DSIIC
DEPARTAMENTO DE SISTEMAS
INFORMÁTICOS Y COMPUTACIÓN

UNIVERSIDAD
POLITECNICA
DE VALENCIA

MÁSTER EN COMPUTACIÓN PARALELA Y DISTRIBUIDA, UPV

Conexión de sistemas distribuidos dinámicos

Javier Sánchez Ríos
Director: Francisco D. Muñoz Escoí

Septiembre 2013

Ante la complejidad y diversidad de sistemas, resulta interesante que puedan colaborar entre sí para aumentar el rendimiento, de la forma más dinámica posible. Una de las soluciones para este desafío es la interconexión de sistemas. Con ella se evitarán costosos reinicios, y se podrá llegar a un nivel de consistencia aceptable.

Índice

Introducción 4
Tipos de consistencia 5
Tipos de difusión 6
Reliable broadcast 6
FIFO Broadcast 6
Causal Broadcast 6
Atomic Broadcast 7
FIFO Atomic Broadcast 7
Casual Atomic Broadcast 7
Timed Broadcast 7
Uniform Broadcast 8
Modelos de consistencia 8
Modelos de consistencia uniformes 10
Atomic consistency 11
Implementación 11
Sequential consistency 11
Implementación 11
Causal consistency 11
Implementación 12
Pipelined RAM (PRAM) 12
Implementación 12
Cache Consistency 12
Implementación 12
Processor Consistency 13
Implementación 13
Slow Memory 13
Implementación 13
Notación 14
Vista 14
Cola de admisión 14
Lista 14
Política de fallos 15
Recuperación del fallo 16
Descripción del protocolo 17
Cosas necesarias 18
Problemática de ejecutar código en diferentes sistemas 18
Protocolos abiertos (Ver Apéndice I) 19
Problemas de concurrencia 21
Solicitud de ingreso a un grupo 21
Versión secuencial de ingreso a un grupo 22
Situación inicial 24
Ingreso de un grupo a nuestro sistema 24
Cosas necesarias 24
Notación 24
Absorción 26

Problemas de la absorción 27
Consistencia global 27
Posibles uniones 28
No Fast <-> No Fast 29
Fast <-> No Fast 30
Fast <-> Fast 30
Compatibilidad con otros protocolos 32
Solicitud de baja de un proceso / o de un grupo de procesos 34
Cambio de consistencia 34
Algoritmos intrusivos 35
Cambio de consistencia con otros protocolos 36
Problemas 37
Saturación 37
Envío masivo de solicitudes de unión al grupo G 37
Fallo del proceso puente 37
Conflicto de variables 38
Pruebas 38
Mantenimiento de la consistencia 39
Al agregar un solo proceso 39
Compatibilidad con otros protocolos 39
Conclusiones 40
Apéndice I Protocolo Abierto 41
Bibliografía 45

Introducción

El objetivo de este documento, es que un sistema de procesos pueda crecer, es decir, aumentar el número de procesos. Así si al sistema le faltan procesos para poder llevar a cabo sus tareas, se pueda fácilmente añadirle más procesos.

Además la consistencia del grupo resultante ha de ser idéntica al grupo previo para mantener la integridad de los procesos del grupo, ya que estos procesos están diseñados para una determinada consistencia.

El objetivo de este documento es crear sistemas dinámicos particionables, es decir el grupo resultante será la unión de sistemas donde cada sistema puede ser un único proceso o un grupo de procesos. En caso de que sea un grupo de procesos, se contará con un proceso que hará de puente entre el grupo y el resto del sistema. Esta idea no es nueva, ya que cuando una empresa se quiere conectar a la red de Internet, lo hace mediante un router (proceso puente).

Cada subgrupo podrá tener una consistencia diferente a la del grupo general. Nuestro sistema sera lo más dinámico posible incluyendo que pueda cambiar de consistencia.

Aunque ya hay formas de hacer esto, por ejemplo el protocolo propuesto por [**JFC08]**, este trabajo llegará un paso más lejos (ya que en los resultados anteriores solo se mantiene la consistencia causal) pudiendo gestionar un buen número de modelos de consistencia.

La idea subyacente es la de la caja negra; es decir cada proceso o subgrupo puede ser visto como una caja negra por el sistema dinámico. Esto nos da ciertas características:

- Simplifica el protocolo : Únicamente nos preocuparemos de la interconexión, no del comportamiento interno de cada componente.
- División del sistema en grupos (o subsistemas): Entendemos por un sistema particionadoun sistema dividido en varios subsistemas, cada uno de ellos trabajando de forma independiente. Así un trabajo se distribuye en varios grupos. La ventaja de que los sistemas estén organizados en subgrupos (con posibilidad de comunicación entre ellos cuando las situaciones de particionado se resuelvan), es que reducimos la carga de envío de mensajes. De esta manera la dividimos (repartiendo la carga entre cada subsistema) pero a costa de relajar algunas restricciones, como se verá más adelante.

  Con ello se facilita una base para tolerar las situaciones de particionado [CKV01, GL02]. Cada subsistema tiene su protocolo propio para realizar las difusiones y, en caso de que haya particiones, seguirá utilizando ese protocolo. Cuando el subsistema logre reconectarse con el resto del sistema distribuido, le resultará muy sencillo obtener y aplicar las modificaciones que se hayan generado durante su intervalo de desconexión.
- Consistencia independiente: Como se ha explicado en la simplificación, no nos ocuparemos del comportamiento interno de cada componente. Esto incluye

también la consistencia de cada componente. Solo nos encargaremos de la unión de los componentes. Así el puente de un grupo tendrá que ejecutar dos algoritmos: por un lado el de la consistencia del grupo, y por otro el de la consistencia del subgrupo.

En concreto, el objetivo principal de este documento es dar una serie de pinceladas para crear un sistema dinámico basado en interconexión de subsistemas. Para ello se ha partido de lo estudiado en varias asignaturas del Máster en las que se presentaron los principios a seguir para desarrollar aplicaciones altamente disponibles. Sin embargo, muchas ideas presentadas en este trabajo extienden esos principios básicos. El autor ha tenido que revisar un buen número de artículos científicos relacionados con la gestión de consistencia en sistemas distribuidos, generando así el presente TFM que supone un avance significativo sobre los conocimientos que debía adquirir un estudiante de esta titulación.

Tipos de consistencia

Cuando se está trabajando de forma distribuida hay que garantizar una integridad o consistencia en los datos. Para que se pueda replicar de forma fiable y no dé lugar a fallos en la ejecución del programa.

Además también hay que tener en cuenta que esto tiene un costo. Por lo tanto según el objetivo de nuestra aplicación podremos escoger un tipo de consistencia u otro

Nuestro sistema global será la unión de varios sistemas cada uno con sus tipos o modelos (se verá en el siguiente capítulo) de consistencia.

Una consistencia B será más débil que A, si y solo si B tiene menos restricciones que A. Hay que tener en cuenta que esto tiene 2 características principales:

- **Ventaja:** Si tiene menos restricciones , más fácil será de implementar y de ejecutarse.
- **Inconveniente:** Si tiene menos restricciones, más fácil es que se pueda acceder a algún dato no actualizado.

Pero como hemos comentado antes, a la hora de escoger un tipo u otro, hay que tener en cuenta para qué se va a usar.

Para lograr un cierto modelo de consistencia se necesitan mecanismos para propagar los cambios de las variables. Uno de estos mecanismos son los algoritmos de difusión.

Según [**HT94**], podemos contar con varios tipos de difusión que se describen seguidamente. Estos tipos de difusión son necesarios para realizar los modelos de consistencia. Tanto los tipos, como los modelos, y cómo se construyen a partir de los tipos, se explican a continuación.

Tipos de difusión

Reliable broadcast

Es la más débil de todas y además todas la demás se basan en ésta. Esto es, las demás han de cumplir todas las propiedades de ésta:

- **Terminación:** Cada proceso correcto entrega los mensajes.
- **Acuerdo:** Todos los procesos reciben los mismos mensajes.
- **Validez:** Todos los mensajes se enviarán a todos los procesos incluso al emisor.
- **Integridad:** Cada proceso recibe solo una vez el mensaje m, si y solo si ha sido difundido por su emisor.

FIFO Broadcast

Debido a que en Reliable Broadcast los mensajes se pueden entregar sin ningún orden , vamos a añadirles un orden ya que puede ser determinante para la ejecución del programa distribuido.

El ejemplo más ilustrativo es el que establece [**HT94].** La cancelación de un billete de avión solo se puede realizar si se ha realizado la reserva previa. Es decir, ningún terminal recibirá la cancelación sin haber recibido previamente la reserva.
FIFO Broadcast cumple todas las propiedades de Reliable Broadcast, pero además añade :

- Si un proceso p difunde un mensaje m1 antes que m2, entonces a ningún proceso correcto se le entrega m2 sin haber entregado previamente m1.

Causal Broadcast

A veces el FIFO no es suficiente para mantener la integridad , por ejemplo puede darse el siguiente caso según [**HT94].** Supongamos un servicio de noticias, el usuario A difunde una noticia, B recibe la noticia y difunde otra. Ningún proceso debería recibir la noticia difundida por el usuario B si no ha recibido previamente la difundida por A.

Causal Broadcast es más estricto que FIFO Broadcast , e impone el siguiente orden:

- Si un proceso p difunde un mensaje m1 que precede causalmente a m2, entonces a ningún proceso correcto se le entrega m2 sin haber entregado previamente m1.

De hecho Causal Broadcast se puede implementar como la unión de FIFO Broadcast y Local Broadcast donde :

- Local Broadcast: 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.

Atomic Broadcast

Si los mensajes no dependen causalmente pueden entregarse en diferente orden, entonces Causal Broadcast no puede ser suficiente. En ese caso se define la siguiente propiedad para establecer Atomic Broadcast:

- Si dos procesos correctos llamados p y q reciben los mensajes m1 y m2 respectivamente, entonces p recibe m1 antes que m2 si y solo si a q se le entrega m1 antes que m2.

Esta propiedad junto con la validez garantiza que todos los mensajes serán entregados en la misma secuencia.

FIFO Atomic Broadcast

Atomic Broadcast puede ser insuficiente ya que según [**HT94**] se puede dar el siguiente caso :

Un proceso tiene un fallo temporal durante la difusión de m, pero puede difundir m´ después del fallo sin haber finalizado la difusión de m. Esto se evita añadiendo la restricción FIFO a Atomic Broadcast.

Casual Atomic Broadcast

El problema de FIFO Atomic Broadcast es el mismo que FIFO Broadcast: se puede enviar mensajes que dependen causalmente en orden incorrecto. Por ejemplo, según [**HT94**]:

> El usuario A difunde un mensaje m1 y falla. B quien es el único que entrega el mensaje difunde el mensaje m2 y también falla. Entonces C podría recibir m2 sin haber recibido el mensaje m1.

Así pues le añadimos el orden Causal a Atomic Broadcast, para hacerlo más fuerte.

Timed Broadcast

A todos estos tipos de difusión se les puede añadir una ventana temporal:

- Si un mensaje no es entregado en un determinado tiempo se descartará.

Uniform Broadcast

Lo malo de lo anteriormente expuesto es que no tienen en cuenta los procesos que fallan, dando lugar a fallos en el orden de la entrega del mensaje.

Entonces podemos hablar de dos propiedades para garantizar el correcto orden:

- **Acuerdo uniforme:** Si un mensaje es entregado a un proceso correcto o fallido entonces se le entrega a los demás procesos correctos que son los únicos que están activos.
- **Integridad uniforme:** Para cualquier mensaje m y cualquier proceso válido o fallido recibe m solo una vez si y solo si ha sido difundido por algún emisor.

Nótese que en la primera propiedad solo se tienen en cuenta a los procesos activos ya que son los únicos capaces de entregar y recibir mensajes.

Además estas propiedades se pueden mezclar con otras propiedades. Por ejemplo:

- **Uniform FIFO Order:** Si un proceso p difunde un mensaje m1 antes de difundir m2, a ningún proceso (correcto o fallido) se le entrega m2 antes de haber entregado m1.
- **Uniform Local Order:** Si un proceso difunde m1 y un proceso recibe m1 antes de difundir m2, a ningún proceso (correcto o fallido) se le entrega m2 antes de haber entregado m1.
- **Uniform Causal Order:** Si un proceso p difunde un mensaje m1 que precede causalmente a m2, a ningún proceso (correcto o fallido) se le entrega m2 antes de haber entregado m1.

