



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

DISEÑO DE UNA NUEVA ARQUITECTURA DE SOFTWARE PARA VIEW SYNCHRONY ESTRUCTURADO Y DEFINICIÓN DE UN FRAMEWORK DE APLICACIÓN PARA SU UTILIZACIÓN

Director: Alfredo Vega Weiss

ALUMNOS: LORENZO JORQUERA, 578/90
JUAN PABLO OLGUIN, 230/90



INDICE

1 ABSTRACT.....	6
2 PRÓLOGO.....	7
2.1 GUÍA DE LECTURA	9
3 OBJETIVOS.....	10
3.1 INTRODUCCIÓN.....	10
4 MARCO CONCEPTUAL PARA LA COMPUTACION GLOBAL (WDVM – WORLD DISTRIBUTED VIRTUAL MACHINE).....	13
4.1 MOTIVACIÓN	13
4.2 ASPECTOS FUNDAMENTALES EN LA DEFINICIÓN DEL MARCO CONCEPTUAL	16
4.3 SERVICIOS PARA LA EJECUCIÓN DE APLICACIONES PARALELAS Y DISTRIBUIDAS.....	18
4.3.1 <i>Mecanismos de comunicación entre procesos</i>	18
4.3.2 <i>Sincronización entre procesos</i>	21
4.3.3 <i>Mecanismo para el booteo de procesos</i>	24
4.3.4 <i>Distribución de código</i>	24
4.3.5 <i>Manejo de versiones</i>	24
4.3.6 <i>Seguridad</i>	24
4.4 SERVICIOS PARA LA ADMINISTRACIÓN DE LA MÁQUINA VIRTUAL	26
4.4.1 <i>Balance de carga</i>	26
4.4.2 <i>Tolerancia a fallas</i>	28
4.4.3 <i>Administración de la red de nodos</i>	30
4.4.4 <i>Compartición de recursos</i>	32
4.4.5 <i>File system distribuido</i>	33
4.4.6 <i>Métricas</i>	33
4.5 LENGUAJES SOBRE LA WDVM	34
4.6 ENTORNO DE DESARROLLO.....	34
4.7 ESTRUCTURA DE LA WDVM.....	35
5 STATE OF THE ART.....	36
5.1 CONCEPTOS DE VIEW SYNCHRONY	36
5.1.1 <i>Introducción a View Synchrony</i>	36
5.1.2 <i>Técnicas de tolerancia a fallas basadas en la replicación</i>	39
5.1.3 <i>Primitivas de comunicación de grupos</i>	40
5.1.4 <i>Reliable Multicast (Failure Atomic Multicast)</i>	43
5.1.5 <i>Ordenes de entrega de multicasts</i>	45
5.1.6 <i>Relación entre atomicidad de fallas y orden de entrega</i>	47
5.1.7 <i>Grupo de procesos (membership)</i>	48
5.1.8 <i>View Synchrony</i>	51
5.2 PROTOCOLOS DE VIEW SYNCHRONY	54
5.2.1 <i>Protocolos de multicast atómico con respecto a fallas</i>	54
5.2.2 <i>Orden de mensajes</i>	58
5.2.3 <i>Protocolo de pertenencia al grupos de procesos</i>	62
5.2.4 <i>Protocolo de view synchrony</i>	64
5.3 VIEW SYNCHRONY ESTRUCTURADO	66
5.3.1 <i>Modelo del sistema</i>	66



5.3.2	<i>Protocolos de Multicast</i>	67
5.3.3	<i>Protocolos de View Change</i>	69
5.4	OTROS SISTEMAS	72
5.4.1	<i>Relacs</i>	72
5.4.2	<i>Horus</i>	78
5.4.3	<i>Transis</i>	81
5.4.4	<i>Totem</i>	85
6	DEFINICIÓN DE LA ARQUITECTURA DE SOFTWARE.....	90
6.1	INTRODUCCIÓN	90
6.2	EL MODELO DEL SISTEMA	90
6.3	LA ARQUITECTURA DEL SISTEMA.....	92
6.4	DESCRIPCIÓN DETALLADA DE LAS CAPAS DE LA ARQUITECTURA	95
6.4.1	<i>La capa Netcomm</i>	95
6.4.2	<i>La capa Rss</i>	95
6.4.3	<i>La capa FD</i>	97
6.4.4	<i>La capa MSS</i>	98
6.4.5	<i>La capa VSS</i>	102
7	FRAMEWORK DE APLICACIÓN	112
7.1	INTRODUCCIÓN	112
7.2	FRAMEWORK	112
	INTERFAZ DEL FRAMEWORK SERVIDOR DE APLICACIÓN.....	114
7.4	ESQUEMA DE USO DEL FRAMEWORK PARA UN PROCESO SERVIDOR	117
7.5	INTERFAZ DEL FRAMEWORK PARA UN PROCESO CLIENTE	118
7.6	ESQUEMA DE USO DEL FRAMEWORK PARA UN PROCESO CLIENTE	119
7.7	DIAGRAMA DE CLASES DEL FRAMEWORK	120
8	IMPLEMENTACIÓN DEL FRAMEWORK.....	121
8.1	INTRODUCCIÓN	121
8.2	COMPOSICIÓN DEL FEVS (PROCESO SERVIDOR).....	121
	COMPOSICIÓN DEL FEVSCLIENT (CLIENTE).....	123
9	PRUEBAS.....	125
9.1	PRUEBA 1 - TIEMPO DE CAMBIO DE VIEW	125
9.1.1	<i>Cambio de view en Structured view synchrony</i>	125
9.1.2	<i>Cambio de view con View Synchrony</i>	130
9.1.3	<i>Comparación tiempo de cambio de view entre view synchrony y structured view synchrony</i> 134	
9.2	PRUEBA 2 – LATENCIA DE ENVÍO DE MENSAJE (ONE WAY LATENCY) CON MENSAJES DINÁMICAMENTE UNIFORMES	135
9.2.1	<i>Structured View Synchrony</i>	136
9.2.2	<i>View Synchrony</i>	140
9.2.3	<i>Comparación latencia de mensajes du entre view synchrony y structured view synchrony</i> 143	
9.3	PRUEBA 3 – LATENCIA DE ENVÍO DE MENSAJE (ONE WAY LATENCY) CON MENSAJES NO DINÁMICAMENTE UNIFORMES	144
9.3.1	<i>Structured View Synchrony</i>	144
9.3.2	<i>View synchrony</i>	148
9.3.3	<i>Comparación latencia de mensajes ndu entre view synchrony y structured view synchrony</i> 152	



