además, en otros protocolos el tamaño del mensaje también se ve incrementado debido a la sobrecarga que introduce la información de control, haciendo que el sistema llegue a un punto en el que deje de ser escalable.

Por todos estos motivos surge una nueva vertiente en la que se opta por gestionar el sistema mediante grupos de procesos ([39],[11],[22]) en vez de ubicar a todos los procesos en un solo grupo ([9],[10],[44]). Esto es debido a que la mayor parte de las aplicaciones distribuidas tienen una estructura organizativa muy clara de forma que se

puede sacar partido de los protocolos de comunicación a grupos, agrupando normalmente aquellos procesos que realicen la misma funcionalidad y/o que se comuniquen con bastante frecuencia, consiguiendo de esta forma reducir la sobrecarga de control que producen, por ejemplo, los protocolos de replicación o la gestión de la consistencia dentro de un mismo grupo.

Así pues, el objetivo de este artículo es proponer una solución generalizada para sistemas escalables en los que se trabaje con composición de grupos, independientemente de los protocolos que se utilicen en cada grupo para enviar un mensaje y de la semántica de entrega global del sistema. Para ello será necesario distinguir las comunicaciones entre los miembros de un mismo grupo, de las comunicaciones entre grupos, siendo el encargado de realizar la comunicación entre grupos un proceso representante de cada grupo (elegido siempre de forma determinista).

Además, esta solución permitirá trabajar con los denominados dominios de orden (*ordering domains*), que fueron descritos por *Birman* en [15]. Los dominios de orden introducen en el sistema una funcionalidad importante ya que permiten que un mismo grupo de procesos trabaje con dos semánticas de entrega diferentes en función de con qué grupos se comunique, mejorando la escalabilidad del sistema. Por ejemplo, imaginemos que en un sistema un determinado conjunto de grupos de procesos necesitan comunicarse mediante entrega total. Si sólo se trabaja con un dominio de orden estaríamos obligando a que todos los grupos del sistema mantuviesen la semántica de entrega total, aunque esto no fuera necesario en todos los grupos, con el consecuente coste adicional. Sin embargo, con los dominios de orden se puede añadir a los grupos que requieran entrega total en un dominio aparte (además de a cualquier otro dominio si es necesario), utilizándose en el resto del sistema entrega FIFO o causal, según requiera el sistema.

Por último, la arquitectura con la que se trabajará requerirá que los procesos de un mismo grupo estén ubicados en la misma red física. De esta forma se conseguirá una semántica de entrega más flexible, reduciendo además la información de estado global que se propagará entre los grupos.

4.4.2 Notación y descripción del sistema

En la arquitectura que se va a utilizar, vamos a asumir que el sistema podrá ser síncrono o asíncrono y que estará formado por un conjunto de procesos P de tamaño n. Estos procesos se clasificarán en grupos $G=\{g_1,g_2,...g_m\}$, de manera que un proceso sólo podrá ser miembro de un solo grupo, al que denominaremos grupo local. De esta forma, si $p_i \in g_j$, p_i es miembro del grupo local g_j, mientras que el resto de grupos se considerarán grupos remotos.

En cuanto a la ubicación física de los procesos, sólo se asumirá que los procesos de un mismo grupo deberán ubicarse en la misma red física. Sin embargo, esta red podrá estar compartida con otros grupos, a pesar de que una red física por cada grupo sería el caso ideal.

Otro punto importante a tratar en la arquitectura del sistema será la entrega de los mensajes. Como en los artículos comentados previamente, un proceso de aplicación no recibirá automáticamente un mensaje de la red, sino que será el gestor de entregas (ver apartado 2.2) el que se encargue de recibir los mensajes y entregarlos de manera ordenada al nivel de aplicación. Un hecho importante a destacar es que los mensajes enviados a un grupo se entregarán en todos los procesos miembros de ese grupo. En realidad este hecho no es obligatorio dada la arquitectura que se plantea, pero simplificará la entrega de los mensajes entre grupos (*intergroup*) y hará que el sistema sea tolerante a fallos.

Por otra parte en el sistema podrán coexistir grupos cerrados o abiertos. Un grupo cerrado es aquel en el que los destinatarios de los mensajes sólo son miembros de ese grupo local, mientras que un grupo abierto será aquel en el que parte de los destinatarios de un mensaje son miembros de un grupo remoto. Es decir, en los grupos cerrados sólo se realizarán comunicaciones *intragroup* mientras que en los grupos abiertos se realizarán tanto comunicaciones *intragroup* como *intergroup*.

En las comunicaciones dentro de un mismo grupo los procesos utilizarán el protocolo de comunicación a grupos habitual, tal y como ocurriría en el caso de que sólo se utilizase un único grupo en todo el sistema. Sin embargo, cuando se desea enviar un mensaje a otros grupos no se puede utilizar tal cual el protocolo de comunicación a grupos puesto que, a diferencia de la arquitectura de margarita, no existe un grupo que intercomunique dos grupos. Por este motivo cada grupo deberá seleccionar al menos un representante (al que denominaremos *router*), que ejercerá la misma funcionalidad que el resto de procesos de aplicación, preocupándose además de intercomunicar dos grupos. Los *routers* presentan puntos importantes de fallo y por lo tanto deberá utilizarse algún mecanismo de replicación para que, en caso de que falle un *router*, otro proceso pueda ejercer de representante[15].

Para las comunicaciones entre *routers* podrán utilizarse desde protocolos no fiables como UDP, hasta protocolos más complejos como pueden ser las difusiones a grupos. Además, es posible que un mismo representante interconecte más de un grupo usando protocolos distintos para cada uno de ellos (ver Figura 26).

La Figura 25 y la Figura 26 muestran posibles ejemplos de sistemas en los que se utiliza la composición de grupos.

[15] No obstante, el artículo se centra más en la semántica del sistema, con lo que no se hará hincapié en las cuestiones referidas a la replicación y a la tolerancia a fallos.

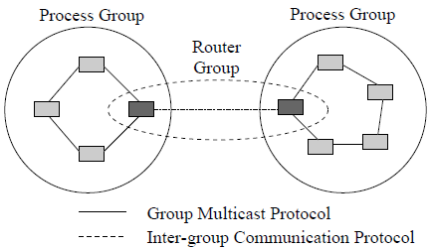


Figura 25. Sistema con 2 grupos interconectados por sus *routers*. [30]

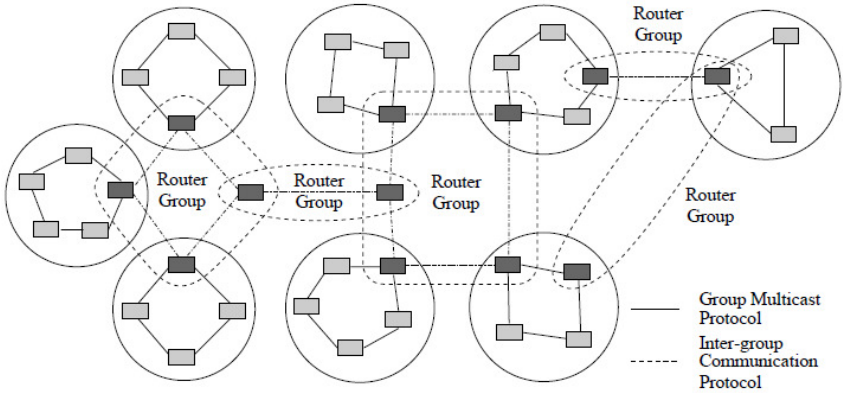


Figura 26. Composición de grupos más compleja con múltiples *routers* y múltiples protocolos de comunicación entre grupos. [30]

Nótese que en topologías tan complejas, como en el caso de la Figura 26, puede haber más de un camino de comunicación entre dos grupos. Por este motivo los *routers* deben tener la capacidad de enrutar y, por tanto, elegir cuál será el siguiente salto en el reenvío de mensajes.

Por otra parte en la Figura 27a se puede observar como un mismo *router* puede ejecutar varios protocolos de comunicación en función de con qué grupo se intercomunique mientras que en la Figura 27b se muestra la pila de protocolos de un proceso de aplicación en un sistema con composición de grupos, destacando la capa de filtrado, que introduce nuevas cabeceras en el mensaje que ayudarán a los *routers* a entregar el mensaje en los grupos destinatarios y además permite que un mensaje enviado a otro grupo se entregue a todos los *routers* del grupo local emisor.

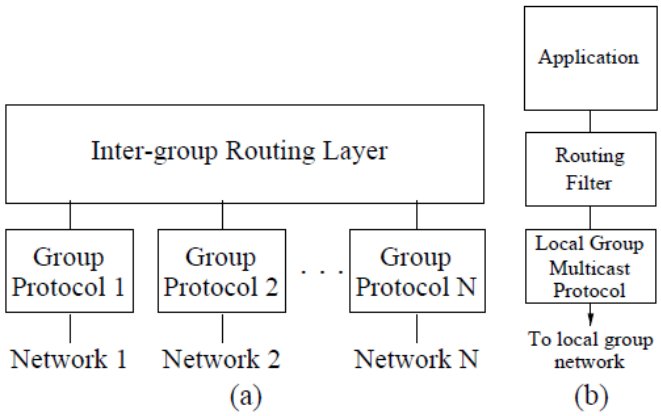


Figura 27. Pila de protocolos de los *routers* (a) y pila de protocolos de los miembros de un grupo (b). [30]

Los *routers* enviarán los mensajes siempre en orden FIFO, garantizando la propiedad *FIFO forwarding* que será crucial para mantener la semántica de entrega en todo el sistema. A continuación se define dicha propiedad:

> *"Dado un grupo de procesos G, que es parte de la composición de grupos del sistema, y un conjunto de mensajes entre grupos M cuyo grupo emisor es G, se asumirá que los mensajes entregados en G se harán mediante un determinado orden semántico S. Si todos los protocolos de comunicación son FIFO ordenados (sender-based FIFO delivery) y todos los mensajes en M se reenvían a través de la misma secuencia de routers para cada destino, entonces todos los mensajes en M se entregarán con la semántica de orden S en todos sus destinatarios."*

En cuanto a cómo se entrega un mensaje en los grupos destinatarios, el sistema se comportará de forma distinta dependiendo de si se entrega un mensaje en el mismo grupo emisor o si se entrega además en algún otro grupo. En el caso de los mensajes *intragroup* el filtro de encaminamiento simplemente pasa el mensaje sin añadir ninguna cabecera, entregándose el mensaje en el grupo local mediante el protocolo que emplee dicho grupo. Si el mensaje es *intergroup*, el filtro de encaminamiento introducirá cabeceras para indicar las direcciones de los grupos destinatarios y un identificador que permitirá detectar mensajes duplicados. Siempre y cuando el mensaje no deba entregarse con semántica de orden total[16], el mensaje también se entregará a los destinatarios del grupo local, tal y como se hacía con los mensajes *intragroup*.

Cada vez que un *router* reciba un mensaje éste lo reenviará al siguiente *router* hasta que se entregue en todos los *routers* de los grupos destinatarios. Por último, los *routers* reenviarán el mensaje a los procesos destinatarios utilizando el protocolo de *multicast* local del grupo. La capa de filtrado de cada proceso se encargará de eliminar las cabeceras, entregando el mensaje original a la aplicación libre de información de control.

Una vez descrito el sistema y su funcionamiento, veremos en el apartado 4.4.3 cómo se clasifican los sistemas en función de la composición de sus grupos.

4.4.3 La semántica de entrega en el sistema

Hasta ahora hemos visto cómo interconectar los diversos grupos de sistema, pudiendo llegar a representar topologías muy complejas. Sin embargo, aún no hemos visto cómo garantizar una semántica de entrega de principio a fin en el sistema. En esta sección se explicarán las diversas composiciones de grupos que pueden darse en un sistema con un solo dominio de orden y se explicará detalladamente cómo conseguir orden FIFO, orden causal y orden total en cada una de estas composiciones.

[16] En el apartado 4.4.3 se explicará por qué el caso del orden total es diferente.