Modelos de consistencia

Según [**Mos93**], un modelo de consistencia es el entorno de ejecución de un determinado sistema. Además según [**SN04]**, los modelos de consistencia se desarrollan para especificar qué valor ha de ser devuelto por una lectura dada en una determinada secuencia de operaciones.

El propósito de los modelos de consistencia es asegurar ciertas propiedades que pueden ser intuitivas para el programador, y por lo tanto escribir un programa correcto

Este se comportará correctamente si se comporta como si se ejecutara un único proceso. Para eso habrá que tener en cuenta si necesita sincronización o si se recuperará ante determinados fallos.
El problema de esto es que es muy estricto y no deja a los compiladores realizar optimizaciones. Por tanto será conveniente usar modelos más relajados.

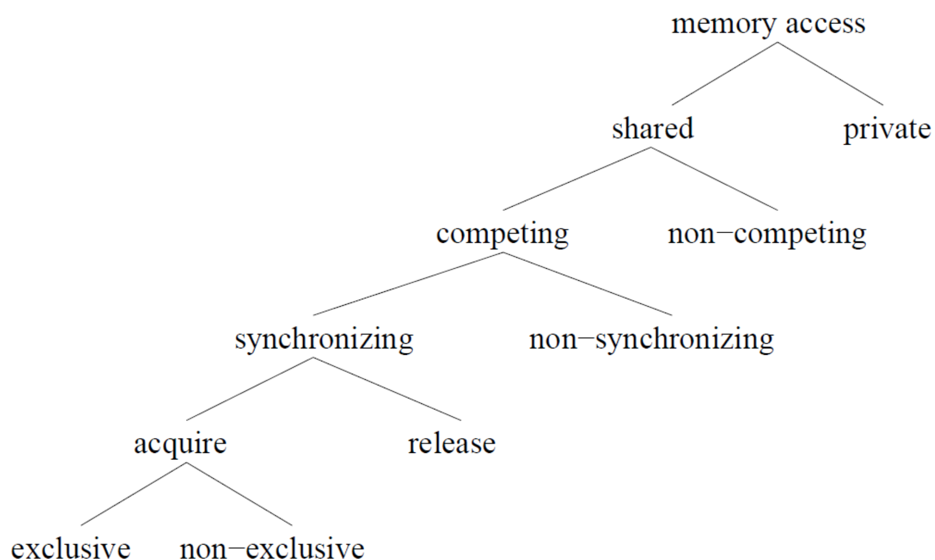
No usaremos modelos formales, solo descriptivos. La descripción formal puede ser revisada en [**ABJ_ 92**] y en [**GLL+90**].

El modelo más débil de consistencia es el que retorna un valor por un acceso de lectura después de un acceso de escritura. Así pues, este sistema de memoria puede escoger cualquier valor previamente escrito.

Implementar modelos débiles puede ser tan complejo como sea necesario para mantener la restricción del orden de ejecución de dos accesos simultáneos.

Los modelos de consistencia se caracterizan según cómo gestionen diferentes aspectos de los accesos a memoria.

- Localización de acceso; esto es, la dirección de memoria a la que se accede.
- Tipo de acceso (lectura, escritura o ambos).
- Valor transmitido en el acceso.
- Si hay causalidad.
- Categoría de accesos.



Las categorías de accesos pueden ser:

- **Acceso de memoria**
    - **Privado:** Solo el proceso dueño puede acceder. Por tanto, estos accesos no generan conflictos.
    - **Compartido:** Pueden acceder varios procesos.
        - **No competitivo:** Todos son de lectura.
        - **Competitivo:** Al menos uno es de escritura.
            - **No sincronizado:** No hay orden.
            - **Sincronizado:** Se establece un orden.
                - **Liberación:** De escritura.
                - **Adquisición:** De lectura.
                    - **No exclusivo:** No se espera a ninguna escritura (liberación).
                    - **Exclusivo:** Se espera a la liberación.

Los modelos de consistencia pueden ser de dos tipos:

- **Híbridos:** Distinguen accesos de memoria. Este tipo de modelos escapan al propósito de este trabajo.

Las motivaciones están descritas en [**AH90**].

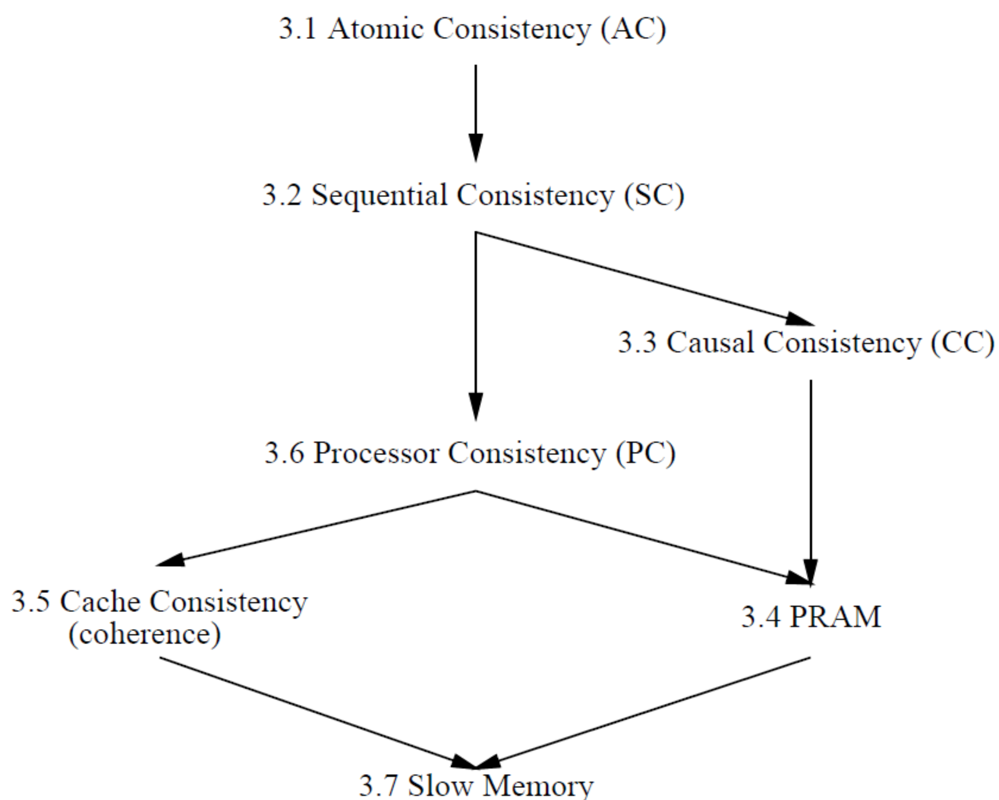
- **Uniformes:** No distinguen.

Modelos de consistencia uniformes

Según [**Mos93**] hay estos tipos de modelos de consistencia:

- **Atomic consistency**
- **Causal consistency**
- **Pipelined RAM (PRAM)**
- **Cache consistency**
- **Processor consistency**
- **Slow Memory**

Donde:



Cada flecha indica que el origen es más estricto que el destino. Por ejemplo, Atomic Consistency es más estricto que Sequential Consistency.

Atomic consistency

Este modelo es el más estricto. En este modelo las operaciones tanto de lectura como de escritura tienen efecto durante un determinado tiempo. Puede haber dos tipos:

- **Estática:** Las operaciones de lectura se realizan al principio de ciclo y las de escritura al final.
- **Dinámica:** Se pueden realizar las operaciones en cualquier momento del intervalo siempre que su resultado sea idéntico a cualquier ejecución secuencial.

Implementación

Para implementar este modelo podemos usar el tipo **FIFO Timed Atomic Broadcast** ya que con la ventana de tiempo nos aseguramos que la propagación no haga efecto durante un determinado tiempo.

Además debe ser FIFO, ya que así nos aseguramos un cierto orden. El problema de esto es que necesitamos mecanismos de sincronización sobre todo en las lecturas haciendo pesada la implementación.

Sequential consistency

Este modelo fue definido por Lamport en 1979 [**Lam79**]**:**

... el resultado de una ejecución es el mismo que si las operaciones de todos los procesos fueran ejecutadas en cualquier orden secuencial, y las operaciones de cada proceso individual aparecieran en la secuencia en el orden especificado por su programa.

Implementación

Aquí se usará **FIFO Atomic Broadcast,** ya que así aseguramos un orden total de los procesos.

Pero además necesitamos FIFO para garantizar el orden en que se envían los mensajes en cada proceso.

Causal consistency

Hutto y Ahamad [**HA90**] introducen consistencia causal, aunque fue Lamport [**Lam78]** el que introduce el concepto de causalidad potencial. Esta noción puede ser aplicada en sistemas de memoria para interpretar una escritura como un envío de mensaje y una lectura como una recepción de mensaje.

Un sistema de memoria causal mantiene el acuerdo sobre el orden de la causalidad de los eventos.

Implementación

Para implementar este modelo se puede usar **Causal Broadcast.**

Pipelined RAM (PRAM)

Lipton y Sandberg [**LS88**] definieron Pipelined RAM. Consideremos muchos procesos donde cada proceso tiene una copia de la memoria compartida. Cada acceso debe ser independiente del instante de acceso de otros procesos.

Se propone:

- Ante una determinada lectura, simplemente se devuelva el valor memorizado en la copia local de la memoria compartida.
- Ante una determinada escritura, actualizará su valor y difundirá el nuevo valor

Es equivalente a decir que todos los procesos ven las escrituras de un mismo proceso en el mismo orden mientras las generadas por procesos distintos podrán ser vistas en diferente orden.

Implementación

Se puede usar el tipo FIFO Broadcast , ya que se propagan en el mismo orden

Cache Consistency

Cache consistency [**Goo89**] y Coherence consistency [**GLL+90**] son sinónimas.

Es un modelo más débil que causal consistency. En este modelo:

- Todos los procesos están de acuerdo en un secuencia de accesos sobre cada una de las variables.
- Coherence solo requiere que los accesos sean secuencialmente consistentes (por variable).

Nótese que sequential consistency implica coherence consistency pero no al revés. Así pues coherence es más débil que secuencial.

Implementación

Similar al **Atomic Broadcast** pero con un secuenciador para cada variable. Cada proceso conoce quien es el secuenciador de cada variable, y le manda la petición de escritura al secuenciador, que luego propaga mediante **FIFO Broadcast.**

Processor Consistency

Goodman propone este modelo en [**Goo89**] aunque de manera informal. Para una definición formal puede referirse a [**ABJ+92**]. Sin embargo otros autores como el grupo DASH en [**GLL+90**] dan su propia definición. Aquí usaremos la de Goodman.

Se define como un modelo intermedio entre sequential consistency y cache consistency. Puede ser interpretado como una combinación de PRAM y cache consistency.

Es fácil pensar que processor consistency es un modelo de consistencia que requiere de un historial para ser como cache consistency y PRAM simultáneamente.

Este modelo se basa en:

- Los procesos deben de estar de acuerdo en el orden de las escrituras para cada proceso pero no para el orden de diferentes escrituras si son en diferentes variables.

Implementación

Igual que el **Cache consistency** pero con **FIFO Broadcast** para comunicarse con el secuenciador.

Slow Memory

Este modelo es más débil que PRAM según [**HA90**]. Se ha escogido este nombre ya que las escrituras se propagan muy lentamente por el sistema.

Requiere que todos los procesos estén de acuerdo en el orden en que se ven las escrituras para cada posición en un proceso.

Implementación

Este modelo se puede implementar mediante **Timed Broadcast** con una ventana muy grande para asegurarnos su comportamiento lento.

Notación

Para explicar nuestro protocolo vamos a usar una serie de herramientas, que se presentan a continuación.

Vista

Llamaremos vista de un grupo a la lista de procesos de un grupo en un determinado instante.

$Vg(i)=\{i,\{p,q,r\}\}$, denota la vista del grupo g en el instante i, y está formada por los procesos p,q,r.