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

10 MODELO DE TOLERANCIA A FALLAS DE LA WDM	154
10.1 INTRODUCCION	154
10.2 ARQUITECTURA DE LA WDM	154
10.3 MODELO DE TOLERANCIA A FALLAS EN LA WDM	156
11 CONCLUSIONES	159
12 BIBLIOGRAFÍA	162



Indice de figuras

Fig. 1 - Esquema de la WDVM.....	14
Fig. 2 - Esquema sin memoria compartida	20
Fig. 3 - Esquema con memoria compartida	20
Fig. 4 - Esquema de llamada RPC.....	21
Fig. 5 - Capas de seguridad en la WDVM	25
Fig. 6 - Estructura de la WDVM.....	35
Fig. 7 - Modelo Primary-Backup.....	39
Fig. 8 - Modelo State Machine.....	40
Fig. 9 - Figura Caída del primario	41
Fig. 10 - Caída del cliente	42
Fig. 11 - Pedidos en distintos órdenes.....	42
Fig. 12 - Entrega de mensajes no dinámicamente uniforme	44
Fig. 13 - Entrega causalmente ordenada de mensajes.....	46
Fig. 14 - Atomicidad de fallas y orden de entregas	47
Fig. 15 - Particionamiento del grupo de procesos.....	50
Fig. 16 - Relación entre entrega de mensajes y cambio de view (1)	52
Fig. 17 - Relación entre entrega de mensajes y cambio de view (2)	53
Fig. 18 - Protocolo de tres fases para un multicast no dinámicamente uniforme.....	56
Fig. 19 - Protocolo de cuatro fases para un multicast dinámicamente uniforme	58
Fig. 20 - View Synchrony Estructurado.....	66
Fig. 21 - Protocolo de cambio de view de View Synchrony Estructurado	70
Fig. 22 - Arquitectura de red del sistema Relacs	73
Fig. 23 - Arquitectura de capas del sistema Relacs.....	74
Fig. 24 - Arquitectura de red del sistema Transis	82
Fig. 25 - Arquitectura de capas del sistema Totem.....	86
Fig. 26 - Arquitectura de capas de la solución	95
Fig. 27 - Estructura del Mss	102
Fig. 28 - Protocolo de cambio de view en un miembro local	106
Fig. 29 - Protocolo de cambio de view en un miembro controlador	109
Fig. 30 - Framework de aplicación.....	112
Fig. 31 - Composición del Servidor FEVS.....	121
Fig. 32 - Composición del Cliente FEVS.....	123
Fig. 32 - Arquitectura de la WDVM.....	155
Fig. 33 - Modelo de tolerancia a fallas de la WVDM.....	158



1 Abstract

El crecimiento de los sistemas distribuidos y de la demanda de aplicaciones complejas y con altos requisitos de confiabilidad que corran sobre estos hace que cobren mucha importancia las abstracciones que faciliten la tarea de razonar y diseñar aplicaciones confiables sobre dichos sistemas distribuidos. Una de estas abstracciones es View Synchrony, que permite al diseñador de una aplicación abstraerse de la mayoría de los problemas debidos a fallas o asincronías.

La utilización de View Synchrony para el desarrollo de aplicaciones distribuidas de gran escala presenta sin embargo problemas debidos a la pobre escalabilidad del protocolo que provoca que este colapse con una cantidad relativamente pequeña de procesos. Por esta razón se ha modificado la abstracción para lograr soluciones que puedan manejar una cantidad superior de miembros sin colapsar. Esta modificación a View Synchrony es conocida como Structured View Synchrony.

En esta tesis se presenta e implementa una arquitectura basada en Structured View Synchrony que ofrece los mecanismos de tolerancia a fallas necesarios el desarrollo de aplicaciones distribuidas en gran escala y un framework que permite la utilización de dicha arquitectura. Además se diseña la arquitectura necesaria para proveer mecanismos de tolerancia a fallas a la máquina global distribuida para internet definida en [EST/00].

The grow of distributed systems and the need of complex and reliable applications that use these systems has increased the importance of abstractions that ease the design of those applications.

One of these abstractions is View Synchrony that allows the designer of a distributed applications to overlook the majority of the problems due to failures and asynchronicities.

However, the utilization of View Synchrony in large scale distributed applications is not posible, due to scalability issues that cause the collapse of the protocol even when working with a small number of process. For that reason, View Synchrony has been modified to allow the management of a larger number of members without collapsing. This modification of View Synchrony is known as Structured View Synchrony.

This thesis presents an architecture based on Structured View Synchrony that provides fault tolerance mechanisms for the development of large scale distributed applications and a framework that allows the utilization of that architecture. Besides, we design the architecture needed to provide fault tolerance to the internet-based global machine defined in [EST/00]



2 Prólogo

Queremos comenzar este documento haciendo un poco de historia. La gestación de esta tesis comenzó hace aproximadamente 3 años cuando el profesor del laboratorio VIII de compiladores Alfredo Vega weiss nos invitó a realizar la tesis de licenciatura bajo su consejo y dirección. La idea original era realizar una tesis sobre algún aspecto del área de compiladores. Así fue como comenzamos a leer e investigar papers acerca de Java (lenguaje que por esos tiempos comenzaba a ser muy conocido y exitoso), optimizaciones de compiladores, lenguajes paralelos y cualquier otro tema relacionado con el área de compiladores. En ese momento éramos un grupo de cuatro alumnos con la intención de encontrar un tema lo suficientemente complejo como para realizarlo en conjunto o dividirlo para realizarlo en grupos de a dos.

