UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE INGENIERIA

APLICACION DE LOS ENFOQUES DE GRUPOS DE PROCESOS Y SINCRONIA VIRTUAL EN EL COMPUTO DISTRIBUIDO

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO MECANICO ELECTRICISTA

(AREA ELECTRICA ELECTRONICA)

P R E S E N T A

JULIO RAMIREZ ALARCON

DIRECTOR DE TESIS:

DR. EN ING. GUSTAVO AYALA MILIAN

MEXICO, D. F. **1997**

Dedico esta tesis a mis padres Mario y Marcela.
por el apoyo que me han brindado siempre.

Agradecimientos

Agradezco de manera muy especial al Dr. Gustavo Ayala Milian por haberme ayudado a la realización de éste trabajo.

A mi novia Laura por su apoyo, comprensión y aportaciones al desarrollo de este trabajo.

A mi tío Ubaldo por su apoyo durante mi carrera.

A mis hermanos a quienes siempre recuerdo con afecto.

INDICE

1.Introducción

1.1 Descripción del problema.

Los sistemas operativos actuales cuentan con herramientas básicas para el desarrollo de aplicaciones distribuidas, que en su mayoría permiten crear aplicaciones cliente/servidor a través de llamadas a procedimientos remotos. Para este tipo de aplicaciones de solicitud-respuesta estas herramientas funcionan adecuadamente, sin embargo, en el caso de aplicaciones en donde se requiere de coordinación entre las acciones que realizan los procesos es necesario usar herramientas más sofisticadas. Este tipo de aplicaciones en las que los procesos tienen que interactuar unos con otros, i.e. cómputo distribuido, son el tema de este trabajo.

La creación de un sistema de computo distribuido es una labor complicada en la que el diseñador del sistema tiene que tomar en consideración una gran cantidad de factores que pueden afectar. Estos van desde, el correcto funcionamiento del sistema en condiciones normales, esto es que se haga realmente lo que uno desea, hasta consideraciones de fallas en el mismo.

En este trabajo se describen los problemas a los que un diseñador de sistemas de computo distribuido se tiene que enfrentar. La manera en la que los procesos se comunican unos con otros, y el orden en que estos ven que los eventos ocurren son aspectos de vital importancia en la creación de un sistema de cómputo distribuido.

El orden de los mensajes puede ser diferente en los diferentes procesos, los mensajes se pueden perder, los procesos pueden fallar en el transcurso de la comunicación, algunos procesos pueden recibir un mensaje mientras otros no. A pesar de que existen protocolos relativamente simples que permiten manejar algunos de estos problemas, su implementación a partir de las herramientas que proporciona la mayoría de los sistemas operativos nos conduce ha sistemas ineficientes con un desempeño pobre.

Uno tendría que implementar sus propios protocolos a un nivel más bajo, lo cual aumentaría la complejidad del sistema y sobre todo el tiempo en el que este se diseña.

Un sistema de cómputo distribuido de propósito específico, construido a partir de las herramientas básicas del sistema operativo, requiere de un elaborado diseño y de una gran cantidad de tiempo para la implementación de la comunicación eficiente y confiable de los procesos que participen en él. Es esta combinación entre eficiencia y confiabilidad la que hace a un sistema de cómputo distribuido difícil de diseñar.

1.2 Antecedentes

Desde 1945, cuando empezó la era de la computadora moderna, hasta cerca de 1985 las computadoras eran grandes y caras. Como resultado, la mayoría de las organizaciones tenían tan solo unas cuantas computadoras, las cuales no se podían comunicar entre sí. A partir de la mitad de los ochentas surgieron dos importantes avances tecnológicos: El primero fue el desarrollo de poderosos microprocesadores con una capacidad de cómputo comparable al de un *mainframe* y a un costo menor. El segundo fue la invención de redes de área local de alta velocidad, esto permitió conectar docenas e incluso cientos de máquinas que podían transferir información entre ellas. Estos avances tecnologicos dieron origen a los sistemas distribuidos.

Un sistema distribuido es un sistema con varios elementos de procesamiento y varios dispositivos de almacenamiento conectados a través de una red. Potencialmente, esto hace a un sistema distribuido mas poderoso que uno centralizado en dos aspectos: Primero, puede ser más confiable debido a que cada función esta replicada varias veces. Cuando un procesador falla, otro puede hacerse cargo del trabajo. Cada archivo puede ser almacenado en varios discos por lo que la falla de un disco no destruye ninguna información. Segundo, un sistema distribuido puede hacer mas trabajo en el mismo tiempo, debido a que varios calculos pueden ser realizados en paralelo. El objetivo de los diseñadores de sistemas distribuidos es por lo tanto crear sistemas que sean capaces de soportar fallas y que exploten al máximo el procesamiento en paralelo.

1.3 Desarrollo propuesto

Los ingenieros en sistemas que se tuvieron que enfrentar a estos problemas por primera vez, diseñaron en un principio sistemas de cómputo distribuido a partir de sus suposiciones con respecto al comportamiento de éstos, descubrieron nuevos problemas e idearon nuevas soluciones. Basados en su experiencia han desarrollado herramientas de propósito general como *ISIS*, *Smartsockets* y otras (además de los sistemas operativos distribuidos que implican aún una mayor complejidad), que facilitan la creación de los sistemas distribuidos. Estos sistemas proporcionan una variedad de herramientas que permiten solucionar de manera eficiente una amplia gama de aplicaciones distribuidas. Lo que se propone con esta tesis es tener el conocimiento de los conceptos fundamentales alrededor de los sistemas de cómputo distribuido, los posibles problemas a los que se puede enfrentar el programador de una aplicación de este tipo y las opciones que tiene para resolver eficientemente estos problemas.

A través de la descripción de una aplicación simple pero en donde se consideran las dos ventajas más importantes de un sistema distribuido (que son la tolerancia a fallas y el procesamiento en paralelo) se muestra de que forma se utilizan las herramientas y los conceptos básicos de los sistemas distribuidos.

El trabajo concluye con una aplicación de gran interés práctico: una red de computadoras móviles, donde además pueden correr aplicaciones distribuidas. A través de el uso de las herramientas adecuadas de *ISIS* se puede obtener este sistema que es más eficiente que las propuestas convencionales ya que proporciona un servicio confiable tolerante a fallas y elimina los protocolos de introducción.

2. Grupos de procesos y sincronía virtual

2.1 Características generales de un sistema distribuido

Es importante señalar que no existe una definición formal de lo que es un sistema de cómputo distribuido y que las características que éstos deben de tener varían desde las más simples y fáciles de obtener hasta las más ambiciosas y complejas. Sin embargo una definición elemental sería la siguiente:

Un sistema de cómputo distribuido es aquel que consiste de un grupo de máquinas que se pueden comunicar entre sí y no tienen memoria compartida.

En cuanto al equipo, existen varias formas en las que éste puede ser organizado, especialmente en términos de como se conectan las máquinas y de que forma se comunican. Existen varias clasificaciones para los sistemas de cómputo con varios *CPU's* (TAN92). Aquí sólo se hara distinción entre aquellos que comparten la misma memoria (multiprocesadores) y los que no (multicomputadoras). Un sistema distribuido puede incluir multiprocesadores, pero un multiprocesador no será considerado un sistema distribuido. El tipo de interconexión en la red puede ser de conmutación o de canal. El sistema puede ser a su vez fuertemente acoplado o débilmente acoplado. En un sistema fuertemente acoplado el tiempo que tarda en llegar un mensaje de una máquina a otra es corto y la tasa de transmisión de información es alta, mientras que en un sistema débilmente acoplado la tasa de transmisión es baja

En un sistema distribuido el software es aun más importante que el hardware, ya que éste debe proporcionar transparencia, flexibilidad, confianza, buen desempeño y ser escalable.

A continuación se describen las características generales de un sistema de cómputo distribuido (TAN92).

Transparencia

El concepto de transparencia se puede aplicar a los siguientes aspectos de un sistema distribuido:

Transparencia de la localización: se refiere al hecho de que en un verdadero sistema distribuido, los usuarios no pueden indicar la localización de los recursos de hardware y software.

Transparencia de migración: significa que los recursos deben de poder moverse de una posición a otra sin tener que cambiar sus nombres.

Transparencia de replica: el sistema operativo es libre de fabricar por su cuenta copias adicionales de los archivos y otros recursos sin que lo noten los usuarios.

Transparencia con respecto a la concurrencia: los usuarios no notan la existencia de otros usuarios.

Transparencia con respecto al paralelismo. el compilador, el sistema de tiempo de ejecución y el sistema operativo deben, en forma conjunta, aprovechar el paralelismo potencial sin que el programador se dé cuenta de ello.

En la actualidad no existe la transparencia con respecto al paralelismo ya que los programadores que desean usar varios *CPU's* deben programar esto de manera explícita. Este último punto es la tarea más ambiciosa de los diseñadores de sistemas operativos distribuidos.

Flexibilidad

En un sistema distribuido las interfases entre las partes del sistema tienen que ser mucho más cuidadosamente diseñadas que en un sistema centralizado. Los sistemas

distribuidos deben ser construidos en una forma mucho más modular que los sistemas centralizados. Las interfases entre los módulos son frecuentemente interfases de llamadas a procedimientos remotos, las cuales automáticamente imponen un cierto estándar.

Debe ser confiable

Dado que los sistemas distribuidos tienen replicas de datos y son redundantes internamente en todos los recursos que pueden fallar, los sistemas distribuidos tienen el potencial para estar disponibles aun cuando ocurran fallas arbitrarias.

La disponibilidad es uno de los aspectos de confianza que puede proporcionar un sistema. Un sistema confiable debe no solo estar disponible, sino que también, debe hacer lo que supuestamente hace, correctamente, aun cuando ocurran fallas. Los algoritmos usados en un sistema distribuido no deben comportarse correctamente sólo cuando el sistema funciona adecuadamente sino también deben de ser capaces de recuperarse ante fallas en el ambiente del sistema.

Buen desempeño

La construcción de un sistema distribuido transparente, flexible y confiable no debe dar como resultado un sistema lento. En el peor de los casos se esperaría que la aplicación no corriera más lento que un sistema centralizado. El problema del desempeño se ve principalmente afectado por las comunicaciones.

Escalable

Los sistemas distribuidos deben ser capaces de crecer. Debe ser posible añadir servidores de archivos o procesadores, para incrementar la capacidad de almacenamiento o procesamiento de un sistema distribuido.

2.2 Grupos de procesos

El hardware en un sistema distribuido permite a un procesador enviar mensajes a otros. Los sistemas operativos utilizan esta facilidad para permitir a un proceso en una máquina enviar mensajes a otro proceso en otra máquina. Una abstracción comúnmente utilizada para la comunicación entre procesos es la llamada a un procedimiento remoto (*RPC*), un proceso se comunica con otro utilizando una interfase que parece un procedimiento. La ventaja de esta abstracción es que simplifica la programación distribuida haciendo que la comunicación con un procedimiento remoto parezca local. Sus limitaciones, sin embargo, son que solo puede ser usado para una comunicación entre 2 procesos: el proceso que llama y el proceso que es llamado. Por lo tanto, las llamadas a un procedimiento remoto son más útiles en programas distribuidos que se ajustan al modelo cliente-servidor, los procesos clientes solicitan servicios de los procesos servidores y estos aceptan las solicitudes y responden a cada una de ellas individualmente. En contraste a la situación anterior, la llamada a un procedimiento remoto no es una abstracción conveniente cuando un programa distribuido está compuesto de varios procesos que tienen un alto grado de dependencia entre ellos y donde la comunicación refleja esta interdependencia. En estos programas, la comunicación normalmente tiene lugar de un proceso a varios procesos. Un ejemplo de esta clase de programas sería un servidor que con la finalidad de proporcionar la tolerancia a fallas y la repartición de la carga de trabajo, es implementado como un grupo de procesos en varios sitios. Si un cliente solicita un servicio del servidor podría entonces enviar una solicitud al grupo, como un todo, en lugar de tener que conocer a los miembros del grupo y tener que comunicarse uno a uno, esto es muy importante ya que el grupo podría cambiar su membresía o su localización en un momento dado(BR93). Si los miembros del grupo desean dividir el trabajo para responder a una solicitud, cada uno de ellos debe asegurarse de que sus acciones son consistentes con respecto a lo que los otros

miembros están haciendo y por lo tanto tendrán la necesidad de comunicarse unos con otros. Lo que se necesita en este caso es una utilería que permita a un proceso enviar un mensaje a un grupo de procesos. A la acción de enviar un mensaje a un grupo de procesos se le llama multitransmisión. En su forma más simple una multitransmisión provoca que un mensaje sea enviado a cada proceso destino. Lo que hace a una multitransmision interesante es que debe ser capaz de manejar la posibilidad de que algunos procesos que participen en el puedan fallar antes de que este haya terminado. Por ejemplo, una falla en el proceso transmisor puede provocar que una multitransmisión de un mensaje sea entregada sólo a algunos de sus destinos. Para que una multitransmisión sea útil a un programador, debe de tener un comportamiento bien definido aun cuando las fallas ocurran. Las multitransmisiones que proporcionan esta garantía son llamadas multitransmisiones confiables. Las multitransmisiones confiables estan implementadas usando protocolos especiales que detectan fallas y toman acciones en compensación. Una de las ideas básicas en el desarrollo de programas distribuidos confiables es el uso de grupos de procesos y herramientas de programacion para grupos (LL86, CD90, Pet87 y KTHB89).

Uno de los principales retos en la creación de programas para computo distribuido es lograr que estos sean confiables: aun en los sistemas no distribuidos el lograr un sistema confiable es algo complicado, y abarca problemas como el correcto funcionamiento del sistema (que haga realmente lo que uno desea), la tolerancia a fallas, que se administre él mismo, que responda en tiempo real y que proporcione protección y seguridad (BSS91). En el caso de los sistemas distribuidos, los problemas son aún mayores ya que constan de multiples procesos que deben cooperar, es decir, no sólo se debe cuidar el funcionamiento de los componentes individuales sino también el funcionamiento de la aplicación como un todo.

Uno podría esperar un buen desempeño en un sistema distribuido a partir del correcto funcionamiento de sus componentes, pero esto no siempre es así. El mecanismo usado para estructurar un sistema distribuido y para implementar la comunicación entre sus componentes juega un papel vital que determina cómo funcionará el sistema. Los sistemas operativos distribuidos actuales, hacen excesivo énfasis en el desempeño de las comunicaciones pasando por alto la necesidad de herramientas que soporten el desarrollo de sistemas complejos. Además, las primitivas de comunicación aunque a menudo son confiables, resultan débiles ante eventos poco comunes como fallas o cambios en la configuración del sistema, por esta razón se pueden desarrollar programas distribuidos rápidos pero poco confiables.

Anteriormente, se mencionó la necesidad de utilizar la multitransmisión de mensajes a un grupo de procesos con la finalidad de que el sistema explote al máximo la tolerancia a fallas y el paralelismo. A continuación, se presentan algunas de las características más comunes en los grupos de procesos.

Existen dos tipos de grupos que se utilizan ampliamente en los sistemas distribuidos (BSS91): Los Grupos anónimos y la Acción coordinada por un conjuntos de programas.

Los grupos anónimos, se utilizan cuando una aplicación publica información acerca de algún tema para otros procesos subscritos a este grupo. Si se desea que una aplicación opere en forma automática y confiable como un grupo anónimo, debe tener ciertas propiedades. En términos generales, debe cumplir con los siguientes puntos:

1. Debe ser posible enviar mensajes al grupo usando una *dirección de grupo*. El programador de alto nivel no se debe de preocupar en expandir las direcciones del grupo dentro de una lista de destinos.

2. Los mensajes deben ser enviados solamente una vez. El programador de aplicaciones no se debe preocupar por la pérdida o duplicación de mensajes.

3. Los mensajes se deben enviar en orden. Hay muchas formas de interpretar este requisito. Pero cuando menos, uno esperaría que los mensajes sean enviados en el orden en que se publican.

4. Debe ser posible mantener la historia de eventos vistos por el grupo. Esto es delicado, ya que pueden haber programas que se unan al grupo cuando este ha estado operando por un tiempo por lo que un nuevo suscriptor frecuentemente necesitará un conocimiento preciso de la historia del grupo. Si *n* mensajes son enviados y el primer mensaje visto por el programa suscrito es el *mi*, uno esperaría que los mensajes *m*1...*mi*-1 estén en el historial y que los mensajes *mi* ... *mn* hayan sido enviados al nuevo proceso.

Acción coordinada por un conjunto de programas. Los programas pueden cooperar entre sí por las siguientes razones: tolerancia a fallas, compartir la carga de trabajo, búsqueda en paralelo en las bases de datos, duplicar los datos y compartir información confidencial.

Los grupos creados de forma explícita comparten los requisitos de comunicaciones de un grupo anónimo, pero tienen necesidades adicionales que se derivan del uso de la información de los miembros del grupo en la aplicación. Por ejemplo, un servicio tolerante a fallas puede tener un miembro primario que realiza algunas acciones y

un grupo ordenado de respaldos que se harán cargo, si éste falla. Si ocurre un cambio en el número de miembros (falla del miembro primario) los miembros que queden del grupo se harán cargo. A menos que sean vistos los cambios en el mismo orden por todos los miembros del grupo, podrían surgir situaciones en las cuales no haya un miembro primario o hayan muchos. De igual forma se puede llevar a cabo una búsqueda en paralelo a una base de datos dividiendo la base de datos en *n* partes, donde *n* es el número de miembros del grupo; cada grupo haría una enésima parte del trabajo, los miembros necesitan una visión precisa del número de miembros para realizar la búsqueda correctamente.

En general, si el grupo mantiene datos replicados, divide una base de datos usando un criterio que varía dinámicamente, o utiliza la composición del grupo como una entrada para el algoritmo ejecutado por los componentes, entonces, será necesario sincronizar los eventos que cambian estos atributos del grupo.

Para esta clase de grupos deben ser considerados otros problemas técnicos:

Soporte para comunicaciones de grupo, incluyen direccionamiento, fallas atómicas (todos los miembros del grupo reciben el mensaje o ninguno lo hace) y entrega ordenada de mensajes.

Uso de la membresía del grupo como entrada, debe ser posible usar la membresía del grupo como información de entrada a un algoritmo distribuido (uno que corre concurrentemente por múltiples miembros del grupo).

Sincronización, para tener un funcionamiento global correcto de las aplicaciones del grupo, es necesario sincronizar el orden en el cual las acciones se llevan acabo.

especialmente cuando los miembros del grupo actúan de manera independiente realizando cambios dinámicos en la información que comparten.