Llamaremos Vista del proceso P sobre el grupo a la lista de procesos que ve el proceso P, en un determinado instante i.

$Wp(i)=\{i,\{p,q,r\}\}$, se denota la vista de P del grupo g en el instante i.

Es obvio pensar que $Wp(i) \leq WG(i) \Leftrightarrow p \, \varepsilon \, G$.

Cola de admisión

Llamaremos cola de admisión a una cola de procesos capaz de admitir a nuevos procesos. Llamaremos proceso bloqueante, al proceso que está en la cabeza. Sólo éste será capaz de aceptar a nuevos procesos y bloquear el sistema. En caso de que falle, el proceso bloqueante será el siguiente en la cola.

Con $Cg=\{i,p,\{q\}\}$ se denota la cola de admisión formada por los procesos q, con proceso bloqueante p en el instante i.

Es obvio pensar que $Cg \leq G$

Lista

Llamaremos Lista del proceso P a la lista de variables que comparte con el Grupo, es decir que son visibles desde otros procesos del grupo.

$Lp=\{i,\{a,b,c\}\}$ Lista de variables públicas que hay en P en el instante i.

Política de fallos

En nuestro protocolo de unión puede pasar que en medio de la unión un proceso falle. A veces podremos contar con el enmascaramiento del fallo como dice [**Cri91**], ya que la unión al grupo se produce en distinto nivel.

Para entenderlo tendremos que distinguir diferentes niveles

- Nivel de usuario: Es el nivel más alto que se puede encontrar. El usuario ejecuta un programa distribuido y éste internamente o por orden de él necesita ingresar un proceso a un determinado grupo.
- Nivel de programa: Son los programas los que se ejecutan de forma distribuida. Estos programas deciden por sí mismos o por orden de los usuarios si conectarse a un grupo o no. Cuando decimos programas nos referimos a todos los programas necesarios: el servicio de unión, los servicios de cada proceso, etc.
- Nivel de grupo: Es donde se ejecuta el grupo de procesos como grupo. Esto es, cada proceso forma parte de un grupo.

Además hay que distinguir entre dos tipos de procesos:

- Crítico: Indispensable para la correcta ejecución del grupo. Puede ser un proceso o un subgrupo de procesos. Esto solo se puede enmascarar en caso de que sea un subgrupo.
- Auxiliar: Un proceso que se une al grupo pero no es indispensable para el correcto funcionamiento del grupo. Son reemplazables.

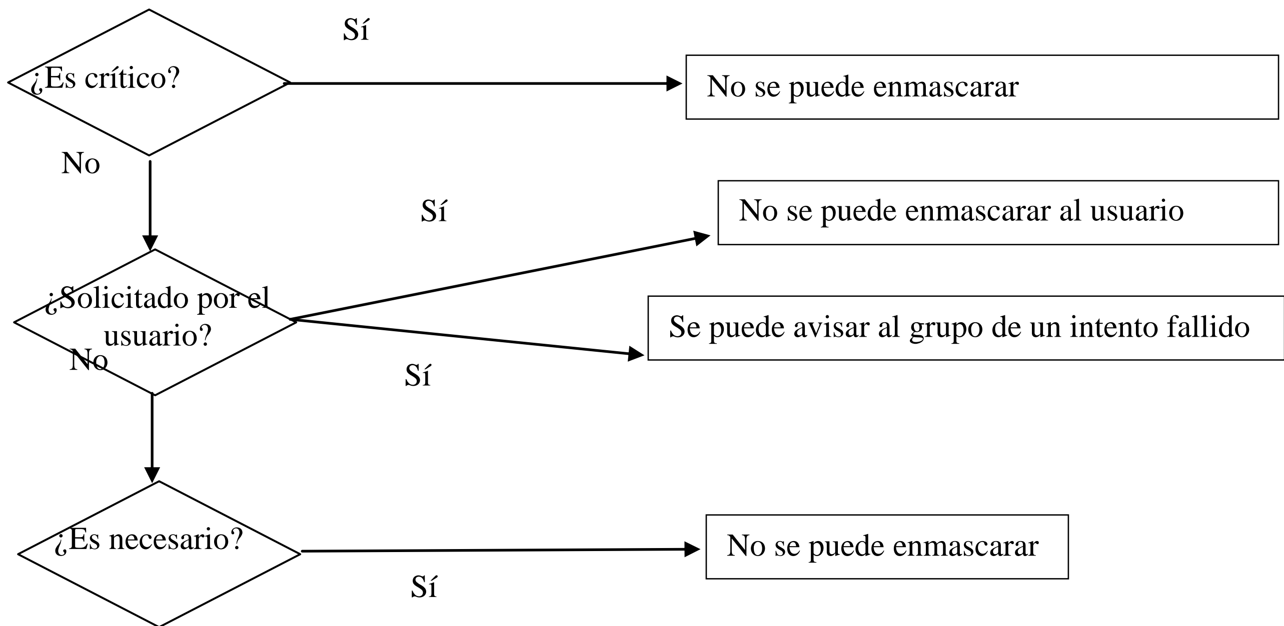
Por ejemplo, supongamos un grid donde hay procesos que se encargarán de sincronizar los datos recibidos. Estos serán procesos críticos. Otros se unirán para realizar los cálculos. Estos son auxiliares.

También puede darse el caso de que el proceso se una por iniciativa propia o por orden de un usuario.

Desde el punto del proceso se puede ver que:

- La unión puede ser indispensable para la correcta ejecución del programa.
- O que el proceso pueda seguir sin unirse al grupo.

En general el enmascaramiento puede seguir este esquema



Recuperación del fallo

Otra problemática a la que nos enfrentamos es que un proceso dentro del grupo G ha fallado. Para unirse al grupo bastará con que ejecute el algoritmo de adhesión al grupo, pero ¿en qué estado?

Aquí hay dos posibilidades según [**Cri91**]

- Amnesia total: El proceso se une con el estado inicial.
- Amnesia parcial: El proceso se une con el estado en que falló. Aquí habría que imponerle algunas restricciones de corrección.

En ambos casos, el proceso tendrá que preguntar al grupo cual es el estado actual y actualizarse. Esto puede realizarse de esta manera:

- Actualizar todo el estado: Cuando se recupera y se une al grupo, todo el estado del grupo es volcado al proceso.
- Actualizar según lo necesite: Cuando acceda a una posición de memoria ya sea de lectura o de escritura pregunta al grupo el estado.

Esto también puede servir como estado inicial del proceso cuando se une a G.

Descripción del protocolo

Aquí se va a describir el protocolo que vamos a proponer. La idea básica de este protocolo está inspirada en las interfaces del paradigma de programación orientada a objetos.

Las interfaces en este paradigma de programación ofrecen un buen sistema de adaptabilidad de los programas.

Lo bueno que tienen las interfaces es que nos permiten cambiar el código de ejecución sin cambiar la forma de conectarse con los demás componentes.

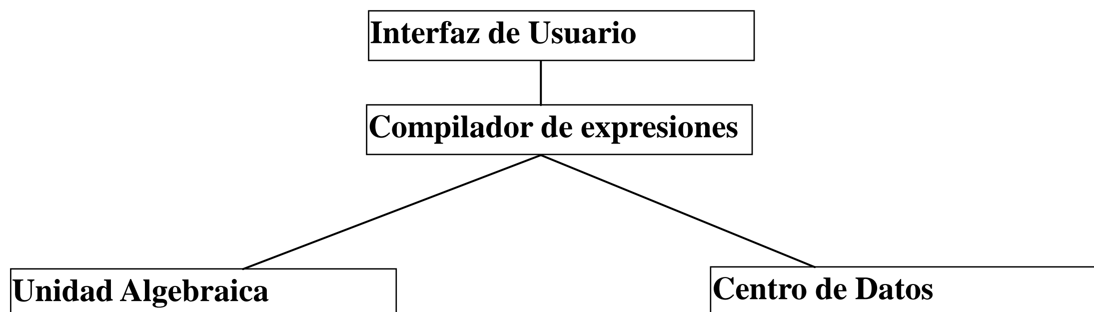
Esto también funciona en el mundo de cada día. Por ejemplo un coche de gasolina y un coche de gasoil; ambos funcionan de forma diferente, pero la interfaz es la misma (llave de contacto, marchas), de tal manera que si se sabe conducir en un coche de gasoil también se sabe conducir en uno de gasolina, y viceversa.

Desde el punto de vista informático, se puede ilustrar con este ejemplo:

Supongamos que tenemos un proceso calculadora, tal que tiene estas operaciones básicas:

- LeerDato (x)
- EscribirDato (x)
- Suma(x,y)
- Resta(x,y)
- Multiplicación(x,y)
- División(x,y)

El sistema está compuesto según el diagrama:



Como se ve el sistema se compone de una interfaz de usuario, de un compilador que transforma las expresiones en operaciones básicas que entiendan los subsistemas de la unidad algebraica y centro de datos.

Para entenderse con el compilador, lo único que tienen que hacer es implementar una determinada interfaz:

Interfaz Algebraica
Suma(x,y) Resta(x,y) Multiplicación(x,y) División(x,y)

Interfaz de Datos
LeerDato (x) EscribirDato (x)

Donde el subsistema de la unidad algebraica implementa la interfaz algebraica y el subsistema de centro de datos implementa la interfaz de datos. Si queremos en cualquier momento cambiar por ejemplo que el sistema pase de acceder a los datos desde un hashtable a desde un vector, bastará con cambiar el subsistema de centro de datos. El único requisito es que implemente la interfaz de datos para asegurar la comunicación.

Otra gran ventaja de las interfaces es la delegación, es decir el compilador delega las operaciones en los subsistemas correspondientes.

El protocolo que emplearemos nosotros solo se encargará de unir procesos y delegará en el sistema resultante el mantenimiento de la consistencia.

Cosas necesarias

Para mantener la consistencia, mejor dicho, para delegar su gestión tendremos que emplear diversos sistemas que garanticen que la delegación se realiza de forma correcta.

Nuestro protocolo no solo tendrá que adicionar procesos, sino también mecanismos para garantizar la delegación.

Todos los procesos tienen código en común que procesan para mantener la consistencia. Esa será la base de nuestro protocolo. Cuando un proceso se una al grupo tendrá que ejecutar este código. Eso hay dos formas de conseguirlo:

- Capacidad de enviar código a ejecutar.
- En caso de que lo anterior no sea posible, ser capaces de crear procesos virtuales que hagan de interfaz entre el grupo G y el nuevo proceso.

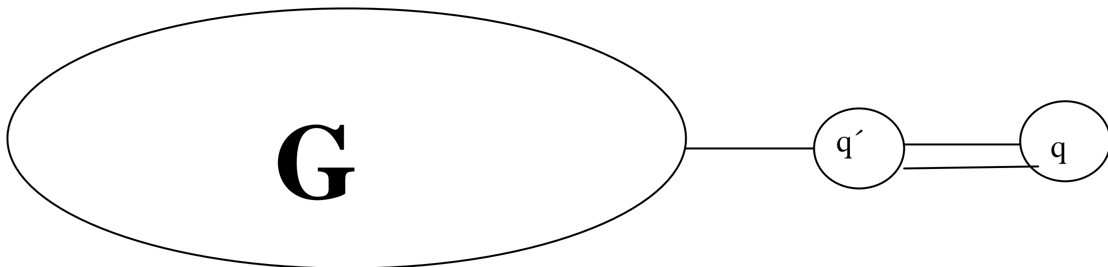
Problemática de ejecutar código en diferentes sistemas

Nuestro protocolo se apoya en poder enviar código que se pueda ejecutar en otros procesos para poder mantener la consistencia. En caso de que no se pueda se creará un proceso virtual.

Para poder ejecutar nuestro código en diferentes procesos, podemos apoyarnos en el uso de Scripts o lenguajes multiplataforma como Java.

Cuando un proceso se una a nuestro protocolo, se le tendrá que enviar al proceso si es capaz de ejecutar el script, o de lo contrario enviarle por qué puerto se comunicará para la escritura, y por qué puerto se comunicará para la lectura. Así nuestro grupo de procesos creará un proceso virtual que se comunicará con el nuevo proceso.