Esta investigación nos llevó desde estudiar los bytecodes de java, la estructura de la JVM y varios lenguajes paralelos hasta estudiar algunos sistemas distribuidos (PVM, MOSIX, DCE, JADE, etc.), y de allí terminó por interesarnos en el ambiente en el cual se pueden llegar a correr dichas aplicaciones, Internet. Por esos tiempos Internet ya se perfilaba como el gran motor de investigación e interés de los siguientes años que luego fue así que nos pareció natural tratar de unir algunos de esos temas relacionándolos con Internet.

En esos momentos Internet se había convertido ya en una red mundial de arquitecturas heterogéneas interconectadas, cuyo fin principal era el intercambio de datos más o menos estáticos (e-mail, FTP, browsing). Con la aparición de Java, y otros lenguajes para desarrollo sobre Web sumado al aumento en la velocidad de la red, se sumó un nuevo uso: como plataforma para la ejecución de aplicaciones.

Fue así como un día dimos con la idea de crear una máquina virtual distribuida sobre Internet que permitiera utilizar a la inmensa cantidad de recursos computacionales conectados como si formarían parte de una única máquina global. Por supuesto que la idea no era utilizar a todas las máquinas del mundo en una supercomputadora sino analizar y estudiar que problemas presentaría una máquina virtual compuesta por muchas máquinas sobre Internet. Notoriamente al poco tiempo comenzamos a encontrar en nuestras investigaciones sobre Internet más y más documentos y proyectos relacionados con nuestra idea.



Hacia comienzos del año 1998 terminamos de definir un documento que presentaba el marco global de todos los conceptos que se deberían tener en cuenta en la definición de una computadora virtual sobre un sistema distribuido débilmente acoplado como Internet (la denominamos WDVM: World Distributed Virtual Machine). El documento enumeraba que aspectos se necesitan atacar y que problemas resolver para la construcción de una máquina con las características citadas. Presentamos este documento a algunos profesores del departamento de computación y los comentarios fueron unánimes: "muy bueno pero imposible de realizar". el tiempo necesario para la definición de tal máquina excedía con creces el poco tiempo disponible incluso por cuatro alumnos con trabajo a tiempo completo.

Un poco desilusionados empezamos a intentar encontrar la forma de dividir la tesis en temas más acotados con el fin de realizar aunque más no fuera la definición de parte de la máquina virtual. Así fue como llegamos a mediados de 1998 cuando atendimos la ECI "Sistemas Distribuidos Asincrónicos" del profesor Alberto Bartoli de la universidad de Pisa. Esta ECI se basó en los últimos adelantos sobre sistemas distribuidos asincrónicos tolerantes a fallas. La idea surgió entonces nítida: Dos tesis; una, la infraestructura de distribución del procesamiento de la máquina virtual y la otra, la de definición de los mecanismos de tolerancia a fallas de dicha máquina.

Así fue como a principios de 1999 comenzamos con la etapa final de esta tesis, la definición de los mecanismos de tolerancia a fallas de una máquina distribuida global. Estudiamos fuertemente toda la teoría relacionada con la tolerancia a fallas y en especial con la abstracción conocida como View Synchrony o sincronía virtual. Este concepto, de alto grado de complejidad, fue interesándonos cada vez más hasta que cobró tanta fuerza que se convirtió en el tema principal de nuestra tesis. Decidimos, entonces, definir la arquitectura necesaria para soportar la abstracción de View Synchrony estructurado (una adaptación de view synchrony que permite una mayor escalabilidad) en el contexto de un grupo de procesos en distintas máquinas que brindan un servicio confiable (basados en la replicación). Por encima de nuestra arquitectura se necesita, además, un framework de aplicación que simplifique el desarrollo de tales servicios y a ello nos avocamos también. Ambos objetivos se convirtieron, como dijimos, en el centro de nuestra tesis y a ellos se debe el título que escogimos. Realizamos, claro está, una implementación de ambos elementos (la arquitectura y el framework) como modo de validar y probar nuestras ideas.

Por último, recientemente, y como una forma de cerrar el círculo que comenzamos hace algunos años retomamos el tema de la máquina virtual distribuida sobre Internet y realizamos una agregado al modelo propuesto por



Esteban/Weinbinder [EST/00] de la infraestructura de la máquina global para agregarle tolerancia a fallas.

2.1 Guía de lectura

Comenzamos presentando el trabajo que dió origen a nuestro estudio de los mecanismos de tolerancia a fallas, la definición del marco conceptual a tener en cuenta en la definición de una máquina global distribuida sobre Internet.

A continuación, para facilitar el entendimiento de los conceptos de la tolerancia a fallas en general y en particular de la abstracción de View Synchrony se realiza una descripción de la teoría, protocolos y algoritmos necesarios en el capítulo de "State of the art (teoría y protocolos de view synchrony)". Luego se presentan otros sistemas similares basados en los mismos conceptos de VS, a saber: Relacs, Transis, Horus y Totem.

A partir de este estudio, en el capítulo "definición de la arquitectura para View synchrony estructurado" nos concentramos en definir la estructura de capas necesaria para una máquina que albergue procesos servidores tolerantes a fallas de acuerdo a SVS. Luego en "framework de aplicación" se presenta la interfaz de nuestra API para desarrollar aplicaciones sobre nuestra arquitectura.

Para verificar el modelo de la arquitectura propuesta realizamos una implemetación completa en cuanto a funcionalidad y eficiencia de algortimos. Los detalles del diseño, pruebas y resultados de las mismas conforma el capítulo "Pruebas"

Por último presentamos la extensión al modelo de la infrestructura de la WDVM (World Distributed Virtual Machine) propuesto por Esteban/Weinbinder agregandole los mecanismos de tolerancia a fallas necesarios. Creemos que este modelo sirve como base para la implementación de una WDVM tolerante a fallas.

Un repaso de los objetivos planteados y su análisis de cumplimiento conforman el capítulo de "conclusiones" del presente trabajo.