A continuación, se presentan las soluciones a los problemas más comunes en un sistema distribuido utilizando las herramientas de los sistemas operativos convencionales.

2.2.1 Construcción de sistemas distribuidos a través de mensajes.

Soluciones convencionales de entrega de mensajes.

Los sistemas operativos actuales ofrecen tres tipos de servicios de comunicación(TAN88):

Datagramas poco confiables.- Esta posibilidad descarta automáticamente mensajes corruptos, pero esto implica un poco de proceso adicional. Desde el punto de vista del usuario, la mayoría de los mensajes llegarán. Sin embargo, bajo ciertas condiciones los mensajes pueden perderse, duplicarse o enviarse fuera de orden en la transmisión.

Llamada a un procedimiento remoto.- En este caso, la comunicación se presenta como una llamada a un procedimiento que regresa un resultado. Este servicio es relativamente seguro, pero cuando algo falla, el que envía la información no puede distinguir entre cuatro casos: el destino pudo haber fallado antes o después de recibir la petición o la red puede estar deshabilitada o retrasada en el envío de la solicitud o la respuesta.

Flujo de datos confiables.- En este caso, la comunicación se realiza sobre canales que ofrecen un control de flujo y envío secuencial de mensajes. Debido a la conexión, el flujo de datos tiene un desempeño mejor al de las llamadas remotas a procedimientos cuando una aplicación envía grandes volúmenes de datos. El problema es si el flujo se rompe, la situación sería la misma que en el caso de las llamadas remotas a procedimientos.

Construcción de grupos con tecnologías convencionales(BSS91).

Direcciones de grupos.

Considérese primero el problema de mapear una dirección del grupo en una lista de miembros, en una aplicación donde la membresía podría cambiar constantemente debido a los procesos que entran y salen del grupo. La solución obvia para resolver este problema involucra un *servicio de membresía*. Este servicio mantiene un mapa con los nombres de los grupos y lista de miembros. Ignorando el aspecto de tolerancia de fallas del servidor, uno podría implementar este servicio con un simple programa que soportara las llamadas a procedimientos remotos con la finalidad de registrar un nuevo grupo o un miembro de grupo, obtener la membresía de un grupo y tal vez enviar un mensaje al grupo. Un proceso puede entonces transmitir un mensaje ya sea direccionándolo, vía el nombre del servicio o a través de buscar la información de la membresía, y, una vez obtenida, transmitir el mensaje directamente. En el último caso, uno necesitaría también un mecanismo para descartar la información de la dirección obtenida cuando la membresía del grupo cambie. La primera aproximación funcionaría mejor para una sola interacción, mientras que la segunda, sería más recomendable en aplicaciones que envían flujos de mensajes hacia el grupo.

Orden en la distribución mensajes.

Aunque ambas aproximaciones al problema de direccionamiento de grupos entregarían los mensajes a los miembros de un grupo, algunos aspectos importantes han sido pasados por alto, éstos tienen que ver con el orden en el cual los mensajes son entregados.

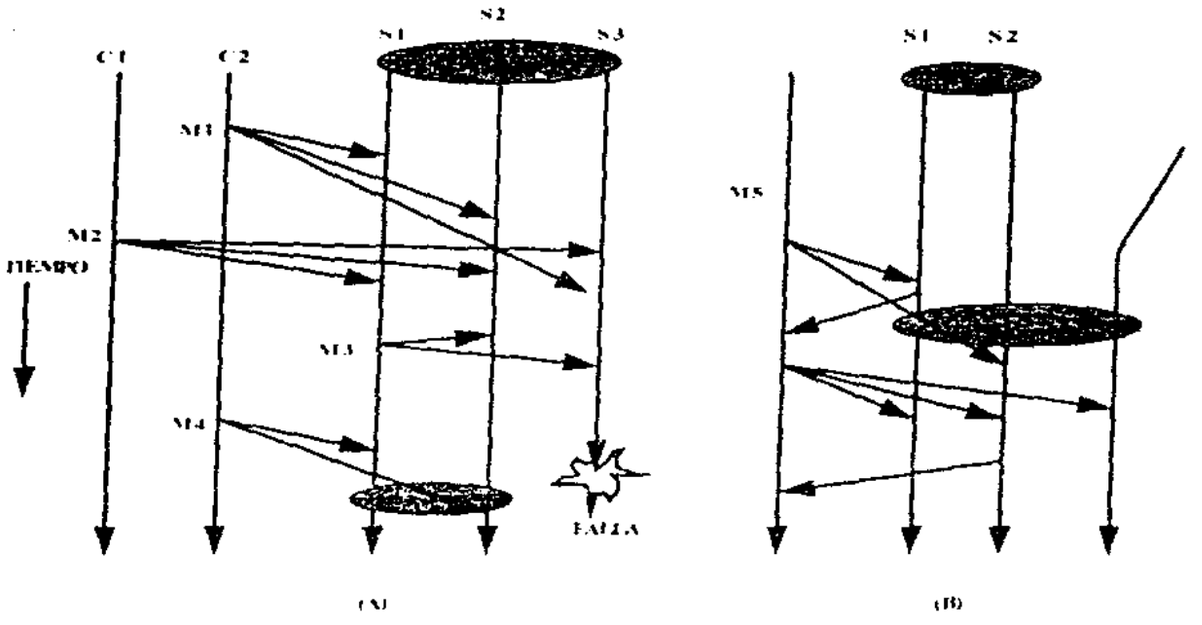


Figura 1: Problemas en el orden de entrega de mensajes.

En la figura 1-A los mensajes *m1* y *m2* son enviados concurrentemente y son vistos en diferente orden por los servidores *s1* y *s3*. En muchas aplicaciones, *s1* y *s3* se comportarían de una manera inconsistente o fuera de coordinación si esto ocurriera. Por ejemplo, un programa podría ver la variacion del mercado, caer y luego levantarse mientras que el otro lo veria levantarse y después caer. La variación del mercado es un parámetro para muchos cálculos financieros y tal secuencia podria fácilmente dejar a los servidores en un estado de inconsistencia.

Un diseñador tendría que anticiparse a las posibles inconsistencias en el orden de los mensajes, ya sea diseñando una aplicación que tolere este desorden o previéndolo explícitamente en el momento en que ocurra, tal vez retrasando el procesamiento de *m1* y

m3 dentro del servidor hasta que un orden haya sido establecido. El verdadero peligro es que el diseñador pase por alto esta situación proporcionando una aplicación que normalmente es correcta, después de todo, dos mensajes simultáneos para el servidor que llegan en diferente orden puede parecer un escenario poco probable, aunque puede mostrar un comportamiento anormal en momentos en que el sistema esté muy cargado.

Desafortunadamente, este es solo uno de varios problemas en el orden de entrega ilustrado en la figura. Considérese el momento en que *s3* recibe el mensaje *m3*. El mensaje *m3* fue enviado por *s1* despues de recibir *m1*, y puede referirse o depender de m1. Por ejemplo *m1* pudo autorizar a un accionista negociar una cuenta particular de una empresa, *m3* podria ser una negociación que el accionista ha iniciado en nombre de esta cuenta. La ejecucion es de tal forma que *s3* no ha recibido todavia *m1* cuando *m3* es enviado. Tal vez, *m1* fue descartado por el sistema operativo debido a la falta de espacio en el buffer. Este será retransmitido, pero sólo despues de un breve retraso durante el cual *m3* puede ser recibido.

Esto puede ser importante ya que *s3* puede estar desplegando órdenes de compra-venta en un piso de negociación. *s3* considerara a *m3* inválido ya que no será capaz de confirmar que la negociación fue autorizada. Una aplicación con este problema puede fallar al llevar a cabo una solicitud de negociacion valida. De nuevo, aunque el problema tiene solución, la pregunta es si el diseñador de la aplicación ha previsto el problema y programado un correcto mecanismo para compensar esta situación. La solución involucra etiquetar los mensajes con suficiente informacion de contexto para reconocer cuando llegan en desorden y retrasar los mensajes apropiadamente.

El mensaje *m4* muestra un problema adicional de orden. Aquí, *s1* cree que el servicio contiene 3 miembros en el momento en que *m4* es recibido. Suponiendo que *m4*

acciona una búsqueda en la base de datos y que el proceso *si* busca en la n-ésima parte de la base de datos. El proceso *s1* buscará en el primer tercio de la base de datos. Sin embargo, el proceso *s2* recibe *m4* después de observar que *s3* ha fallado por lo que busca en la segunda mitad. Una sexta parte de la base de datos no ha sido revisada y las dos respuestas serán inconsistentes.

Transferencia de estados.

La figura 1-B nos muestra un problema un poco diferente. Aquí se desea transferir el estado del servicio a un nuevo miembro, tal vez, a un programa que ha reinicializado después de una falla (habiendo perdido su estado anterior) o un servidor que ha sido añadido para distribuir la carga. Intuitivamente, el estado del servicio será una estructura de datos que reflejará los datos manejados por el servicio que han sido modificados por las actualizaciones que se hicieron antes de que el nuevo miembro se uniera al grupo. Sin embargo, en la ejecución mostrada, un mensaje ha sido enviado al servidor concurrentemente con el cambio de membresía. Una consecuencia es que el nuevo miembro *s3* recibirá un estado que no refleja el mensaje *m5*. Esto dejaría a *s3* en inconsistencia con *s1* y con *s2*. Resolver este problema involucra un complejo algoritmo de sincronización que puede ir más allá de la habilidad de un típico programador de aplicaciones distribuidas.

Tolerancia de fallas.

Hasta el momento las fallas han sido ignoradas. Las fallas traen consigo varios aspectos, aquí sólo consideraremos uno. Supongamos que el remitente de un mensaje falla después de que algunos, pero no todos los destinos recibieron el mensaje. Los

destinos que tienen una copia necesitarán completar la transmisión. El protocolo usado debe exactamente una vez efectuar el envío de cada mensaje.

Los protocolos para resolver este problema pueden ser complejos, pero una solución bastante simple puede estar basada en el protocolo de acciones atómicas (indivisibles) de tres fases sin bloqueo (*non-blocking three-phase atomic commit*) de Skeen(SKE82).

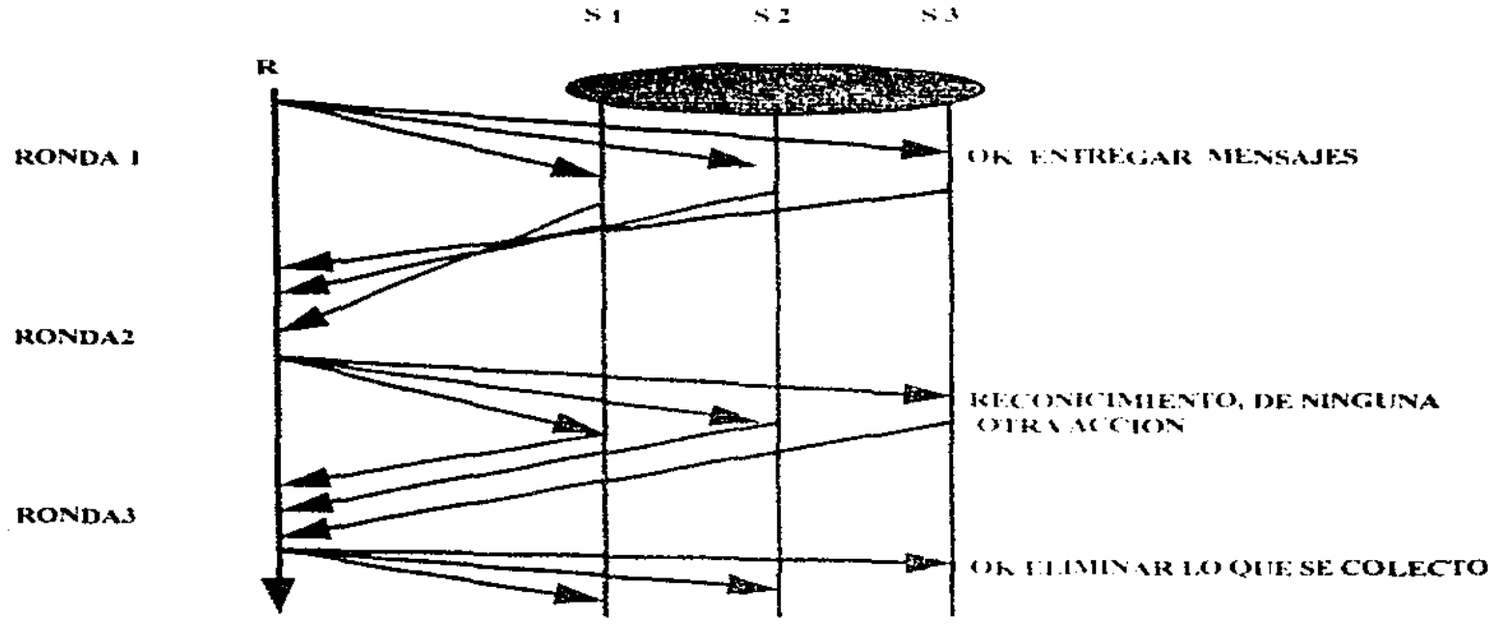


Figura 2:Multitransmision confiable de tres rondas

El protocolo usa una secuencia de tres llamadas a un procedimiento remoto para los destinos, como se muestra en la figura 2. Durante la primera vuelta el remitente envia los mensajes a los destinos los cuales avisan su llegada. Aunque los destinos pueden

enviar el mensaje en este momento, ellos necesitan guardar una copia, debido a que si el remitente falla durante esta primera ronda, los procesos destinos que han recibido las copias necesitarán terminar el protocolo por su propia cuenta. Si no ocurre alguna falla, el remitente le dirá a todos los destinos que la primera vuelta ha terminado. Ellos reconocen este mensaje y anotan que el destinatario está entrando a la tercera vuelta, pero no realizan otra acción. Durante la tercera ronda, cada destino descarta toda la información acerca del mensaje, borra la copia salvada del mensaje y cualquier otra información que haya sido almacenada.

Para manejar las fallas, se asume primero que los destinos tienen una forma de detectar la falla del remitente. Cuando una falla ocurre, un proceso que ha recibido una primera o segunda ronda de mensajes puede terminar el protocolo. La idea básica es tener algún miembro del conjunto de destinos haciéndose cargo de la ronda que el remitente estaba ejecutando cuando falló, los procesos que ya han recibido mensajes en esa ronda detectan los duplicados y responden a ellos como ellos respondieron después de la recepción original.

Desafortunadamente, la detección de fallas no es trivial. Muchos sistemas utilizan un tiempo límite como un esquema de detección de fallas, pero tal aproximación no sería correcta en el protocolo mencionado arriba. Supongamos que el proceso p comienza enviando la primera ronda de mensajes a los procesos $s1, \ldots sn$, pero es retardado después de enviar el mensaje a $s1$. Ya habiendo recibido este mensaje, $s1$ empezará a monitorear a p y, eventualmente, el tiempo límite se cumplirá. El proceso $s-1$ se hará cargo ahora y correrá el protocolo para terminarlo, esto es, todos los procesos recibirán y ejecutarán los mensajes y se olvidarán completamente de la interaccion. Ahora, consideremos esta misma situación cuando p halla experimentado un problema transitorio que corrige el mismo, esto hará que se vuelva a hacer cargo de la transmisión enviando los mensajes a s-

2...x-n. Ninguno de estos destinos reconocerá que estos mensajes están duplicados, por lo que cada uno de ellos los aceptará una segunda vez.

Una forma de resolver este problema sería salvar algo de información, para cada destino, después de enviar un mensaje, para ser usado en el reconocimiento de duplicados. Pero un sistema de comercio puede generar 1000 mensajes por segundo. Si fueran usados 16 bytes para cada mensaje, un proceso podría consumir un megabyte de almacenamiento cada minuto. Una mejor solución, es sustituir un mecanismo confiable para fallas, en el cual se llegue a un acuerdo en el tiempo de detección de falla (RB91), entre otras funciones, este filtra mensajes para prevenir a un proceso que haya fallado, que interactúe con procesos en operación sin ejecutar primero el protocolo de recuperación.

Como se mencionó al inicio de este documento, este es un protocolo de comunicación tolerante a fallas relativamente simple. De hecho, este protocolo no funciona para obtener cualquier flujo de datos asincronos o conectados cuando es invocado sucesivamente y el uso de llamadas a procedimientos remotos limita el grado de concurrencia de comunicación durante cada ronda (sería mejor enviar todos los mensajes al mismo tiempo y recibir las respuestas en paralelo). Mejores protocolos de comunicación han sido descritos en la literatura, pero el mejor desempeño frecuentemente viene acompañado del costo de la mayor complejidad. Incluso, la programación de grupos de procesos trae consigo otros aspectos relacionados con la tolerancia a fallas, tales como la tolerancia a fallas de los mecanismos de direccionamiento de grupos. Ninguno de estos problemas es inmanejable, pero ellos nos llevan a una colección compleja de mecanismos que deben trabajar armónicamente.

Los puntos anteriores, muestran algunos de los obstáculos potenciales que tiene que enfrentar un programador que desea resolver los problemas sobre un sistema operativo convencional, tal como UNIX: (1) expansión de las direcciones del grupo, (2) envío ordenado para mensajes concurrentes, (3) envío ordenado para secuencias de mensajes relacionados, (4) transferencia de estado y (5) atomicidad de fallas (BC90). Esta lista no es exhaustiva, se han pasado por alto cuestiones relacionadas con la garantía de envío en tiempo real así como con los archivos y bases de datos persistentes.

A los sistemas operativos modernos les faltan herramientas necesarias para desarrollar programas basados en grupos, aunque todos estos problemas pueden ser resueltos, la complejidad asociada a la búsqueda de soluciones y a la integración de estas en un único sistema sería inmanejable para programadores no expertos. La única aproximación práctica es resolver estos problemas en el mismo ambiente de cómputo distribuido o en el sistema operativo. Esto permite que los sistemas sean diseñados de tal forma que tengan un buen desempeño predecible y aprovechen por completo las características del hardware y del sistema operativo. Incluso el proveer un grupo de procesos como una herramienta base permite al programador concentrarse en el problema a tratar, como en el caso de soportar la construcción de una interfase gráfica.