La idea del proceso virtual viene a ser



Donde cada vez que q quiera escribir una variable o consultar una variable, pasará primero por q´, siendo este la comunicación con G.

Realmente solo hará falta las operaciones para q´:

- Difundir(m)
- Entregar(m)

Lo cual tampoco es ninguna sobrecarga para la máquina donde se ejecute. Incluso todos los procesos virtuales pueden ejecutarse en una única máquina, y todos los demás procesos en máquinas diferentes.

Como se puede comprobar G no ve a q, solo ve a q´, luego la idea que subyace es la de la delegación. Se delega en q´ el paso de información de G a q y viceversa.

Protocolos abiertos (Ver Apéndice I)

Para facilitar la problemática de ejecución multiplataforma, vamos a denotar el concepto de protocolo abierto.

Definiremos como protocolo abierto aquel que no está predeterminado, si acaso solo por el servidor. Así el cliente solo pregunta. El servidor no es más que un sitio conocido donde los clientes van solicitando peticiones (Ver Apéndice I)

Para eso usaremos un compilador de expresiones dinámicas, ya que ofrece varias ventajas:

- Es la manera óptima de tratar expresiones que se ejecutan en caliente.
- Da mucha flexibilidad al sistema por ser la clave de los protocolos abiertos.
- Son de fácil implementación.
- Además es totalmente compatible con la reflexión de los lenguajes modernos.

Vamos a ver un pequeño ejemplo.

Supongamos que hay una serie de cálculos, y los clientes se van anotando. A priori los clientes no saben qué tienen que hacer.

- El cliente solicita una petición.
- El servidor la acepta.
- El servidor envía el código que ha de ejecutar el cliente que sería por ejemplo
  - X=5;Y=7;Z=suma(X,Y);escribir(Z);desconectar();
- Por su parte el servidor se pone a ejecutar el código análogo pero para servidor que sería
  - leer(Z);desconectar();actualizar(Z);

Existe una limitación en este protocolo, y es que el cliente no comparte las mismas bibliotecas que el servidor o que las tenga desactualizadas. Para evitar eso añadiremos al protocolo la actualización.

- El cliente solicita una petición.
- El servidor la acepta.
- El cliente envía su número de versión.
- El servidor la comprueba, y en caso de que no esté actualizada, envía la actualización.
- El cliente se actualiza.
- El servidor envía el código que ha de ejecutar el cliente que sería por ejemplo
  - X=5;Y=7;Z=suma(X,Y);escribir(Z);desconectar();
- Por su parte el servidor se pone a ejecutar el código análogo pero para servidor

que sería

- ◦ leer(Z);desconectar();actualizar(Z);

Problemas de concurrencia

Se puede dar el caso que dos o más procesos quieran unirse a nuestro grupo al mismo tiempo. Eso crearía un problema de concurrencia ya que puede darse:

Sean p, q dos procesos del grupo G, y s, t dos procesos que quieren unirse a G. El proceso s solicita la admisión a p, y t solicita la admisión a q. Entonces cuando se propaga el cambio p propagará G U {s} sin tener en cuenta a t, y q propagará G U {t} sin tener en cuenta a s. Esto implica que algunos procesos de G vean a s y otros a t, haciendo inestable el sistema.

Así que nuestro protocolo también tendrá que impedir el problema de concurrencia. No basta con mantener la consistencia, sino también la estabilidad del sistema.

Para eso, dentro de G habrá una lista de procesos, tales que solo el primero de la lista será capaz de decidir si se admite a un proceso. Así la propagación del grupo resultante solo estará a cargo de un único proceso.

Para eso usaremos la cola de admisión, que ya ha sido presentada en el capítulo de notación.

Solicitud de ingreso a un grupo

Sea G un grupo de procesos {p0,...,pN} de tal manera que Wp0=Wp1=...=WpN. Nótese que esta propiedad implica que para cualquier proceso pi, entonces pi ε WpI, al que el proceso q se quiere unir.

Además, es evidente que Lp0=Lp1=...=Lpn por la duplicación.

Sea q un proceso que quiere unirse. Antes tenemos que tener en cuenta que el sistema resultante ha de tener esta propiedad fundamental:

- El grupo resultante G´=G U {q}, ha de tener el mismo modelo de consistencia que G.
- Es obvio que VG´={i+1,VG{i} U {q}}. Eso implicará ciertas acciones.

Cuando q quiera ingresar en el grupo:

- Envía una solicitud de ingreso, con certificado digital a cualquier proceso p tal que pertenezca a VG.
- p consulta su cola de admisión, y envía la petición a la cabeza, que denominaremos s.
- s difunde el mensaje de bloqueo a todos los procesos de G.
- Cada proceso perteneciente a G se bloquea. No se mandan más mensajes y si se recibe un mensaje, éste se pone en la cola de mensajes sin que sea entregado.
- Cuando un proceso se bloquea envía un mensaje a s diciendo que se ha bloqueado.
- Cuando s comprueba que todos se han bloqueado, atiende a q.
- El proceso s envía Ws al proceso q, y hace que Wq=Wp U {q}.
- El proceso s hace que Wp=Wp U {q}.
- Para mantener la consistencia se envía a q el código de difundir mensaje, entregar mensaje mediante protocolos abiertos, en caso de que no se pueda crear a q´.
- El proceso s difunde que se ha agregado q (o q´). Cuando los demás procesos reciben este mensaje se desbloquean.
- Además s difunde el código necesario a q o se salta este paso si se ha creado q´, para mantener la consistencia con G.
- Cuando q necesite una variable, solicitará la replicación de esta variable (evita sobrecarga de la red), siempre empleando el código que se ha mandado.

Versión secuencial de ingreso a un grupo

Supongamos que empleamos el orden FIFO para entregar los mensajes. El problema de esto es que para la incorporación al grupo es más importante el orden en que se entregan los mensajes que el orden en que se reciban como dice [**HT94**].

Por ello hay que controlar con especial cuidado la secuencia en que los mensajes se entreguen, pues esa secuencia determinará el orden en que llegarán a aplicarse las modificaciones sobre cada una de las variables utilizadas por las aplicaciones de nuestro sistema.

Vamos a ver aquí qué problemas puede conllevar cada tipo de difusión.

Tipo de difusión	**Problema encontrado**
Reliable Broadcast	Este es el más problématico de todos, pues no hay ningún orden.
Fifo Broadcast	Puede llegar un mensaje m2 antes que m1, pero aunque no lo entregue ejecutará m2, dando el sistema inconsistente, aún así podría arreglarse, mediante una máquina de estados, y que se almacenara qué estados ha recibido.
Casual Broadcast	Como según [**HT94**] Causal=FIFO+Total, bien podría ser candidato.
Atomic Broadcast	Orden total, es el propuesto.

Además en el momento de la unión q aún no pertenece a G, luego no tiene por qué mantener la consistencia de G.

Para entender esto, cada mensaje dentro del protocolo propuesto tendrá un identificador que denotará el orden en que se ha insertado. Los procesos tendrán en cuenta el orden. No se tratará ningún mensaje t sin haber tratado previamente todos los mensajes 1...t-1.

Así el protocolo propuesto queda de la manera que sigue:

- Envía una solicitud de ingreso, con certificado digital a cualquier proceso p tal que pertenezca a VG, t=1.
- p consulta su cola de admisión, y envía la petición a la cabeza, que denominaremos s, t=1.
- s difunde el mensaje de bloqueo a todo proceso de G.
- Cada proceso perteneciente a G se bloquea. No se mandan más mensajes y si se recibe un mensaje se pone en la cola de mensajes sin que sea entregado.
- Cuando un proceso se bloquea envía un mensaje a s diciendo que se ha bloqueado.
- Cuando s comprueba que todos se han bloqueado, atiende a q.
- El proceso s envía Ws al proceso q, y hace que Wq=Wp U {q}, t=2.
- El proceso s hace que Ws=Ws U {q}.
- Para mantener la consistencia, se envía a q el código de difundir mensaje, entregar mensaje mediante protocolos abiertos, en caso de que no se pueda crear a q´.
- El proceso s difunde que se ha agregado q (o q´), cuando reciben este mensaje se desbloquea t=3.
- Además s difunde el código necesario a q o se salta este paso si se ha creado q´,

para mantener la consistencia con G, t=4.

- Cuando q necesite una variable, solicitará la replicación de esta variable (evita sobrecarga de la red), siempre empleando el código que se ha mandado.

Situación inicial

Hay que pensar en qué estado estará inicialmente nuestro proceso. Aquí hay varias opciones:

- Cuando se ingresa, se realiza un volcado de todos los datos de G en en el nuevo proceso.
- O se va realizando el volcado a medida que el nuevo proceso vaya necesitando las nuevas variables.

Ingreso de un grupo a nuestro sistema

Cosas necesarias

Antes de empezar, vamos a tener en cuenta que ambos grupos tienen su propia consistencia. Lo ideal sería que ambos grupos tuvieran un sistema de encapsulación, es decir que el mensaje esté encapsulado con la información necesaria para mantener la consistencia. Aquí tendremos que distinguir dos cosas:

- Información: La información del mensaje, es decir, el contenido.
- Cápsula: La información necesaria que necesitan los procesos para mantener la consistencia. Por ejemplo identificadores, marca de tiempo, etc.

Este modelo es también muy usado en la vida real, sobre todo en la logística, ya que una cosa es el objeto en sí, (un ordenador, una colonia, etc.) y otra cosa los mecanismos para asegurar su mantenimiento y su entrega (cajas, códigos de barras, etc.)

Con esto nos aseguramos la compatibilidad entre sistemas, independientemente de su consistencia, como se verá más adelante.

Esto nos plantea un problema de rendimiento, ya que añadimos un paso más, la sustitución de una capa por otra.

Notación

> **Cg(M) :** Cápsula del grupo G para tratar el mensaje m con la consistencia de G

Sean dos grupos H y G, cada uno con su consistencia. G entrega un mensaje a H, entonces ha de seguir estos pasos:

- G entrega a H el mensaje **Cg(M)**

- H tiene que transformar **Cg(M)** por **Ch(M)**

Para facilitar la tarea, lo que es la parte de la cápsula tiene que estar muy bien delimitado de lo que es la información del mensaje. Además G tendrá que entregar a H, el código necesario para desencapsular el mensaje entregado por G. Vamos a simplificar suponiendo que todos los procesos de H tienen el mecanismo necesario para desencapsular los mensajes de H.

Sean dos grupos G y H con VG={i,{g1,g2,g3,g4....gn}} y VH={i,{h1,h2,h3}}, donde H quiere unirse a G. Cada grupo tiene su propia consistencia. Así la consistencia de G es CG y la consistencia de H es CH.

Para unir el grupo usaremos el hi ε VH , entonces hi dará los pasos necesarios para unirse a VG, formando VG´={{i+1},VG U {hi}}, y además VG' ∩ VH = {hi}.

Nótese que desde el punto de vista de G, VH será visto como un solo proceso, es decir el proceso hi. La comunicación entre el proceso hi y el grupo G mantendrá la consistencia CG, según lo visto en el punto anterior.

Véase que la comunicación entre los procesos del grupo H se mantendrá como antes y será invisible a G. Por tanto G no podrá interferir en la comunicación de los procesos dentro de H.

La comunicación entre G y hi seguirá la consistencia CG, pero el grupo H mantendrá la consistencia CH. ¿Qué ocurre si hi cae? Entonces la visión de G es que el grupo H habrá caído, y para H que se ha perdido la comunicación con G. H tendrá que elegir otro proceso hj ε VH para que comience la unión con G.

Aquí no solo se pierde la visión de H , sino también la consistencia global con H (ver más adelante) , pues G pierde la colaboración de H.

Nótese que no tiene por qué haber un único proceso hk ε VH, sino que puede haber varios. La visión de G es que habrá diferentes procesos unidos a él.

Esto plantea un problema en H. ¿Qué proceso escoger para la comunicación con G? La selección ha de ser un proceso que no sea relevante, es decir que no contenga información del grupo. ¿Por qué? Para evitar sobrecarga, ya que este proceso controlará todo el flujo de información. Si aparte contiene información de control de flujo de H, el proceso tendría demasiado trabajo.