3 OBJETIVOS

- **Diseño de una nueva arquitectura de software para View Synchrony Estructurado y definición de un framework de aplicación para su utilización.**
- **Aplicación de la arquitectura propuesta para la implementación de los mecanismos de tolerancia a fallas de una máquina distribuida global sobre internet.**

3.1 Introducción

Los últimos años han visto un crecimiento enorme en la escala y extensión geográfica de los sistemas distribuidos debido fundamentalmente al éxito de Internet. Al mismo tiempo también se ha incrementado la demanda para aplicaciones cada vez más complejas y con requisitos de confiabilidad cada vez más exigentes. Areas como la bancaria, el comercio internacional, las telecomunicaciones, el control de procesos industrial, el trabajo cooperativo y muchas otras requieren, cada vez más, soluciones de sistemas distribuidos de gran escala. [SAR/98] [BAR/96] [SET/99] [GRI/97] [OSF/96] [CHR/96] [EST/00] [GEI/94] [GRI/94] [BAL/91] [BAL/92]

Sin embargo las extensiones geográficas -cada vez más grandes- y la masificación en la utilización de estos sistemas, productos de la globalización, conspiran en contra de la confiabilidad requerida. Tanto la extensión geográfica de un sistema como la impredecibilidad de la carga impuesta por muchos usuarios hacen que estos sistemas sean caracterizados como eminentemente **asincrónicos** en el sentido de que los retardos en la comunicación y las velocidades de cómputo relativas no pueden ser acotados con certeza. Las aplicaciones que deben ser desarrolladas en esos ambientes suelen ser complejas y difíciles de implementar debido a la multiplicidad de los escenarios de fallas posibles. Esto es así ya que la imposibilidad de comunicarse en estos sistemas no puede ser atribuida a una sola causa: el destino puede haber fallado, puede estar sobrecargado y, por lo tanto, lento, el link de comunicación puede estar roto o con problemas que provocan un gran retardo en las comunicaciones. [BIR/96] [BAB/95] [BAR/98]

Una abstracción muy útil para razonar y diseñar aplicaciones distribuidas para el difícil ambiente antes descrito es **view synchrony** en el contexto de **grupos de procesos**. Definiremos a un **grupo** como a un conjunto de procesos que



comparten un estado y cooperan para lograr un objetivo en común. La composición del grupo es dinámica debido a que los procesos pueden unirse o desconectarse del grupo (voluntariamente o por un fallo). El concepto de **view synchrony** permite abstraerse de la mayoría de los problemas debidos a fallas o asincronía. En primer lugar transforma las fallas de un proceso en eventos de cambios de pertenencia al grupo a través de **views** que son acordadas entre todos los miembros que sobrevivieron al fallo. Por otra parte VS provee garantías de atomicidad acerca del conjunto de mensajes entregados globalmente en el grupo ante un cambio de view. Así, View Synchrony asegura que todos los procesos que sobrevivieron al cambio de view recibieron y procesaron el mismo conjunto de mensajes en la view anterior y además que todos tienen la misma percepción respecto a los integrantes de la nueva view – es decir, saben quien falló y quien no-. [BIR/96] [BAR/98] [BAB/94] [BA2/94] [BAB/95] [BAB/96] [SCH/96] [VER/96]

La aplicación de view synchrony para facilitar el desarrollo de aplicaciones distribuidas de gran escala presenta, sin embargo, un inconveniente fundamental: su escalabilidad. La implementación de un servicio de VS implica como costo primario un incremento en la cantidad de mensajes que se intercambian entre los miembros del grupo. A medida que la cantidad de miembros se incrementa crece también la cantidad de mensajes que se deben intercambiar ante una falla de un miembro (durante el protocolo de cambio de view) afectando grandemente la performance del sistema. Pruebas realizadas han comprobado que el protocolo colapsa con una cantidad de miembros mayor o igual a cuarenta (40).

Una modificación a view synchrony [Birman] conocida como **view synchrony estructurado** permite aumentar la escalabilidad del protocolo. La idea es dividir al conjunto de miembros del grupo en una estructura de árbol de 2 niveles que permita disminuir la cantidad de mensajes que debe manejar cada miembro del grupo durante el protocolo. Se ha demostrado que con esta modificación se pueden lograr grupos de hasta 800 miembros. [GOU/94]

El objetivo primario del presente trabajo es diseñar una arquitectura de software que ofrezca los mecanismos de tolerancia a fallas necesarios para el desarrollo de aplicaciones distribuidas de gran escala basados en la abstracción de View Synchrony Estructurado (SVS). Además definiremos un framework de aplicación para que la abstracción pueda ser utilizada para implementar aplicaciones distribuidas.

El objetivo secundario es aplicar la arquitectura definida para proveer los mecanismos de tolerancia a fallas de la máquina global distribuida para internet definida en [EST/00].



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

Para verificar la arquitectura propuesta realizaremos una implementación total de la misma en cuanto a funcionalidad. Es decir una implementación que cumpla con las propiedades requeridas para view synchrony estructurado. Nos orientamos hacia una implementación eficiente de los algoritmos y protocolos aunque no es nuestra prioridad la optimización. De cualquier manera dejaremos marcado cuando eso suceda y de que forma se puede mejorar.



4 MARCO CONCEPTUAL PARA LA COMPUTACION GLOBAL (WDVM – World Distributed Virtual Machine)

En este capítulo definimos un marco conceptual que debería considerarse en el diseño de una máquina distribuida global denominado WDVM-World Distributed Virtual Machine. Para esta definición hemos tenido en cuenta varias de las soluciones propuestas por el mundo científico en el área de estudio de la computación global.

4.1 Motivación

A medida que la velocidad de la comunicación en sistemas de computación distribuidos débilmente acoplados se incrementa, tales sistemas se vuelven cada vez más atractivos para ejecutar aplicaciones paralelas. El mejor ejemplo de este fenómeno es Internet [OSF/96], [CHR/97], [SAR/98]. En la actualidad, Internet se ha convertido en una red mundial de arquitecturas heterogéneas interconectadas, cuyo fin principal es el intercambio de datos (e-mail, FTP, browsing). Con la aparición de Java, y otros lenguajes para desarrollo sobre Web, se ha sumado un nuevo uso: como plataforma para la ejecución de aplicaciones.