Como se ha comentado anteriormente, la arquitectura que se propone permite el uso de múltiples dominios de orden. De manera formal definiremos un dominio de orden de la siguiente forma:

> *"Un dominio de orden D será un subconjunto de grupos de una aplicación distribuida, tal que D⊆G, y una semántica de entrega S concreta. Si intercambian un conjunto de mensajes M enviados por D y cuyos destinatarios incluyen grupos de D, todos los mensajes de M se entregarán a todos los destinatarios de D en un orden consistente con la semántica de entrega S".*

Los dominios de orden serán muy útiles para dividir el sistema en problemas más pequeños, facilitando enormemente la resolución de estos y llegando incluso a mejorar el rendimiento del sistema.

A continuación se describen las diferentes composiciones de grupos que se pueden dar en un sistema de uno o varios dominios, teniendo en cuenta que un sistema de varios dominios puede estar formado por las composiciones que aquí se describen (ver Figura 28).

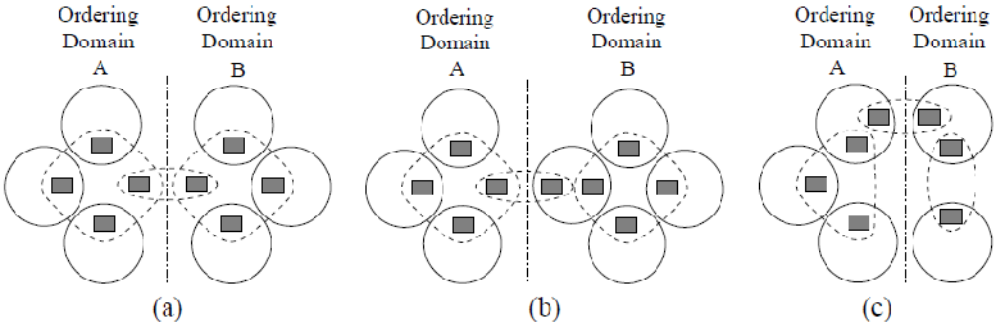
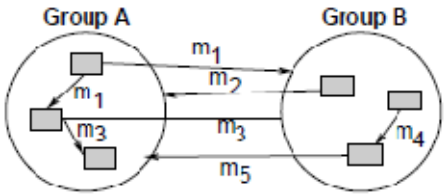


Figura 28. Métodos para conectar dominios de orden (sólo se muestran los grupos y sus *routers*). [30]

Composición de pares de grupos (un único dominio)

En la Figura 25 hemos visto un ejemplo de la estructura más simple y frecuente que se da en la composición de grupos de un sistema. Pese a ser una composición muy simple ya permite observar las posibilidades que tiene de ofrecer una semántica de entrega concreta en todo el sistema en función de la semántica de cada uno de los grupos y de la semántica de interconexión de ambos grupos.

Por ejemplo, si observamos el ejemplo de la Figura 29, vemos como los mensajes se han entregado y enviado en los grupos cumpliendo con las restricciones del orden causal. Esto no hubiera sido posible si los mensajes enviados de un grupo a otro no se hubieran enviado manteniendo el orden causal, que en este caso, al tratarse de la interconexión de únicamente dos grupos, es equivalente a mantener el orden FIFO al enviar los mensajes de un grupo a otro.



- Un proceso en G_a envía m_1 a ambos grupos.
- Al mismo tiempo G_b envía m_2 a G_a.
- Cuando G_a recibe m_2 envía m_3 a ambos grupos.
- Cuando un proceso de G_b recibe m_3 envía m_4 a G_b.
- Por último G_b recibe m_4 y envía m_5 a G_a.

Figura 29. Ejemplo de entrega causal en una composición de 2 grupos. [30]

Así pues, diremos que podemos interconectar una composición de dos pares de grupos que realicen comunicaciones *intragroup* y/o *intergroup*, manteniendo una semántica de entrega causal siempre que cada grupo realice entrega causal y la comunicación entre ambos grupos se realice con entrega *sender-based FIFO*.

Nótese que la interconexión de grupos con semántica causal implica la interconexión de grupos con semántica FIFO. Si se desea trabajar con esta semántica no sería necesario realizar entrega causal en cada grupo, bastaría con utilizar la entrega FIFO.

Sin embargo, la semántica de orden total es un poco más compleja. Por ejemplo, si tenemos un escenario compuesto por dos grupos en el que un proceso del grupo G_a envía un mensaje m_1 al mismo tiempo que un proceso del grupo G_b envía un mensaje m_2, por definición G_a recibe (*deliver*) m_1 y G_b recibe (*deliver*) m_2 antes de que el *router* de cada grupo envíe el mensaje al otro grupo, violando el orden total. Para solucionar este problema los mensajes deberán entregarse al *router* de cada grupo local antes de entregarse localmente en el grupo. De este modo el protocolo de comunicación entre grupos, convirtiéndose en un protocolo intrusivo, se encargará de ordenar los mensajes y de volver a entregarlos en el orden apropiado en cada *router*. A partir de este momento los *routers* podrán entregar el mensaje a los destinatarios de su grupo local cumpliendo con el orden total y sin entrar en conflicto con el otro grupo. Este método de reenvío de mensajes se denominará "reenvío de mensajes con orden total" (*totally ordered message forwarding*) o simplemente *total forwarding*. Además, para que se mantenga una semántica de orden total en todo el sistema, dentro de cada grupo deberá emplearse la entrega con orden total FIFO.

Formalmente diremos que, dada una composición de dos grupos en la que los *routers* realicen *total forwarding* y entrega total, si cada grupo ejecuta entrega total FIFO, entonces todos los mensajes se entregarán a todos los destinatarios en orden total.

Pese a que todos los mensajes deberán ser ordenados por un proceso antes de entregarse, esta arquitectura seguirá siendo más eficiente que utilizar un solo grupo ya que el orden total sólo debe llevarse a cabo entre los *routers* que interconectan ambos sistemas y no en todos los procesos del sistema.

La siguiente tabla (ver Tabla 3) resume qué tipo de semántica de entrega debe ejecutarse tanto en los grupos como en la interconexión de grupos para conseguir una determinada semántica de entrega global:

	Semántica de			Semántica global
	Emisores	Entre grupos	Destinatarios	
Composición de 2 grupos	*Cualquiera*	*FIFO*	*Cualquiera*	*FIFO*
	causal	*FIFO*	*causal*	*causal*
	total	*total forwarding*	*total FIFO*	*total*

Tabla 3. Semántica global en composiciones de dos grupos

Composición multigrupo (un único dominio)

Con la composición multigrupo seguiremos trabajando con un único protocolo de intercomunicación de grupos pero con más de dos grupos, formando un gran dominio de orden. La Figura 30 nos muestra un ejemplo de composición multigrupo. Si observamos la estructura de esta composición nos damos cuenta de que es equivalente a la configuración de margarita y que la composición de dos grupos no es más que el caso base de una composición multigrupo.

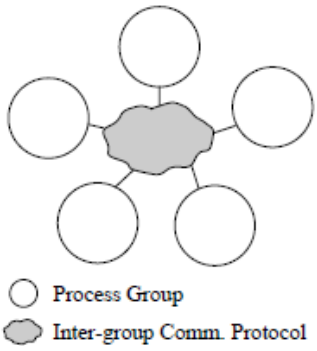


Figura 30. Ejemplo de composición multigrupo (sólo se muestran los grupos). [30]

En este tipo de composiciones se pueden dar dos tipos de comportamiento: que solamente exista un grupo que emita información a otros grupos (relación uno a muchos) o que exista más de un grupo que emita información a otros grupos (relación muchos a muchos). A continuación se describen estos comportamientos:

- Comunicación uno a muchos: En este tipo de comportamiento solamente un grupo enviará información a uno o varios grupos. Tal y como hemos visto en el caso de las composiciones de dos grupos, los protocolos de comunicación enviarán los mensajes con orden FIFO (*sender-based FIFO*) de forma que todos los mensajes se entregarán a sus destinatarios manteniendo la semántica, dado que sólo existirá un camino (*path*) entre cada par de grupos. Este tipo de comunicación puede darse cuando un grupo necesita publicar sus resultados en el resto de grupos.

- Comunicación muchos a muchos: En este tipo de comunicaciones existen al menos dos grupos que envían información a dos o más grupos. En este caso el orden FIFO en los envíos se vuelve insuficiente para semánticas de entrega más restrictivas que las de orden FIFO, puesto que hay muchos emisores y receptores, con lo que no se garantiza la semántica de entrega. Sin embargo, si

utilizamos un protocolo de comunicación entre grupos causal, podremos garantizar una semántica de entrega causal en todo el sistema.

Así pues, diremos que obtendremos entrega causal con múltiples emisores para un conjunto de mensajes M cuando dado un conjunto de mensajes M enviados en una composición de varios grupos por un conjunto de grupos emisores G_S a un conjunto de grupos G_R, tal que G_S y G_R pueden solaparse, si cada grupo en G_S y el protocolo de comunicación entre grupos ejecutan entrega causal y cada grupo en G_R ejecuta al menos entrega FIFO basada en el emisor.

Como en el caso de la composición de dos grupos, la semántica de entrega causal incluye a la semántica de entrega FIFO. Sin embargo, tal y como ocurría en la composición de 2 grupos, la semántica de entrega total continúa necesitando de cierta coordinación.

Con las composiciones multigrupo podemos dividir la entrega total en dos tipos de caso: cuando sólo un grupo pueda ser emisor y receptor (entrega total sin solape) y el caso más general en el que cualquier grupo puede enviar o recibir mensajes entre grupos. A continuación se detallan estos dos casos:

- Entrega total sin solape: Consideremos un conjunto de mensajes M enviados por un conjunto de grupos G_S a un conjunto de grupos G_R tal que $| G_S \cap G_R |=1$, es decir, sólo hay un único grupo que sirve de puente entre los grupos que puedan difundir mensajes y los que puedan recibirlos. Si todos los grupos en G_S realizan entrega total y, tanto los grupos de G_R como el protocolo de interconexión ejecutan entrega total FIFO, entonces todos los mensajes de M se entregarán en orden total en todos los destinatarios.
  Esto es debido a que como máximo sólo existe un grupo que realiza tanto envíos como recepciones de mensajes entre grupos, con lo que actuará de secuenciador y por tanto los mensajes serán recibidos trivialmente en orden total por todos los destinatarios. Además, por este mismo motivo no hace falta utilizar *total forwarding*, ya que la entrega total FIFO es suficiente.

- Entrega total (caso general): Consideremos un grupo de mensajes M enviados en una composición multigrupo. Si los *routers* llevan a cabo el reenvío con orden total (*total forwarding*) y todos los protocolos de comunicación ejecutan la entrega con orden total FIFO, entonces todos los mensajes de M se entregarán a los destinatarios en orden total.
  En este caso, para evitar que los grupos envíen simultáneamente mensajes violando la entrega total, sí que deberá usarse el *total forwarding* en los *routers*.

Debe observarse que muy rara vez una aplicación necesita que todos sus procesos trabajen con una semántica de entrega de orden total. En este caso se puede hacer uso de los dominios de orden para dividir el problema y obtener un sistema más eficiente.

En la Tabla 4 podemos ver, a modo de resumen, los tipos de semántica de entrega que deben darse en los grupos emisores y receptores y en el protocolo de interconexión de grupos para obtener una semántica de entrega global determinada en un sistema multigrupo:

	Semántica de			Semántica global
	Emisores	**Entre grupos**	**Destinatarios**	
Composición multigrupo	*Cualquiera*	*Cualquiera*	*Cualquiera*	*FIFO*
	Cualquiera	*FIFO*	*Cualquiera*	*FIFO*
	causal	*causal*	*causal*	*causal*
	total	*total forwarding*	*total FIFO*	*total*

Tabla 4. Semántica global en composiciones multigrupo

Composición multigrupo jerárquica (un único dominio)

Este tipo de composiciones surgen porque en ocasiones, en aplicaciones de un solo dominio de orden, una composición multigrupo con muchos grupos y muchos procesos por grupo puede llegar a no ser escalable. Este mismo problema ya se estudió en el apartado 4.3.3 cuando vimos la arquitectura de margarita. Cuando la margarita tenía demasiados procesos resultaba más eficiente dividir los procesos en una nueva margarita de acuerdo a la frecuencia con la que se comunicaban e intercomunicar ambas margaritas en vez de tener una sola margarita con muchos procesos.