2.3. Sincronía virtual

Para simplificar la programación de grupos de procesos y resolver los problemas mencionados, se propone un método inspirado en el control de concurrencia de las bases de datos. Este enfoque se presenta en dos estados. Primero, se discute un modelo de ejecución llamado casi síncrono. Este modelo es usado para después entender mejor el de sincronía virtual que es el modelo propuesto en esta tesis.

Se propone a los programadores asumir un estilo de ejecución distribuida casi síncrono (BJ89, SCH88) en el cual un evento ocurre en un tiempo. Aquí el término evento es usado en forma vaga, significando no sólo un único mensaje sino cualquier evento único que múltiples miembros de un grupo pueden observar. Esto es:

La ejecución de un proceso consiste de una secuencia de eventos, los cuales pueden ser cálculos internos, transmisión de mensajes, entrega de mensajes y cambio en la membresía de grupos, unión a grupos o creación de los mismos.

Una ejecución global del sistema consiste de un conjunto de ejecuciones de procesos. A manera global, uno puede referirse a los mensajes enviados como multitransmisión a grupos de procesos (un único mensaje dirigido a todos los miembros de un grupo de procesos).

Cualquier par de procesos que observen el mismo evento global (esto es recibiendo mensajes de la misma multitransmisión o participando en el mismo grupo) ven los correspondientes eventos locales en el mismo orden.

Un mensaje que se envía a los miembros de un grupo, es entregado a todos sus miembros he interpretado cuando la entrega ha sido programada por el sistema. Aquí asumimos que el remitente especifica un grupo de procesos destino, usando una dirección que es expandida por el protocolo de multitransmisión, que incluye el conjunto de procesos destinos presente.

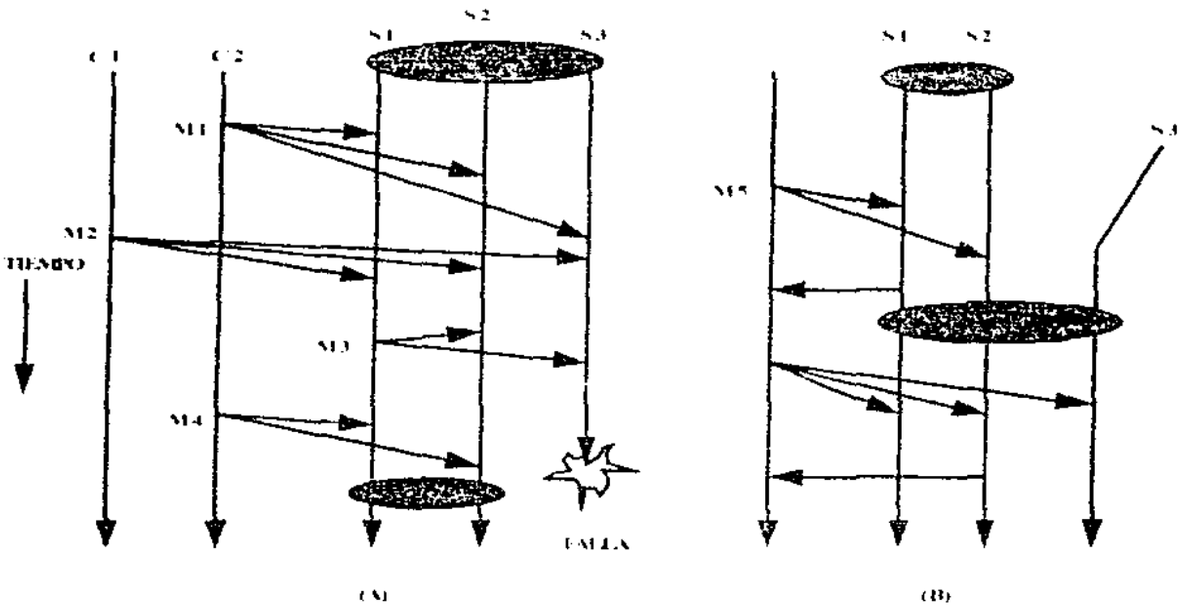


Figura 3: Ejecución casi síncrona.

El modelo casi síncrono nos proporciona varias ventajas, de hecho, como se muestra en la figura 3 elimina todos los problemas identificados en la sección anterior:

(1) Expansión de direcciones de grupo.- En la ejecución casi síncrona, la membresía de un grupo de procesos es determinada en el instante en el que el mensaje es entregado. Un sistema que implemente expansión de direcciones de grupos en sincronía

cercana, necesitaría sincronizar eventos de comunicación con cambios de membresía de grupos. Por ejemplo, retrasa los cambios de membresía hasta que todos los mensajes anteriores hayan sido entregados a sus destinos, y retrasa los nuevos mensajes hasta después de que cualquier cambio de membresía pendiente se haya completado. Este orden es invisible para el programador de aplicaciones, aunque, algunas veces, produce ligeros retrasos en las comunicaciones.

(2)*Entrega ordenada para mensajes concurrentes.*- En una ejecución casi síncrona, multitransmisiones concurrentes serían tratadas como eventos diferentes. Serían, por lo tanto, manejadas en el mismo orden por cualquier destino que tengan en común. En la práctica, esto significaría que un sistema que soporta el modelo transmitiría mensajes usando un protocolo que recoge una orden de entrega y la impone.

(3)*Entrega ordenada para secuencias de mensajes relacionados.*- En la figura 5-a el proceso *s1* envía el mensaje *m3* después de recibir *m1*. Podríamos decir que el envío *m1* ocurre antes que el envío *m3*. En la ejecución de procesos, en un mundo casi síncrono nunca se verá algo que contradiga la secuencia en que ocurren los eventos. En términos prácticos, un sistema necesitará añadir suficiente información extra a *m3* para que este pueda ser retrasado en la recepción si *m1* no ha sido todavía recibido.

(4)*Transferencia de estado.*- La transferencia de estado ocurre en un instante bien definido en el modelo. Si un miembro del grupo controla el estado del grupo en el instante en que un miembro es añadido, o envía algo basado en el estado al nuevo miembro, el estado estará bien definido y completo.

(5)*Fallas atómicas (fallas indivisibles).*- El modelo casi síncrono provee implícitamente atomicidad de fallas, a través de tratar una multitransmisión como un

único evento lógico. Los sistemas que soportan el modelo casi síncrono tendrían que implementar una multitransmisión usando un protocolo tolerante a fallas.

Desafortunadamente, aunque la ejecución casi síncrona simplifica el diseño de aplicaciones distribuidas, sería muy costoso emplearlo en casos prácticos. El problema más serio se origina en el acoplamiento entre el envío de un mensaje y su entrega.

De acuerdo a la primera regla, una multitransmisión a una dirección de grupo será entregada a todos los miembros de un grupo de procesos, e interpretado en el momento de su entrega. Aun cuando un proceso no necesite respuestas de los destinos de una multitransmisión, el modelo bloqueará al remitente de una multitransmisión hasta que las entregas hayan ocurrido, para que el inicio y la entrega de una multitransmisión pueda ser presentada como un único evento indivisible.

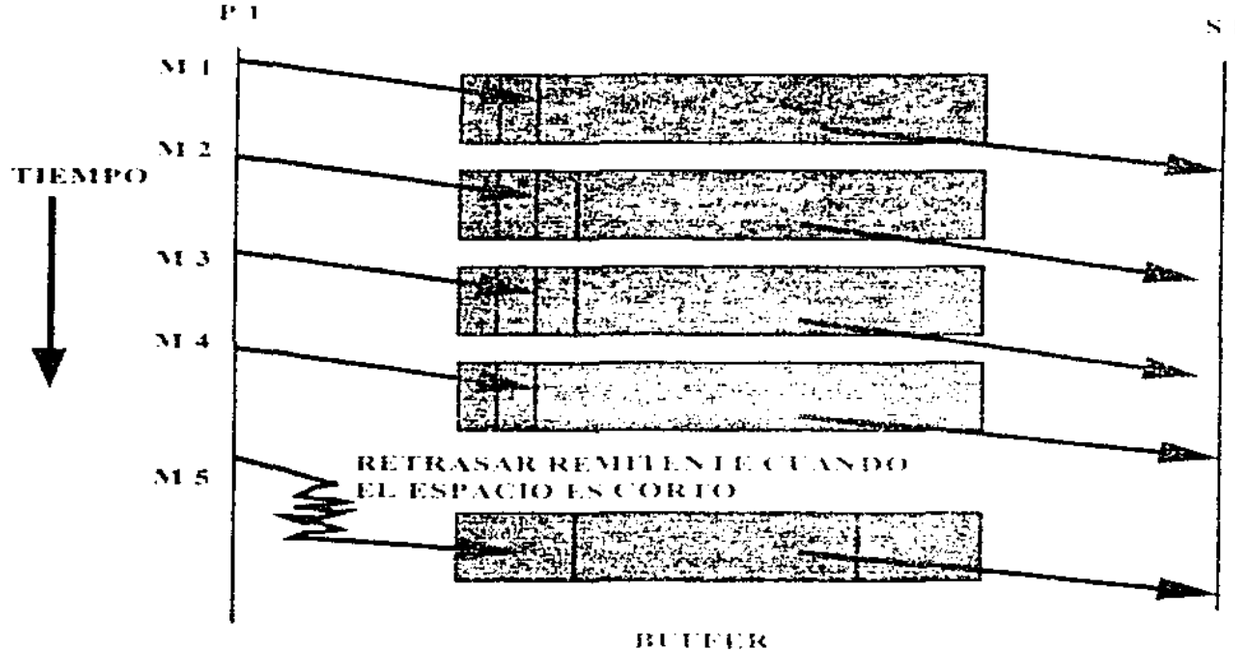


Figura 6: Conexion asincrona

En los sistemas distribuidos, el alto desempeño está dado por las interacciones asincronas: patrones de ejecución en los cuales es permitido al remitente de un mensaje continuar ejecutando sin esperar la entrega. Una aproximación asincrona, trata los sistemas de comunicaciones como un buffer limitado, bloqueando al remitente sólo cuando la tasa de generacion de datos excede la tasa de consumo, o cuando el remitente necesita esperar por una respuesta o alguna otra entrada (Figura 4). La ventaja de este enfoque, es que el retraso entre el remitente y los destinos no afecta a la tasa de transmisión de datos; el sistema opera en forma conectada, permitiendo a ambos, el remitente y el destino permanecer continuamente activos. Una ejecución de casi sincrona impide tal conectividad, retrasando la ejecución de los procesos que envían un mensaje hasta que este es entregado.

El modelo de sincronia cercana ofrece varias ventajas pero a un precio elevado. El sistema propuesto es similar a uno casi sincrono. La idea es que por cada aplicación los eventos sean sincronizados, sólo al grado en que la aplicación sea sensible al orden de los eventos. En algunos casos, este enfoque será identico al modelo casi sincrono, por ejemplo, cuando el estado del grupo es transferido a un nuevo miembro. Aquí, es importante que el estado visto por el nuevo miembro corresponda al que es visto por los miembros viejos, en el instante lógico de su incorporación. Los mensajes anteriores a su unión deben ser transportados, y la secuencia de eventos vista por cada proceso estrictamente controlada. Pero, en otras situaciones, se pueden entregar los mensajes en diferentes ordenes en procesos distintos sin que la aplicación lo note. Esto permite una ejecución mas asincrona. Esto es muy similar a la manera en como los sistemas de bases de datos implementan la seriación usando dos fases. Los servidores de datos, algunas veces, procesan requisiciones 'fuera de orden' pero la ejecución resultante no se puede diferenciar de una serial (BHG87).

2.3.1 Sensibilidad al orden en los sistemas distribuidos.

Existen dos formas de orden que pueden ser necesarias, una es más fuerte que la otra pero también es más costosa. Considérese un sistema con dos procesos, *s1* y *s2*, que envían mensajes a un grupo *G* compuesto por los miembros *g1* y g2. *s1* envía el mensaje *m1* a *G* y, concurrentemente, *s2* envía *m2*. En un sistema casi sincrono, *g1* y *g2* recibirían estos mensajes en el mismo orden. Si por ejemplo los mensajes actualizaran una estructura de datos replicada dentro del grupo esta propiedad podría ser usada para asegurarse que las replicas permanecen idénticas en el transcurso de la ejecución del sistema. Una multitransmision con esta propiedad se dice que logra un orden de entrega atómico. A esta primitiva se le conoce como *ABCAST*. *ABCAST* facilita la programación pero su implementacion tiene su precio. Un mensaje *ABCAST* solo puede ser entregado cuando no existe un *ABCAST* anterior que no haya sido entregado. Esto provoca un retraso: los mensajes *m1* y *m2* deben ser retrasados antes de que puedan ser entregados a *g1* y *g2*. Este retraso en la entrega puede no ser visible en la aplicación. Pero en los casos en que *s1* y *s2* necesitan respuestas de *g1* y o de *g2*, o donde los transmisores y receptores son los mismos, la aplicacion experimentará un retraso significativo cada vez que un *ABCAST* sea transmitido. Este retraso puede ser muy alto dependiendo de como se diseñe el protocolo *ABCAST*.

No todas las aplicaciones requieren de un orden de entrega tan fuerte y costoso. Los sistemas concurrentes normalmente proporcionan alguna forma de sincronización o mecanismo de exclusión mutua para asegurar que las operaciones que pueden entrar en conflicto sean llevadas a cabo con algun orden. En un ambiente paralelo de memoria compartida esto se hace normalmente usando semaforos alrededor de secciones criticas del código. En un sistema distribuido esto se haria normalmente a través del uso de alguna forma de bloqueo o paso de estafeta.

Considérese un sistema distribuido que tiene la propiedad de que dos mensajes pueden ser enviados concurrentemente al mismo grupo solo cuando sus efectos en el grupo son independientes. En el ejemplo anterior a *s1* y *s2* se les impediria que enviaran

mensajes concurrentemente (esto es, si m1 y m2 tienen conflictos potenciales que afecten el estado de los miembros de G). Si se les permite enviar concurrentemente, los ordenes de entrega podrían ser impuestos arbitrariamente, ya que la recepción de los mensajes puede ocurrir en diferente orden.

Parece ser que el orden de entrega que se necesitaría es primera entrada-primera salida o *FIFO*. Sin embargo esto no es completamente cierto como se ilustra en la figura 7.

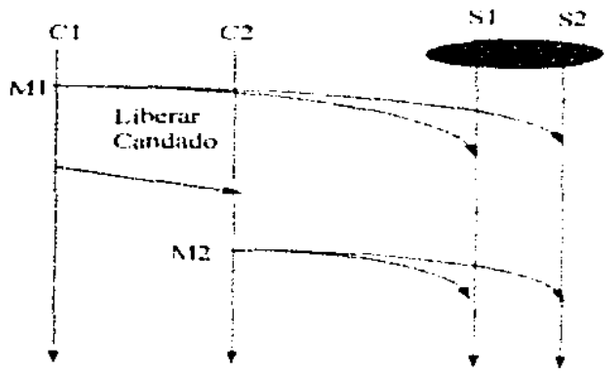


Figura 7: Orden Causal

Aquí se observa una situación en la que *s1* , que sostiene una exclusión mutua, envía el mensaje *m1*, y luego libera su bloqueo hacia *s2*, el cual envía *m2*. Tal vez *m1* y *m2* son actualizaciones a una misma información; el orden de entrega podría ser entonces muy importante. Aunque ciertamente existe la sensación de que *m1* fue enviado primero, es importante notar que el orden de entrega *FIFO* no lograría el orden deseado, ya que el orden *FIFO* normalmente es definido como un par (transmisor, receptor) y aquí tenemos dos transmisores. La propiedad de orden necesaria para este ejemplo es que si *m1* es anterior causalmente a *m2*, entonces *m1* debe ser entregado antes que *m2* en los destinos compartidos, a esto le corresponde la primitiva de multitransmisión *CBCAST*. *CBCAST*

es más débil que *ABCAST* ya que permite que los mensajes enviados concurrentemente sean entregados en diferente orden a destinos traslapados.

La principal ventaja de *CBCAST* sobre *ABCAST* es que no esta sujeta al tipo de retraso citado anteriormente. Un mensaje *CBCAST* puede ser entregado tan pronto como el mensaje anterior haya sido entregado y toda la información necesaria para determinar si los mensajes anteriores son importantes puede ser incluida, a un bajo, costo en el mismo mensaje *CBCAST*. Excepto en casos poco comunes, donde un mensaje anterior es por alguna razón retrasado en la red, un mensaje *CBCAST* sera inmediatamente entregado después de la recepción.

El poder usar un protocolo como *CBCAST* depende de la naturaleza de la aplicación. Algunas aplicaciones tienen una estructura de exclusión mutua para las cuales el orden de entrega causal es adecuado, mientras que otras necesitan utilizar alguna forma de bloqueo para utilizar *CBCAST* en lugar de *ABCAST*. Básicamente *CBCAST* puede ser usado cuando cualquier conflicto de multitransmisión es solo ordenado a través de una cadena causal única. En este caso las garantías que ofrece *CBCAST* son lo suficientemente fuertes para asegurar que todos los conflictos de multitransmisión son vistos en el mismo orden por todos los recipientes- especialmente el orden de dependencia causal. Un sistema de ejecución con estas caracteristicas es virtualmente sincrono ya que el resultado de la ejecucion es el mismo que se obtendria si un orden de entrega atómica fuera utilizado.

El patrón de comunicación *CBCAST* se utiliza mas frecuentemente en un grupo de procesos que maneja informacion replicada que utiliza bloqueos para ordenar actualizaciones. Los procesos que actualizan tal información primero adquieren el candado, y luego realizan un flujo de actualizaciones asincronas, para liberar después el candado. Normalmente habrá un candado de actualización para cada clase de información relacionada, por lo que la adquisicion del candado de actualización descarta el conflicto entre actualizaciones. Sin embargo la exclusión mutua se puede lograr a partir de otras propiedades de un algoritmo, por lo que este patrón puede ser utilizado aun sin un estado

de bloqueo explícito. A través del uso de *CBCAST* para la comunicación se puede lograr un flujo eficiente de información conectada. En particular, no hay necesidad de bloquear al transmisor de una multitransmisión, ni siquiera momentáneamente, a menos que la membresía del grupo cambie en el momento en que el mensaje es enviado.