En caso de que falle el proceso encargado en H de mantener la comunicación con G, será el propio H quien escoja otro proceso que se unirá a G. La visión de G será que ha caído el anterior proceso y que otro intenta unirse. G no ve grupos, solo procesos.

La consistencia de G será mantenida en el grupo resultante G U {hk} por el propio G, y la consistencia de H, incluyendo a hk, por H. El protocolo de unión que proponemos no interfiere a los modelos de consistencia de cada grupo por separado. Solo se encarga de unir grupos.

La idea subyacente es la delegación: cuando un mensaje tenga que ir de G a H, G delegará este cometido al proceso de H que se ha unido a G. Algo parecido pasa con el servicio DNS. Está el nodo raíz y los nodos que cuelgan de él (com, es, etc.). Cuando una petición va a un nodo auxiliar, el nodo delega esta petición a los nodos que cuelgan de él.

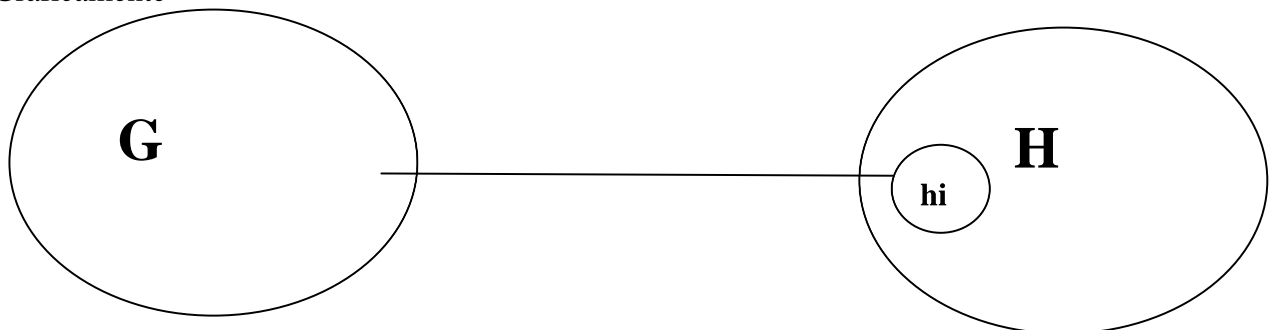
Delegamos, eso es cierto, pero la delegación hasta qué punto es efectiva. Se puede unir todo tipo de consistencias con todo tipo de consistencias. En algunos tipos de consistencia no bastará con unir los grupo , sino que ambos grupos tendrán que mantener la misma consistencia, y para eso se necesitarán algoritmos intrusivos, es decir que modifiquen el estado de cada grupo, y sus protocolos.

En algunos casos esto será posible, en los protocolos fast. En otros será imposible sin recurrir a los algoritmos intrusivos como se verá más adelante.

Antes de esto, veamos cómo funciona:

1. G entrega un mensaje siguiendo la consistencia de G al proceso puente hk.
2. hi entrega a H, el mensaje siguiendo la consistencia de H al grupo H.

Gráficamente



Absorción

Puede darse el caso en que no queramos que cuando se unan dos grupos existan ambos grupos unidos por un proceso puente, sino que un grupo absorba a otro.

En este caso el algoritmo para conseguir esto es:

- Para cada h ε H
    - Solicita la unión a G, no como puente sino como grupo separado.
    - Solicita a H quitarse del grupo.

Cuando H se quede sin ningún proceso desaparecerá de forma lógica. Para invertir el proceso cada proceso h ε H, tendrá que guardar el origen.

Problemas de la absorción

- Incompatibilidad de datos: Podría darse el caso de que G y H tengan nombres de variables en común, pero con datos disjuntos. La forma de arreglar esto es dar preferencia a G. Lo contrario sería propagar todas las variables de H en G.
- H es muy grande: Esto puede producir saturación en G se ve más adelante.

Consistencia global

¿Qué pasa cuando se unen dos grupos?

Llamaremos consistencia global a la consistencia obtenida cuando se pasan mensajes de G a H a través de ih.

Supongamos que G y H sean dos grupos cada uno con la consistencia CG y CH.

Hay que tener en cuenta que CH y CG pueden ser diferentes, y por tanto su "velocidad" también lo podrá ser. Según [**AF96]** un protocolo es rápido si y solo si el tiempo de cada operación es sensiblemente más breve que el retardo de la red. Esto es debido a que la toma de decisiones pasa a ser local; es decir que tanto las lecturas como las escrituras, se pueden gestionar de manera local sin necesidad de conocer lo que hayan hecho otros procesos del sistema.

Los modelos Fast son Pipeline Ram, weak consistency y causal memory.

Los modelos rápidos según [**AF96]** no soportan la exclusión mutua, pues eso supondría un retardo.

Según [**AF96]** los modelos no *fast*, se podrán conectar, pero se obtiene una consistencia más relajada, mientras que los *fast* mantendrán sus propiedades

La demostración está en **[AF96]** y no entra dentro de los objetivos de este trabajo.

En general según **[AF96],** solo los modelos *fast* se podrán componer y mantener sus propiedades , independientemente de sus propiedades fisicas , ya que puede tomar recursos globales

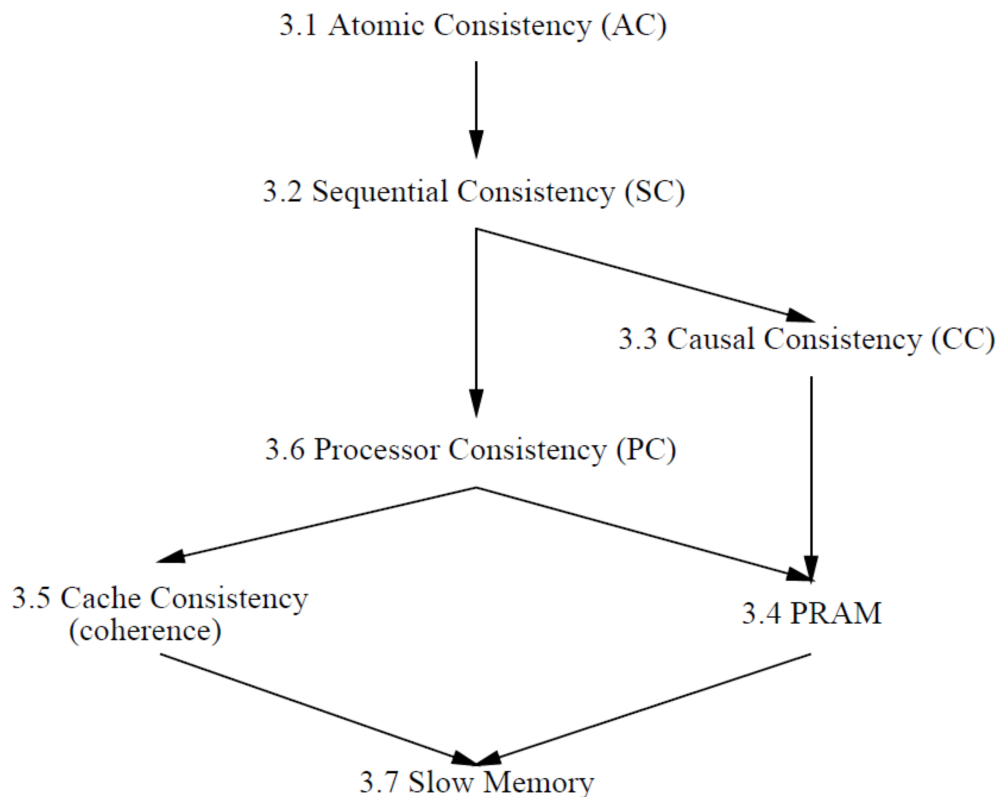
El problema puede llegar a ser realmente complejo, pues aun teniendo ambos grupos consistencias *fast*, se puede dar el caso de que T(G)<T(H) debido a propiedades físicas de cada grupo (velocidad de la red , velocidad de las máquinas) etc. En esa situación se daría el problema de saturación sobre el proceso puente.

Esto se puede solucionar mediante :

- Antes de unir 2 grupos, mediante tests escoger cual debe ser el proceso óptimo que haga de puente.
- Si T(G)<<<T(H) , simplemente descartar la unión.
- Hacer de H un grupo satélite. Esto es, solo usarlo como apoyo de ciertas operaciones.

Posibles uniones

Tengamos en mente el dibujo de



Aquí antes de realizar nuestro analisis , vamos a recordar como son los modelos , es decir si son fast o non fast

Modelo	**Fast / Non Fast**	**Observaciones**

Atomic Consistency	Non Fast	Se necesita un conocimiento global del grupo, es decir un histórico. Aparte, se apoya en el orden total exigido por la consistencia secuencial y esa consistencia tampoco era "fast".
Sequential Consistency	Non Fast	Se necesita un conocimiento global del grupo , ya que todos los procesos han de estar de acuerdo en el orden total a imponar sobre todo el sistema. Resulta imposible manejar un protocolo distinto de orden total en cada subgrupo y que la unión de todos ellos genere un único orden total global.
Processor Consistency	Non Fast / Fast	Este modelo de consistencia admite dos interpretaciones diferentes. Una de ellas es "fast" y fue formalizada por [AF96]. La otra requiere acuerdo sobre el orden global de las escrituras sobre cada variable (aparte del orden FIFO de cada proceso) y ese acuerdo común resulta imposible bajo un modelo "fast".
Cache Consistency [**AF96**]	Fast	Se sacrifica la concurrencia
PRAM	Fast	Para el orden FIFO no hace falta conocimiento global
Slow	Fast	Simplemente se pasan los mensajes con una ventana muy grande

Una vez echa una primera imagen , vamos a analizar cada posible unión

No Fast <-> No Fast

Los modelos "no fast" no pueden tomar las decisiones en base a conocimiento exclusivamente local. Por tanto, deben "preguntar" al otro subsistema sobre la

ordenación de escrituras que va a adoptar. Con ello, no se podrá garantizar de manera global esa consistencia “no fast”, ya que el otros subsistema no utiliza el mismo protocolo. Aquí aparecen dos vías para resolver esta interconexión. La primera alternativa consiste en adoptar un protocolo de consistencia “intrusivo” (que altere las decisiones tomadas en cada subsistema) y es la solución descrita por [Johnson99]. Con ella se puede llegar a interconectar subsistemas con consistencia secuencial, para obtener una consistencia global también secuencial.

La segunda alternativa consiste en respetar los principios de interconexión explicados hasta ahora, que no permiten algoritmos intrusivos. Como consecuencia de esto, sólo se podrá obtener un modelo de consistencia global que será “fast” y que, dentro de la jerarquía de Mosberger presentada arriba, sería el más cercano (pero siempre más relajado) a los dos “no fast” que pretendíamos interconectar. Por ejemplo, si quisiéramos interconectar dos sistemas secuenciales bajo este principio obtendríamos una consistencia causal.

Fast <-> No Fast

Aqui la consistencia global será una consistencia “fast” siempre más relajada que la “no fast” a interconectar, pero lo más estricta posible, pudiendo coincidir con la “fast” que se pretendiera conectar. Por ejemplo, si se intenta conectar un sistema secuencial (“no fast”) con otro sistema causal (“fast”) obtendremos consistencia causal. Sin embargo, si interconectamos la variante “no fast” de la consistencia procesador con otro sistema causal (“fast”) solo obtendremos consistencia FIFO o PRAM , por ser la consistencia “fast” más fuerte inmediatamente más relajada que la “procesador” (y también más relajada que la causal que pretendíamos conectar).

Fast <-> Fast

Cuando se intente conectar un sistema “fast” A con otro sistema “fast” B se obtiene la consistencia “fast” más relajada de ese par, siempre y cuando exista una ordenación entre ellos. Por ejemplo, si A es causal y B es PRAM, se obtendrá PRAM, pues PRAM es más relajado que causal. Si los modelos A y B utilizados no se pueden ordenar según su grado de relajación (véase figura con los modelos de Mosberger), entonces se seguirán las reglas explicadas para los modelos “no fast”. Por ejemplo, al interconectar un sistema causal con otro caché, sólo se podrá obtener consistencia “slow”.