Para interconectar varias composiciones multigrupo jerárquicamente utilizaremos una estructura de árbol (grafo acíclico) que será muy útil para evitar ciclos en los envíos de mensajes. Además, la estructura de árbol (ver Figura 31), por definición, hará que entre dos grupos sólo exista un camino de comunicación. Cuanto más balanceado sea el árbol más rápida será la comunicación entre dos grupos, sobre todo si los grupos que se comunican más a menudo penden de la misma raíz.

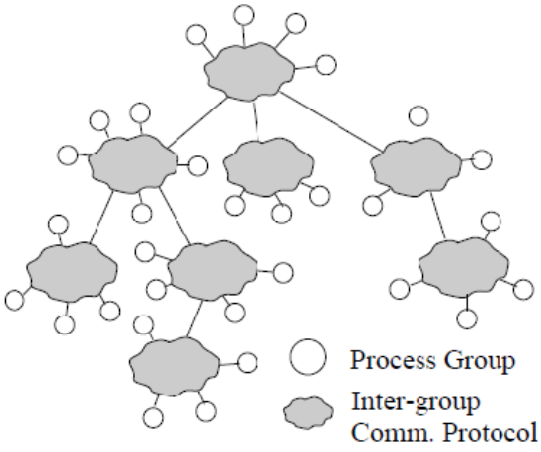


Figura 31. Ejemplo de sistema multigrupo jerárquico. [30]

En sistemas de este tipo, si queremos tener una semántica de entrega FIFO en todo el sistema simplemente bastaría con que todas las comunicaciones entre grupos fueran *sender-based FIFO*.

Si lo que queremos es una semántica de entrega global causal, tanto los grupos emisores como los receptores y el propio protocolo de comunicación entre grupos deberán ejecutar orden causal.

Si por el contrario lo que se desea es tener una semántica de entrega global con orden total deberemos aplicar una nueva propiedad basada en el *total forwarding* a la que denominaremos reenvío total jerárquico o *hierarchical total forwarding*. En este caso, cuando un mensaje sea enviado a un conjunto de grupos destinatarios G_R éste se entregará en la raíz del árbol más pequeño que incluya todos los grupos destinatarios de G_R. Así pues, dada una composición multigrupo jerárquica, si todos los grupos de procesos y los protocolos de comunicación trabajan con orden total FIFO y se utiliza el *hierarchical total forwarding* en la entrega de los mensajes en los protocolos de comunicación entre grupos, entonces todos los mensajes se entregarán en orden total en todos los destinatarios.

En la Tabla 5 se resumen los tipos de semántica de entrega que deben darse en los grupos emisores y receptores y en el protocolo de interconexión de grupos para obtener una semántica de entrega global determinada en un sistema multigrupo jerárquico:

	Semántica de			**Semántica global**
	Emisores	**Entre grupos**	**Destinatarios**	
Composición multigrupo jerárquica	*Cualquiera*	*FIFO*	*Cualquiera*	*FIFO*
	causal	*causal*	*causal*	*causal*
	total FIFO	*Hierarchical total forwarding*	*total FIFO*	*total*

Tabla 5. Semántica global en composiciones multigrupo jerárquicas

Composición multigrupo segmentada (múltiples dominios)

Además de todas las composiciones de grupos de varios dominios que se pueden hacer en un sistema jugando con las composiciones anteriormente comentadas, existe una composición muy especial, con múltiples dominios de orden, que se da comúnmente en sistemas como por ejemplo los utilizados en ingeniería industrial. En esta composición los grupos de procesos estarán divididos en etapas donde cada etapa se corresponderá con un dominio de orden que recibirá información de la etapa anterior y enviará información a la etapa siguiente.

En la Figura 32 se muestra un ejemplo de este tipo de sistemas.

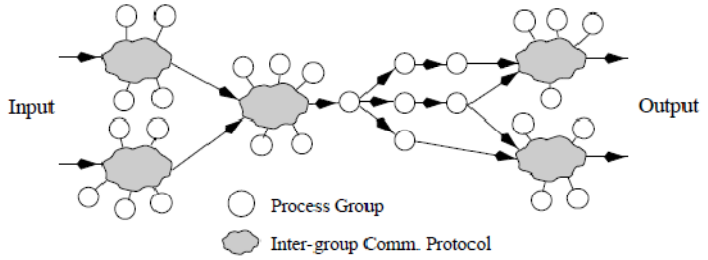


Figura 32. Ejemplo de sistema segmentado con múltiples dominios de orden. [30]

4.4.4 Análisis de rendimiento y conclusiones

En este artículo los autores realizan una evaluación de rendimiento del sistema[17] (latencia del mensaje y productividad del sistema) mediante simulación. Para ello emplearán la aplicación Opnet y como protocolo de difusión a grupos utilizan RTCAST [1], que está diseñado por ellos mismos.

El objetivo de la simulación es comparar la escalabilidad en un sistema de un solo grupo, un sistema compuesto por dos grupos y un sistema multigrupo formado por cuatro grupos. Cada uno de estos grupos se ejecutará en una LAN exclusiva y los procesos se distribuirán de manera equitativa en todos los grupos.

En cuanto a los protocolos de comunicación entre grupos, en el caso de 2 grupos se utilizará un protocolo *sender-based FIFO*, mientras que en la composición multigrupo utilizarán RTCAST.

Para garantizar que los mensajes se envíen siempre en el peor de los casos, éstos siempre se generarán justo después de que el nodo haya perdido el *token*. Además, con las mismas probabilidades los mensajes reenviados por los *routers* serán enviados a todos los grupos o sólo a un grupo.

Los resultados demuestran que incluso en el caso peor donde el 100% de los mensajes se envían a los cuatro grupos, la latencia del mensaje es menor que en el caso de un sistema con un solo grupo.

En cuanto a la productividad, los resultados presentados demuestran que ésta crece casi proporcionalmente con el número de grupos del sistema, haciendo que la latencia de los mensajes *intragroup* disminuya. Esto es debido a que conforme aumenta el número de grupos en el sistema, menor es el número de miembros en cada grupo.

Así pues, con la simulación se ha demostrado que como norma general será más escalable y eficiente dividir los procesos de un sistema en grupos de acuerdo a la frecuencia con la que se comuniquen en vez de utilizar un solo grupo en todo el sistema.

También hemos visto que, gracias a la cantidad de composiciones de grupos que se pueden hacer con esta arquitectura, se pueden representar incluso topologías muy complejas, manteniendo siempre la semántica de entrega de principio a fin.

En la Tabla 6 se muestra un resumen de las semánticas de entrega que garantizarán una semántica de entrega final en sistemas con un único dominio de orden.

[17] Los resultados al completo del análisis pueden consultarse en [30].

	Semántica de			Semántica global
	Emisores	**Entre grupos**	**Destinatarios**	
Composición de 2 grupos	*Cualquiera*	*FIFO*	*Cualquiera*	*FIFO*
	causal	*FIFO*	*causal*	*causal*
	total	*total forwarding*	*total FIFO*	*total*
Composición multigrupo	*Cualquiera*	*Cualquiera*	*Cualquiera*	*FIFO*
	Cualquiera	*FIFO*	*Cualquiera*	*FIFO*
	causal	*causal*	*causal*	*causal*
	total	*total forwarding*	*total FIFO*	*total*
Composición multigrupo jerárquica	*Cualquiera*	*FIFO*	*Cualquiera*	*FIFO*
	causal	*causal*	*causal*	*causal*
	total FIFO	*Hierarchical total forwarding*	*total FIFO*	*total*

Tabla 6. Resumen de semánticas globales según el tipo de composición

4.5 Interconexión de sistemas de paso de mensajes

En este apartado se va a analizar el artículo *On the interconnection of message passing systems* [6], que fue publicado recientemente (en 2007).

Pese a ser un artículo relativamente reciente veremos que las propuestas que plantean son muy parecidas a las vistas en el artículo que se ha analizado en el apartado 4.4, estando enfocadas, en este caso, a la interconexión de sistemas en vez de a la interconexión de grupos. Por este motivo, y dada la estrecha relación que guarda con la arquitectura de los sistemas de consistencia vistos en los apartados 3.2 y 3.3, se incluirá en este análisis, teniendo en cuenta que esta vez estará orientado a la interconexión de sistemas de paso de mensajes. Además, en este artículo se estudiará la arquitectura base sobre la que se trabajará en el apartado 4.6.

4.5.1 Introducción al problema

Hasta ahora en todos los artículos que se ha analizado hemos visto cómo ha ido cambiando la arquitectura del sistema. Primero empezamos teniendo un solo sistema en el que nos preocupábamos por reducir el tamaño de la información de control [44]. Después vimos que dividir los procesos de un gran sistema en grupos según la frecuencia con la que se comunicasen e interconectarlos en vez de trabajar con un solo grupo en todo el sistema aumentaba la escalabilidad del sistema, además de proporcionar tolerancia a fallos gracias al sistema de comunicación a grupos ([11],[30]). Sin embargo, los autores de [6] observan el sistema desde otro punto de vista. En esta ocasión no tenemos un sistema global que queremos dividir en subsistemas, sino que tenemos sistemas existentes que deseamos unir para formar un sistema más grande manteniendo siempre la semántica de entrega global del sistema. Esto puede ocurrir por ejemplo en sistemas de *cloud computing* o en redes P2P.

Tal y como ocurría en los apartados 3.2 y 3.3, construir un sistema de paso de mensajes a partir de dos o más subsistemas tiene dos ventajas importantes:

- Se pueden interconectar sistemas existentes sin necesidad de modificarlos. Es decir, cada sistema puede utilizar el protocolo local de difusión que desee.

- Si la red de cada sistema es mucho más rápida que los enlaces entre sistemas, es mejor formar un sistema interconectando los subsistemas que formar un solo sistema. Esto es debido a que si tratásemos el sistema como un solo gran sistema, las difusiones entre nodos ubicados en redes distintas serían muy lentas, mientras que si se forma el sistema interconectando los subsistemas, sólo un mensaje atravesaría la red por cada difusión entre pares de sistemas.

En la literatura existen muchos protocolos que interconectan sistemas causales formando un gran sistema causal ([11],[2]) y por tanto, dado que los sistemas causales son por definición sistemas FIFO, se puede afirmar que en la literatura existen

numerosas arquitecturas y protocolos de interconexión de sistemas FIFO. Sin embargo, no existen tantos protocolos que interconecten sistemas exclusivamente FIFO.

En este caso, los autores proponen un protocolo de interconexión de sistemas FIFO no necesariamente causales. Además, veremos por qué no es posible interconectar sistemas de orden total sin violar la restricción de mantener intactos los sistemas a interconectar[18].

4.5.2 Notación y descripción del sistema

Como en los casos anteriores, el sistema estará compuesto por nodos interconectados a través de una red. Dentro de cada uno de estos nodos es donde se ejecutarán los procesos de aplicación. Estos procesos se comunicarán entre ellos mediante mensajes que serán difundidos/recibidos en todo el sistema a través del servicio de *broadcast*, teniendo en cuenta que todos los mensajes enviados se recibirán en todos los procesos del sistema (propiedad de viveza).

En este caso los autores no son muy estrictos con la terminología de "entrega/recepción", asumiendo un modelo de difusión en el que existe una operación de envío de mensajes (*bc_send*) y una operación de recepción de mensajes (*bc_recv*). El motivo por el que no se distingue entre recibir o entregar un mensaje (ver apartado 2.2), es que, como veremos, en el caso de los sistemas FIFO, si el enlace que une un par de sistemas es FIFO ordenado, los mensajes se recibirán ordenados, con lo que podrán entregarse directamente al nivel de aplicación. De todas formas, continuando con la notación que se ha estado siguiendo en todo este trabajo, se utilizará el término de "entrega" cuando se haga referencia a que un mensaje se ha entregado al nivel de aplicación manteniendo la semántica de entrega global, ya que lo que nos interesa es mantener dicha semántica. Además, dado que la mayor parte de los sistemas de comunicación a grupos utilizan esta estructura, podríamos identificar con mayor facilidad un grupo con lo que antes hemos denominado como subsistema.

La Figura 33 muestra la arquitectura del sistema en un sistema con dos nodos, cada uno de ellos con dos procesos de aplicación:

[18] Nótese que en [6] los autores no hacen referencia al trabajo publicado en [30] donde se explican las directrices que hay que seguir en la interconexión de sistemas con orden FIFO y en la interconexión de sistemas con orden total.