La enorme ventaja de *CBCAST* sobre *ABCAST* puede no ser inmediatamente evidente. Sin embargo, cuando se considera qué tan rápidos son los modernos procesadores en comparación con los dispositivos de comunicación, debe quedar claro que cualquier primitiva que espera innecesariamente antes de entregar un mensaje puede provocar una sobrecarga substancial. Es común para una aplicación que replica una tabla de solicitudes pendientes dentro de un grupo multitransmitir cada nueva solicitud para que todos los miembros puedan mantener copias idénticas de la tabla. En tales casos si la forma en la que la solicitud es manejada es sensible a los contenidos de la tabla, el transmisor de la multitransmisión debe esperar hasta que la multitransmisión sea entregada antes de actuar sobre la solicitud. Usando *ABCAST* el transmisor necesitará esperar hasta que el orden de entrega pueda ser determinado. Usando *CBCAST* la actualización puede ser realizada de manera asincrona y aplicada inmediatamente a la copia mantenida por el transmisor. De esta forma el transmisor evita un retraso potencialmente largo, y puede inmediatamente continuar su computo o responder a la solicitud.

Cuando un transmisor genera una cantidad enorme de actualizaciones la ventaja de *CBCAST* sobre *ABCAST* es aun mayor debido a que múltiples mensajes pueden ser enviados en un paquete dando un efecto de conexión.

2.3.2 Beneficios de la sincronía virtual

A continuación se mencionan algunos de los beneficios que se obtienen con el uso de la sincronía virtual.

- Permite que el codigo sea desarrollado asumiendo un modelo simplificado de ejecución casi síncrono.
- Proporciona una noción significativa del estado del grupo y de la transferencia de estado, cuando el grupo maneja información replicada, y cuando el cómputo es dividido dinámicamente entre los miembros del grupo.
- Comunicación conectada y asíncrona.
- Se trata a la comunicacion, al cambio en la membresía del grupo de procesos y a las fallas a través de un único modelo de ejecución orientado a eventos.
- Manejo de las fallas a través de una lista de membresía presentada consistentemente integrada con el subsistema de comunicación. En contraste a la clásica aproximación de detección de fallas a través tiempos límites y canales rotos, los cuales no garantizan consistencia.

La aproximación también tiene sus limitaciones:

- Disponibilidad reducida durante fallas de partición de la red *LAN*: sólo permite la continuidad en una particion, y por lo tanto tolera a lo mucho n/2-1 fallas simultáneas, donde n es el número de sitios que estén operando.
- El riesgo de clasificar incorrectamente un sitio operacional o proceso como fallo.

El modelo de sincronía virtual se caracteriza por ofrecer estos beneficios dentro de un único marco de trabajo.

3. Protocolos

3.1. Protocolo de multitransmisión atómica

Una de las propiedades más simples que provee un protocolo de multitransmisión es la atomicidad, esto es, un mensaje es recibido por todos los destinos que no fallan o por ninguno de ellos. Puede ocurrir que no haya entrega, sólo si el transmisor falla antes de que el protocolo finalice. Un protocolo de multitransmisión atómica nunca provocará que un mensaje permanezca sin entregar en algun destino que no falle si este ha sido entregado en algunos otros (aun si algunos destinos fallan antes de que el protocolo termine). Esta es una propiedad muy útil ya que un proceso que recibe una multitransmisión puede actuar con el conocimiento de que todos los destinos que estén operando recibiran una copia del mismo mensaje. Esto reduce el peligro de que un recipiente realice acciones que son inconsistentes con las acciones realizadas por otros procesos. Considerese el caso de un número de procesos en el que cada uno tiene que mantener una copia de un conjunto replicado de elementos, y se realiza una multitransmisión a estos procesos solicitandoles que añadan un elemento particular a este conjunto. Si un protocolo de multitransmision atomico es usado, cada recipiente puede añadir el elemento a su copia del conjunto con la certeza de que todos los demás destinos harán lo mismo, y por lo tanto todos sus conjuntos contendran información idéntica. Sin la atomicidad, el implementador del conjunto repticado tendria que tomar las medidas necesarias para asegurar que una falla no provoque que algunos procesos no realicen las actualizaciones, lo cual provocaria que las copias del conjunto fueran inconsistentes.

A primera vista un protocolo de multitransmisión atomica puede parecer trivial de implementar, especialmente si la capa de transporte proporciona transmisión punto a punto confiable. El iniciador podría simplemente enviar el mensaje a cada sitio destino, y un recipiente podria simplemente entregarlo a cualquier proceso destino en aquel sitio. Pero ¿qué pasa si el iniciador falla después de que ha enviado el mensaje a algunos (pero no a todos) de los sitios destinos? Esto lleva precisamente a la situación que se esta

tratando de evitar: algunos destinos han recibido el mensaje mientras otros no. Para empeorar las cosas, los destinos que no han recibido el mensaje no tienen idea de que ellos deben de recibir alguno. Esto significa que debe ser necesario para uno o más recipientes detectar que el iniciador ha fallado y direccionar los mensajes a los sitios que no lo recibieron. Esto por supuesto implica guardar una copia del mensaje por un momento- al menos hasta que se sabe que todos los destinos lo han recibido. Y ya que las copias de un mensaje no pueden ser guardadas permanentemente, se debe de proporcionar algun medio para que un recipiente sepa que el mensaje ha sido recibido en todas partes para luego poder descartarlo. Esto introduce aún más complejidad. Si una copia duplicada de un mensaje llegara a un sitio despues de saber que el mensaje fue descartado, este podria ser entregado erroneamente una segunda vez. Por esto se necesita estar seguro de que antes de que el sistema descarte un mensaje, todas las copias del mensaje hayan sido purgadas de cualquier proceso activo y canales de comunicación.

La figura 3.1 muestra un protocolo simple que implementa la multitransmisión atómica tolerante a fallas (SCH86). Cuando un sitio recibe un mensaje por primera vez, este retransmite un copia del mensaje a todos los destinos. Por lo que si un sitio recibe un mensaje y permanece operando, todos los sitios recibirán una copia del mensaje. Por lo que se garantiza la atomicidad. Sin embargo esta propiedad se logra a través de incrementar la comunicación debido a las retransmisiones. El protocolo también hace uso de espacio en memoria porque el mensaje (o alguna parte de él) debe ser almacenado en un recipiente hasta que todas las copias retransmitidas lleguen, de otra forma no habrá manera de identificar a estas copias como duplicados del primer mensaje. Este protocolo puede ser modificado para retransmitir mensajes sólo si se observa que el iniciador ha fallado. La mayoria de la comunicación extra sólo se llevaría a cabo en caso de falla lo cual es más razonable. Aunque no existan fallas, este protocolo incurriría en costos de almacenamiento y comunicación extra. Cada recipiente debe almacenar el mensaje hasta que se le notifique que éste ha sido entregado a todos los destinos a los que fue direccionado y esta notificación requiere de mensajes extras. En general, dependiendo de las propiedades que un protocolo de multitransmisión proporciona, éste producirá costos en terminos de latencia (tiempo que transcurre entre que el mensaje es enviado y es

entregado a sus destinos), comunicación (por los mensajes extras o mensajes más grandes) y consumo de memoria.

En el iniciador:

envía el mensaje *m* a todos los sitios donde hay un proceso destino

En el sitio receptor del mensaje:

Si el mensaje *m* no ha sido recibido todavía
envía una copia de *m* a todos los otros sitios donde hay un proceso destino
entrega *m* a cualquier proceso destino en este sitio

Figura 3.1. Un protocolo de multitransmisión atómica simple.

3.2. Protocolo de orden causal

La primitiva de comunicación *CBCAST* garantiza la entrega ordenada de mensajes que están relacionados causalmente. El protocolo *CBCAST* funciona de la siguiente manera(BSS91, TAN92): Si un grupo tiene *n* miembros, cada proceso mantiene un vector con *n* componentes, uno por miembro del grupo. El *i-ésimo* componente de este vector es el numero del ultimo mensaje recibido proveniente del proceso *i* en secuencia. Los vectores son manejados por el sistema en tiempo de ejecución y no por los mismos procesos de usuario y son inicializados a cero, como se muestra en la parte alta de la figura 3.2

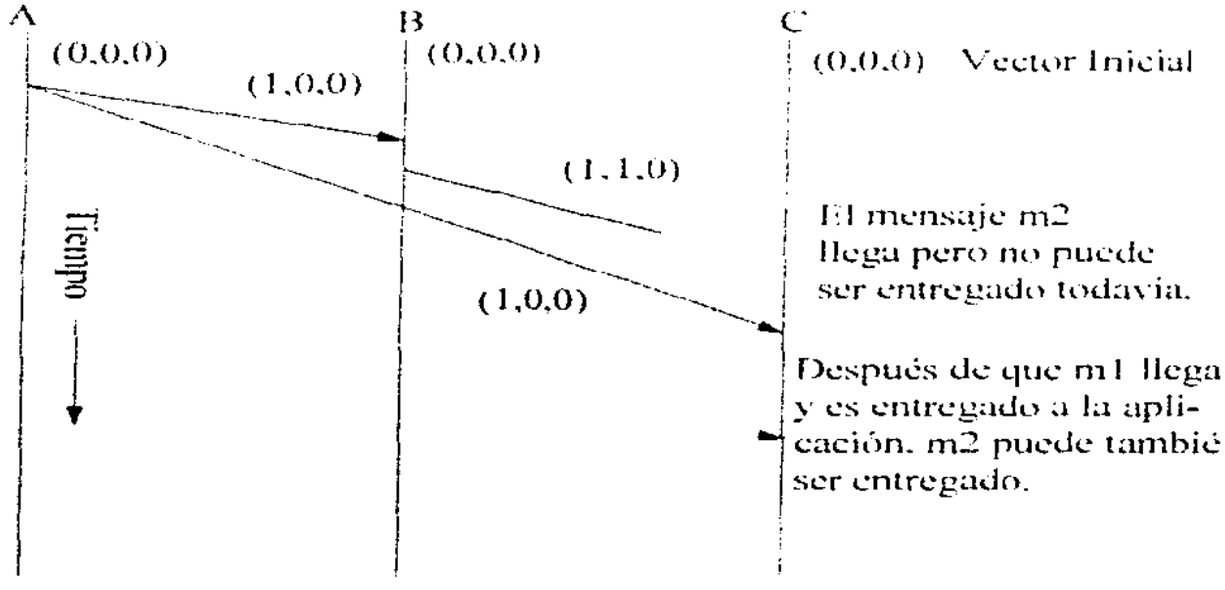


Figura 3.2. Mensajes relacionados causalmente.

Cuando un proceso tiene un mensaje a enviar, este incrementa su propia posición en su vector y envía el vector como parte del mensaje. Cuando *m1* en la figura 3.2 llega a *B*, se realiza una revisión para ver si este depende de algo que *B* no haya visto todavía. El primer componente del vector es un número mayor por uno que el primer componente del vector de *B*, lo cual es esperado (y requerido) para un mensaje proveniente de *A*, y los

otros son iguales, por lo que el mensaje es aceptado y pasado al miembro del grupo que corre en *B*.

Ahora *B* envía un mensaje por su cuenta, *m2*, a *C*, el cual llega antes que *m1*. A partir del vector, *C* ve que *B* ha recibido ya un mensaje proveniente de *A* antes que *m2* fuera enviado y por lo tanto no ha recibido nada de *A* todavía. *m2* es almacenado hasta que un mensaje proveniente de *A* llegue. Bajo ninguna condición puede éste ser entregado antes que el mensaje de *A*.

El algoritmo general para decidir si se pasa un mensaje que llega al proceso usuario o se retrasa puede ser establecido de la siguiente manera. Supóngase que *Vi* es el *i-ésimo* componente del vector en el mensaje que llega y que *Li* es el *i-ésimo* componente del vector almacenado en la memoria del receptor. Supóngase que el mensaje fue enviado por *j*. La primera condicion para que sea aceptado es *Vj =Lj + 1*. Esto simplemente indica que este es el siguiente mensaje de la secuencia proveniente de *j*, esto significa que no han faltado mensajes. (Los mensajes provenientes de un mismo iniciador estan siempre causalmente relacionados). La segunda condición para que los mensajes sean aceptados es que *Vi =Li* para toda *i* diferente de *j*. Esta condición establece que el iniciador no ha visto algún mensaje que al receptor le haya faltado. Si un mensaje que llega pasa ambas pruebas, el sistema de tiempo de ejecución puede pasar el mensaje al proceso usuario sin retrasarlo. En caso contrario este debe esperar.

En la figura 3.3 se muestra un ejemplo más detallado del mecanismo de vector. Aquí el proceso *0* ha enviado un mensaje que contiene el vector (4,6,8,2,1,5) a otros 5 miembros de su grupo. El proceso *1* ha visto exactamente los mismos mensajes que el proceso *0*, por lo que el mensaje pasa la prueba, es aceptado y puede ser pasado al proceso usuario. Al proceso *2* le falta el mensaje *6* enviado por el proceso *1*, por lo que el mensaje que llega debe ser retrasado. El proceso *3* ha visto todo lo que el iniciador ha visto y además el mensaje *7* proveniente del proceso *1*, el cual aparentemente no ha llegado al proceso *0*, por lo que el mensaje es aceptado. Al proceso *4* le falta el mensaje anterior proveniente del mismo proceso *0*. Esta omisión es seria, por lo que el nuevo

mensaje tendrá que esperar. Finalmente el proceso *5* esta ligeramente adelantado al proceso *0*, por lo que el mensaje puede ser aceptado inmediatamente.

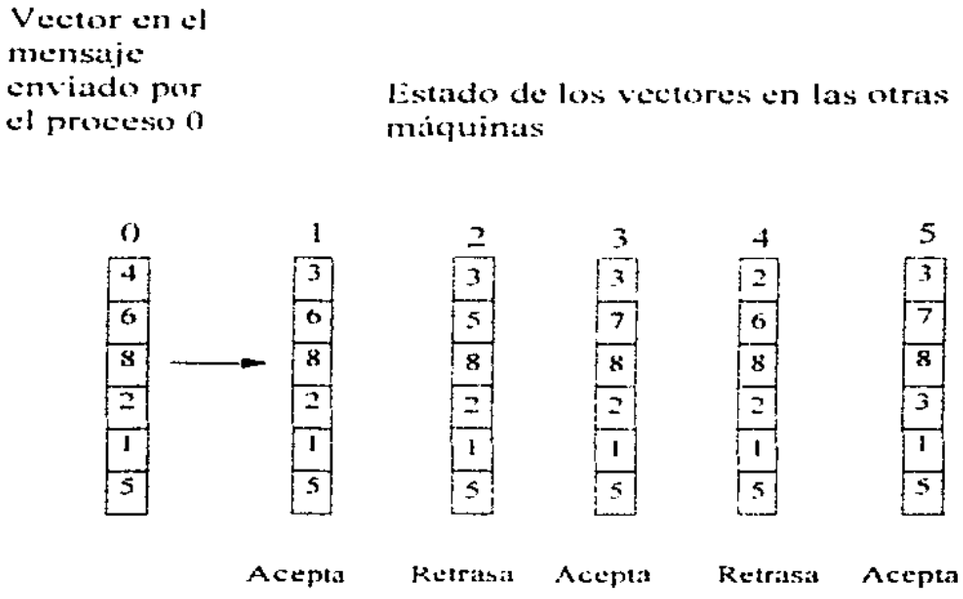


Figura 3.3 Ejemplo de vectores usados para *CBCAST*

3.3. Protocolos de multitransmisión ordenada

Si dos sitios transmiten un mensaje a grupos de destinos que coinciden, es posible que estos mensajes lleguen a destinos comunes en diferentes ordenes. La característica esencial de un protocolo de multitransmisiones, es por lo tanto, que un mensaje sea entregado sólo cuando todos los recipientes lleguen a un acuerdo en como ordenar su entrega con relacion a los otros mensajes. Esto normalmente aumenta la tardanza, provocando comunicación adicional y requiere que el mensaje sea almacenado el tiempo que dure el protocolo.

El siguiente protocolo (BJ87) opera asignando a cada multitransmision una etiqueta de tiempo y entregando los mensajes en el orden de las etiquetas de tiempo. (Estas etiquetas no necesitan tener relacion con el tiempo real (todo lo que necesita es una secuencia de números que se incrementa). Cuando un sitio recibe un nuevo mensaje lo almacena en una cola de pendientes y lo marca como no entregable. Luego envía un mensaje al iniciador con una etiqueta de tiempo propuesta para el mensaje. Esta etiqueta de tiempo es escogida para ser la mas grande que cualquier otra etiqueta de tiempo que este sitio haya propuesto o recibido en el pasado (para hacer la etiqueta de tiempo única, a cada sitio se le asigna un unico número que el añade a sus etiquetas de tiempo como un sufijo). El iniciador colecta las etiquetas de tiempo de todos los recipientes, escoge el más grande de los valores que recibe, y envía este valor de regreso a los recipientes. Esta se vuelve la etiqueta de tiempo final para la multitransmisión. Cuando un recipiente recibe una etiqueta de tiempo final este asigna la etiqueta de tiempo al mensaje correspondiente en la cola de pendientes y marca el mensaje como entregable. La cola de pendientes es entonces reordenada para quedar en orden creciente de etiquetas de tiempo. Si el mensaje al inicio de la cola de pendientes es entregable, entonces es sacado de la cola y entregado. Esto se repite hasta que la cola se vacía o el mensaje al inicio de la cola es no entregable. Si hay mensajes entregables despues de este no entregable, estos permanecen en la cola hasta que los mensajes adelante de ellos hayan sido todos entregados o movidos después de ellos en la cola.

La siguiente figura muestra como funciona este protocolo. Asúmase que tres sitios (procesos) están tratando de multitransmitir los mensajes *m1*, *m2* y *m3* al mismo conjunto de destinos, los sitios 1, 2 y 3. Asúmase que las etiquetas de tiempo más grandes vistas en los sitios 1, 2 y 3 son 14, 15 y 16 respectivamente. El paso 1 muestra los mensajes llegando a los recipientes en diferentes órdenes. Todos son colocados en la cola de pendientes y marcados como no entregables (*u*) con etiquetas de tiempo propuestas como se muestra. Es importante notar que el número de sitio es usado para eliminar ambigüedades con respecto a las etiquetas de tiempo iguales.

En el paso 2 el iniciador de *m1* toma sus etiquetas de tiempo propuestas (16.1, 17.2 y 17.3), y calcula el máximo (17.3) y envía su valor a los recipientes como su etiqueta final de tiempo. Los recipientes mandan el mensaje como entregable (*d*) y reordenan su cola de pendientes.

El paso 3 muestra la cola de pendientes después de que el iniciador de *m2* envía su etiqueta de tiempo final, y el paso 4 muestra la cola después de que el iniciador de *m3* hace lo mismo. En este momento todos los mensajes pueden ser sacados de la cola de pendientes y entregados. Se observa que los mensajes son entregados en todos los sitios en el orden *m1*, *m3* y después *m2*, que fue el orden de sus etiquetas de tiempo finales.