Los millones de máquinas interconectadas actualmente a Internet están la mayor parte del tiempo ociosas, ofreciendo un potencial recurso computacional gigantesco [BAR/96], [SAR/98]. La posibilidad de utilizar estas máquinas como nodos de una super máquina distribuida ha sido estudiada desde varios enfoques y por varios grupos de investigación, entre estos podemos mencionar a Javelin, [CHR/97], Charlotte [BAR/96], Bayanihan [SAR/98], Legion [GRI/94] y OSF DCE [OSF/96].

El marco que presentamos en este capítulo tiene como objetivo determinar y mencionar los aspectos de diseño básicos para el desarrollo de una máquina distribuida global, que permita ejecutar aplicaciones sobre un conjunto de máquinas distribuidas conectadas a la WEB basándose en un sistema distribuido que brinde independiencia de hardware y de sistema operativo. Las máquinas participantes serán vistas por dichas aplicaciones como una única máquina. La máquina distribuida global virtual (WDVM) estará compuesta por máquinas heterogéneas, representadas como nodos, conectadas a la WEB.



El diseño de la máquina virtual se asemeja a la siguiente figura:

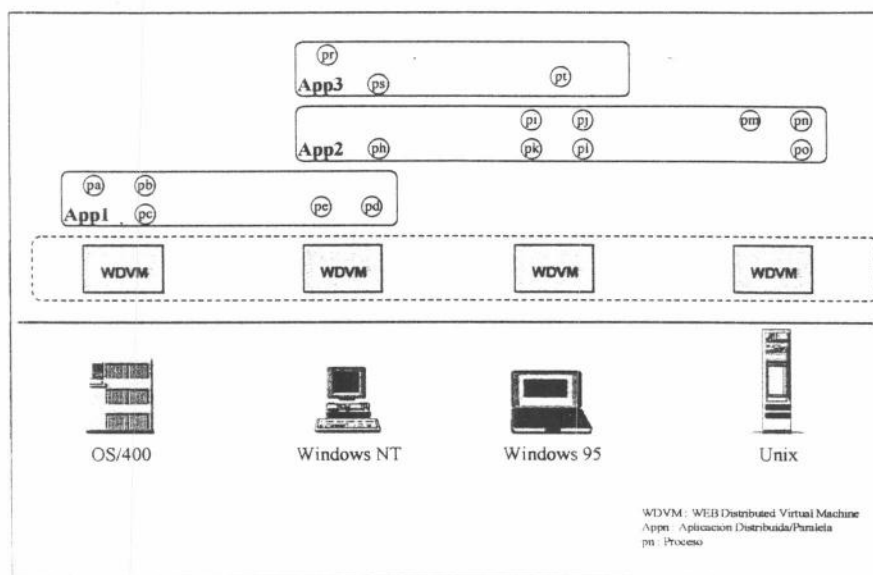


Fig. 1 - Esquema de la WDM

Una definida en este entorno posee las siguientes ventajas potenciales :

- Escalabilidad:

En la actualidad, una aplicación no puede usar más recursos (CPU, memoria, disco, etc.) que los disponibles originalmente en el sistema sobre el cual corre. En cambio con la WDM los recursos disponibles originalmente para la aplicación pueden ir creciendo en forma gradual a través del ingreso de nuevos nodos a la máquina virtual puede crecer tanto como máquinas ociosas hay en el mundo conectadas a Internet.

- Potencia de cálculo:

Existen en la actualidad más de 179 millones de máquinas conectadas a la WEB y su crecimiento es exponencial, con lo que la WDM podría potencialmente hacer uso de todo ese poder de cálculo.

- Disponibilidad

Dada la ventaja de la naturaleza paralela de un sistema distribuido, la disponibilidad de los servicios brindados puede incrementarse en medida de los recursos disponibles, esto es, una instancia de un servicio puede continuar corriendo en el caso que otra instancia del mismo halla fallado cualquiera sea el motivo de la falla, entre otros, sobrecarga, falla temporal



de la red, o fallas de los equipos donde corre el servicio. Dada la cantidad de recursos disponibles se pueden generar tantas instancias de los servicios esenciales como para asegurar una disponibilidad del 100% en todo momento.

- Acceso globalizado:
Como la WDVM corre sobre la WEB, cualquier usuario que disponga de un browser podría acceder a una supercomputadora desde su casa o desde donde él se encuentre.
- Procesos eternos:
En los sistemas de cómputo distribuido o paralelo la cantidad de recursos es limitada. Por lo tanto, un proceso CPU-Bound que esté corriendo desde hace cierto tiempo o bien tiene exclusividad (con lo cual el costo de correr ese proceso es altísimo), o es bien es penalizado disminuyéndole su prioridad o inclusive sacándolo de ejecución para permitir que otros procesos se ejecuten.
En la WDVM, debido a la virtualmente infinita disponibilidad de recursos, podrían existir procesos libre de penalización y procesos que corran eternamente. Así por ejemplo, se podría lanzar un proceso que calculara la progresión del número π hasta una cantidad infinita de decimales y una vez por año entrar al site para ver como va el cálculo.
- Facilidad para el desarrollo y ejecución de aplicaciones compartidas:
Dado que la plataforma virtual creada por la WDVM es de naturaleza distribuida se vería facilitado tanto el desarrollo como la ejecución de aplicaciones para grupos de personas que se encuentren en distintas locaciones.

Veamos ahora las dificultades que acarrea definir una máquina global sobre un sistema distribuido débilmente acoplado como la WEB:

- La poca estabilidad de los nodos y sus conexiones atentan contra la integridad de las aplicaciones que corran sobre estos [TA2/96].
- Al ser la WEB un sistema abierto se incrementa el riesgo de accesos indebidos a los datos o aplicaciones [LAW/96]
- Las impredecibles fluctuaciones que se producen en el uso de la red dificultan la comunicación entre los nodos. Esto afecta la performance de los procesos corriendo sobre la WDVM.
- La congestión en los enlaces debido a una administración centralizada puede provocar un incremento desmedido de la latencia.