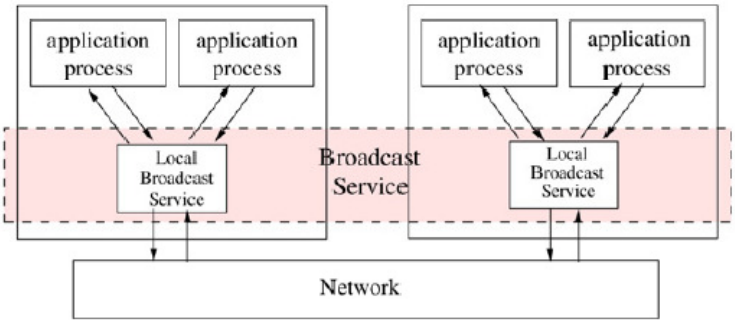


Figura 33. Arquitectura de sistema de paso de mensajes. [6]

Para interconectar varios sistemas FIFO manteniendo la arquitectura de los sistemas a interconectar, se hará uso de un Sistema de Interconexión (IS) del mismo modo que se utilizó en los apartados 3.2 y 3.3. Así pues, dentro de cada sistema se escogerá un proceso de aplicación que además asumirá la responsabilidad de ejercer como *router*[19]. Tal y como vimos en el apartado 4.4, los *routers* interconectarán pares de sistemas a través de un enlace, enviando y recibiendo mensajes de un sistema a otro mediante un protocolo de interconexión (protocolo-IS) que no afectará en modo alguno al protocolo de difusión local que emplee cada sistema (protocolos no intrusivos).

En la Figura 34 se observa el resultado de interconectar dos sistemas iguales al visto en la Figura 33, donde un proceso de aplicación en cada sistema ha adquirido la funcionalidad de *router*:

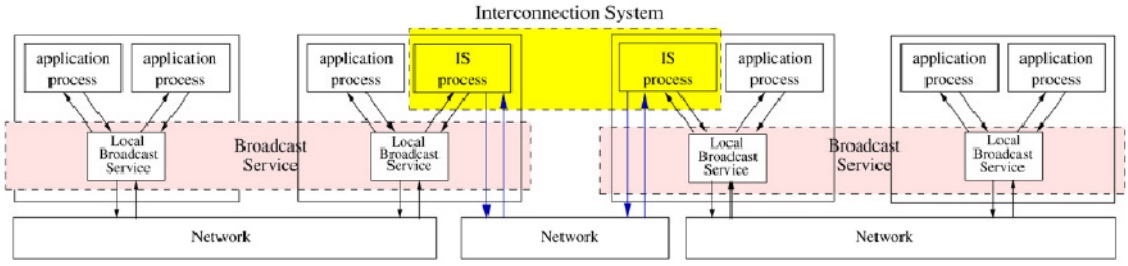


Figura 34. Interconexión de dos sistemas de paso de mensajes. [6]

4.5.3 Interconexión de sistemas con orden total

En el apartado 4.4 vimos que dos grupos (o sistemas en este caso) con semántica de entrega total no podían ser interconectados a no ser que se modificase el protocolo de *broadcast* de cada grupo. Esto es debido a que para mantener el orden total entre varios grupos es necesario utilizar un proceso coordinador (uno de los *routers*) que reciba y ordene todos los mensajes y que los devuelva ordenados a todos los *routers*, de manera que cada *router* difunda localmente los mensajes ordenados en su sistema. Sin embargo, esto rompería con la filosofía de los autores de utilizar protocolos no intrusivos para mantener intactos los sistemas, ya que obligaríamos a cambiar el protocolo de difusión interno de cada sistema.

[19] Notación de [30]. Los autores de [6] denominan este proceso proceso-IS (*Interconecting system processes*) o *isp*.

Por ejemplo, imaginemos dos grupos (o sistemas) A y B donde un proceso p_i de A envía mediante *broadcast* un mensaje m_1 mientras que un proceso p_j de B envía un mensaje m_2. Si los procesos reciben "al instante" sus propios mensajes no se cumpliría con la restricción de orden total, ya que m_2 se entregaría a p_i después de que lo hiciera m_1 mientras que m_1 se entregaría en p_j después de que lo hiciera m_2.

Por lo tanto, cuando un proceso envía un mensaje no puede difundirlo directamente en su grupo local, sino que debe enviarlo únicamente al *router* del grupo. Si esto no se hiciera así el mensaje se entregaría de forma casi inmediata, sin dar tiempo a que el *router* acordase el orden de los mensajes con el resto de *routers*.

La principal diferencia entre la explicación de los autores de [6] con respecto a lo que se ha visto en el apartado 4.4 es la nomenclatura (sistemas en [6] y grupos en [30]) y que en [6] los autores demuestran formalmente[20] por qué no se pueden interconectar los sistemas de orden total mediante el protocolo-IS.

4.5.4 Interconexión de sistemas con orden FIFO

Hasta ahora sabemos que se pueden interconectar sistemas FIFO porque hemos visto diversas arquitecturas y protocolos que interconectan sistemas causales que, por definición, son equivalentes a un sistema FIFO al que se le aplica además la restricción de orden local.

Con el protocolo-IS que se propone a continuación siempre se podrán interconectar todo tipo de sistemas FIFO, incluso los que no sean causales. Para ello vamos a empezar con el caso base, que consistirá en interconectar dos sistemas (S^0 y S^1). Como ya hemos visto en la descripción del sistema, cada uno de estos sistemas escogerá un proceso de aplicación para que ejerza de proceso de interconexión o *router*.

Para interconectar sistemas FIFO, los *routers* ejecutarán dos tareas atómicas adicionales que serán imprescindibles para la interconexión de los sistemas. A continuación se describen dichas tareas asumiendo que se van a interconectar dos sistemas S^k y $S^{k'}$:

- $Propagate^k_{out}$: transferirá un mensaje generado en S^k al sistema $S^{k'}$ a través del enlace que los une. Esta tarea se ejecutará inmediatamente después de que el mensaje se entregue en el *router* de S^k pero sólo se ejecutará si el mensaje proviene de S^k y no de $S^{k'}$ para evitar bucles.

- $Propagate^k_{in}$: Recibirá el mensaje enviado por $Propagate^{k'}_{out}$ y lo difundirá localmente a todos los procesos de S^k.

Además, para que los mensajes sean recibidos por un *router* en el mismo orden en el que se enviaron, el enlace tendrá que ser FIFO ordenado (*sender-based FIFO*). Este enlace podrá implementarse mediante paso de mensajes o con memoria compartida.

---

[20] La demostración formal puede encontrarse en la sección 3 de [6].

La Figura 35 muestra el pseudocódigo correspondiente a cada una de estas tareas donde isp^k sería el *router* de S^k y $isp^{k'}$ sería el *router* de $S^{k'}$.

Por otra parte, en la Figura 36 vemos cómo interactúan estas tareas en el protocolo-IS.

$Propagate^k_{out}(m)$ :: task which is	$Propagate^k_{in}(m)$ :: task which is
activated immediately after	activated immediately after
$bc\text{-}recv_{isp^k}(m, i)$ is executed	message m is received from $isp^{\overline{k}}$
begin	begin
if m was not received from $isp^{\overline{k}}$ then	$bc\text{-}send_{isp^k}(m)$
transfer m to $isp^{\overline{k}}$	end
end	

Figura 35. Psuedocódigo de las tareas *Propagate*$^k_{out}$ y *Propagate*$^k_{in}$. [6]

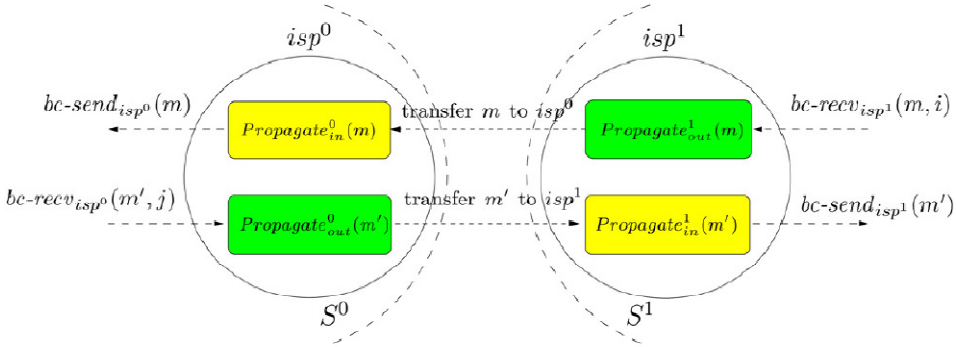


Figura 36. Esquema del protocolo-IS en sistemas FIFO. [6]

El protocolo-IS para interconectar sistemas FIFO, manteniendo la semántica de entrega FIFO en todo el sistema, podrá extenderse para interconectar más de dos sistemas simplemente teniendo en cuenta que el enlace que unirá cada par de sistemas tendrá que ser siempre FIFO ordenado. La demostración formal de que pueden interconectarse *n* sistemas mediante el protocolo-IS puede encontrarse en [6].

4.5.5 Análisis de rendimiento y conclusiones

El objetivo de este artículo es demostrar que los sistemas FIFO pueden interconectarse formando un gran sistema FIFO. Por este motivo los autores no se han centrado en la eficiencia. Sin embargo sí que comparan la eficiencia de su arquitectura con respecto a utilizar un único gran sistema, asumiendo para ello que siempre se utilizará el mismo protocolo de *broadcast*.

Tal y como ocurría en los apartados 3.2 y 3.3, el tiempo de respuesta que un proceso observa cuando se realiza una operación de difusión no se verá afectado, puesto que cada sistema continuará utilizando el mismo protocolo de *broadcast* localmente.

En cuanto al tráfico de red, vamos a asumir que el protocolo de difusión genera un mensaje por cada proceso al que haya que entregarle un mensaje difundido:

- En un sistema global con *n* procesos, cada operación de difusión generaría *n-1* mensajes.

- Con el protocolo-IS, para 2 subsistemas se generarían *n+1* mensajes dado que se añaden 2 *routers* y se envía un mensaje de un *router* a otro.

- Por tanto, con el protocolo-IS, para *m* subsistemas se generarían *n+m-1* mensajes, de forma que si *m* crece mucho el número de mensajes puede suponer un cuello de botella.

Por último, en cuanto a la latencia del sistema (tiempo que pasa desde que una difusión es visible en un proceso cualquiera del sistema), se considerarán despreciables las operaciones de cómputo local. Asumiendo *m* subsistemas interconectados en estrella y teniendo en cuenta que un único sistema tiene una latencia *l* y que el retraso obtenido al recibir un mensaje enviado entre dos *routers* es *d*, la latencia sería *3l+2d* en el peor de los casos.

En mi opinión, pese a que tanto la interconexión de sistemas FIFO como la interconexión de sistemas de orden total se estudiaron previamente en [30] bajo el punto de vista de la interconexión de grupos de comunicación, los autores de este artículo utilizan demostraciones más sencillas para probar algunas propiedades ya conocidas.

En el siguiente apartado veremos que el sistema de interconexión que se ha planteado servirá como base para la solución que se propone en [31], donde se mejorará el ancho de banda en la interconexión de sistemas FIFO y causales.

4.6 Interconexión de sistemas de difusión con múltiples canales FIFO

En este apartado se analiza un artículo reciente (2009) que culmina el análisis de los protocolos y arquitecturas que se han ido viendo para mejorar la escalabilidad del sistema en sistemas de paso de mensajes en la actualidad. El artículo en cuestión es *Parallel Interconection of Broadcast Systems with Multiple FIFO Channels* [31] y, como su propio nombre indica, en él se va a proponer un protocolo para interconectar sistemas de paso de mensajes FIFO y sistemas de paso de mensajes causales con múltiples enlaces FIFO entre pares de sistemas, a diferencia de lo que hemos visto en el apartado 4.5.4, donde los sistemas se interconectaban con un único enlace FIFO. Este protocolo será muy útil en sistemas en los que no sea posible delegar en la capa de red la elección de qué camino es el mejor para evitar la congestión, donde la paralelización en la capa de transporte o en la de aplicación puede ser la solución. De esta forma se podrá incrementar el ancho de banda del sistema y además se podrá reducir el problema de que el enlace que interconecta dos sistemas se convierta en cuello de botella en el caso de que se realicen muchas difusiones (se generen muchos mensajes en un sistema).