El protocolo *ABCAST* asigna a cada multitransmisión una etiqueta de tiempo final única y todos los mensajes son entregados en el orden de sus etiquetas de tiempo final. Esto asegura que la multitransmisión será entregada en el mismo orden en todos los destinos. Debido a que el iniciador escoge la más grande de todas la etiquetas de tiempo cambiando la etiqueta de tiempo de un mensaje de su etiqueta propuesta a la final solo puede causar que este sea movido atrás de otros mensajes en la cola de pendientes y nunca adelante de ellos. Por esto un mensaje puede tener que esperar a que otros mensajes sean entregados antes de que sea entregado pero nunca se presentara una situación donde sea necesario entregar un mensaje antes de uno que haya sido sacado de la cola de pendientes y entregado (lo cual provocaría que el protocolo fallara).

Este protocolo tiene un costo asociado: Primero, cada mensaje no puede ser entregado tan pronto como es recibido, este tiene que permanecer en la cola de pendientes hasta al menos una segunda vuelta de intercambio de mensajes haya ocurrido y haya sido asignada una etiqueta de tiempo definitiva, tambien se tiene que esperar a que todos los mensajes con etiquetas menores de tiempo sean entregados, lo que representa un costo por retardo. Segundo, cada multitransmisión produce una mayor comunicación más allá del hecho de enviar los mensajes a cada sitio destino. Cada recipiente también debe enviar su etiqueta de tiempo propuesta de regreso al iniciador y el iniciador debe responder a todos ellos con la etiqueta de tiempo final. Finalmente, el mensaje debe ser salvado en la cola de pendientes desde el momento en que es recibido hasta el momento en que es entregado. Esto representa el costo de almacenamiento. Realmente el costo de almacenamiento es aún mayor. Alguna información acerca del mensaje tiene que ser retenida en cada recipiente hasta que se sepa que este ha sido entregado en todos los destinos.

Sitio 1

m_3	m_1	m_2	
15.1	16.1	17.1	. . .
u	u	u	

Sitio2

m_2	m_1	m_3	
16.2	17.2	18.2	. . .
u	u	u	

Sitio3

m_1	m_3	m_2	
17.3	18.3	19.3	. . .
u	u	u	

Paso 1

m_3	m_2	m_1	
15.1	17.1	17.3	. . .
u	u	d	

m_2	m_1	m_3	
16.2	17.3	18.2	. . .
u	d	u	

m_1	m_3	m_2	
17.3	18.3	19.3	. . .
d	u	u	

Paso 2

m_3	m_1	m_2	
15.1	17.1	19.3	. . .
u	d	d	

m_1	m_3	m_2	
17.3	18.2	19.3	. . .
d	u	d	

m_3	m_2		
18.3	19.3		. . .
u	d		

Paso 3

m_1	m_3	m_2	
17.3	18.3	19.3	. . .
d	d	d	

m_3	m_2		
18.3	19.3		. . .
d	d		

m_3	m_2		
18.3	19.3		. . .
d	d		

Paso 4

4. Desarrollo de una aplicación utilizando *ISIS*

4.1. La herramienta de programación ISIS

Isis es una herramienta que fue desarrollada en la universidad de Cornell para la programación distribuida y proporciona un conjunto de mecanismos de alto nivel que sirven para crear y manejar grupos de procesos.

Isis incluye herramientas basadas en los paradigmas más importantes del cómputo distribuido con la finalidad de poder manejar comunicaciones asincronas. A continuación se mencionan algunas de ellas:

Grupos de procesos: creación, borrado y unión (transferencia de estado).

Multitransmisión al grupo: multitransmisiones CBCAST, ABCAST, con número de respuestas a recolectar variable.

Sincronización: candados, basados en las estafetas distribuidas. La detección o la evasión del abrazo mortal (deadlock) debe ser contemplada por el programador.

Información replicada: implementada a través de multitransmisiones de las actualizaciones. Transferencia de valores a los procesos que se unen al grupo usando la herramienta de transferencia de estado. Reconfiguración dinámica del sistema usando información de la configuración replicada.

Herramientas de monitoreo: monitoreo de un proceso o un sitio, disparo de acciones después de fallas o recuperaciones. Monitoreo de los cambios en la membresía de un grupo de procesos, fallas en los sitios, etc.

Herramientas para la ejecución distribuida: Calculos redundantes (todos los miembros del grupo realizan la misma acción). Uso del enfoque servidor principal y sus respaldos.

Recuperación automática: inicialización automática de los programas cuando un sitio se recupera. Almacenamiento del estado del grupo en un archivo bitacora para el primer sitio que se recupera e inicio de una transferencia de estado para los demás miembros que se unen al grupo.

Comunicación en una WAN: paso de mensajes y transferencia de archivos a través de redes de area amplea.

Actualmente se han desarrollado interfaces para Isis en C, C++, Fortran, Common Lisp, Ada y Smalltalk. Existen puertos para Isis en estaciones de trabajo UNIX y de los principales mainframes, asi como para Mach, Chorus, ISC y SCO UNIX, el sistema DEC VMS y el sistema operativo Linx de Honeywell. Los datos dentro de los mensajes son representados en el formato binario usado por la maquina que lo envia, y convertido al formato de la máquina destino después de la recepción de manera automátea y transparente.

A continuación se presentan algunas de las ideas principales para programar en *ISIS*.

Una aplicación tipica en *ISIS* tiene partes que corren en varias máquinas diferentes y puede ser expandida para incluir más máquinas o quitarse de algunas máquinas aún cuando la aplicación este corriendo. La razón para distribuir una aplicación en varias máquinas puede ser compartir el trabajo, para obtener un tiempo de respuesta menor, o poder continuar la operación a pesar de la falla de algunas máquinas. La siguiente aplicacion muestra las caracteristicas básicas de *ISIS*, y muestra como escribir un programa distribuido que se puede expander dinámicamente y es tolerante a fallas.

La aplicación consiste de un servicio distribuido llamado "tarjeta de tiempo" para una organización con varios departamentos. En esta, la organización contrata un número

de empleados temporales, quienes pueden trabajar en varios departamentos diferentes en una semana dada, cubriendo el exceso de trabajo donde y cuando es requerido. Cada departamento de manera independiente graba el número de horas que cada empleado temporal trabaja en ese departamento. El objetivo del servicio de tarjeta de tiempo es permitir que alguien de al servicio el nombre de un empleado y obtener el número de horas que el empleado ha trabajado en los diferentes departamentos la semana anterior. (Al empleado se le pagaría supuestamente con base en número de horas). El servicio de tarjeta de tiempo tendrá que buscar el número de horas trabajadas por un empleado a través de los registros de los departamentos individuales antes de dar su respuesta.

Si la organizacion tiene un numero de estaciones de trabajo conectadas a través de una red, este servicio podria ser implementado para correr en varias máquinas en lugar de correr en una única máquina. Una ventaja es que los registros de los diferentes departamentos pueden ser inspeccionados en paralelo en varias máquinas en lugar de uno después de otro en una única máquina. Esto implica que las consultas serán contestadas más rápido. Una segunda ventaja es que se puede asegurar que el servicio permanece disponible aún cuando algunas de las máquinas no estén operando debido a una falla o por que han sido sacadas de la linea para mantenimiento o por otras razones. Se asume que cada departamento mantiene sus registros en un archivo llamado *departamento1*, *departamento2*, *departamento3* y así sucesivamente, y que cada registro es simplemente una línea con el nombre del empleado seguido por el número de horas que él trabajó. También se asume que estos archivos están disponibles en todas las máquinas en las cuales el servicio corre.

4.2. Grupos de procesos y multitransmisión

Uno de los mecanismos básicos que *ISIS* proporciona es un medio para agrupar procesos y nombrarlos como una unidad. Un grupo de procesos puede contener un único miembro, pero normalmente consistirá de un número de procesos que residen en máquinas que se encuentran en cualquier lugar del sistema. La membresía de un grupo de procesos puede cambiar con el tiempo, conforme nuevos procesos se unen al grupo o los miembros existentes lo abandonan, ya sea porque ellos lo decidieron o debido a una falla en alguna parte del sistema. Un proceso puede ser miembro de uno o más grupos de procesos.

ISIS también proporciona un mecanismo de multitransmisión que permite enviar un mensaje de un proceso a un grupo de procesos. Para hacer esto el proceso que envía el mensaje le pide a *ISIS* que busque ("*look up*") el nombre del grupo de procesos y obtiene una dirección ("*address*"). Luego realiza la multitransmisión dando esta dirección, el mensaje y otra información relevante como argumentos. El efecto de esto es enviar una copia del mensaje a cada uno de los miembros actuales del grupo de procesos.

Todos los miembros eventualmente recibirán el mensaje, aunque ellos lo pueden recibir en diferentes tiempos.

El mecanismo de multitransmisión también permite al recipiente del mensaje enviar una respuesta. Un proceso que realiza una multitransmisión puede indicar que quiere esperar por un número específico de respuestas o que quiere una respuesta de todos los recipientes del mensaje. La llamada a la función de multitransmisión regresa cuando el número requerido de respuestas ha sido recibido. Si el grupo no es lo suficientemente grande o si varios recipientes han terminado (posiblemente debido a una falla) y el número requerido de respuestas no puede ser colectado, *ISIS* colectará el mayor número de respuestas que le sea posible y notificará al iniciador de esto. Este mecanismo de respuesta permite a la multitransmisión de *ISIS* ser usada como un mecanismo general de llamadas a procedimientos remotos.

La razón más común para hacer a un conjunto de procesos un grupo de procesos es permitir la multitransmisión de mensajes al grupo como un todo, aun cuando su membresia pueda estar cambiando. Como un caso especial, la forma más simple de enviar un mensaje a un proceso individual es hacerlo miembro de un grupo de procesos que lo contiene a él mismo, y realizar la multitransmición a este grupo. Si un proceso recibe mensajes como un miembro individual y también como un miembro de un grupo de procesos, se puede hacer miembro de los dos grupos de procesos. Los grupos de procesos en *ISIS* son economicos, por lo que esto no debe de traer problemas de desempeño.

ISIS proporciona herramientas fáciles de usar que permiten a los miembros de un grupo de procesos accesar a la información compartida o replicada, para realizar ciertas formas de ejecución distribuida coordinada, y poder tolerar o recuperarse de fallas.

Los grupos de procesos proveen una forma conveniente de dar un nombre abstracto al servicio implementado por los miembros del grupo. Los otros procesos interactuan con el servicio usando el nombre del grupo y la multitransmisión, no tienen que preocuparse de la membresia real del grupo. Ésto implica que el grupo que implementa el servicio, puede crecer, hacerse más chico o moverse a otras máquinas, o se le pueden añadir nuevas capacidades sin ninguna interrupción del servicio, sin que los usuarios del servicio se preocupen por estos cambios. Es esta caracteristica la que permite el desarrollo de aplicaciones modulares, dinámicamente expandibles y tolerantes a fallas.

4.3. Tareas y entradas

Los procesos en *UNIX* tienen un espacio de direcciones privado dentro del cual un único hilo de control vive. Algunos sistemas operativos más nuevos como MACH han introducido la noción de tareas ligeras que coexisten con un único proceso, compartiendo el espacio de direcciones. Aunque *ISIS* fue construido sobre *UNIX* se necesito un mecanismo de tareas para implementar el sistema. Por lo que un proceso en una aplicación de *ISIS* esta internamente estructurado en un número de tareas. Una tarea en *ISIS* parece una funcion en *C* y comparte el mismo espacio de direcciones y variables globales como otras tareas y funciones en el proceso. La diferencia es que una tarea (y no sólo la funcion llamada *main*) puede ser invocada por el sistema y empezar a ejecutarse en respuesta de ciertos eventos, el más comun de ellos es la entrega de un mensaje. Una tarea que es iniciada en respuesta de la entrega de un mensaje es llamada una "entrada" (*entry*). Un proceso puede tener varias entradas y a cada una se le da un numero de entrada diferente. Cuando un mensaje es enviado este es direccionado a un número de entrada particular. En la entrega un apuntador al mensaje es pasado como un parámetro a la entrada, el cual normalmente lee el contenido del mensaje y actúa de acuerdo a éste.

La programación con tareas (*ISIS*) no es muy diferente a la programacion con funciones en C, excepto por tres cosas. La primera es que se necesita ligar bibliotecas especiales. Otra es que cuando una tarea realiza ciertas llamadas al sistema de *ISIS*, es posible que una nueva tarea se inicie y empiece a ejecutarse antes de que la llamada al sistema regrese. La tarea original después continuara donde se quedo. Como ejemplo considérese un tarea que hace una llamada al sistema para realizar un multitransmisión y ahora esta esperando las respuestas. Si un nuevo mensaje llega antes que las respuestas, otra tarea (una entrada) puede iniciarse para encargarse del nuevo mensaje aun cuando la primera tarea no haya terminado. Normalmente esto no causa problemas pero si la segunda tarea cambia los valores de las variables globales, entonces la primera tarea tiene que estar consciente del hecho de que sus valores pueden cambiar entre el momento en que realiza una llamada al sistema para una multitransmisión y cuando la llamada al sistema regresa. Las llamadas al sistema que permiten a otras tareas iniciarse antes de que

regrese son llamadas "bloqueadas" ya que bloquean la ejecución de la tarea que realizo la llamada.

4.4. Monitoreo de eventos

Un proceso puede indicarle a *ISIS* que le notifique cuando cierto tipo de eventos ocurra. Esto lo hace dándole a *ISIS* el nombre de la tarea a invocar (y un argumento para pasárselo a la tarea) cuando este tipo de evento se presente. Entre los tipos de eventos que pueden ser monitoreados están los cambios en la membresía del grupo, terminación de procesos y fallas de sitios. Esta facilidad permite la repartición de la carga de trabajo cuando un nuevo miembro se une o abandona el grupo, o para hacerse cargo del trabajo de un proceso o un sitio que ha fallado, entre otras cosas.

4.5. El servicio de tarjeta de tiempo

El servicio de tarjeta de tiempo será implementado como un grupo de procesos que consiste de un número de procesos que corren en diferentes máquinas. El grupo se llama *servicio_tiempo*. Todos los miembros del servicio de tarjeta de tiempo ejecutan el mismo programa. El cual será llamado el programa "servicio". Otro programa llamado "consulta" es usado para consultar el servicio.

La figura 4.1 muestra una parte del programa servicio. El valor *<port no>* es el numero de puerto usado por *ISIS* para hablar a las aplicaciones. Si el número de puerto es 0, entonces *ISIS* primero buscara la variable de ambiente *ISISPORT* y usa este número si lo encuentra. Si no *ISIS* buscará en el archivo /etc/services.

```
#include "../include/isis.h"
#include <stdio.h>
#define       ENTRADA_CONSULTA     1

address*dirgrp_p;
int               mi_indice;
int               mi_depto=0;


main()
{
        int  tarea_principal();
        int  cambio_grupo();
        int  recibe_consulta();

        isis_remote_init ((char*)0, 0, 0, 0);

        /* Declare tasks and entry points */
        isis_task (tarea_principal, "tarea_principal");
        isis_task (cambio_grupo, "cambio_grupo");
        isis_entry (ENTRADA_CONSULTA, recibe_consulta, "recibe_consulta");

        isis_mainloop (tarea_principal);
}
```

```
tarea_principal()
{
      int    cambio_grupo();

      /*Se une al grupo de procesos y monitorea los cambios en la membresia*/

         dirgrp_p = pg_join ("servicio_tiempo", PG_MONITOR, cambio_grupo, 0, 0);

      isis_start_done();

}

cambio_grupo (gview_p, arg)
 groupview *gview_p;
 int  arg;
{
        int i;
        i=0;
        while (!addr_ismine (&gview_p->gv_members[i]))
                i++;
        mi_indice=i+1;
        mi_depto=my_indice;
}
```

Figura 4.1

En una aplicación con *ISIS* la función *main* normalmente inicializa *ISIS*, declara tareas y entradas, y establece el ciclo principal. El argumento de *isis_mainloop* es la primera tarea que corre. La primera cosa que hace *tarea principal* es unirse al grupo de procesos *servicio tiempo* y establecer un monitor para los cambios en la membresía. El primer argumento de *pg join* es el nombre del grupo, y el último argumento es siempre un 0. Entre estos dos argumentos se puede especificar un número opcional de palabras clave y los argumentos correspondientes a éstas. En el programa la palabra clave *PG_MONITOR* especifica que la tarea *cambio grupo* va ha ser llamada con la nueva vista del grupo y el argumento dado (en este caso 0) siempre que la membresía del grupo cambie. La función *pg join* regresa la dirección del grupo, la cual es almacenada en la variable global *dirgrp p*. Cuando la tarea *main* se inicia, *ISIS* inhibe la entrega de nuevas solicitudes provenientes de otros procesos. Esto asegura que se puede hacer toda la

inicialización antes de que se pida responder a otros eventos como mensajes que llegan. La llamada *isis_start_done* le dice al sistema que la secuencia de inicialización ha sido terminada.

Los procesos en *ISIS* pueden estar activos aun cuando no haya tareas activas dentro de ellos. *ISIS* fue diseñado asumiendo, que las tareas se van ha iniciar, hacer el trabajo que se supone deben hacer, y finalmente terminar(a través de regresar). Esto se aplica a todas las tareas incluso *main*. Un proceso que no tiene tareas activas de hecho esta esperando trabajo para realizar en *isis_mainloop*. La rutina *cambio_grupo* es llamada en cada miembro del grupo (todos los miembros están ejecutando la misma pieza de código) siempre que la membresía del grupo cambie. Las rutinas que monitorean la membresía del grupo son siempre llamadas con un apuntador a la estructura vista del grupo como su primer argumento. La estructura *groupview* contiene información acerca del grupo de procesos. En particular, contiene una lista de las direcciones de todos los miembros actuales *gv-members*. La estructura de vista de grupo siempre ordena esta lista de acuerdo al "rango" de los miembros. El miembro más viejo en el grupo tiene el rango 0, el segundo mas viejo el rango 1 y así sucesivamente. Por lo que el rango puede ser usado para dar a cada miembro un único indice que lo distingue de los demás. En el programa *mi_indice* contiene el valor del rango + 1. La tarea *cambio_grupo* es invocada en un proceso cuando el proceso se une al grupo por primera vez (esto también es considerado un cambio en la membresía), por lo que cuando *isis_start_done* es llamada *mi_indice* tendrá un valor definido. Este indice cambiara cada vez que un miembro se una o abandone el grupo.