- Lograr la disponibilidad parcial o total de recursos en una Intranet es mucho más sencillo que convencer a los usuarios de Internet que cedan sus máquinas.

El objetivo de este marco conceptual es trasladar las ventajas antes mencionadas a las aplicaciones minimizando la degradación potencial producida por los problemas previamente comentados.

4.2 Aspectos fundamentales en la definición del marco conceptual

De nuestra investigación surgió la necesidad de contemplar ciertos aspectos fundamentales para definir el marco conceptual que servirá para alcanzar el objetivo propuesto.

Las aplicaciones requerirán de servicios de **migración de procesos** para brindar la capacidad de trasladar un proceso desde un procesador hacia otro, ya sea por pedido del proceso (explícito) [LAW/96] [LAN/97] o por necesidades externas a éste (implícito). También necesitarán **mecanismos de comunicación entre procesos** para ofrecer facilidades para el intercambio de datos. Entre estos mecanismos podemos enumerar el **pasaje de mensajes** y la **memoria virtual distribuida** (permite simular un espacio único de memoria para todos los procesadores) [TAN/96].

Para acceder en forma ordenada a los recursos compartidos los procesos necesitarán **mecanismos de sincronización** [TAN/96].

Las aplicaciones se ejecutarán sobre un entorno distribuido débilmente acoplado por lo que mecanismos relacionados con la **seguridad** que brinda el entorno y los elementos que permitan monitorear la ejecución de una aplicación (**debugger**) cobran significativa importancia.

Tanto el uso de memoria compartida como la sincronización entre procesos o cualquier otro mecanismo de comunicación deberá tener en cuenta la resistencia a demoras en los accesos a datos remotos o las fallas de los nodos que los contienen (**tolerancia a la latencia**).

La ejecución de aplicaciones de alta performance requiere de mecanismos inteligentes que permitan manejar los recursos disponibles en la red de forma eficiente y transparente, distribuyendo la carga de trabajo dinámicamente entre los distintos componentes de la red [LAD/95]. Las técnicas principales para la



distribución dinámica del trabajo son el **balance y la compartición de la carga**, que usan alguna forma de ejecución remota y **migración de procesos**. Estas técnicas deben ser supervisadas por los distintos algoritmos, para poder tomar ventaja de los recursos disponibles en la red.

Por otro lado, se deberá proveer de facilidades para la **administración y configuración de la red de nodos** que forman la máquina virtual [TAN/96]. Deben existir mecanismos para permitir el acceso a los recursos de cada nodo. Estos mecanismos deben ser configurables en base al porcentaje de utilización de los recursos que se asignan a la máquina virtual, y seguros, para no permitir accesos indebidos o no autorizados a los recursos del "anfitrión" (llamaremos anfitriones a los nodos que sólo ceden una parte de sus recursos para la ejecución de aplicaciones en la WDVM).

La inestabilidad de las conexiones en la WEB obliga a considerar aspectos relacionados con la **tolerancia a fallas** más que en cualquier otro sistema distribuido [TAN/96].

Por último, la motivación que nos llevó a la definición de este nuevo marco conceptual, trae aparejada también la necesidad de analizar la definición de una metodología de desarrollo que facilite el acercamiento a las soluciones de los problemas inherentes a este marco. Es decir, dar un nuevo enfoque al análisis y desarrollo de aplicaciones que enfrenten los problemas en el área de sistemas distribuidos/paralelos.

Asociado al concepto de metodología de desarrollo se encuentran la creación de un entorno de desarrollo y la definición de un lenguaje de manera de completar así este marco conceptual.

En las próximas secciones analizaremos más en detalle la problemática de estos conceptos clasificándolos, según su naturaleza, en *Servicios para la ejecución de aplicaciones paralelas y distribuidas* y *Servicios para la administración de la máquina virtual*.



4.3 Servicios para la ejecución de aplicaciones paralelas y distribuidas

En esta sección expondremos los aspectos de diseño relacionados con la ejecución de aplicaciones, que deberían contemplarse en el desarrollo de una máquina virtual global.

4.3.1 Mecanismos de comunicación entre procesos

La máquina virtual debe brindar soporte para computación distribuida (múltiples usuarios independientes trabajando con diferentes proyectos) [BAL/91] así como también para computación paralela (una sola aplicación branch and bound usando el conjunto completo de procesadores) en un ambiente de Web con múltiples máquinas heterogéneas interconectadas.

Debido a esto, una aplicación típica consistirá de un conjunto de procesos relacionados ejecutándose en la misma máquina o en máquinas distintas. Para hacer más sencillo y coherente el modelo de programación, la WDVM debería especificar un único mecanismo de comunicación entre procesos para los 2 casos.

Un mecanismo de comunicación sencillo y homogéneo es el de pasaje de mensajes (message passing) [TAN/96] [BRA/93] que permite a un proceso WDVM enviar un mensaje a cualquier otro proceso de la WDVM. Se debe brindar un mecanismo de naming y direccionamiento de procesos para identificar a los procesos individuales dentro de una misma aplicación y su ubicación; esto está relacionado con la administración y migración de procesos.

Como facilidades básicas el sistema debe contar con primitivas para enviar y recibir mensajes. Un modelo posible, utilizado exitosamente en PVM [GEI/94], es el siguiente :

- facilidad proceso a proceso
 - ⇒ send - Asincrónico
 - ⇒ receive - Bloqueante (blocking)
 - No Bloqueante (non – blocking)
- facilidad de comunicación de grupos (de un proceso a un grupo de procesos)



- provee un mecanismo de envío de mensajes confiable y ordenado dentro de un grupo de tareas, a través del cual a todo receptor se le asegura que recibirá el mismo conjunto de mensajes en el mismo orden.