4.6.1 Introducción al problema

Sabemos que existen muchos trabajos que proponen mejoras en la interconexión de sistemas de paso de mensajes para aumentar la escalabilidad de estos sistemas. Muchos de ellos eran sistemas causales ([11],[30],[21])[21] en los que el objetivo era reducir la información de control tanto en los mensajes como en los procesos. Para ello, los sistemas se intercomunicaban a través de enlaces FIFO ordenados o, como en el caso de la arquitectura de margarita, mediante difusiones causales. Además, desde [44] y pasando por ([11],[30]), hemos visto que dividir un sistema causal en subgrupos hace que se reduzca la información de control de manera significativa.

Del mismo modo, esto se podía aplicar a la interconexión de sistemas FIFO ([30],[6])[22], donde vimos que se puede aumentar la escalabilidad de un gran sistema dividiendo el sistema en partes más pequeñas para evitar que las difusiones colapsaran los enlaces físicos que interconectaban los sistemas.

Sin embargo, en sistemas muy grandes en los que es necesario difundir grandes cantidades de mensajes, los enlaces que interconectan pares de sistemas pueden ser el cuello de botella del sistema. Recordemos que cuando hablamos de grandes sistemas podemos estar hablando de sistemas que ni siquiera se encuentran físicamente en el mismo edificio, sino que pueden estar distribuidos en ciudades distintas, como por ejemplo las redes P2P o el *cloud computing*. Como norma general, cada subsistema suele estar implementado sobre una *LAN* muy rápida, mientras que los enlaces físicos mediante los cuales se interconectan los subsistemas son mucho más lentos.

---

[21] Se incluye a [21] porque los sistemas de memoria podrían implementarse como sistemas de paso de mensajes.

[22] Nótese como en este artículo los autores tienen en cuenta el trabajo publicado en [30].

Así pues, para mejorar el ancho de banda en el sistema permitiendo el envío de mensajes en paralelo y siempre sin modificar los sistemas existentes que se quieren interconectar, en las siguientes secciones se adaptará el protocolo de interconexión (protocolo-IS) de sistemas FIFO visto en el apartado 4.5.4 y el protocolo de interconexión de sistemas causales visto en 3.2.4, de forma que se permitan múltiples enlaces FIFO entre pares de sistemas. Además, veremos cómo mantener la tolerancia a fallos con estos protocolos.

4.6.2 Notación y descripción del sistema

La descripción del sistema será prácticamente igual a la descrita en [6] (ver apartado 4.5.2) sólo que en este caso se especifica que trabajaremos con sistemas asíncronos. De nuevo, se utilizará la notación de [30] puesto que, en mi opinión, es la notación más genérica de todas las que hemos visto hasta ahora.

Consideraremos posibles los fallos de tipo caída [25] y asumiremos que los canales de comunicación serán fiables, es decir, aunque podrán fallar temporalmente los mensajes se entregarán en sus destinos, siempre y cuando el nodo no esté caído.

Las operaciones de difusión *bc_send* (envío) y *bc_recv* (recepción) serán uniformes (ver apartado 2.4.2) con lo que trabajaremos con difusiones uniformes de orden FIFO en la interconexión de sistemas FIFO y con difusiones de orden causal en la interconexión de sistemas causales. Las difusiones uniformes nos permitirán una mayor tolerancia a fallos ya que con ellas nos aseguramos de que todos los procesos recibirán cualquier mensaje que haya sido difundido.

Nótese que todo el rato se está hablando de interconectar sistemas FIFO y sistemas causales. Obviamente podríamos interconectar sistemas causales cuyo resultado fuera un sistema FIFO, pero este no es el objetivo. Así pues, cuando se interconecten sistemas de un determinado tipo, consideraremos que un sistema estará correctamente interconectado si la semántica de entrega global del sistema final es del mismo tipo que la de los sistemas interconectados.

Cuando trabajábamos con un único enlace entre pares de sistemas, era necesario que en cada sistema un proceso de aplicación asumiese el rol de *router* (o *isp*, según la nomenclatura de [6]). Puesto que ahora puede ocurrir que un sistema se interconecte con otro a través de varios enlaces, será necesario añadir más de un *router* por sistema.

Por ejemplo, vamos a asumir que se interconectan dos sistemas S^k tal que $k \in \{0,1\}$ y $S^{k'}=S^{1-k}$. Cada uno de estos sistemas tendrá *n routers*, de forma que $router_v^k$ denotará el *router* *v*-ésimo[23] en el sistema S^k. Una vez definido el número de *routers* en cada sistema los procesos se dividirán en grupos, siendo obligatorio que todos los procesos estén asignados en un único grupo, y cada grupo se asociará a un *router*, de manera que $set_w(router_v^k)$ representará al grupo *w* asociado al $router_v^k$. En la Figura 37 podemos

[23] Posteriormente se analizará cual es el número adecuado de *routers* en un sistema.

ver un ejemplo en el que se aplica el protocolo-IS paralelo en dos sistemas. En este sistema se observa cómo cada subsistema tiene un número determinado de procesos agrupados en conjuntos. En el sistema S^0 cada grupo tiene asociado un *router* propio, mientras que en S^1 los grupos $set_1(router_2^1)$ y $set_2(router_2^1)$ comparten el $router_2^1$.

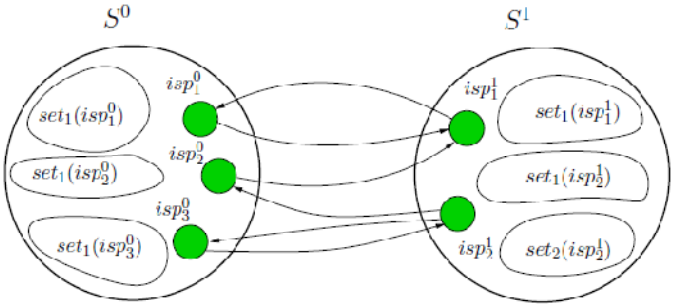


Figura 37. Interconexión de dos sistemas con el protocolo-IS paralelo. [31]

Los *routers* serán un proceso de aplicación más del sistema, con lo que podrán utilizar el sistema de comunicación de S^k como si de un proceso de aplicación corriente se tratara. Sin embargo excluiremos estos procesos al considerar el sistema resultante S^T.

En los siguientes apartados veremos qué tareas ejecutaran dichos *routers* dependiendo del tipo de semántica de entrega del sistema.

4.6.3 Interconexión de sistemas FIFO con múltiples enlaces FIFO

Una vez conocida la arquitectura del sistema con el protocolo-IS paralelo, sólo queda saber cómo se comportará cada *router* para que el sistema interconectado sea un sistema FIFO ordenado. Como en el apartado 4.5.4, los *routers* ejecutarán dos tareas atómicas que se definen a continuación:

- $Propagate_{out}(router_v^k, m)$: transfiere el mensaje m enviado por un proceso del grupo $set_w(router_v^k)$ de S^k a $S^{k'}$. Concretamente, el mensaje se enviará por un determinado enlace $link_w(router_v^k)$ a $S^{k'}$. Se debe tener en cuenta que un *router* puede enviar mensajes a muchos *routers* y recibir mensajes de muchos *routers* que no tienen por qué ser los mismos.

- $Propagate_{in}(router_v^k, m)$: recibe los mensajes enviados por $S^{k'}$ y los reenvía a S^k, difundiendo el mensaje a todos los procesos del sistema sin importar a qué grupo pertenecen.

Además de que los *routers* ejecuten estas tareas, recordemos que los enlaces tendrán que ser FIFO ordenados (*sender-based FIFO*), es decir, los mensajes mandados por un *router* en S^k se entregarán en el *router* correspondiente de $S^{k'}$ en el mismo orden en el que fueron enviados. Teniendo en cuenta esto, se puede afirmar que dos sistemas FIFO pueden interconectarse correctamente (formar un sistema FIFO). Esto es debido a que se puede considerar que un subsistema de [6] no es más que el caso base de la arquitectura que estamos viendo ahora, dado que podríamos considerar que todos los procesos del subsistema están agrupados en un único grupo que a su vez está asociado

al único *router* del subsistema. Como cada proceso sólo se encuentra en un grupo y cada grupo tiene asignado un único *router*, los mensajes de un mismo proceso siempre se enviarán por orden FIFO y como gracias al canal FIFO ordenado los mensajes se entregarán en este orden, podemos afirmar que se mantiene el orden FIFO en todo el sistema. Del mismo modo podríamos llegar a interconectar *n* sistemas[24].

En la Figura 38 se muestra el pseudocódigo de las tareas $Propagate_{out}$ y $Propagate_{in}$ que ejecutarán en este caso los *routers*[25].

$Propagate\_out(isp_v^k, m)$ :: task which is activated once $bc\text{-}recv_{isp_v^k}(m)$ is executed begin if m was sent by a process in $set_w(isp_v^k)$ then transfer m to $link_w(isp_v^k)$ end	$Propagate\_in(isp_v^k, m)$ :: task which is activated immediately after message m is received from $\overline{S^k}$ begin $bc\text{-}send_{isp_v^k}(m)$ end

Figura 38. Protocolo de interconexión en cada *router*. [31]

Con este protocolo se consigue mejorar los resultados que ofrecía el protocolo-IS visto en el apartado 4.5.4. El tiempo de respuesta y la latencia continuarán siendo los mismos que en el protocolo-IS sin enlaces paralelos, pero se consiguen reducir los problemas de cuello de botella en el tráfico de red que podía ocasionar tener sólo un enlace entre cada par de sistemas.

En resumen, los pasos que se deben seguir para conseguir una arquitectura en la que se interconecten correctamente *n* sistemas FIFO son los siguientes:

- Paso 1: asignar a cada proceso *p* de S^k un *router* al que llamaremos *router(p)*.
- Paso 2: para cada *router(p)*, escoger una serie de *paths* a otros *routers*. Llamaremos a estas rutas *paths(p)*. Se debe tener en cuenta que estas rutas sólo pasarán por un mismo *router* de un sistema. Además, diferentes rutas podrán compartir enlaces.
- Paso 3: los mensajes generados por *p* se transferirán al resto de sistemas a través de *router(p)* por las rutas indicadas en *paths(p*).
- Paso 4: Cuando un *router* recibe un mensaje lo difunde a todos los procesos de su sistema.

En la Figura 39 se muestra un ejemplo de sistema interconectado con 4 subsistemas. Nótese que un mismo *router* siempre se interconecta con un sistema a través del mismo enlace (ver flechas del mismo color), evitando posibles ciclos. En el siguiente apartado veremos que, en el caso de la interconexión de sistemas causales, los ciclos serían un

[24] Ver demostración formal en [31].
[25] Recordemos que en la notación de [31] los *routers* se denominan *isp*.

grave problema porque evitarían que la semántica de entrega global del sistema fuera causal.

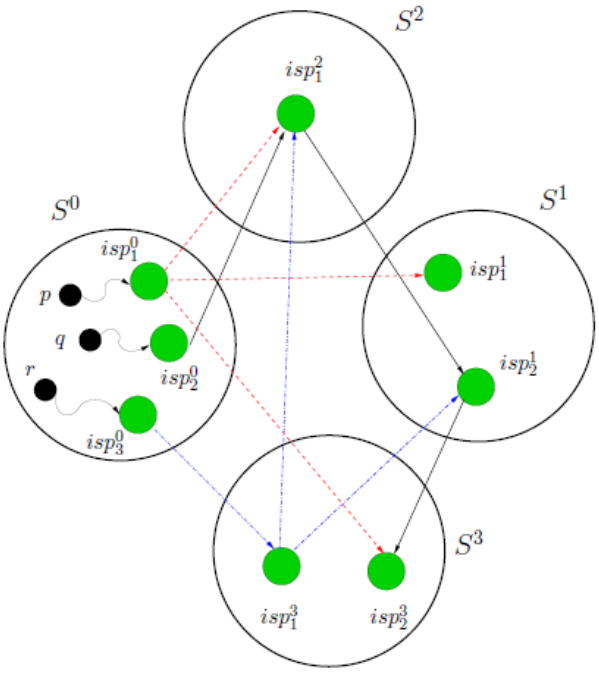


Figura 39. Ejemplo de interconexión de 4 sistemas con el protocolo-IS con enlaces FIFO paralelos. [31]

4.6.4 Interconexión de sistemas causales con múltiples enlaces FIFO

En este apartado veremos cómo interconectar apropiadamente sistemas de orden causal, es decir, que la semántica de entrega global del sistema interconectado sea causal.