Para recibir un mensaje que llega se debe definir una entrada. La sentencia *isis_entry* declara esta entrada, dándole el número de entrada *ENTRADA_CONSULTA*. A continuación se muestra el código para esta entrada. Se asume que un mensaje de consulta que llega contiene una cadena dando el nombre de un empleado. Por el momento se asume que hay al menos tantos miembros en el grupo de procesos como departamentos hay, y que cada miembro es responsable de buscar a través del archivo cuyo número de departamento este en su variable *mi_depto*. Los miembros extra tendrán

el valor de 0 en *mi depto* y no harán nada. Más adelante se verá como deciden los miembros de que departamentos son responsables y como manejan la situación en la que hay menos miembros que departamentos.

```
recibe_consulta(msg_p)
message *msg_p;
{
        char nombre_consulta[NOMBREL_MAX], nombre[NOMBREL_MAX];
        int horas_consulta, horas;

        if(mi_depto != 0)
        {
                /*Lee en el mensaje el nombre del empleado*/
                msg_get (msg_p,"%s", nombre_consulta);

                /*Busca a través de los archivos el número de horas*/
                /*trabajadas en mi_depto y las almacena en horas_consulta*/

                /*Envia el mensaje de respuesta*/
                reply (msg_p, "%d%d", mi_depto, horas_consulta);
        }
        else
                /*No soy responsable de algún departamento*/
                reply(msg_p, "%d%d", 0, 0);
}
```

Figura 4.2

Una entrada es llamada cuando un mensaje direccionado a su número de entrada llega al proceso. Su primer argumento es un apuntador al mensaje. Este apuntador puede ser usado para leer la información del mensaje usando *msg_get*, que tiene una interfase similar a *fscanf*. En este caso, lee la cadena de caracteres del mensaje y la almacena en *nombre_consulta*.

La figura 4.3 muestra el programa de consulta. Tiene una secuencia de inicio similar al programa al programa de servicio y simplemente lee un nombre de la terminal, envia un mensaje al grupo de procesos *servicio_tiempo* e imprime las respuestas.

```
#include        "../include/isis.h"
#define         ENTRADA_CONSULTA    1
#define         NOMBREL_MAX         64
#define         NDEPTOS             5

main()
{
        int     consulta_tprincipal();

        isis_remote_init ((char*)0, 0, 0, 0);

        isis_task (consulta_tprincipal, " consulta_tprincipal ");

        isis_mainloop (consulta_tprincipal);
}

consulta_tprincipal()
{
        address*dirgrp_p;
        char            nombre[NOMBREL_MAX];
        int             depto[NDEPTOS], horass[NDEPTOS];
        int             i, valret;

        /*Encuentra la dirección del grupo de procesos servicio_tiempo*/
        dirgrp_p=pg_lookup("servicio_tiempo");
        if(addr_isnull(dirgrp_p))
        {
                printf("Lo siento!, el servicio no esta disponible\n");
                exit();
        }
        isis_start_done();

        /*Ciclo de consultas*/
        printf("De el nombre del empleado(^C para salir):");
        while(scanf("%s", nombre)==1)
        {
                /*Envia el mensaje que contiene el nombre y */
                /*colecta las respuestas*/
                do
                {
                        valret=bcast(dirgrp_p, ENTRADA_CONSULTA, "%s", nombre,
                                        ALL, "%d%d", depto, horas);

                        if(valret<=0)
                        {
```

```
                        isis_perror("Lo siento! beast error");
                        exit();
                }
        }
while(isis_nreplies != isis_nsent);

printf("Tarjeta de tiempo para %s (En base a  %d respuestas):\n",
nombre, isis_nreplies);
printf("    Departamento.        Horas\n");
for(i=0; i<isis_nreplies; i++)
        if(depto[i])
                printf("%8d%8d\n", depto[i], horas[i]);

printf("\n De el nombre del empleado (ctrl C to quit): ");

}
/* Salir terminando éste proceso*/
exit(0);
}
```

Figura 4.3

Es importante notar el uso de *pg_lookup* para obtener la dirección del grupo de procesos. La parte mas importante del código es la llamada a *beast* para enviar el mensaje y recoger las respuesta. La llamada primero especifica la dirección y el número de entrada, seguidos por una descripción de la información que será puesta en el mensaje (de una manera similar a *fprintf*). Después viene el número de respuestas que se desean. La constante *ALL* especifica que se quiere una respuesta de todos los procesos a los cuales se les envío una copia del mensaje. Esto es seguido por una descripción de donde poner la información que es leída de las respuestas (De manera similar a un *fscanf*). A diferencia de *fscanf* cada elemento es un apuntador a un arreglo del tipo especificado, porque podría haber mas de un mensaje de respuesta. La información de cada respuesta va dentro de cada elemento del arreglo. La llamada a *beast* en el programa de consulta es similar a las llamada *msg_get* y *reply* en el programa del servicio.

Cuando una llamada a *beast* regresa, la variable global *isis_nsent* contiene el número de procesos a los que les fue enviada una copia del mensaje e *isis_nreplies* contiene el número de respuestas colectadas. (*isis_nreplies* es el valor que regresa *beast*,

a menos que un error ocurra, en cuyo caso regresa el código de error). En este ejemplo los dos valores de regreso son normalmente iguales. Sin embargo, si un proceso al que fue enviado el mensaje termina antes de responder (posiblemente debido a una falla), *ISIS* detecta esto y la llamada a *bcast* regresa sin colectar la respuesta de este miembro. Un proceso que termina es automáticamente sacado de todos los grupos de procesos a los que pertenece, por lo que el programa de consulta vuelve a solicitar la multitransmisión si esto ocurre. Esta siguiente vez el mensaje será enviado a la nueva membresía del grupo y al menos que más procesos terminen se recibirá una respuesta de todos ellos.

Por el momento se asume que hay tantos miembros como departamentos hay, a continuación se muestra como se asignan los departamentos a los miembros. Asúmase que *NDEPTOS* es el número de departamentos, anteriormente se asigno un índice único a cada miembro (*mi_indice*). Una regla simple sería asignar al miembro con índice i como responsable del departamento *i*, para *i* = *NDEPTOS*. Un miembro con un índice *i* > *NDEPTOS* no hace nada a menos que un miembro activo salga del grupo (posiblemente debido a una falla). Si el índice de un miembro previamente inactivo se vuelve menor o igual a *NDEPTOS*, entonces este empezara a participar en la búsqueda. Estos procesos son llamados en espera. Los procesos en espera hacen que una aplicación sea tolerante a fallas, y son usados frecuentemente en *ISIS*. Esta aplicación tolerara tantas fallas como procesos en espera existan.

El código completo del programa de servicio se muestra en la figura. Siempre que la membresía del grupo cambia, cada miembro vuelve a calcular *mi_indice* y abre el archivo *departamento* (y cierra cualquier otro anteriormente abierto).

```
#include      "../include/isis.h"
#define       ENTRADA_CONSULTA    1
#define       NOMBREL_MAX         64
#define       NDEPTOS             5

address*dirgrp_p;
int           mi_indice;
int           mi_depto;
```

```
FILE            *mi_archivo;

main()
{
        /* Permanece igual*/
}


tarea_principal()
{
        /* Permanece igual*/
}


cambio_grupo (gview_p, arg)
 groupview *gview_p;
 int  arg;
{
        char    nombre_archivo[16];
        int i;

        /* Calcula un indice único para éste miembro */
        i=0;
        while (!addr_ismine (&gview_p->gv_members[i]))
                i++;
        mi_indice=i+1;

        /* Si existe un archivo abierto entonces lo cierra */
        if(mi_depto!=0)
                fclose (mi_archivo);

        if(mi_indice<=NDEPTOS)
        {
        mi_depto=mi_indice;

        /*Abre el archivo de importancia*/
        sprintf(nombre_archivo, "departamento%d", mi_depto);
        mi_archivo=fopen(nombre_archivo,"r");
                if(mi_archivo==NULL)
                {
                        printf("No pudo abrir el archivo %s\n", nombre_archivo);
                        exit();
                }
```

```
        }
        else
                mi_depto=0;
}
recibe_consulta(msg_p)
message *msg_p;
{
        char nombre_consulta[NOMBREL_MAX], nombre[NOMBREL_MAX];
        int horas_consulta, horas;

        if(mi_depto != 0)
        {
                /*Lee en el mensaje el nombre del empleado*/
                msg_get (msg_p,"%s", nombre_consulta);

                /*Busca a través de los archivos el número de horas*/
                /*trabajadas en mi_depto y las almacena en horas_consulta*/
                horas_consulta=0;
                rewind(mi_archivo);
                while (fscanf (mi_archivo, "%s %d", nombre, &horas)==2)
                        if(strcmp (nombre_consulta, nombre) == 0)
                        {
                                horas_consulta +=horas;
                                break;
                        }
                /*Envia el mensaje de respuesta*/
                reply (msg_p, "%d%d", mi_depto, horas_consulta);
        }
        else
                /*No soy responsable de algun departamento*/
                reply(msg_p, "%d%d", 0, 0);
}
```

Figura 4.4

A partir de este momento se tiene una implementación completa del servicio de tarjeta de tiempo. Se pueden iniciar varias instancias del programa servicio, posiblemente en máquinas diferentes, y varias instancias del programa de consulta si se desea, posiblemente también en diferentes máquinas y empezar a realizar consultas. Si se desea probar la tolerancia a fallas se podría añadir un ciclo al programa de consulta, para que por cada nombre que se teclee sea multitransmitida la misma consulta, una y otra vez, por ejemplo 50 veces. Después mientras el servicio esta siendo consultado repetidamente, se pueden añadir nuevas instancias del servicio en cualquiera de las máquinas o simular una falla a través de matar a los miembros existentes. Mientras haya una cantidad de

miembros igual a *NDEPTOS* el servicio continuará operando correctamente. Con la finalidad de que el sistema opere correctamente aún cuando el número de miembros sea menor que *NDEPTOS*, se pueden realizar algunos cambios simples, en este caso algunos de los miembros se harán cargo de más de un departamento. En este caso las variables *my dept* y *my file p* se vuelven arreglos se vuelven arreglos, y las respuestas llevan arreglos también. Es importante notar los cambios en *bcast*, *msg get* y *reply* para manejar arreglos.

4.6. Transferencia de estado

La regla que se utilizó para dividir el trabajo tiene una gran desventaja. Cada vez que la membresía del grupo cambia, un miembro se puede volver responsable de un grupo de departamentos completamente nuevos, esto es, tiene que cerrar todos sus archivos y abrir los nuevos. En una implementación real, partes del archivo (o todo el archivo) se leería en la memoria principal y se accesaría rápidamente a través de estructuras construidas para buscar a través de la información. Esto se tendría que volver a hacer cada vez que la membresía cambie. Por esta razón se considerara otra forma de dividir el trabajo que elimina la reasignación innecesaria. Cuando un miembro abandona un grupo, sus departamentos se le asignan al miembro responsable del menor número de departamentos. Un miembro que se une al grupo se hace cargo de la mitad de los departamentos del miembro con mas departamentos. Para realizar la nueva asignación bajo esta regla, no es suficiente que un miembro sepa solo de sus departamentos asignados a él. Cada miembro tiene que saber las asignaciones de todos los miembros. A continuación se muestra el código con las modificaciones necesarias.

```
#include "stdio.h"
#include "../include/isis.h"
#define         ENTRADA_CONSULTA    1
#define         NOMBREL_MAX         64
#define         NDEPTOS             5
#define         MIEMBROS_MAX   10

int     n_asignaciones;
struct
{
        address     su_dir;
        int         sus_ndeptos;
        int         su_depto[NDEPTS];
}       asignacion[MIEMBROS_MAX];

address*dirgrp_p;
int             mi_indice;
int             mis_ndeptos=0;
int             mi_depto[NDEPTOS];
FILE            *mi_archivo[NDEPTOS];
```

```
main()
{
        /* Permanece igual*/
}

tarea_principal ()
{
        /* Permanece igual*/

}


cambio_grupo (gview_p, arg)
 groupview *gview_p;
 int  arg;
{
        char            nombre_archivo[16];
        int             n_miembros;
        int             i, menor_i, fallo_i, nmenor, mayor_i, union_i, nmayor, pasar;
        addressmenor_dir, mayor_dir, *fallo_dir_p, *union_dir_p;

        /* Calcula un único índice para éste miembro */
        i=0;
        while (!addr_ismine (&gview_p->gv_members[i]))
        i++;
        mi_indice= i+1;

        /* Guarda el número de miembros en el grupo*/
        n_miembros= gview_p->gv_nmemb;

        union_dir_p=&gview_p->gv_joined;

        fallo_dir_p=&gview_p->gv_departed;

     if(n_asignaciones <n_miembros)
        {
                if(n_asignaciones ==0)
                {
                        asignacion[n_asignaciones].su_dir=*union_dir_p;
                        asignacion[n_asignaciones].sus_ndeptos=NDEPTOS;
                        for(i=mi_indice; i<=NDEPTOS; i++)
                        {
                                asignacion[n_asignaciones].su_depto[mis_ndeptos]=i;
                                mi_depto[mis_ndeptos]=i;
                                sprintf(nombre_archivo, "departmento%d", i);
                                mi_archivo[mis_ndeptos]=fopen(nombre_archivo, "r+");
                                if(mi_archivo[mis_ndeptos]==NULL)
        -
```

```
                {
                        printf("No pudo abrir el archivo %s\n",
                                nombre_archivo);
                        exit();
                }
                mis_ndeptos ++;
        }
n_asignaciones ++;
}
else
{
        asignacion[n_asignaciones]. su_dir=*union_dir_p;
        union_i= n_asignaciones;

        nmayor=0;
        for(i=0; i< n_asignaciones; i++)
        if(asignacion[i].sus_ndeptos>nmayor)
        {
                mayor_i=i;
                nmayor = asignacion[i].sus_ndeptos;
                mayor_dir=asignacion[i].su_dir;
        };

        for(i=0; i<nmayor/2; i++)
        {
                if(nmayor/2+ nmayor -2> nmayor) pasar= nmayor /2+1+i;
                else    pasar= nmayor -1-i;

                asignacion[union_i].su_depto[
                        asignacion[union_i].sus_ndeptos++]=
                                asignacion[mayor_i].su_depto[pasar];
                asignacion[mayor_i].sus_ndeptos--;

                if(addr_ismine(&mayor_dir))
                {
                        fclose(mi_archivo[pasar]);
                        mi_depto[pasar]=0;
                        mis_ndeptos--;
                }

                if(addr_ismine(&gview_p->gv_joined))
                {
                        mi_depto[i]= asignacion[mayor_i].su_depto[pasar];
                        sprintf(nombre_archivo, "departmento%d",
                                        mi_depto[mis_ndeptos]);
                        mi_archivo[mis_ndeptos]=
```

```
                                                fopen(nombre_archivo, "r+");
                                        if(mi_archivo[mis_ndeptos]==NULL)
                                {
                                        printf("No pudo abrir el archivo %s\n", filename);
                                exit();
                                }
                                        mis_ndeptos ++;
                                }
                                asignacion[mayor_i].su_depto[pasar]=0;
                        }
                n_asignaciones ++;
                }
        }
        else
        {
                nmenor=NDEPTOS+1;
                for(i=0; i< n_asignaciones; i++)
                {
                        if(addr_isequal(&asignacion[i].su_dir, fallo_dir_p))
                                continue;

                        if(asignacion[i].sus_ndeptos<nmenor)
                        {
                                menor_i=i;
                                nmenor = asignacion[i].sus_ndeptos;
                                menor_dir=asignacion[i].su_dir;
                        }
                }

                fallo_i=0;
                while(!addr_isequal(&asignacion[fallo_i].su_dir, fallo_dir_p))
                fallo_i++;

                for(i=0; i< asignacion[fallo_i].sus_ndeptos; i++)
                {
                        asignacion[menor_i].su_depto[
                                asignacion[menor_i].sus_ndeptos++]=
                                        asignacion[fallo_i].su_depto[i];
                        if(addr_ismine(&menor_dir))
                        {
                                mi_depto[mis_ndeptos]=
                                        asignacion[fallo_i].su_depto[i];
                                sprintf(nombre_archivo, "departmento%d",
                                                mi_depto[mis_ndeptos]);
                                mi_archivo[mis_ndeptos]=fopen(nombre_archivo, "r+");
                                mis_ndeptos++;
```

```
                }
            }

            for(i=fallo_i; i<n_asignaciones-1; i++)
                asignacion[i]=asignacion[i+1];
            n_asignaciones --;
        }
}


receive_query (msg_p)
 message  *msg_p;
{
        /* Permanece igual*/
}
```

Figura 4.5

El entero *n_assignments* y arreglo *assignment* describen la forma en que el trabajo se divide entre los miembros. Este es un ejemplo de "información de configuración" la información es la que describe el estado de un grupo de procesos como un todo.

Aún se sigue teniendo un problema. ¿Cómo va a ser ésta información de configuración inicializada?. Cuando un nuevo proceso se une al grupo, sus estructuras de información de configuración deben contener la asignación que se tenía justo antes de su unión, de otra forma *group change* tendría problemas cuando se llamara. En otras palabras, el estado del grupo en el momento de la unión debe ser de alguna forma transferido al proceso que se une.

El función *pg_join* proporciona una opción que permite la transferencia de estado. Cuando un proceso se une a un grupo, el sistema elige uno de los miembros existentes y llama a una rutina especificada por el usuario "rutina de envía" (*sendo routine*) en ese proceso para obtener el estado que va ha ser transferido. Una rutina de envío llama a la función *xfer_out* una o más veces, cada vez dándole un apuntador a una pieza del estado, su tipo, su tamaño (si es un arreglo) y un entero no negativo "*localizador*" asociado con

esta pieza del estado. Cada llamada a *xfer_out* se asemeja a una llamada a *beast* que no espera por respuestas, y conceptualmente produce el envío de un mensaje al miembro que se une. En la práctica, más de uno de estos mensajes es empaquetado en uno más grande y enviados todos juntos, esto simplemente se hace para mejorar el desempeño.