Ejemplo de Algoritmo

Como ejemplo de algortimos vamos a usar el que hace referencia a [**JFC08**] , en este documento , mediante un unico algoritmo se implementan las consistencias causal , cache y secuencial

El tipo de consistencia , esta parametrizado mediante un parametro , el cual decide que tipo de consitencia se realiza

```
Initialization ::
begin
  turn_p ← 0
  updates_p ← ∅
end

w_p(x)v :: atomic function
begin
  x_p ← v
  if ((x, ·) ∈ updates_p) then
    remove (x, ·) from updates_p
  include (x, v) in updates_p
end

r_p(x) :: atomic function
begin
  if (model = sequential) and (updates_p ≠ ∅) and ((x, ·) ∉ updates_p)
then
    wait until turn_p = p
  return(x_p)
end

send_updates() :: atomic task activated whenever turn_p = p
begin
  /* send to all processes, except itself */
  broadcast(updates_p)
  updates_p ← ∅
  turn_p ← (turn_p + 1) mod n
end

apply_updates() :: atomic task activated whenever turn_p = q, p ≠ q, and
the set updates_q from process q is in the receiving buffer of process p
begin
  take updates_q from the receiving buffer
  while updates_q ≠ ∅ do
    extract (x, v) from updates_q
    if (model = causal) or ((x, ·) ∉ updates_p) then
      x_p ← v
  turn_p ← (turn_p + 1) mod n
end
```

Comienza el algoritmo con una inicialización

Objeto	Significado
Turnp	Conjunto de identificadores de los procesos según p , tambien identifican el turno
Updatesp	Actualizaciones pendientes del proceso p

Seguido de una función de escritura W(p)v donde se ve que adjutna la nueva variables a la lista de actualizaciones

Seguido de una funcion de lectura R(x) , donde se ve que esta operación no es fast en caso de ser secuencial , pues tiene que esperar hasta que sea su turno , es decir hasta que no se haya recorrido todos los procesos , no se hace la lectura , esto se realiza para garantizar que sea secuencial.

Vemos que el turno solo se actualiza cuando se envian las acutalizaciones o se aplican , es decir cuando se tiene en cuenta todo el sistema.

Como se ve ya no es fast , por que ya tiene un cose de n procesos , no se realiza de inmediato como las demás operaciones

Una funcion de difusion send_upadtes , y se activa unicamente cuando le corresponde , donde se difunden las actualizaciones , y se actualiza el turno

Y otra denominada apply_dates donde se aplican esos cambios , como se ve cuando es causal siempre se aplican esos cambios.

También se ve que en la forma cache , opta por **[AF96]**, es decir sacrifica la concurrencia , en la operación de lectura no se espera a a que sea su turno , no es concurrente con las otras operaciones y por lo tanto con el sistema , así que es fast.

Resumiendo

Modo	**Razon**
Causal	No hay espera en la lectura , y siempre se aplica el cambio este o no este dentro de las actualizaciones de p , por tanto fast , no depende del entorno
Secuencial	Hay una espera de que se hayan aplicado todos los cambios del sistema en el proceso que este ejecutando , para la lectura de la variable
Cache	Le quita la parte secuencial , es decir sacrifica la concurrencia , para hacerlo fast

Compatibilidad con otros protocolos

Vamos a hacer que nuestra solución propuesta sea compatible con otras soluciones propuestas.

Cuando un proceso se una a nuestro grupo, no hay que olvidar que se unen procesos aunque estos hagan de puente entre nuestro grupo G y el suyo. G no ve grupos, solo procesos. Podrá pedir la inclusión de un proceso que haga de interfaz e incluso, mediante protocolos abiertos, que ese proceso se implemente en la misma máquina que hace de puente.

Sean dos grupos G y H con VG={i,{g1,g2,g3,g4....gn}} y VH={i,{h1,h2,h3}}, donde H quiere unirse a G. Cada grupo tiene su propia consistencia. Así la consistencia de G es

CG, la consistencia de H es CH y además CH tiene un sistema de interconexión (SI) con otros sistemas que denominaremos IH.

El protocolo resultante será:

- q cualquier proceso de VH, envía una solicitud de ingreso con certificado digital a cualquier proceso p tal que pertenezca a VG.
- p consulta su cola de admisión y envía la petición a la cabeza, que denominaremos s.
- s se pone en contacto con q, t=1.
- q responde a s, t=2.
- s difunde el mensaje de bloqueo a todo proceso de G, t=3.
- Cada proceso perteneciente a G se bloquea. No se mandan más mensajes y si se recibe un mensaje, este se pone en la cola de mensajes sin que sea entregado.
- Cuando un proceso se bloquea envía un mensaje a s diciendo que se ha bloqueado.
- Cuando s comprueba que todos se han bloqueado, atiende a q, t=4.
- El proceso s envía Ws al proceso q, y hace que Wq=Wp U {q}, t=5.
- El proceso q solicita a G la creación de un proceso intermedio, t=6.
- El proceso crea q´, y se lo hace saber a q.
- El proceso s hace que Wp=Wp U {q'}.
- El proceso s difunde que se ha agregado q'. Los procesos se desbloquean cuando reciben este mensaje, t=7.
- Cuando q necesite una variable, solicitará la replicación de esta variable (evita sobrecarga de la red).

Nótese que solo q´ verá q. Desde el punto de vista de G es que se ha agregado a q´. Luego q´ solicitará a H su inclusión. Para ello mediante protocolos abiertos, ejecutará el código que H le envíe para su ingreso.

Pero q´ se unirá a H como representación de G, es decir, como un grupo. H tendrá que seguir un protocolo de unión de grupos, creando el correspondiente IH, cuando G quiera comunicarse con H, lo hará a través de q´, y éste con H mediante IH. Y cuando H quiera comunicarse con G, lo hará con q´ mediante IH.

Simplificando el esquema q´ es el SI de G, que se comunica con el SI de H.

Véase que realmente la consistencia de G y H no tienen por qué ser la misma, ya que G

no ve a H, sino a q´, y H no ve a G, sino a q´. Cuando q´ se comunique con G usará los protocolos de G, y tendrá la consistencia de G, y cuando q´ se comunique con H usará los protocolos de H, teniendo la consistencia de H.

Las ideas subyacentes siguen siendo la delegación y el uso de interfaces del paradigma de programación orientado a objetos. Cuando G se comunica con H o viceversa se delegará en q´.

Solicitud de baja de un proceso / o de un grupo de procesos

Aquí se asume un grupo G = {g1,g2,g3.....gn} y gn quiere darse de baja en el grupo. Este proceso será similar a cuando un proceso p ε {g1,g2,g3,....gn-1}, se ponga en contacto con gn, y vea que falle la conexión. La red le devolverá fallo; entonces se eliminará de su lista el proceso gn, y además se propagará este cambio.

El problema que presenta este método es el retardo. Supongamos que por causas desconocidas el mensaje tarda x tiempo en llegar; más de lo que cualquier proceso p espera; entonces se le dará de baja sin motivos.

Podemos hacer que haya unos m intentos. Si al intento m no hay respuesta se produce la baja del grupo de forma irremediable. Si el proceso vuelve a la vida, enviará un mensaje solicitando la unión al grupo. Si no se ha dado cuenta de su baja, el grupo le responderá que ya no pertenece y por tanto tendrá que volver a solicitar la unión al grupo.

La ventaja de este método es que no propone sobrecarga y es fácil de implementar. Basta con que el proceso p deniegue toda respuesta al grupo, dando la sensación de que ha caído.

Otra opción es que envíe expresamente una solicitud de baja pero esta opción es menos eficiente que la otra. El algoritmo será análogo al de adición de un proceso, pero solicitando la baja.

También puede darse el caso de que se expulse el proceso p del grupo. En este caso p recibirá la notificación de la expulsión y los demás procesos borrarán a p de su lista.

Cambio de consistencia

Vamos a dotar a nuestro protocolo de más dinamismo, y es que vamos a poder cambiar la consistencia en caliente, es decir sin tener que cerrar las aplicaciones

Supóngase que tenemos un sistema que funcione con la consistencia FIFO, pero se nos ha dado un caso en las pruebas en que los datos se han corrompido y queremos aumentar el grado de fiabilidad de nuestro grupo.

Vamos a cambiar la consistencia de FIFO a causal. Para hacer esto nos apoyaremos en la idea de la interfaz del paradigma de programación orientado a objetos. Todos los procesos de G ya sean procesos reales o procesos virtuales, tendrán que ser capaces de aceptar *scripts*.

El protocolo para poder cambiar la consistencia vendrá a ser:

- Sea cualquier proceso p perteneciente a G.
- p recibe la orden de cambiar de consistencia.
- p consulta su cola de admisión, y coge el que esté en la cabeza s.
- p envía a s la orden de cambio de consistencia.
- s bloquea el grupo.
- Cada proceso perteneciente a G se bloquea. No se mandan más mensajes y si se recibe un mensaje, éste se pone en la cola de mensajes sin que sea entregado.
- s envía a todo proceso perteneciente a G los nuevos *scripts*.
- s recibe que todos los procesos han recibido el *script*, si no es así se aborta el proceso.
- s difunde el desbloqueo.
- A partir de entonces se entregan y se difunden los mensajes con los nuevos *scripts*, excepto los que estén en la cola, que se entregan con los antiguos, así mantenemos la vista.
- Los relojes vectoriales se ponen todos a 0, con la llegada de los nuevos *scripts* (se produce un reinicio).

Como se ve en ningún momento se reinicia el sistema. El cambio de consistencia se hace en caliente, evitando los problemas que implica reiniciar un sistema. Desde el punto de vista del usuario, se observa que el sistema ha estado bloqueado un pequeño periodo de tiempo en comparación con reiniciar el sistema.

Los encargados de ejecutar los nuevos *scripts* serán los procesos unidos a G, o los procesos virtuales que estén en G y que hagan de puente entre G y los demás procesos.

Algoritmos intrusivos

Para poder aplicar los tipos de consistencia no fast en diversos grupos, necesitamos algoritmos intrusivos, es decir, aquellos que hacen que ambos grupos compartan el mismo estado, tal como requiere la consistencia no fast.

Podemos apoyarnos en esta idea para enviar el cambio en caliente sin necesidad de reiniciar el sistema.

Aquí tenemos el problema principal de que el estado de los grupos o de cada proceso en caso de que la consistencia sea fast, no tiene por qué ser la misma, produciendo contradicciones. La idea más fácil es hacer que un estado de un proceso escogido (como maestro, al azar, etc.) sea el estado a imponer a los demás.

Con el script también llegará el nuevo estado, como actualización.

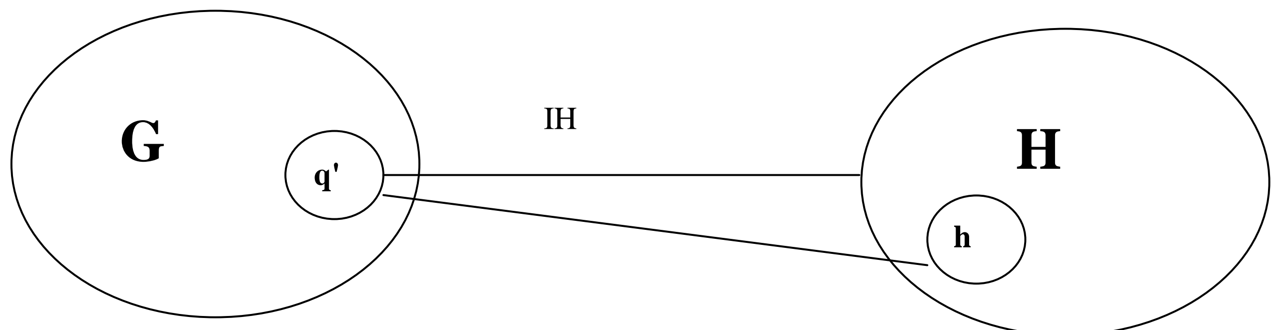
Aquí se hará en dos pasos :

- Se envía el estado a los dos grupos.
- Luego el código del nuevo protocolo.

Cambio de consistencia con otros protocolos

Aquí vamos a ver cómo cambia la consistencia G cuando está unido a otros grupos con sus propios protocolos de interconexión.

Tengamos en mente cómo se hace la unión de interconexión.