En el apartado anterior hemos visto que modificando la arquitectura de los subsistemas FIFO podíamos interconectar sistemas FIFO. Para ello simplemente había que añadir tantos *routers* como fueran necesarios en cada subsistema y asignar a cada proceso del sistema un *router* para que enviase siempre los mensajes por el mismo *router*. De hecho, la arquitectura del protocolo visto en 4.5.4 podía considerarse un caso base de este nuevo protocolo con múltiples enlaces, dado que en 4.5.4 todos los procesos pertenecían a un único grupo y, por tanto, todos los mensajes se enviaban al resto de los subsistemas a través del único *router* que existía en el subsistema.

Sin embargo, el protocolo visto en el apartado anterior no sirve para interconectar sistemas causales manteniendo la semántica de entrega causal. Por ejemplo, vamos a asumir que se van a interconectar dos sistemas causales S^0 y S^1 tal que S^0 tiene 2 *routers* ($router_1^0$ y $router_2^0$) y que S^1 tiene otros 2 *routers* ($router_1^1$ y $router_2^1$). Como ya sabemos, en el caso de los sistemas FIFO, dentro de un mismo sistema los routers envían y reciben mensajes sin necesidad de coordinarse con el resto de *routers* del sistema. Imaginemos que un proceso p de S^0 envía un mensaje m_1, que *router(p)*=$router_1^0$ y que $router_1^0$, una vez recibe m_1, lo envía a $router_1^1$. Imaginemos además un proceso q de S^0 que en cuanto se le entrega m_1 envía un nuevo mensaje m_2 que será transferido desde $router_2^0$ a $router_2^1$. Si a q se le entrega m_1 localmente y acto

seguido envía m_2, puede ocurrir que m_2 se transfiriera por $router_2^0$ antes que m_1 por $router_1^0$. Si además $router_2^1$ recibe m_2 antes de que $router_1^1$ reciba m_1 (o incluso al mismo tiempo) y lo difunde sin esperar la llegada de m_1, es muy probable que a algunos procesos de S^l se les entregue m_2 antes que m_1, lo que rompería la relación de orden causal del sistema.

Como el objetivo de interconectar sistemas es mantener la semántica de entrega, en este caso tenemos dos opciones: trabajar con un único *router* por sistema como en las soluciones anteriores o modificar el protocolo-IS paralelo para que los procesos-IS o *routers* se coordinen de forma que el sistema mantenga la causalidad en la entrega de todos aquellos mensajes que no sean concurrentes.

Así pues, el objetivo en este caso es que todos los enlaces entre dos pares de sistemas se comporten globalmente como si fueran un solo enlace FIFO ordenado, cumpliendo además con el teorema *FIFO forwarding* de [30]. Para ello se tomará como base el protocolo visto en 4.6.3 y se extenderá de la siguiente manera para el caso de la interconexión de dos sistemas:

- En cada sistema S^k uno de los *routers* asumirá el rol de secuenciador (será elegido de forma determinista) y se denominará $router_{seq}^k$. El *router* secuenciador mantendrá un número de secuencia *sn* que se incrementará y se asignará cada vez que se difunda un nuevo mensaje en S^k. Esta idea está basada en el protocolo ABCAST de ISIS donde se garantiza orden total causal sobre el protocolo de entrega causal CBCAST [14]. Sin embargo, en este caso el protocolo de difusión causal local de cada sistema no se verá afectado, ya que el número de secuencia será utilizado únicamente por el protocolo-IS en los *routers*.

- Cuando un proceso *p* de S^k envíe un mensaje *m*, cuando al *router(p)* se le entregue *m*, *router(p)* esperará a que $router_{seq}^k$ le envíe el número de secuencia *sn(m)* que $router_{seq}^k$ le enviará cuando *m* se entregue en $router_{seq}^k$. Cuando lo haya recibido, *router(p)* enviará *m* a través del enlace que lo interconecta con $S^{k'}$.

- En $S^{k'}$ todos los *routers* mantendrán una lista de *recibidos* que acumularán los mensajes recibidos de S^k. Concretamente se añadirá un nuevo elemento a la lista cada vez que se entregue un mensaje enviado por algún *router* de $S^{k'}$.

- Cuando el mensaje *(m,sn(m))* se entregue en el *router* correspondiente de $S^{k'}$, dicho *router* difundirá causalmente *m* al resto de procesos de $S^{k'}$ tan pronto como *sn(m)=recibidos+1*, según la variable *recibidos* del propio *router*.

De esta forma se mantendrá el orden FIFO global entre todos los enlaces entre pares de sistemas pudiendo interconectar, por tanto, sistemas causales[26]. Además, también se

[26] La demostración formal de este hecho puede encontrarse en el teorema 3 de [31].

mejorará el ancho de banda del sistema ya que los mensajes se enviarán en paralelo, a pesar de que en el sistema destinatario deben coordinarse para ser enviados en orden.

En la Figura 40 se muestran las 3 tareas que deberán ejecutar en esta ocasión los *routers* de cada sistema, además de la tarea de secuenciación que ejecutará únicamente el *router* secuenciador.

$Sequence\_out(isp^k_{seq}, m)$ :: task which is activated once $bc\text{-}recv_{isp^k_{seq}}(m)$ is executed begin $sn(m) = ++seq\_num$ send $\langle id(m), sn(m)\rangle$ to $isp(sender(m))$ end	$Receive(isp^k_v, m)$ :: task activated once m is received from any isp^k_j begin $received{+}{+}$ end
$Propagate\_out(isp^k_v, m)$ :: task which is activated once $bc\text{-}recv_{isp^k_v}(m)$ is executed begin if m was sent by a process in $set_w(isp^k_v)$ then wait for receiving $\langle id(m), sn(m)\rangle$ transfer $\langle m, sn(m)\rangle$ to $link_w(isp^k_v)$ end	$Propagate\_in(isp^k_v, \langle m, sn(m)\rangle)$ :: task activated once message $\langle m, sn(m)\rangle$ is received from $S^{\overline{k}}$ begin wait until $sn(m) = received{+}1$ $bc\text{-}send_{isp^k_v}(m)$ end

Figura 40. Protocolo de Interconexión (protocolo-IS) en sistemas causales. [31]

Este protocolo puede extenderse a *n* sistemas siempre y cuando se eviten los ciclos en el sistema global. Es decir, en caso de que un sistema tenga múltiples enlaces, estos enlaces no podrán interconectar sistemas distintos ya que podrían definir ciclos en el sistema global. Los ciclos son problemáticos porque permiten enviar mensajes por dos o más caminos distintos, invalidando la coordinación que se da entre los *routers* de un mismo sistema. Por ejemplo, si queremos interconectar un sistema S^i y un sistema S^j y hay más de un camino para llegar de S^i a S^j, se mantendría un número de secuencia en los *routers* que encaminan por un sistema S^x y otro distinto en los *routers* que encaminan por otro sistema S^y, siendo imposible mantener una relación entre dichos números de secuencia en S^j.

Además, para mantener los números de secuencia correctamente deberá existir un único *path* entre cada par de sistemas. Interconectar los sistemas manteniendo una estructura de árbol nos proporcionará un único camino entre pares de sistemas además de evitar los ciclos, de forma que siempre podremos interconectar apropiadamente los sistemas causales.

4.6.5 Tolerancia a fallos en el protocolo de interconexión de sistemas

La mayor parte de los sistemas tolerantes a fallos, como por ejemplo la arquitectura de margarita, utilizan la entrega uniforme (ver apartado 2.4.2), es decir, un mensaje no se entrega hasta que el sistema de comunicación a grupos sabe que todos los procesos del sistema han recibido dicho mensaje, aunque esta restricción puede relajarse [18] y los mensajes pueden entregarse tan pronto como se reciban los mensajes en un mínimo de procesos, siempre y cuando se cumpla con las restricciones de orden pertinentes.

Además, los procesos emisores de un mensaje no podrán eliminar dicho mensaje hasta que este mensaje no se considere estable.

El protocolo de interconexión de sistemas también puede adaptarse para ser tolerante a fallos siguiendo las siguientes reglas:

- Un *router* $router_v^i$ no reportará la entrega uniforme de un mensaje *m* dentro de su sistema S^i hasta que no obtenga confirmación de los *routers* a los que previamente les ha enviado *m* ($router_w^j$).

- Un mensaje *m* que se ha encaminado en S^j se reportará como uniforme dentro de S^j siguiendo el protocolo de difusión local de S^j. Cuando el *router* receptor de *m* ($router_w^j$) sepa que *m* se ha entregado uniformemente, $router_w^j$ comunicará este hecho a $router_v^i$.

- Si el proceso de interconexión $router_w^j$ falla, otro proceso de aplicación de S^j asumirá el rol de *router*. En este caso se pueden dar dos escenarios diferentes:

    - El *router* fallido pudo difundir todos los mensajes recibidos que fueron enviados por el $router_v^i$. Este caso no es problemático ya que el nuevo *router* podrá reportar los mensajes como uniformes según determine el protocolo de difusión local del sistema.

    - El *router* fallido no pudo difundir todos los mensajes recibidos que fueron enviados por el $router_v^i$. Si esto pasa, el nuevo $router_w^j$ no podrá reportar el estado de estos mensajes porque no sabrá de su existencia. Para solventar esto, tras un periodo de tiempo sin obtener respuesta, $router_v^i$ volverá a enviar los mensajes que no hayan sido reportados al nuevo $router_w^j$. Además, si se están interconectando sistemas causales, el resto de *routers* en S^j se bloquearán hasta que $router_w^j$ recupere los mensajes perdidos y podrán decirle a $router_w^j$ el número de secuencia de dichos mensajes, con lo que $router_w^j$ podrá preguntar a $router_v^i$ por los mensajes asociados a un determinado número de secuencia.

- Si uno de los procesos de interconexión emisores cae ($router_v^i$), un nuevo proceso de aplicación asumirá el rol de *router*. Si alguno de los mensajes encaminados aún no han sido reportados, el nuevo $router_v^i$ lo sabrá, de forma que podrá reenviarlos en caso de que sea necesario. Además, si el *router* caído es un *router* secuenciador, se escogerá de forma determinista un nuevo *router* secuenciador.

Con estas sencillas reglas se evitará la pérdida de mensajes y nos aseguraremos de que cualquier mensaje difundido se entregará en todos los nodos de los sistemas destinatarios, de manera que el sistema podrá recuperarse rápidamente ante un posible fallo.

4.6.6 Número óptimo de *routers*

Hasta ahora hemos asumido que existía un número determinado de procesos de interconexión o *routers* en cada sistema. Sin embargo no se ha especificado cual es el número adecuado de *routers* en un sistema para ser interconectado con otro sistema, que es el objeto de este apartado.

En este análisis necesitaremos definir los siguientes parámetros:

- *n_routers*: número de *routers* existentes en el sistema emisor.
- *ibw*: ancho de banda de la red de los subsistemas (Mbps).
- *idelay*: tiempo de transmisión de un mensaje dentro de un subsistema (segundos).
- *sr*: tasa media de envío en un sistema de difusión (mensajes/segundo).
- *ms*: tamaño del mensaje (Mbits). Permite calcular el ancho de banda en Mbps de la siguiente forma: *sr x ms*. Así pues, $sr < \frac{ibw}{ms}$.
- *ebw:* ancho de banda en Mbps del enlace que interconecta ambos sistemas. Este parámetro es el que introduce el cuello de botella en el sistema, así que se deberá tener en cuenta que $sr < \frac{ebw\ x\ n\_routers}{ms}$.
- *edelay:* tiempo de transmisión de un mensaje entre sistemas (segundos).

Intuitivamente podemos deducir que el rendimiento del protocolo-IS en sistemas FIFO es directamente proporcional al número de enlaces entre dos sistemas. Sin embargo, en el caso de los sistemas causales, el número de enlaces debe calcularse teniendo en cuenta que en el protocolo-IS causal los *routers* deben coordinarse entre ellos para mantener un orden global FIFO en dichos enlaces.