En el proceso que se une, una rutina especificada por el usuario "rutina de recepción" es llamada una vez por cada llamada a *xfer_out* y se le da su mensaje correspondiente y su *localizador*. Este lee la información contenida en el mensaje y el *localizador*. La información la lee del mensaje utilizando *msg_get*. Si el proceso que envía el estado falla antes de haber transferido el estado completo, entonces el sistema elige a otro miembro y llama a su rutina de envío. El valor del último *localizador* entregado al proceso que se une es pasado como un parámetro a la rutina de envío, para que el nuevo transmisor pueda continuar la transferencia desde donde se quedo. (Si no se ha transferido alguna pieza del estado, la rutina de envío es llamada con -1 en lugar del *localizador*.) Dependiendo de la aplicación, un nuevo transmisor del estado puede decidir reiniciar la transferencia desde el principio o transferir algo completamente diferente. El sistema simplemente transporta la información del actual transmisor al proceso que se une, el cual usa el valor del *localizador* para interpretar la información. En este ejemplo una falla en medio de una transferencia invalida la información de configuración actual, por lo que una transferencia reinicializada debe transferir el estado completo desde el inicio.

Cuando un grupo es creado por primera vez, no hay estado que transferir por lo que no se llama a las rutinas de envío o recepcion. En su lugar se invoca a una rutina de inicialización. Esta es especificada en la opción *PG_INIT* de *pg_join*, la cual le da el nombre a la rutina de inicialización. En este ejemplo la rutina de inicialización establece la estructura de información de configuración de la forma apropiada para un grupo vacío.

Cuando todos los miembros de un grupo fallan normalmente *ISIS* llama a una rutina de inicialización. Pero también es posible registrar el estado en un archivo del

esta pieza del estado. Cada llamada a *xfer out* se asemeja a una llamada a *bcast* que no espera por respuestas, y conceptualmente produce el envío de un mensaje al miembro que se une. En la práctica, más de uno de estos mensajes es empaquetado en uno más grande y enviados todos juntos, esto simplemente se hace para mejorar el desempeño.

En el proceso que se une, una rutina especificada por el usuario "rutina de recepción" es llamada una vez por cada llamada a *xfer out* y se le da su mensaje correspondiente y su *localizador*. Este lee la información contenida en el mensaje y el *localizador*. La información la lee del mensaje utilizando *msg get*. Si el proceso que envía el estado falla antes de haber transferido el estado completo, entonces el sistema elige a otro miembro y llama a su rutina de envío. El valor del último *localizador* entregado al proceso que se une es pasado como un parámetro a la rutina de envío, para que el nuevo transmisor pueda continuar la transferencia desde donde se quedó. (Si no se ha transferido alguna pieza del estado, la rutina de envío es llamada con -1 en lugar del *localizador*.) Dependiendo de la aplicación, un nuevo transmisor del estado puede decidir reiniciar la transferencia desde el principio o transferir algo completamente diferente. El sistema simplemente transporta la información del actual transmisor al proceso que se une, el cual usa el valor del *localizador* para interpretar la información. En este ejemplo una falla en medio de una transferencia invalida la información de configuración actual, por lo que una transferencia reinicializada debe transferir el estado completo desde el inicio.

Cuando un grupo es creado por primera vez, no hay estado que transferir por lo que no se llama a las rutinas de envío o recepción. En su lugar se invoca a una rutina de inicialización. Esta es especificada en la opción *PG_INIT* de *pg join*, la cual le da el nombre a la rutina de inicialización. En este ejemplo la rutina de inicialización establece la estructura de información de configuración de la forma apropiada para un grupo vacío.

Cuando todos los miembros de un grupo fallan normalmente *ISIS* llama a una rutina de inicialización. Pero también es posible registrar el estado en un archivo del

disco, especificando la opción *PG_LOGGED* a *pg_join*. De esta forma el estado será recuperado del *log* después de recuperarse de una falla total del grupo.

Para mostrar una de las formas en las que el argumento *localizador* puede ser usado, se dividió el estado en cuatro piezas basándose en el tipo, y enviando cada una con una llamada diferente a *xfer_out*. Es importante notar que las rutinas de recepción y envío coinciden y que el *localizador* es usado para distinguir las piezas del estado. Normalmente el estado estaría dividido en piezas basandose en la función en vez del tipo.

La operación de transferencia de estado muestra otro aspecto de la sincronía virtual. Aún cuando la transferencia de estado es una operación compleja y puede involucrar una serie de mensajes, el programa fue escrito como si la transferencia de estado y la unión ocurrieran instantáneamente e indivisiblemente. Por ejemplo no hay necesidad de preocuparse por mensajes de consulta que fueran entregados al proceso que envía o al que recibe el estado mientras la transferencia de estado esta todavía realizándose, aún cuando el servicio podría estar recibiendo consultas mientras la transferencia de estado se esta llevando a cabo. A continuación se muestra el programa con las rutinas que permiten la transferencia de estado.

```
#include <stdio.h>
#include "../include/isis.h"
#define         ENTRADA_CONSULTA    1
#define         NOMBRE1_MAX         64
#define         NDEPTOS             5
#define         MIEMBROS_MAX        10

#define         NASIGN_LOC      1
#define         DIR_LOC         2
#define         NDEPTOS_LOC     3
#define         DEPTOS_LOC      4

int     n_asignaciones;
struct
{
    address     su_dir;
    int         sus_ndeptos;
```

```
        int         su_depto[NDEPTS];
} asignacion[MIEMBROS_MAX];

address*dirgrp_p;
int             mi_indice;
int             mis_ndeptos=0;
int             mi_depto[NDEPTOS];
FILE            *mi_archivo[NDEPTOS];


main()

{
        /* Permanece igual*/
}

tarea_principal()
{

        int   cambio_grupo();
        int   envia_estado(), recibe_estado(), inicia_grupo();


        /* Unión al grupo de procesos,  monitoreo de los cambios en*/
        /* la membresia, y establecimiento de la transferencia de estado     */

        dirgrp_p = pg_join ("grupo",
                        PG_MONITOR, cambio_grupo, 0,
                        PG_XFER, 0, envia_estado, recibe_estado,
                        PG_INIT, inicia_grupo,
                        0);
        if(addr_isnull(dirgrp_p))
        exit();
        isis_start_done();

}

envia_estado (last_locator)
int     last_locator;
{
        addressdir[MIEMBROS_MAX];
        int             ndeptos[MIEMBROS_MAX];
        int             depto[MIEMBROS_MAX][NDEPTOS];
        int             i, j;

        for(i=0; i<= n_asignaciones; i++)
```

```
        {
                dir[i]=asignacion[i].su_dir;
                ndeptos[i]= asignacion[i].sus_ndeptos;
                for(j=0; j<ndeptos[i]; j++)
                        depto[i][j] = asignacion[i].su_depto[j];
        }

        xfer_out (NASIGN_LOC, "%d", n_asignaciones);
        xfer_out (DIR_LOC, "%A", dir, n_asignaciones);
        xfer_out (NDEPTOS_LOC, "%D", ndeptos, n_asignaciones);
        xfer_out (DEPTOS_LOC, "%D", depto, n_asignaciones *NDEPTOS);
}

recibe_estado(locator, msg_p)
int     locator;
message *msg_p;
{
        addressdir[MIEMBROS_MAX];
        int             ndeptos[MIEMBROS_MAX];
        int             depto[MIEMBROS_MAX][NDEPTOS];
        int             i, j, long;


        switch(locator)
        {
                case    NASIGN_LOC:
                {
                        msg_get(msg_p, "%d", & n_asignaciones);
                        break;
                }
                case    DIR_LOC:
                {
                        msg_get(msg_p, "%A", dir, &long);
                        for(i=0; i<long; i++)
                                asignacion[i].su_dir=dir[i];
                        break;
                }
                case    NDEPTOS_LOC:
                {
                        msg_get(msg_p, "%D", ndeptos, &long);
                        for(i=0; i<long; i++)
                                asignacion[i].sus_ndeptos=ndeptos[i];
                        break;
                }
                case    DEPTOS_LOC:
                {
```

```
                        msg_get(msg_p, "%D", depto, &long);
                        for(i=0; i< n_asignaciones; i++)
                                for(j=0; j< asignacion[i].sus_ndeptos; j++)
                                asignacion[i].su_depto[j]=depto[i][j];
                        break;
                }
        }
}

inicia_grupo()
{
        /* Inicialización para un grupo vacio*/
        n_asignaciones =0;
}

cambio_grupo (gview_p, arg)
 groupview *gview_p;
 int  arg;
{
        /* Permanece igual*/
}

recibe_consulta(msg_p)
message *msg_p;
{
        /* Permanece igual*/
}
```

Figura 4.6

5. Aplicación de la sincronía virtual y grupos de procesos en la computación móvil.

5.1. Introducción

La siguiente propuesta se basa en la comunicación de grupos de procesos y en la sincronía virtual, para permitir el movimiento de los usuarios de una red inalámbrica. Los usuarios con sus computadoras móviles pueden viajar a través de la red sin perder la comunicación, ya que un grupo de servidores en una red estática se encargan de transferir el control de la comunicación de uno a otro en caso de que ésta salga de su alcance. El sistema también aprovecha las ventajas de la tolerancia a fallas, de esta forma los usuarios pueden operar en un ambiente confiable de manera que los problemas de comunicación pasan desapercibidos.

El sistema se basa en dos estrategias: la primera utiliza un servidor principal y un grupo de servidores de respaldo en caso de que el servidor principal falle, la segunda consiste en mantener una copia del estado de la comunicación en todos los servidores.

Estas dos estrategias son utilizadas para simular la multitransmisión por parte del cliente, y poder así seguirlo y proporcionarle una comunicación confiable.

La migración de un cliente de una célula a otra es manejada como un cambio de servidor principal intencional. (Para lograr un alto desempeño, los servidores están replicados a un nivel de *sliding-window*.) Este esquema provee un abstracción simple de migración, elimina los complicados protocolos para cambio de célula, proporciona tolerancia a fallas y es implementado dentro del mecanismo de comunicación de grupo de *ISIS*.

5.2. Modelo del sistema

El sistema consiste de dos grupos distintos de computadoras: los servidores estáticos y los clientes móviles (Figura 5.1). Un grupo de servidores que corre *ISIS* en una red de área local estática proporciona servicios a los clientes móviles. Los servidores actúan como un representante del cliente móvil. Los clientes móviles se comunican con los servidores a través de un canal inalámbrico. La red inalámbrica esta organizada a través de células geográficamente definidas. Los clientes móviles pueden atravesar los límites de las células sin perder la comunicación. La red estática y la red inalámbrica están conectadas a través de servidores *gateways* o estaciones de soporte móvil que son los puntos de control de los clientes móviles.

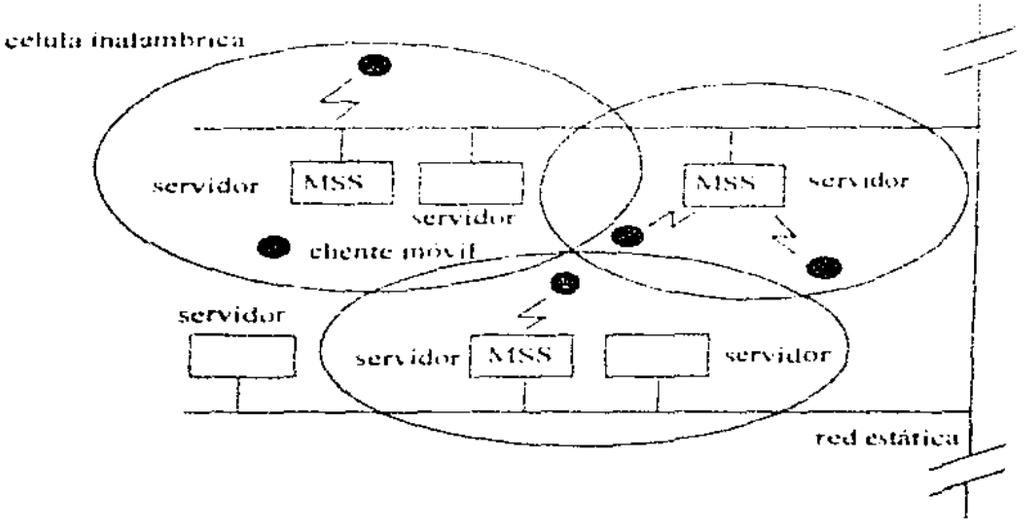


Figura 5.1 Modelo del sistema

5.3 Ventajas de la sincronía virtual y los grupos de procesos

Una de las funciones más importantes en las comunicaciones móviles es el cambio de célula en el transcurso de la comunicación (conocido como *hand-off*). El *hand-off* requiere que los sistemas mantengan una conexión punto final a punto final en presencia de cambios dinámicos en la topología de la red.

El objetivo es que un grupo de servidores se haga cargo de la comunicación del cliente. El cliente mantiene su comunicación conforme se va moviendo de manera que los cambios de célula o las fallas de los servidores en la red estática nos los nota.

La comunicacion de grupos entre servidores se puede utilizar en una topología de red que se reconfigura dinámicamente (debido a la falla de un servidor o la incorporación de un nuevo servidor al sistema con la finalidad de hacerlo más confiable), ya que está intrínsecamente relacionada con el cambio en la membresía del grupo. El sistema puede ser programado para tomar decisiones a partir del conocimiento de los miembros que forman el grupo. Otra ventaja de la comunicación de grupo es que los emisores no necesitan conocer la membresía del grupo ya que los mensajes son enviados a grupos abstractos. El manejo de la membresía de grupos permite abandonar o unirse al grupo dinámicamente, por lo que un miembro puede abandonar el grupo, regresar al mismo grupo y continuar trabajando con los nuevos miembros. Otra propiedad muy útil es la multitransmisión ordenada que garantiza que todos los miembros del grupo verán el mismo conjunto de mensajes en el mismo orden, por lo que la sincronización de múltiples procesos o la coordinación de acciones se puede lograr fácilmente. De esta forma la comunicacion de grupo facilita la tolerancia a fallas.

El verdadero poder de la comunicacion de grupo reside en soportar copias consistentes (a través de la transferencia de estado), éste mecanismo puede ser usado para la movilidad de los usuarios ya que todos los servidores pueden saber con precisión que servidor se esta haciendo cargo de la comunicación y el desarrollo de la misma, de esta

forma el *hand-off* se realiza a través de comunicar un cambio de punto de control entre servidores. Este esquema provee una abstracción simple de migración, elimina los protocolos de *hand-off* complicados y el tiempo de *hand-off*, proporciona tolerancia a fallas y es implementado dentro del mecanismo de comunicación de grupos existente.

Un canal uno a muchos es implementado entre un cliente y servidores. Un cliente ve un único y confiable servidor en la otra terminacion del canal; los *hand-offs* o fallas del servidor están completamente ocultas al cliente, por otro lado, los servidores ven los mensajes de los clientes y mantienen copias del servidor consistentes actuando como una máquina de estado. Todos los servidores observan un mismo conjunto de mensajes que llegan de los clientes pero realmente solo el servidor principal envia mensajes al cliente. Un *hand-off* o una falla del servidor principal invoca un cambio de servidor principal en la cual una de las otras copias del servidor se hace cargo del canal al instante. Un *hand-off* es disparado por un único mensaje de multitransmisión al grupo de servidores; no es necesario un protocolo de *hand-off* entre un cliente y los servidores. Para lograr un alto desempeño, los servidores son replicados a un nivel de *sliding-window*. El mismo *sliding-window* es diseñado como una máquina de estado.

5.4. Modelo de Comunicación

La estructura que se utiliza en el sistema es un cliente móvil que se comunica con un grupo de servidores que se encuentran en una red estática. *ISIS* soporta varios tipos de estructuras de grupos pero el sistema utiliza dos de ellas que se ajustan a las aplicaciones móviles. Una es la llamada a procedimientos remotos (*RPC*); la otra es la comunicación por difusión. En el caso de la *RPC* los clientes interactúan con los servidores a través de un estilo solicitud-respuesta. El estilo de difusión es un tipo de grupo cliente-servidor en el cual los servidores realizan una multitransimisión de mensajes al grupo completo de clientes. Los clientes están pasivos y simplemente reciben los mensajes. Un estilo de difusión funciona bien para cualquier aplicación que transmite información a un gran número de sitios.

5.5. Enfoques de máquina de estado y de respaldo principal

Para implementar un servicio de gran disponibilidad, el servicio debe estar replicado y distribuido entre múltiples servidores. Las acciones deben ser coordinadas para que aún cuando un servidor falle, el servicio permanezca disponible. El uso de réplicas en la información tiene una ventaja adicional en el desempeño. En general, el uso de réplicas es implementado de dos formas, una de ellas es el enfoque de máquina de estado, el cual, no tiene control centralizado y la otra es el enfoque del respaldo principal, que tiene un control centralizado.

En la máquina de estado un cliente hace una solicitud a través de una multitransmisión a todos los servidores. Todos los servidores cambian sus estados de forma idéntica en un estado de bloqueo. Cuando se utiliza este enfoque una falla del servidor es invisible a los clientes y no provoca ningún retraso en la respuesta ya que cada servidor puede tomar acciones por su propia cuenta.

En el enfoque del respaldo principal un cliente realiza una solicitud enviando un mensaje sólo al servidor principal. Si el servidor principal falla, un servidor de respaldo se hace cargo. Las solicitudes pueden ser perdidas en una falla del servidor principal y los clientes necesitan ser informados acerca de la falla para volver a intentar la solicitud perdida, ya que no tiene forma de saber del desarrollo de la comunicación entre el servidor principal y el cliente, el tratar de retomar la comunicación implicaría un intercambio de mensajes entre el cliente y el servidor que tomarían un cierto tiempo.

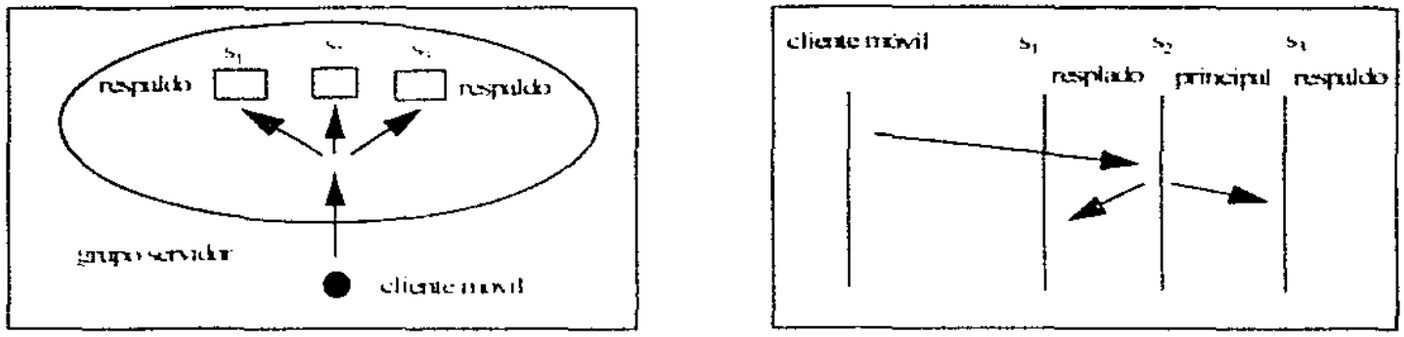
En general, el enfoque del respaldo principal es más simple y menos costoso, especialmente del lado del cliente, ya que el enfoque de la máquina de estado requiere de multitransmisión ordenada confiable.