Al implementar un mecanismo de comunicación de este tipo en un ambiente inseguro como el de la WEB se presentan algunas dificultades y particularidades que deben ser tenidos en cuenta al momento del diseño.

Debido al costo de la comunicación, [TA2/96] se debe tratar de minimizar el tráfico de mensajes entre procesos en máquinas distantes. Esto también debe ser tenido en cuenta por los mecanismos de balance de carga al migrar procesos [LAD/95] y por el usuario o programador al enviar procesos a otras máquinas [REN/95] [LAW/96].

Otra variable a minimizar es la cantidad de mensajes que 2 procesos dados intercambien, debido a que una mayor cantidad de mensajes indica una gran interdependencia y deberían entonces estar en una misma máquina [TA2/96] [LAD/95].

El pasaje de mensajes en la computación distribuida permite administrar el conjunto de datos que una aplicación dada maneja de una manera razonablemente eficiente. Los datos sólo son enviados adonde se necesiten (on demand) aunque la carga de realizar esta división recae sobre el programador [TAN/96] [LEW/92].

En cuanto a la transparencia para compartir datos y sincronizarse, el pasaje de mensajes permite un grado de transparencia no muy alto debido a que un proceso determinado debe conocer a sus procesos pares, adonde se encuentran y encargarse el mismo del envío y la recepción de la información. Un mecanismo de comunicación de más alto nivel con el cual se logra una mayor transparencia es el de Memoria Distribuida Virtual, en la cual se simula un espacio de memoria compartido por todos los procesos de una aplicación dada. De esta manera los procesos pueden compartir datos (crearlos, leerlos y manipularlos) y sincronizarse sin importar adonde se encuentren y sin tener que manejar ellos mismos la comunicación a través de la red (network transparency) [TAN/96] [BRA/93].

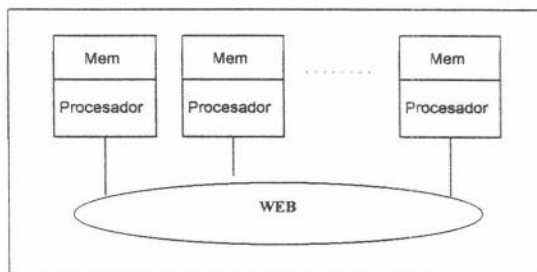


Fig. 2 - Esquema sin memoria compartida

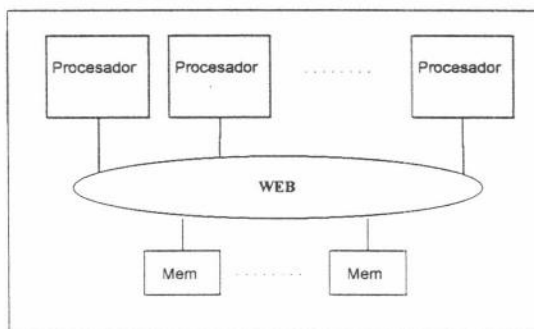


Fig. 3 - Esquema con memoria compartida

Típicamente una facilidad de este tipo se implementa como un administrador de memoria local en cada site, el cual se ocupa de manejar y actualizar en forma consistente los objetos que se van creando en los espacios de memoria de los otros sites (bajo la supervisión de administradores pares). La administración de los objetos, en cada administrador, puede ser lazy (solo en el momento en que un objeto va a ser accedido es solicitado a su administrador de memoria local) o eager (cada vez que un objeto se actualiza es enviado en forma broadcast a todos los sites). Una implementación híbrida combina a las dos estrategias anteriores, enviando al objeto actualizado solo a los sites que tienen una referencia previa a él [TAN/96] [REN/95] [LAW/96].

Un problema que tiene este mecanismo de comunicación y compartición de datos es el overhead de comunicación que causa, ya que la implementación de esta memoria distribuida virtual se realiza en base a una gran cantidad de mensajes entre administradores.

En todos estos casos es el sistema el que se ocupa del traslado de los objetos entre nodos cuando son requeridos o actualizados por un proceso dado. Una alternativa distinta es darle semántica a los objetos de datos, definiendo, por ejemplo, patterns de comunicación que puedan ser asociados con cada tipo de

objeto de manera que el programador tenga control del uso de la red (network awareness). Para esto es necesario diferenciar objetos con estado de objetos sin estado. Los objetos con estado residen en un solo nodo (se pueden referenciar desde cualquier otro nodo) y son migrados cuando necesitan ser actualizados. En cambio los objetos sin estado se pueden replicar (state awareness) y por ello no necesitan mecanismos de migración [REN/95].

Una solución posible podría utilizar ambos mecanismos de comunicación entre procesos en forma combinada. Definiendo una implementación en capas, se puede definir la capa inferior para que brinde servicios de pasaje de mensajes y una capa encima de esta que implemente la memoria distribuida. Así cada proceso podría elegir utilizar una u otra abstracción dependiendo de cada aplicación en particular. De esta manera incluso los procesos administradores de la WDVM podrían convertirse en procesos comunes utilizando el mecanismo de comunicación que sea necesario (teniendo en cuenta performance, abstracción, memoria disponible, etc.) en cada momento.

Otro mecanismo de comunicación entre procesos es la llamada a un procedimiento remoto (RPC) que permite a un proceso llamar a procedimientos localizados en otras máquinas. De esta manera cuando un proceso en un nodo A llama a un procedimiento en un nodo B, el proceso que realiza la llamada se suspende y la ejecución se continúa en el nodo B. La estructuras de datos necesarias se transportan de un lugar a otro mediante el pasaje de parámetros [TAN/96] [TA2/96] [BAL/91].

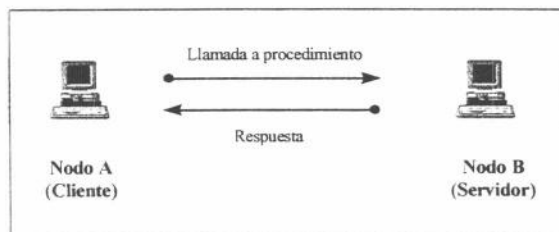


Fig. 4 - Esquema de llamada RPC

Así como la memoria compartida distribuida simula un mismo espacio de memoria para todos los procesos, el RPC simula un único espacio de procesamiento (procesador distribuido) para toda la aplicación.