Donde:

- El proceso q´ es un proceso virtual que se conecta con q.
- El proceso q´ es un proceso virtual que se conecta mediante IH con H
- El proceso q´ es un proceso virtual que se conecta mediante IH con G

Si G cambia de consistencia, solo cambiará la consistencia con q´. El protocolo de cambio de consistencia no debería pasar de q´. A esto le podríamos añadir una

excepción y es saber si H también puede cambiar de consistencia. En este caso el cambio de consistencia se propagaría mediante IH

Problemas

Saturación

Uno de los problemas que podemos tener sobre todo en la conexión de grupos es la saturación. Esto puede darse si las velocidades de G y H, debido no solo a la consistencia de G y H, sino a las propiedades físicas de ambos sistemas (los tiempos de propagación) son diferentes.

De todas maneras, siempre se puede hacer que todos los procesos de H se unan a G. G verá todos los procesos de H, pero no las conexiones entre los procesos de H. Esto se ve más detenidamente anteriormente en la sección de consistencia global.

Envío masivo de solicitudes de unión al grupo G

También una manera de evitar el ataque del sistema de un hacker será mediante la identificación mediante certificado de cada mensaje. Todos aquellos mensajes que no cumplan con el certificado serán descartados

Aquí un usuario malicioso puede enviar solicitudes de procesos que se unan a G, para atacarlo. No hace falta ni que sean de diferente máquina , sino basta con que sean de diferentes procesos .

Una idea para solucionarlo, es mediante certificados. Todo proceso o mensaje que no cumpla el certificado es automáticamente desechado .

Fallo del proceso puente

Si cae el proceso puente, de esto se darán cuenta tanto H como G. Como H es el que se ha unido a G, solicitará con otro proceso la unión a G.

Para detectar que ha caído un proceso, los procesos deberán enviar cada X tiempo pings a, para evitar sobrecarga, sus vecinos. Si el ping falla, los mensajes que se han enviado desde el último ping, se sabrá que pueden no haberse entregado a ese vecino. Si se ha podido entregar a los demás, dará por efecto que se han logrado difundir y se puede completar haciendo que cada proceso envíe al emisor vecino una muestra de que se ha podido difundir un determinado mensaje. Esto ya no entra en el objetivo de este documento pues ya sería un protocolo dentro del grupo, y no de la conexión de sistemas dinámicos.

También se puede mantener una caché de mensajes difundidos. Cuando se llene la caché, entonces se enviará un ping. Si ha sido exitoso entonces la caché se vaciará. Si no se descargará la caché mediante difusiones en los demás vecinos, además de propagar el fallo con su vecino.

Si el vecino se entera de que ha habido un fallo, entonces éste intentará restablecer la conexión. Si no puede, deberá notificar al administrador del sistema sobre ese fallo. Si no se intenta la reconexión con pi, G asumirá que el proceso ha abandonado el grupo, y por tanto se difundirá la baja.

La idea del ping sí que puede ser utilizada para saber si el proceso puente q´ ha caído o no, siendo responsabilidad de H volver a hacer la conexión con G.

Conflicto de variables

Puede ser que cuando un grupo se una a otro mediante un proceso puente, puede darse que ambos grupos compartan nombres de variables.

Aunque se espera en un primer momento este conflicto, pues el objetivo al unir subrupos es compartir trabajo, puede ser que estas variables en conflicto se usen para objetivos diferentes, produciendo un problema mayor. O que el cambio de valor de una variable compartida deje inconsistente a un grupo.

Entonces aquí puede haber conflictos de nombre. Habrá que distinguir dos tipos de variables:

- Locales al grupo : Solo los procesos del grupo podrán ver esas variables. Estas variables nunca se propagarán del proceso puente al otro grupo.
- Globales al grupo : Son visibles por los grupos. Aquí puede haber conflicto, pues antes de la unión las variables escogidas pueden tener el mismo nombre. H no conoce las variables de G y viceversa.

Para solucionar este conflicto, en las globales, pueden darse varias opciones:

- El proceso puente identifica las variables de G, cuando se envían a H, mediante un prefijo o sufijo.
- Se da preferencia a las de G, es decir cuando se envía una actualización, H borra sus datos.

El proceso puente tiene que saber en todo momento que variables son locales y cuales no , para eso tanto G como H , deberan llevar en cada variables una propiedad que indique cuales se pueden propagar al otro grupo , y cuales no

Pruebas

Vamos aquí a comprobar que todo lo que se ha descrito anteriormente se mantiene coherente, es decir, que se mantiene consistente.

Mantenimiento de la consistencia

Al agregar un solo proceso

Si el grupo G tiene la consistencia C, ¿G U {q} tiene la misma consistencia?

Esto se demuestra fácilmente por reducción al absurdo. Supongamos que tenemos un grupo de procesos denominado G y a éste se une un proceso q mediante el protocolo anterior sin necesidad de ningún proceso virtual.

Entonces G manda a q el script necesario para difundir y entregar mensajes. Al principio y esto es importante q no tiene ningún dato de G, sino que se propagará a medida que vaya necesitando los datos.

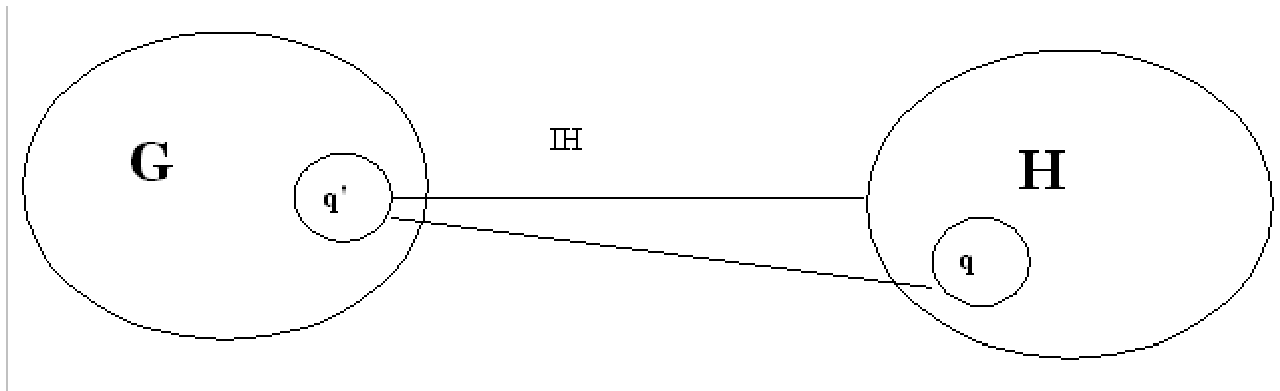
Supongamos que el primer mensaje que se entrega a q es m, y el segundo m´. Aquí pueden pasar dos cosas:

- que lleguen a q desde un proceso de G de tal manera que no mantenga la consistencia de C. Esto es absurdo ya que entonces el proceso que ha entregado el mensaje a q no puede pertenecer a G, pues no tiene la consistencia de G, pero el emisor pertenece a G.
- que q cuando reciba los mensajes de G, los propague de tal manera que no mantenga la consistencia de G. Esto también es absurdo pues propaga los mensajes usando los *scripts* de G. Si no mantiene la consistencia de G los *scripts* significa que están mal, pero esto es absurdo pues G tiene la consistencia C.

La demostración es análoga cuando se agrega el proceso q mediante el proceso virtual q', y a los procesos puente.

Compatibilidad con otros protocolos

Tengamos en mente el diagrama:



Aquí respecto a G, la demostración es exactamente idéntica al apartado anterior.

Conclusiones

Este TFM describe los conceptos y protocolos necesarios para interconectar grupos con sistemas de protocolos de consistencia diferentes. El impacto de estas soluciones es mínimo, solamente se necesita añadir el script de difundir / entregar un mensaje o la creación de un proceso virtual que implantan así el protocolo de interconexión.

La novedad de nuestro protocolo respecto a otros protocolos es:

- Simplicidad: No hace falta grandes conocimientos para implantarlo y entenderlo, solo un poco de programación orientado a objetos.
- Dinamismo: No nos hemos conformado en mantener una determinada consistencia, sino que sea aplicable a cualquier consistencia, incluso darse el caso de que dos grupos estén unidos aun teniendo consistencias diferentes. Un hecho que incrementa su dinamismo es la capacidad de cambiar la consistencia global en caliente, en función de la consistencia presentada por otros subsistemas.

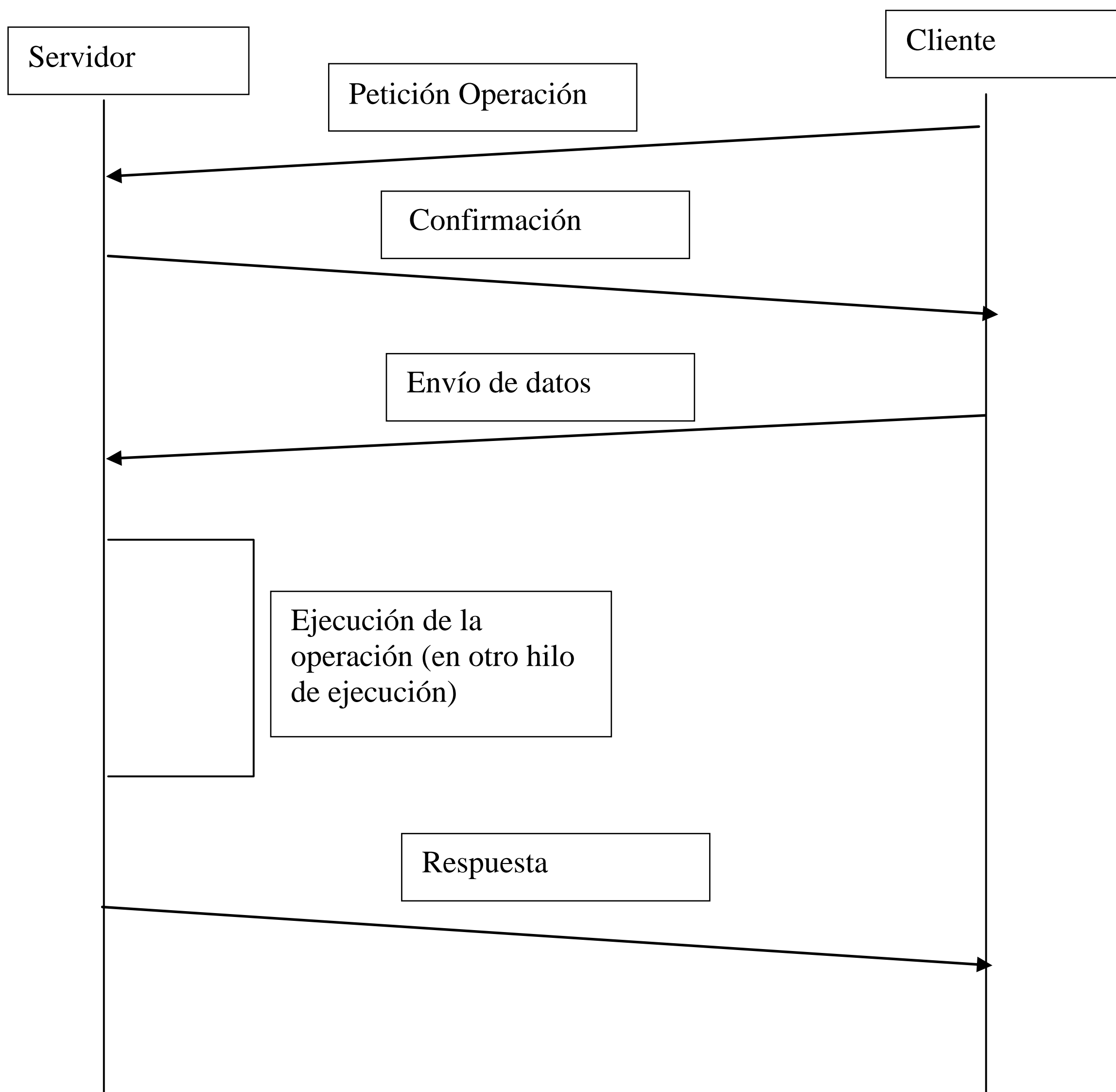
Apéndice I Protocolo Abierto

Los protocolos abiertos son un sistema ideado en esta tesis para resolver el problema de comunicación ante sistemas de comportamiento heterogéneo.