Para los clientes móviles la opción del respaldo del servidor principal es más atractivo. Los clientes no necesitan implementar multitransmisión que puede ser considerablemente más complicada que una comunicación punto a punto. Además los

clientes móviles en una red celular inalámbrica pueden alcanzar un número limitado de servidores dentro del rango de comunicación por lo que la multitransmisión a todos los servidores podría no ser posible.

Por otro lado, *ISIS* soporta el enfoque de máquina de estado y proporciona un paradigma de programación sofisticado en el cual los programas realizan acciones de acuerdo a eventos. La aproximación de máquina de estado es más flexible por varias razones: el mecanismo es completamente descentralizado, un programa puede realizar acciones sólo a partir de su estado local, los clientes pueden esperar respuestas en tiempo real aún en presencia de fallas y los sistemas pueden soportar fallas arbitrarias.

El sistema emplearía una combinación de estos dos esquemas: el enfoque del respaldo principal entre un cliente y los servidores y el de máquina de estado entre los servidores. La figura 5.2a muestra la estructura del canal del sistema. Un cliente habla únicamente al servidor principal pero este transmite los mensajes que le llegan a los respaldos con la finalidad de proveer la ilusión de que el cliente tiene la capacidad de realizar una multitransmisión (figura 5.2b). Esto permite a los programadores utilizar la máquina de estado del lado del servidor, desde el punto de vista de los usuarios, los clientes ven un canal confiable y las fallas del servidor principal pasan desapercibidas. La programación del lado del servidor está basada en la máquina de estado similar al modelo de *ISIS* y el mecanismo de respaldo principal es escondido en el sistema.

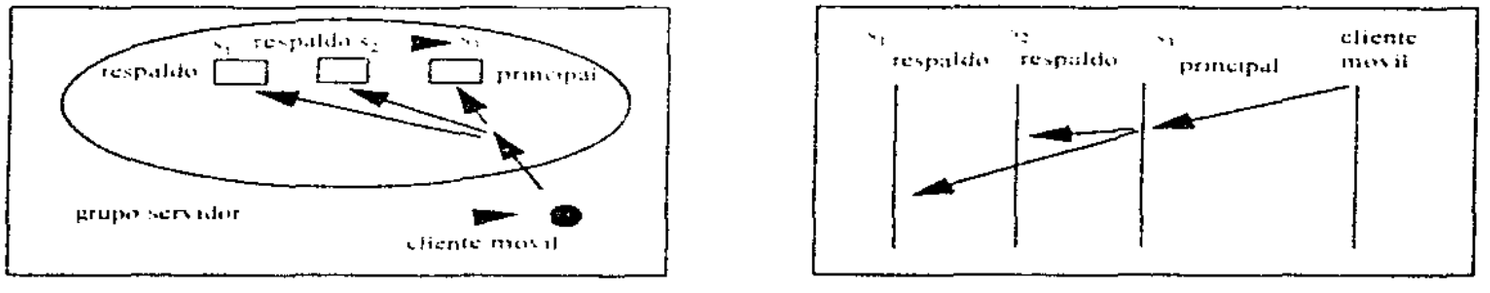


(a) El cliente habla con el principal (b) El principal direcciona el mensaje a los respaldos

Figura 5.2 Principal-Respaldo

5.6. Migración como un cambio de servidor principal intencional.

La característica principal del sistema es que la migración de clientes móviles es manejada como un cambio de servidor principal intencionado. En el enfoque del respaldo-principal una recuperación a la falla normalmente ocurre sólo cuando el servidor principal falla. Sin embargo, este mismo mecanismo puede ser usado para controlar la migración con la finalidad de mover el servidor principal a un respaldo cercano, especialmente en las redes celulares inalámbricas, este mecanismo permite al sistema poner el servidor principal dentro del rango de comunicación deseado. Esto es, el estado del servidor principal puede ser pasado de un servidor a otro para seguir al usuario geográficamente. La figura 5.3 muestra un cambio del servidor principal $s2$ a $s3'$. En el enfoque principal/respaldos los respaldos estan diseñados para hacerse cargo del servidor principal en cualquier momento en caso de que falle, por lo que no hay ninguna dificultad en invocar un cambio del servidor principal en cualquier momento. La única diferencia es que el cambio del servidor principal sera invocado por una migración de un cliente móvil en vez de una falla del servidor principal.



(a) (b)

Figura 5.3 La migración como un cambio de servidor principal intencionado.

5.7. Replicación del *Sliding-Window*

Para cualquier comunicación es necesario un mecanismo de sincronización. Cuando varios procesos necesitan sincronizarse se necesita un protocolo para el establecimiento de la comunicación (*handshake*) que introduce un tiempo extra y, por lo tanto, es necesario el almacenamiento de los mensajes provenientes de otros procesos durante este periodo. El protocolo de *handshaking* y el almacenamiento (*buffering*) originan las dificultades en los protocolos de cambio de célula (*hand-off*). Estos procedimientos de *hand-off* requieren *handshaking* entre tres partes: un nuevo servidor, un viejo servidor y un cliente móvil. Sin embargo, el sistema propuesto no necesita ningún *handshaking* explícito, ya que se logra automáticamente a través de las estructuras subyacentes.

El *handshake*, entre los servidores, puede lograrse en el modelo de sincronía virtual de *ISIS* por las siguientes razones. La sincronía virtual es una ilusión en la que todos los eventos ocurren de manera síncrona en el sistema. Eventos, tales como la recepción de mensajes, son síncronos en términos de un tiempo lógico, aunque cada proceso recibe un mismo mensaje en un diferente tiempo físico. La sincronía virtual puede ser usada para sincronizar múltiples procesos y liberar a los programadores de los problemas de *handshaking* y *buffering*.

El sistema convierte los mensajes de los clientes móviles en eventos de *ISIS* virtualmente síncronos, a través de enviar mensajes desde los clientes móviles a todos los servidores. Por lo que todos los servidores observan el mismo grupo de mensajes que llegan en el mismo orden, lo cual permite a los servidores adoptar el enfoque de la máquina de estado. Cuando un cambio de celula es solicitado por un cliente móvil a través de la multitransmisión al grupo servidor, todos los servidores pueden llevar a cabo las acciones apropiadas sin ningún *handshaking* ya que todos los servidores tienen una visión consistente del estado del sistema. La necesidad de un *handshake* entre dos servidores puede por lo tanto ser eliminado a través del uso de la sincronía virtual.

El *handshake* entre un cliente móvil y los servidores esta confinado a un mecanismo de transporte de bajo nivel. Los protocolos de *hand-off* anteriores (los clásicos) asumen un canal *FIFO* entre un servidor y un cliente. Aunque un canal *FIFO* es una abstracción útil para una red estática éste no funciona bien para un procedimiento de *hand-off*.

El problema de un canal *FIFO* es que un procedimiento de *hand-off* necesita terminar la conexión con el servidor anterior y entonces establecer una nueva conexión con el nuevo servidor. Estos dos procedimientos requieren *handshaking* entre las dos terminaciones e introduce retrasos y almacenamiento, además un canal *FIFO* no es tolerante a fallas. Cuando un servidor falla otro servidor debe hacerse cargo del canal para tolerancia a fallas. Ya que no hay forma de saber en un canal *FIFO* el estado de la otra parte son necesarios protocolos adicionales entre el cliente y el nuevo servidor para almacenar los estados. Una falla de servidor en medio de un *hand-off* necesitara más protocolos complicados.

Sin embargo, si se utiliza una abstracción de transferencia de más bajo nivel como la de paquetes, el estado de la comunicación puede estar replicado en base a los paquete. Si un nuevo servidor conoce el último paquete transferido o de reconocimiento éste puede hacerse cargo y continuar la comunicación donde haya quedado, sin necesidad del *handshaking*. Más aún, los protocolos de este nivel ya asumen que los paquetes pueden ser perdidos o duplicados, así que el protocolo es inherentemente robusto para perdida de paquetes, duplicados o tiempos de retraso. Por lo que no es necesario un protocolo para *hand-off* y el *handshaking* explicito entre un cliente móvil y los servidores puede ser eliminado.

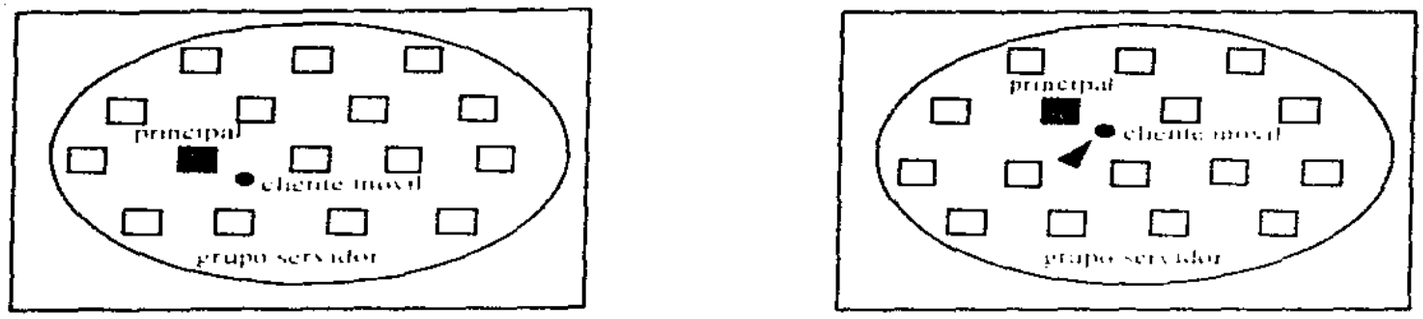
El sistema implementa un protocolo *sliding-window* para la comunicación entre un cliente móvil y los servidores, y replicaria el estado del *sliding-window* del lado del servidor. El protocolo *sliding-window* se utiliza para implementar un canal *FIFO* y controlar el flujo de mensajes en el canal de comunicación. El protocolo puede hacer uso total del ancho de banda de la red si se afina cuidadosamente, lo cual es muy importante

especialmente en los medios de comunicación inalámbricos debido a su relativamente bajo ancho de banda. La replica de los estados de *sliding-window* permite al sistema lograr un alto desempeño así como mantener el modelo de máquina de estado.

5.8. Crecimiento del sistema

Un sistema de cómputo móvil debe tener una capacidad de crecimiento razonable. Una estructura de jerárquica grupos se utiliza para hacer que el sistema pueda crecer. Cuando un servicio necesita estar disponible para cientos de usuarios móviles en decenas de sitios es deseable replicar el servicio en cada sitio pero sólo replicar el estado de un cliente en un pequeño subgrupo de sitios. El grupo queda estructurado en subgrupos; cada subgrupo corresponde a un usuario móvil. Cada subgrupo consiste de los sitios vecinos a la localización actual del usuario. La figura 5.4(a) muestra un ejemplo de la distribución de los sitios; cada cuadrado representa un sitio y todos los sitios constituyen un grupo del servicio. Los sitios sombreados forman un subgrupo alrededor del sitio principal de color negro donde reside el usuario actualmente.

Los grupos cambian su membresía dinamicamente para poder soportar la movilidad de usuario. La idea es que el subgrupo se mueva junto con el usuario. Conforme el usuario viaja, nuevos sitios, localizados en la direccion a la cual el usuario se dirige se unen al subgrupo. Entonces los miembros localizados en los sitios detrás del usuario abandonan el grupo con la finalidad de mantener el mismo tamaño de grupo. En la figura 5.4(b), el subgrupo se mueve desde 5.4(a) para seguir al usuario. Cuando un usuario desconectado establece una nueva conexión en un sitio remoto a la localización anterior, un nuevo subgrupo surge en la nueva localización y entonces el viejo subgrupo se desvanece.



(a) (b)

Figura 5.4 Un subgrupo sigue al usuario conforme éste se mueve

Los sitios que no pertenecen a ningún subgrupo pueden abandonar el grupo de servicio para reducir el tamaño del grupo de servicio. Estos grupos se volverán a unir al grupo de servicio cuando un usuario se acerque a ellos. A través de mantener únicamente los sitios miembros de subgrupos, el grupo de servicio entero cambia la forma para adaptarse a los movimientos de los usuarios. Para que el sistema pueda crecer hasta miles o millones de sitios se necesita una mayor jerarquización.

6. Conclusiones

En esta tesis se describen los sistemas distribuidos desde un enfoque que va de lo más general, como son las características y clasificaciones de los mismos, a aspectos relacionados con el correcto funcionamiento y buen desempeño. Se analizan principalmente los problemas relacionados con los procesos que interactúan continuamente y donde el orden de entrega de los mensajes es de vital importancia. Se plantean posibles soluciones a estos problemas y se analiza el impacto en el desempeño del sistema. Se propone el uso de los enfoques de sincronía virtual y comunicación de grupos, y se describen dos aplicaciones distribuidas que hacen uso de estos. A partir de lo expuesto a lo largo de esta tesis es importante resaltar los siguientes puntos:

- Las aplicaciones distribuidas donde se requiere que los procesos trabajen de manera coordinada como en el caso del procesamiento en paralelo y la tolerancia a fallas son difíciles de programar con herramientas convencionales. El considerar todos las situaciones que se pueden presentar y el diseñarlo de una manera eficiente son los principales obstáculos.

- Un ambiente de programación síncrono facilita la tarea del programador, pero la sincronía trae un costo asociado en términos de tardanza en la entrega de mensajes y por lo tanto en la respuesta del sistema en general, y en consumo de memoria. El costo de la sincronía puede ser muy alto, dando como resultado un sistema ineficiente.

- Un ambiente asíncrono permite que las actividades se lleven acabo concurrentemente, dando como resultado un sistema que responde rápidamente. Si se requiere que los procesos observen un mismo orden de eventos para el correcto funcionamiento del sistema la tarea del programador se complica notablemente especialmente si se impone un límite en el tiempo de respuesta.

- La sincronía virtual permite aprovechar la asincronía cuando es posible y facilita la programación. Como en el caso del protocolo *CBCAST* donde sólo se establece un orden entre eventos que dependen entre sí. De esta forma se facilita la programación y se aprovechan las ventajas de las operaciones asincronas.

- Los grupos de procesos permiten implementar diferentes esquemas de comunicación en un mismo sistema. algunos procesos que requieran de un orden total de comunicación se comunicaran entre sí a través del protocolo *ABCAST*, mientras que otros podrían utilizar *CBCAST*. Los procesos se pueden agrupar de acuerdo a las funciones que realizan en el sistema.

- El manejo de grupos dinámicos en los que los miembros pueden abandonar el grupo o unirse a el sin preocuparse acerca de la entrega de mensajes o el orden en que los miembros del grupo observan los mensajes, hacen a este enfoque especialmente atractivo para aplicaciones como la computación móvil.

- Otra ventaja del uso de grupos de procesos es la transferencia de estado que permite que los procesos puedan tomar acciones por su cuenta así como utilizar la membresía como una entrada a un algoritmo, por ejemplo para realizar una búsqueda en paralelo a una base de datos.

- Las herramientas como *ISIS* permiten que el programador se concentre en lo que desea que su sistema realice en vez de resolver los problemas de comunicación y tolerancia a fallas.

- Una herramienta de propósito general permite al menos potencialmente que uno escoja el comportamiento de su sistema de la manera más adecuada de acuerdo a sus necesidades. El conocimiento de los protocolos de comunicacion así como el de la teoría relacionada con los sistemas distribuidos es muy importante si se desea crear sistemas de computo distribuido que se comporten de manera óptima.

7. Referencias

(BJ89) Kenneth P. Birman y Thomas A. Joseph. *Exploiting replication in distributed systems*, páginas 319-368, New York, 1989. *ACM Press*, AddisonWesley.

(BR93) Kenneth P. Birman y Robbert Van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press.

(BSS91) Kenneth P. Birman, A. Schiper y P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, páginas 272-314, Agosto de 1991.

(CB94) Kenjiro Cho y Kenneth P. Birman. A Group Communication Approach for Mobile Computing. Department of Computer Science, Cornell University.

(CD90) Flaviu Cristian y Robert Dancey. Fault-tolerance in the advanced automation system. Reporte teenico RJ7424, Laboratorio de investigación de IBM, San Jose, California, Abril de 1990.

(KTFHB89) M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn-Hummel y Henri E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, páginas 5-19, Octubre de 1989.

(LAM78) Leslie Lamport. Tiempo, relojes y el orden de eventos en un sistema distribuido. *Communications of the ACM*, páginas 558-565, Julio de 1978.

(LL86) Barbara Liskov y Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. En *Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, paginas 29-39, Calgary, Alberta, Agosto de 1986.

(PET87) Larry Peterson. Perserving context information in an IPC abstraction. En el *Sixth Symposium on reliability in Distributed Software and Database Systems*, páginas 22-31. IEE, Marzo de 1987.

(RB91) Aleta Ricciardi y Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environments. *En Proceedings of the Fifth ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, Agosto de 1991.

(SCH86) Fred. B. Schneider (1986) 'A paradigm for reliable clock sincronization'. Proceedings Advanced Seminar on Real-Time Local Area Networks, Bandol, Francia, Abril de 1986.

(SCH88) Frank Schumuck. *The use of Efficient Broadcast Primitives in Asychronous Distributed System*. Tesis de doctorado, Universidad de Cornell,1988.

(SKE82) Dale Skeen. *Crash recovery in a distributed database system*. Tesis de doctorado. Universidad de California en Berkeley. Departamento de EECS. Junio de 1982.

(TAN88) Andrew S. Tanenbaum. *Computer networks*. Prentice Hall. segunda edición 1988.

(TAN92) Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall.

(TKT91) The Isis Distributed Toolkit. Version 3.0. User Reference Manual. Isis Distributed Systems, Inc.