Si tuviésemos un gran sistema causal global en el que se ejecutase un protocolo de *broadcast* fiable, con *n-1* mensajes se podría difundir un mensaje a todos los procesos del sistema, incluyendo en estos mensajes relojes vectoriales que se interpretarían en la fase de entrega. Además, cuando un mensaje es entregado en un *router*, antes de que éste lo reenvíe debe esperar el correspondiente mensaje adicional en el que se incluirá el número de secuencia. La probabilidad de que un *router* sea el *router* secuenciador es $\frac{1}{n\_routers}$, con lo que este mensaje adicional sólo se producirá con una probabilidad de $1 - \frac{1}{n\_routers}$. Una vez se ha obtenido el número de secuencia, el mensaje se encamina a través del enlace correspondiente y se difunde en el sistema destinatario una vez se recibe. Así pues, si no se considera el tiempo de proceso, el tiempo mínimo de transmisión de un mensaje enviado por S^i y recibido por S^j es $\left(3 - \frac{1}{n\_routers}\right) x \left(idelay + \frac{ms}{ibw}\right) + \left(edelay + \frac{ms}{ebw}\right)$, donde la constante *3* representa 2

difusiones en cada sistema para difundir el mensaje en todos los procesos más una adicional en el caso de que haya que enviar el mensaje con el número de secuencia (el peor caso).

Si la carga de trabajo es insignificante, el número óptimo de *routers* sería *n_routers*=1, ya que nos ahorraríamos la fase del número de secuencia. Sin embargo, si se considera una carga de trabajo elevada, y teniendo en cuenta que el cuello de botella es el enlace que interconecta dos sistemas, se puede representar el sistema con un modelo de colas tal y como se indica en [36]. En el mejor de los casos, *ebw* será igual a *ibw*, en cuyo caso el número óptimo de procesos de interconexión será *n_routers*=1. En cualquier otro caso, $n\_routers = \left\lceil \frac{ibw}{ebw} \right\rceil$ [27].

4.6.7 Análisis de rendimiento y conclusiones

El objetivo de los protocolos que se han presentado es paralelizar el envío de mensajes entre dos sistemas para mejorar el ancho de banda del sistema ya que el único enlace que interconecta dos pares de sistemas podría suponer un cuello de botella en un momento dado. Por ejemplo, imaginemos que la red interna de cada sistema es SCI (20 Gbps) o una red Ethernet de 10 Gbps, mientras que la red que interconecta dos sistemas es una WAN con un ancho de banda de 100 Mbps y un retraso (*delay*) aproximado de 50 ms. En una situación como esta los protocolos de interconexión y la red pueden sobrecargarse fácilmente. El beneficio que ofrece el protocolo-IS con múltiples enlaces en sistemas FIFO (apartado 4.6.3) frente al protocolo con un solo enlace (apartado 4.5.4) es proporcional al número de enlaces que existen entre dos sistemas puesto que el tiempo de transmisión no se incrementa. Sin embargo, en el caso del protocolo-IS en sistemas causales el beneficio no es tan directo, puesto que los procesos que ejercen como *routers* deben coordinarse entre ellos para mantener la causalidad. Por este motivo se va a analizar el rendimiento de un sistema causal, comparándolo con los sistemas causales en los que se aplica la arquitectura de margarita [11].

Sabemos que en la arquitectura de margarita con único nivel de jerarquía, para que un mensaje de un grupo llegue a un proceso de otro grupo, el mensaje debe difundirse 3 veces: una en el grupo local del proceso emisor, otra en el grupo de *routers* (servidores) y una última difusión en el grupo local del proceso destinatario.

Vamos a suponer que hemos interconectado 2 sistemas S^0 y S^1, cada uno de ellos con *3n* nodos, tal que *ibw>ebw* y que *idelay<edelay*. Con la arquitectura del protocolo-IS dividiremos los procesos de cada sistema en 3 grupos de *n* procesos, o en 3 grupos locales según la arquitectura de margarita (S^{00}, S^{01}, S^{01} y S^{10}, S^{11}, S^{12}).

Si un proceso *p* difunde un mensaje *m* en S^0 utilizando el protocolo-IS:

- Se generarán 3n-1 mensajes para difundir el mensaje en S^0.

[27] En la sección 8.1 de [31] los autores detallan cómo se ha obtenido esta expresión utilizando el modelo de colas de [36].

- Si $router(p) \neq router_{seq}$, estaremos en el peor caso y deberemos esperar a que $router_{seq}$ genere un número de secuencia de mensaje.

- Se reenviará el mensaje a través del enlace correspondiente.

- Se generarán 3n-1 mensajes para difundir el mensaje en S^1.

En total se generarán *6n-1* mensajes (4 saltos) en el peor de los casos y *6n-2* (3 saltos) en el mejor, más un mensaje en la red de interconexión.

En el caso de la arquitectura de margarita, si un proceso *p* de S^{00} difunde un mensaje *m*:

- Se generarán *n-1* mensajes para difundir *m* en el grupo local S^{00}.

- Como hay 6 grupos locales, se generarán 5 mensajes para difundir *m* dentro del grupo de *routers*. Para estar en igualdad de condiciones, S^{01} y S^{02} tendrán enlaces más rápidos que el resto de grupos.

- Dentro de cada grupo local se necesitarán *n-1* mensajes para difundir *m*. Puesto que hay 5 grupos, se generarán *5n-5*.

En total se están generando *6n-1* mensajes en 3 pasos, pero 3 mensajes se han enviado por enlaces más lentos, mientras que con el protocolo-IS sólo se atraviesa un enlace lento. Por todos estos motivos los autores de [31] afirman que el protocolo-IS es más escalable que la arquitectura de margarita.

En mi opinión, creo que con este ejemplo los autores no han llegado a plasmar claramente cuál es el objetivo de realizar esta división de procesos en la arquitectura de margarita. Para empezar, puede parecer que los grupos dentro de un subsistema del protocolo-IS no son equiparables a los grupos locales de la arquitectura de margarita, sino que más bien habría que considerar el propio subsistema como un grupo local de la arquitectura de margarita. Este hecho podría interpretarse así porque cuando un mensaje se difunde en el protocolo-IS de múltiples enlaces, tanto en el subsistema emisor como en el subsistema destinatario, no se distingue si un proceso está en un grupo o en otro, sino que se difunde a todos los procesos del sistema. Es decir, el sistema de comunicación a grupos utiliza un único grupo para hacer difusiones en este subsistema. Así pues, con el protocolo-IS en un grupo del sistema de comunicación a grupos se están difundiendo mensajes a *3n* procesos, mientras que, con el reparto de procesos que se ha hecho, en un grupo del sistema de comunicación a grupos (grupo local) de la arquitectura de margarita sólo se difunden mensajes a *n* procesos, de manera que un mensaje se difunde mucho antes en un grupo local de la arquitectura de margarita que en un subsistema del protocolo-IS.

Evidentemente en sistemas donde la red interna es muy rápida apenas se notaría la diferencia entre difundir un mensaje entre *3n* procesos o difundirlo entre *n* procesos, pero también hemos visto a lo largo del apartado 4 que reducir el tamaño de la información de control en un sistema causal también es importante para hacer un

sistema más escalable y rápido y, sin duda, los grupos locales de *n* procesos en la arquitectura de margarita trabajan con menos información de control y pueden reportar antes un mensaje como estable.

Por todos estos motivos, podría parecer que lo justo sería agrupar los *3n* procesos de S^0 en un grupo local y los *3n* procesos de S^1 en otro grupo local, formando los *routers* de ambos sistemas el grupo de servidores causales. De esta forma, cuando un proceso *p* de S^0 envíe un mensaje *m*:

- Se generarán *3n-1* mensajes para difundir *m* dentro del grupo local S^0.
- Se difundirá *m* en el conjunto de servidores causales, que en este caso son dos, con lo que únicamente se generaría un mensaje.
- Se generarán *3n-1* mensajes para difundir *m* dentro del grupo local S^1.

Es decir, en total se generarían *6n-2* en las redes internas más un mensaje en la red de interconexión, ganando al protocolo-IS en lo que a número de mensajes se refiere, ya que lo que realmente está pasando es que la arquitectura de margarita, en este caso, es equivalente al protocolo-IS en el mejor de los casos (cuando $router(p)=router_{seq}$), con la ventaja de que en la arquitectura de margarita no hay ni mejor ni peor caso, ya que siempre se va a generar el mismo número de mensajes.

Así pues, cabe aclarar que cuando los autores hacen el reparto de procesos en la arquitectura de margarita lo que se pretende es paralelizar de forma artificial su interconexión. En ocasiones esto será necesario ya que, como ya sabemos, puede ocurrir que los enlaces de interconexión sean mucho más lentos (con mayores retardos y menor ancho de banda) que los de cada red interna y, en tal caso, no habrá otra solución que enviar a través de varios canales. Por tanto, si fuera admisible utilizar un único canal de interconexión entre los dos sistemas, la arquitectura de margarita sería la clara vencedora, pero si es necesario utilizar más de un canal, deberá utilizarse el protocolo-IS con múltiples canales FIFO.

4.7 Conclusiones.

A lo largo del apartado 4 hemos estudiado las diferentes propuestas que se han publicado desde 1995 a 2009 para mejorar la escalabilidad en los sistemas distribuidos de paso de mensajes. Por un lado, hemos visto la evolución de los grandes sistemas causales, donde la identificación de los procesos que ejercían de separadores causales o *routers* permitía reducir la información de control al identificar, en cierto modo, grupos dentro de un mismo sistema [44].

Una vez identificados los grupos de un sistema y los procesos representativos de cada grupo (*routers*) que actuarían de puente entre su grupo local y el resto de grupos, haciendo uso de un sistema de comunicación a grupos se podía reducir aún más la información de control si los mensajes se difundían a todos los miembros de un grupo empleando difusiones uniformes. De esta forma, además de reducir la información de control de una matriz a un vector con tantos elementos como procesos existen en el grupo, los sistemas aumentaban la tolerancia a fallos debido a que, ante un fallo por parte de un proceso, los mensajes perdidos podían recuperarse fácilmente ya que todos los procesos de un mismo grupo lo habían recibido. En este sentido la arquitectura de margarita [11], además de ser tolerante a fallos, era escalable ya que solventaba el problema de realizar difusiones muy pesadas a muchos procesos permitiendo dividir los procesos en nuevas composiciones e incluso aumentando los niveles de jerarquía.

También hemos visto que reducir la información de control y aumentar la escalabilidad del sistema dividiendo el sistema en grupos no debía quebrantar la semántica de entrega del sistema global. Para ello era importante definir protocolos de interconexión de grupos según el tipo de semántica de entrega [30]. Al realizar el estudio de las semánticas de entrega más habituales (FIFO, causal y total), se ha llegado a la conclusión de que, en los sistemas FIFO y en los sistemas causales, los grupos podían interconectarse empleando un protocolo de interconexión no intrusivo, es decir, sin necesidad de modificar el algoritmo de difusión local de dichos grupos. Sin embargo, esto no era posible en los sistemas de orden total, donde la coordinación de los procesos de un grupo con su *router* y la coordinación entre los *routers* de los pares de grupos que se interconectaban era necesaria para mantener el orden total en la entrega de todos los mensajes, siendo necesario emplear un protocolo de interconexión intrusivo.

Por último, ante el auge de los sistemas de *cloud computing* y las redes P2P, se ha analizado la posibilidad de interconectar sistemas ya existentes donde la red dentro de un mismo sistema es mucho más rápida que la red que interconecta los sistemas ([6],[31]). Tal y como ocurría en [30], el objetivo era mantener la misma semántica de entrega en todo el sistema, pero en esta ocasión se exigía mantener los sistemas a interconectar intactos. Por este motivo se ha propuesto un protocolo de interconexión no intrusivo capaz de interconectar sistemas FIFO y sistemas causales, pero no sistemas de orden total. Además, se ha mejorado el protocolo de interconexión poniendo varios enlaces entre pares de sistemas para evitar que, en caso que sea necesario enviar muchos

mensajes entre sistemas, el único enlace que interconecta un par de sistemas en el protocolo de interconexión original se convirtiera en el cuello de botella.

Lo cierto es que hoy en día los sistemas distribuidos son cada vez más grandes y considerar un sistema global como un único grupo sería inviable debido a la gran cantidad de mensajes que deberían circular por la red para difundir un solo mensaje en todo el sistema, atravesando tanto enlaces lentos como enlaces ultrarrápidos. La evolución de todas estas optimizaciones ha conseguido reducir la información de control tanto en mensajes como en procesos, además de reducir el tráfico de mensajes entre grupos/sistemas a un único mensaje por difusión entre pares de sistemas.