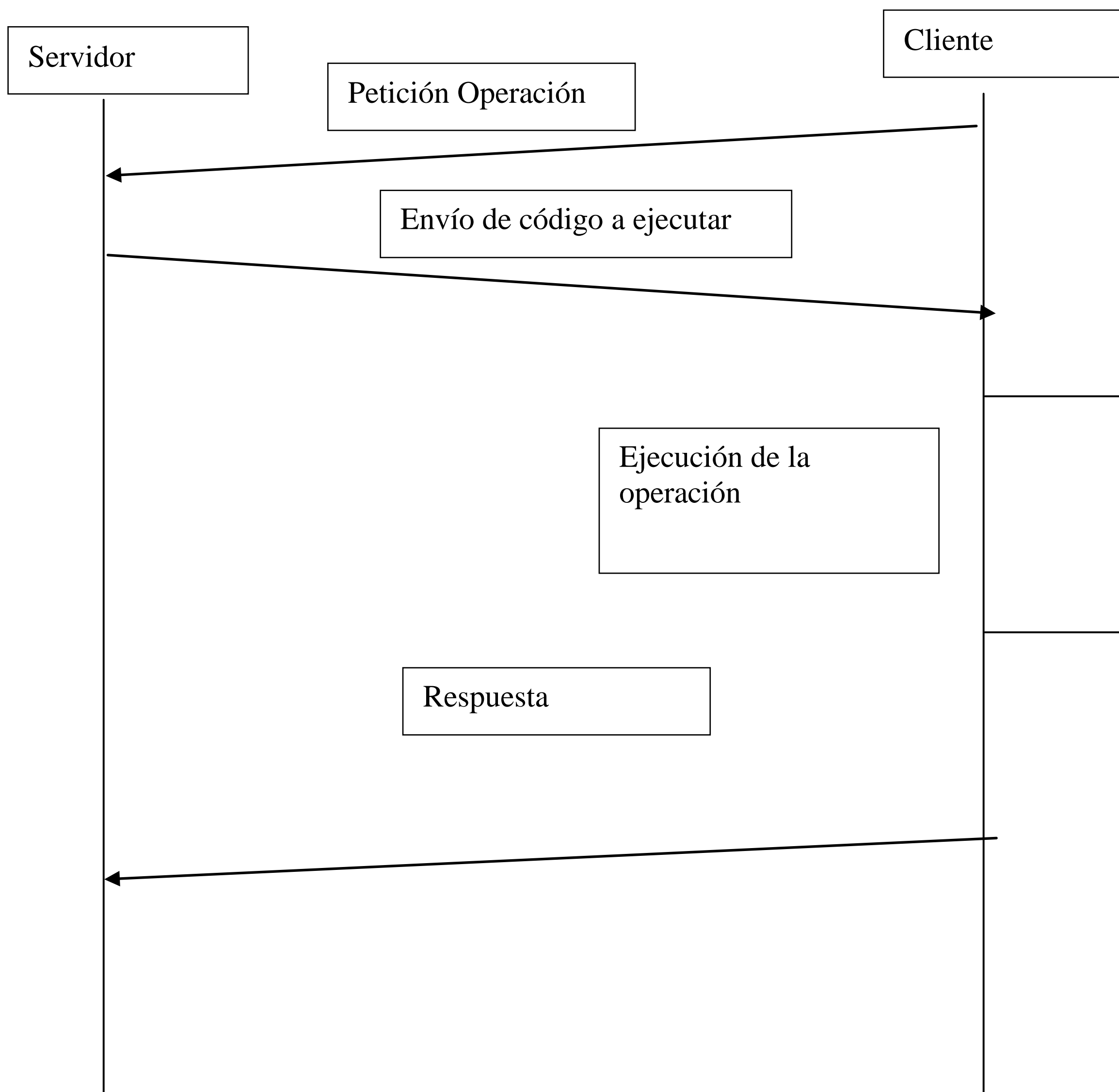
El problema radica que el cliente a priori sabe que tiene que hacer algo, pero no cómo hacerlo.

En un protocolo normal, el cliente se conecta, realiza una petición al servidor y éste la ejecuta y devuelve los resultados. En un protocolo abierto se invierten los papeles. El cliente es el que sirve la petición, pero como es cliente no sabe qué tiene que hacer.

Por ejemplo en un protocolo normal



Ahora sería

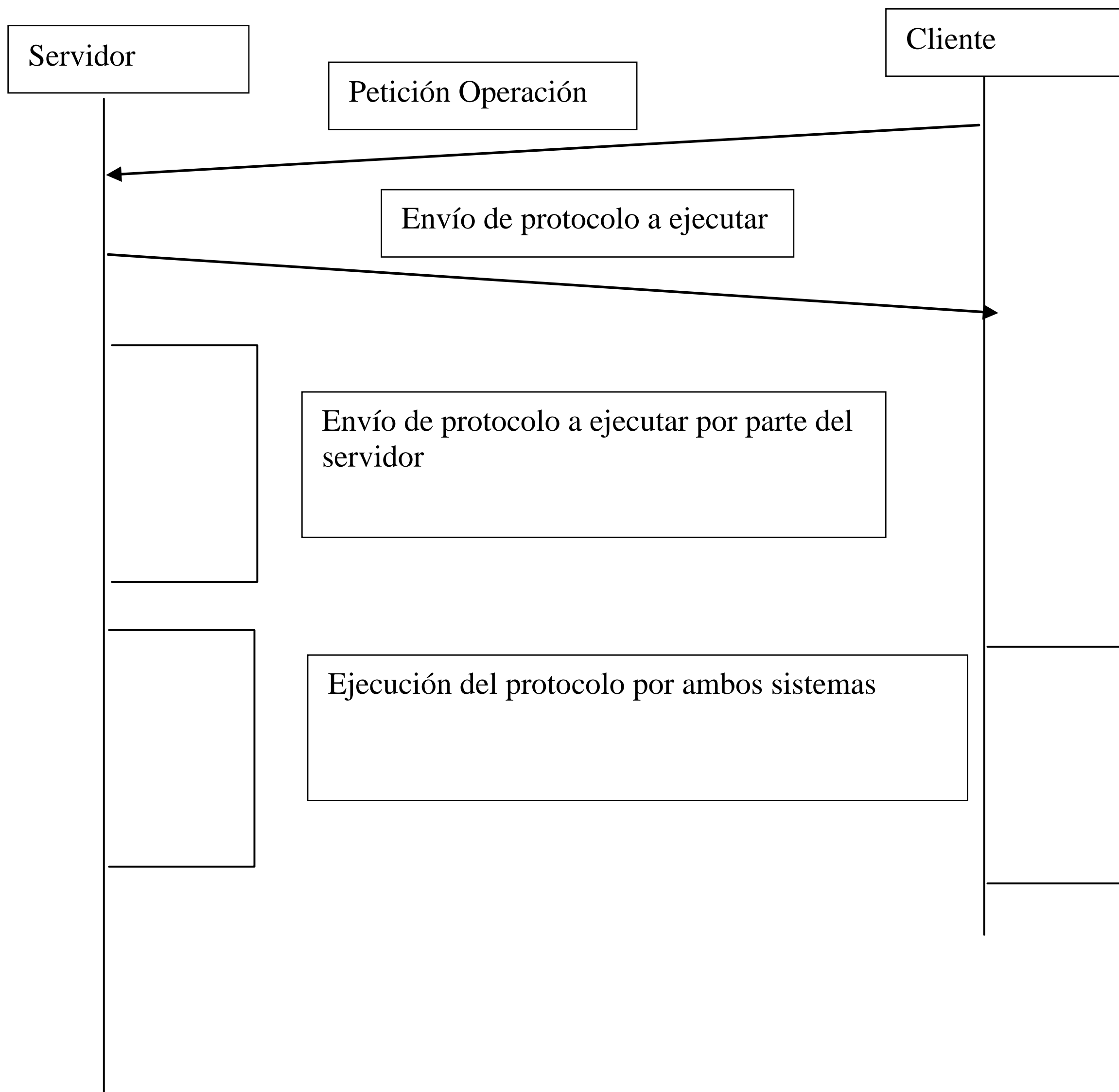


Aún podemos ir más allá. De hecho lo de arriba no deja de ser más que un caso concreto de protocolo normal, donde en vez de enviar el resultado, se envía el código a ejecutar.

En un protocolo abierto, es el servidor quien decide el protocolo de comunicación entre ambos sistemas.

Y lo decide en el momento, es decir sin que se establezca a priori, según las necesidades del sistema.

Entonces quedaría así:



Esto nos deja un sistema totalmente abierto. Por ejemplo puede venir varios clientes:

Cliente A

- Al cliente se le envía las operaciones :
    - Leer(X),Leer(Y),Z=SUMA(X,Y),Enviar(Z)
- El servidor tiene que que ejecutar
    - Enviar(X),Enviar(Y),Esperar_Cliente,Actualizar(Z)

Pero otro cliente puede llegar y entonces

- Al cliente se le envía

    - Iniciar(D,0),Iniciar(H,5),Ejecutar(MiPrograma(D,H),EnviarResultado
- El servidor ejecutará
    - EnviarCódigo(MI_Programa),EsperarCliente,GuardarResultado

Como se ve no tiene nada que ver un protocolo con otro. Son diferentes, e incluso el cliente por un breve tiempo se convierte en servidor.

Bibliografía

[ABJ92] Mustaque Ahamad, Rida Bazzi, Ranjit John, Prince Kohli, and Gil Neiger: "The power of processor consistency". Technical Report GIT-CC-92/34, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA, 1992.

[AF96] Hagit Attiya, Roy Friedman: "Limitations of Fast Consistency Conditions for Distributed Shared Memories". *Inf. Process. Lett.* 57(5): 243-248 (1996)

[AH90] Sarita Adve and Mark Hill: "Weak ordering: A new definition". In *Proceedings of the 17th Annual International Symposium on Computer Architecture,* Seattle, WA, USA, 1990, pgs. 2-14.

[CKV01] Gregory Chockler, Idit Keidar, Roman Vitenberg: "Group communication specifications: a comprehensive study". *ACM Comput. Surv*. 33(4): 427-469 (2001)

[Cri91] Flaviu Cristian: "Understanding Fault-Tolerant Distributed Systems". *Commun. ACM* 34(2): 56-78 (1991)

[GL02] Seth Gilbert, Nancy A. Lynch: "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". *SIGACT News* 33(2): 51-59 (2002)

[GLL+90] K. Gharachorloo, D. Lenoski, J. Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy: "Memory consistency and event ordering in scalable shared-memory multiprocessors". *Proceedings of the 17th Annual International Symposium on Computer Architecture,* Seattle, WA, USA, 1990, pgs. 15-26.

[Goo89] James R. Goodman: "Cache consistency and sequential consistency". Technical Report 61, SCI Committee, March 1989.

[HT94] Vassos Hadzilacos and Sam Toueg: "A modular approach to fault-tolerant broadcasts and related problems", Technical Report, Cornell University, Ithaca, NY, USA, 1994.

[JFC08] Ernesto Jiménez, Antonio Fernández, Vicent Cholvi: "A parametrized algorithm that implements sequential, causal, and cache memory consistencies". *Journal of Systems and Software* 81(1): 120-131 (2008)

[Lam78] Leslie Lamport: "Time, clocks, and the ordering of events in a distributed System". *Communications of the ACM*, 21(7):558–565,1978

[Lam79] Leslie Lamport: "How to make a multiprocessor computer that correctly executes multiprocess programs". *IEEE Transactions on Computers*, 28(9):690–691, September 1979

[LS88] R. J. Lipton and J. S. Sandberg: "PRAM: A scalable shared memory". Technical Report CS-TR-180-88, Princeton University, September 1988.

[Mos93] David Mosberger: "Memory Consistency Models". *Operating Systems Review* 27(1): 18-26 (1993)

[SN04] Robert C. Steinke, Gary J. Nutt: "A unified theory of shared memory consistency". *Journal of the ACM* 51(5): 800-849 (2004)