5. Conclusiones generales

A lo largo de este trabajo se han analizado diversas propuestas de tipos de soluciones de interconexión de sistemas distribuidos para aumentar la escalabilidad que, con la evolución actual, deberán aplicarse en los tipos de sistemas para los que hayan sido diseñados.

En lo referente a los sistemas distribuidos de memoria compartida, el algoritmo de interconexión no intrusivo que se ha analizado (protocolo-IS) se implementará asumiendo la existencia de un sistema de consistencia de memoria (MCS) con un determinado modelo de consistencia. En concreto, hemos estudiado que, de forma no intrusiva, sólo podrán interconectarse los sistemas que utilicen un modelo de consistencia PRAM, causal o caché. Además, únicamente el modelo caché podrá interconectarse en cualquier tipo de sistema, mientras que en los otros dos modelos sólo será posible la interconexión si y sólo si:

- Sistemas PRAM: podrán interconectarse correctamente si los sistemas son FIFO ordenados.

- Sistemas causales: Podrán interconectarse correctamente si los sistemas son globalmente ordenados.

El protocolo-IS modificará las tareas que deberán ejecutar los *routers* de cada sistema para adaptarse al modelo de consistencia que se esté utilizando en el sistema.

En cuanto a los sistemas distribuidos de paso de mensajes, hemos estudiado por qué deben utilizarse protocolos de interconexión en los sistemas causales. En este tipo de sistemas en particular, hemos visto que la arquitectura de margarita es la arquitectura más apropiada para interconectar los procesos de un gran sistema causal, sobre todo si el ancho de banda y el retraso de los enlaces que unen los *routers* en el grupo de *routers* son aproximados al ancho de banda y al retraso de los enlaces que interconectan los procesos en los grupos locales.

Después se han clasificado los tipos de composiciones de grupos que pueden existir en un sistema y, en base a esta clasificación, se ha estudiado de forma genérica qué orden debía implementar el algoritmo de interconexión en los grupos emisores, los grupos destinatarios y los enlaces que interconectan dichos grupos para interconectar sistemas con orden FIFO, causal o total, con el objetivo de mantener la semántica de entrega global deseada en el sistema. Además, hemos visto que siempre es posible interconectar todos estos sistemas siguiendo las directrices indicadas en la Tabla 6, aunque no siempre de forma no intrusiva. Esto es debido a que, por ejemplo, el protocolo-IS, que es no intrusivo, podía adaptarse para interconectar sistemas FIFO pero nunca podría utilizarse para interconectar sistemas de orden total ya que los sistemas con semántica de entrega de orden total siempre necesitarán la coordinación de sus procesos para mantener dicho orden.

Por último, se ha analizado la posibilidad de utilizar más de un enlace en la interconexión de pares de sistemas interconectados mediante el protocolo-IS en sistemas FIFO y en sistemas causales. Esta arquitectura de interconexión podrá aplicarse en cualquiera de estos tipos de sistemas porque el algoritmo de interconexión en ambos casos continúa siendo no intrusivo. Además, deberá utilizarse siempre que el ancho de banda en los enlaces que interconectan pares de sistemas sea mucho menor que el ancho de banda de los enlaces intrasistema, obteniendo los sistemas causales en estas condiciones mejores resultados que en la arquitectura de margarita.

Así pues, aunque en la bibliografía exista una gran cantidad de publicaciones que aporten otras soluciones para aumentar la escalabilidad en sistemas distribuidos utilizando algoritmos de interconexión, en este estudio bibliográfico se han presentado un conjunto de soluciones que podrán utilizarse como base en el caso de que deseemos implementar nuestro propio algoritmo de interconexión.

Referencias bibliográficas

[1] T. Abdelzaher, A. Shaikh, F. Jahanian, and K. Shin. *RTCAST: Lightweight multicast for real-time process groups*. In Proceedings IEEE Real-Time Technology and Applications Symposium (RTAS '96), pages 250–259, June 1996. También disponible en: Technical report, Dept. of Electrical Engineering and Computer Science, University of Michigan, 1997.

[2] N. Adly, M. Nagi. *Maintaining causal order in large scale distributed systems using a logical hierarchy*. In: Proceedings of the 12th IASTED International Conference on Applied Informatics, 1995, pp. 214-219.

[3] S.V. Adve, K. Gharachorloo. *Shared Memory Consistency Models: A Tutorial*. IEEE Computer 29(12): 66-76 (1996).

[4] M. Ahamad, R. Bazzi, R. John, P. Kohli, G. Neiger. *The power of processor consistency*. Technical Report GIT-CC-92/34, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA, 1992. También disponible en: Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures, 1993, pp. 251-260.

[5] M. Ahamad, G. Neiger, J. Burns, P. Kohli, P. Hutto. *causal memory: definitions, implementation and programming*. Distrib. Comput. 9 (1) (1995) 37–49.

[6] A. Álvarez, S. Arévalo, V. Cholvi, A. Fernández, E. Jiménez. *On the interconnection of message passing systems*. Inf. Process. Lett. 105(6): 249-254 (2007)

[7] H. Attiya, J. Welch, *Sequential consistency versus linearizability*. ACM Trans. Comput. Systems 12 (2) (1994) 91–122.

[8] H. Attiya, R. Friedman. *Limitations of fast consistency conditions for distributed shared memories*. Inform. Process. Lett. 57 (1996) 243-248.

[9] Ö. Babaoglu, A. Schiper. *On group communication in large-scale distributed systems*. ACM SIGOPS Operating Systems Review, 29(1):612–621, Jan. 1995.

[10] Ö. Babaoglu, A. Bartoli, G. Dini. *Enriched view synchrony: A programming paradigm for partitionable asynchronous asynchronous distributed systems*. IEEE Transactions on Computers, 46(6):642–658, June 1997.

[11] R. Baldoni, R. Friedman, R. van Renesse. *The hierarchical daisy architecture for causal delivery*. Proceedings of the 17th International Conference on Distributed Computing Systems.(1997), pp 570-577.

[12] K.P. Birman, T.A. Joseph. *Reliable communication in the presence of failures*. ACM Transactions on Computer Systems, 5(1):47-76, February 1987.

[13] K.P. Birman, R. Cooper, T.A. Joseph, K.K. Kane, F.B. Schmuck. *Isis – A distributed Programming Environment*. June 1990.

[14] K.P. Birman, A. Schiper, P. Stephenson. *Lightweight causal and atomic group multicast*. ACM Trans. Comput. Syst. 9(3) (1991) 272–314.

[15] K.P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., pages 308–309, 1996.

[16] T.D. Chandra, S. Toueg. *Personal communication*. 1992.

[17] J. Chang, N.F. Maxemchuk. *Reliable broadcast protocols*. ACM Transactions on Computer Systems, 2(3):251–273, Aug. 1984.

[18] G. Chockler, I. Keidar, R. Vitenberg. *Group communication specifications: a comprehensive study*. ACM Comput. Surv. 33(4) (2001) 427-469.

[19] V. Cholvi, E. Jiménez and A. Fernández. *Interconnection of Distributed Memory Models*. Journal of Parallel and Distributed Computing 69(3):295-306 March 2009.

[20] F. Cristian. *Reaching agreement on processor group membership in synchronous distributed systems*. Technical Report RJ5964, IBM Research Laboratory, October 1990.

[21] A. Fernández, E. Jiménez, V. Cholvi. *On the interconnection of causal memory systems*. J. Parallel Distrib. Comput. 64(4) (2004): 498-506. También disponible en: PODC 2000: 163-170.

[22] U. Fritzke, Jr., P. Ingels, A. Mostefaoui, M. Raynal. *Fault-tolerant total order multicast to asynchronous groups*. Technical report, Centre National de la Recherche Scientifique, Jan. 1998. To appear in SRDS '98.

[23] K. Gharachorloo, D. Lenoski, J. Laudon, Phillip Gibbons, Anoop Gupta, John Hennessy. *Memory consistency and event ordering in scalable shared-memory multiprocessors*. Computer Architecture News, 18(2):15–26, June 1990.

[24] J.R. Goodman. *Cache consistency and sequential consistency*. Technical Report 61, SCI Committee, March 1989.

[25] V. Hadzilacos, S. Toueg. *Fault-Tolerant Broadcasts and Related Problems*. In Mullender, S., ed.: Distributed Systems. 2nd edn. ACM Press (1993) 97-146.

[26] S. Haldar, K. Vidyasankar. *On specification of read/write shared variables*. J. ACM 54 (6) (2007) 31.

[27] P.W. Hutto, M. Ahamad. *Slow memory: Weakening consistency to enhance concurrency in distributed shared memories*. In Proceedings of the 10th International Conference on Distributed Computing Systems, pages 302–311,May 1990.

[28] E. Jiménez, A. Fernández, V. Cholvi. *A parametrized algorithm that implements sequential, causal, and cache memory consistencies*. J. Syst. Softw. 81 (1) (2008) 120-131. También disponible en: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (Euro PDP 2002), IEEE Computer Society Press, Canary Islands, Spain, 2002.

[29] E. Jiménez, A. Fernández, V. Cholvi. *Decoupled Interconnection of Distributed Memory Models*. OPODIS 2003: 235-246.

[30] S. Johnson, F. Jahanian, J. Shah. *The Inter-group Router Approach to Scalable Group Composition*. 1999.

[31] R. de Juan-Marín, V. Cholvi, E. Jiménez, F.D. Muñoz Escoi. *Parallel Interconection of Broadcast Systems with Multiple FIFO Channels*. 2009.

[32] R. Ladin, B. Liskov, L. Shrira, S. Ghemawat. *Lazy replication: Exploiting the semantics of distributed services*. Technical Report MIT/LCS/TR-84, 1990.

[33] L. Lamport. *Time, clocks, and the ordering of events in a distributed system*. Communications of the ACM, 21(7):558–565,1978.

[34] L. Lamport. *How to make a multiprocessor computer that correctly executes multiprocess programs*. IEEE Transactionson Computers, C-28(9):690–691, September 1979.

[35] L. Lamport. *On interprocess communication: Parts I and II*. Distrib. Comput. 1(2) (1986) 77-101.

[36] E.D. Lazowska, J. Zahorjan, G.S. Graham, K.C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc. (1984)

[37] R.J. Lipton, J.S. Sandberg. *PRAM: A scalable shared memory*. Technical Report CS-TR-180-88, Princeton University, September 1988.

[38] D. Mosberger. *Memory Consistency Models*. Operating Systems Review 27(1):18-26 (1993).

[39] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. *Totem: A fault tolerant multicast group communication system.* Communications of the ACM, 39(4):54–63, Apr. 1996.

[40] L. Peterson, N. Buchholz, R. Schlichting. *Preserving and using context information in interprocess communication*. ACM Transactions on Computer Systems, 7(3), Aug. 1989.

[41] M. Raynal, A. Schiper, S. Toueg. *The causal ordering abstraction and a simple way to implement it*. Information Processing Letters, 39:343–350, 1991.

[42] M. Raynal, M. Ahamad. *Exploiting write semantics in implementing partially replicated causal objects*. In: Proceedings of the Sixth EUROMICRO Conference on Parallel and Distributed Computing, 1998, pp. 157–163.

[43] A. Ricciardi, K. Birman. *Using process groups to implement failure detection in asynchronous environments*. In Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing, pages 341-351. ACM Press, August 1991.

[44] L. Rodrigues, P. Veríssimo. *causal separators for large-scale multicast communication*. In: Intnl. Conf. on Distr. Comp. Syst. (ICDCS), Vancouver, Canada, IEEE-CS Press (May 1995) 83–91.

[45] F.B. Schneider. *Replication management using the state-machine approach*. En S.J. Mullender, editor, Distributed Systems, capítulo 7, p 166-197. Addison-Wesley, 2nd ed, 1993.

[46] R. Schwarz, F. Mattern. *Detecting causal Relations in Distributed Computing: in Search of the Holy Grail*. Distributed Computing, 7(3):149–174, 1994.

[47] M. Singhal, A. Kshemkalyani. *An efficient implementation of vector clocks*. Technical report, Ohio State University, Dep. of Computer and Information Science, Oct. 1990.

Índice de figuras

Índice de tablas