4.3.2 Sincronización entre procesos

La necesidad de sincronización surge cuando 2 o más procesos deben acceder y actualizar estructuras de datos compartidas en forma concurrente. En un sistema



con un solo procesador, los accesos a estas regiones críticas se protegen con semáforos o monitores.

En un sistema distribuido como la WDVM, la necesidad de sincronizarse es aún más importante ya que las estructuras de datos compartidas se encuentran diseminadas en diferentes máquinas a lo largo de la WEB. Por ello se hacen necesarios nuevos mecanismos de sincronización para facilitar esta tarea [TAN/96] [BRA/93].

Los mecanismos de sincronización distribuidos pueden clasificarse en mecanismos de **alto** o **bajo** nivel de acuerdo al esfuerzo que requieran de parte de cada proceso para realizar la sincronización. Ambos intentan ordenar en forma mutuamente excluyente los accesos a las estructuras compartidas como si estuvieran entrando en una región crítica en un solo procesador [TAN/96].

Los mecanismos de bajo nivel exigen que cada proceso realice o participe activamente de la tarea de sincronizarse. Así, un ejemplo de un algoritmo de este tipo es el llamado "algoritmo centralizado", en el cual se elige un procesador coordinador dueño de todos los accesos a regiones críticas. Si un proceso quiere actualizar una estructura distribuida, envía una solicitud al coordinador, indicando la región crítica a acceder. Si ningún otro proceso está utilizando en ese momento la estructura compartida, el coordinador envía un permiso al proceso solicitante [TAN/96].

Todos los algoritmos de este tipo deben ser resistentes a pérdidas de mensajes y asegurar la exclusión mutua aún en un ambiente inestable como la WEB [TAN/96] [BRA/93].

Para obtener una abstracción de mayor nivel y evitar que el programador se deba ocupar del manejo de los accesos a las regiones críticas, de evitar deadlocks y recuperarse ante fallas, surge el concepto de Transacción.

En una transacción un proceso anuncia que quiere comenzar una transacción con otros procesos. Cada proceso se prepara entonces para actualizar las estructuras a las que quiere acceder el proceso iniciador. Cuando todos los participantes están listos se realiza la actualización de los datos. El modelo asegura que todas las operaciones se llevarán a cabo o que no se realizará ninguna (si un proceso falla y la transacción no puede completarse se realiza ROLLBACK) [TAN/96].

El modelo transaccional y sus algoritmos han sido ampliamente estudiados y desarrollados para las bases de datos distribuidas, así que habría que estudiar su posible implementación para la WDVM [TAN/96].



En este ultimo caso es el sistema el que se ocupa de manejar los aspectos de bajo nivel de la transacción (pérdida de mensajes, evitar deadlocks, locks distribuidos, etc.), por lo que a cada proceso se limita a manejar las siguientes primitivas [TAN/96]:

BEGIN TRANSACCTION
COMMIT
ROLLBACK



4.3.3 Mecanismo para el booteo de procesos

Se refiere a los pasos que deben realizarse para convertir un fuente en un código ejecutable y como "cargar" este ejecutable en la máquina virtual para su ejecución.

4.3.4 Distribución de código

Para la distribución de código entre los distintos nodos de la WDVM se pueden considerar dos tipos, dinámica y estática. La distribución dinámica se realiza cuando una aplicación envía parte de los cálculos a otros nodos durante la ejecución. Aquí primero se lanza una aplicación y luego se distribuye su código. Mientras que en una distribución estática, el sistema distribuirá primero el código, posiblemente a través de agentes móviles, para luego comenzar la ejecución de manera concurrente [LAN/97] [LAW/96].

4.3.5 Manejo de versiones

En un sistema de este estilo, debería existir un manejador de versiones ya que en un momento dado, el código de una aplicación en ejecución puede estar distribuido en miles de nodos y debe ser reemplazado por una versión nueva.

4.3.6 Seguridad

Uno de los principales problemas de un sistema distribuido es la seguridad. Aspectos como protección de confidencialidad y control de acceso deberán ser enfocados principalmente para prevenir accesos no autorizados entre las distintas aplicaciones que se ejecuten sobre la WDVM y así preservar la privacidad e integridad de dichas aplicaciones [TAN/96] [TA2/96] [LAW/96].

Aunque existe mucha experiencia en estrategias para lograr protección en un sistema distribuido débilmente acoplado (como la WDVM), no es sencillo lograr simultáneamente seguridad junto con otros atributos de garantía de funcionamiento porque suelen resultar objetivos contrapuestos. Confidencialidad y



disponibilidad son vistos usualmente como dos objetivos en conflicto: el único método para hacer que las aplicaciones paralelas sean eficientes será distribuyéndolas y replicándolas por toda la WEB y esto las hace más difícil de proteger de accesos indebidos [TAN/96] [LAW/96].

La seguridad será una preocupación fundamental. Los usuarios deben tener una idea clara de quién puede utilizar cual recurso y seguramente casi todos ellos querrán proteger los recursos "propios" (aquellos recursos del nodo anfitrión que no son cedidos a la máquina virtual).

En este entorno cobran importancia las definiciones relacionadas con identificadores únicos de nodos conectados a la máquina virtual, autenticación a nivel de nodo y a nivel de proceso, tablas de autorización para el acceso a cada recurso del anfitrión, entre otros.

Una primera clasificación de los conceptos de seguridad sobre la WDVM sería la división entre seguridad interna y externa. La primera está relacionada con los mecanismos necesarios para proteger y limitar el acceso a los recursos y datos entre las distintas aplicaciones en ejecución. Cada aplicación deberá tener un entorno privado en el cual deposite sus datos locales y que sea libre de colisiones o accesos no autorizados.

Por otro lado la seguridad externa es aquella que regula la segura interacción de las aplicaciones WDVM dentro del nodo anfitrión.

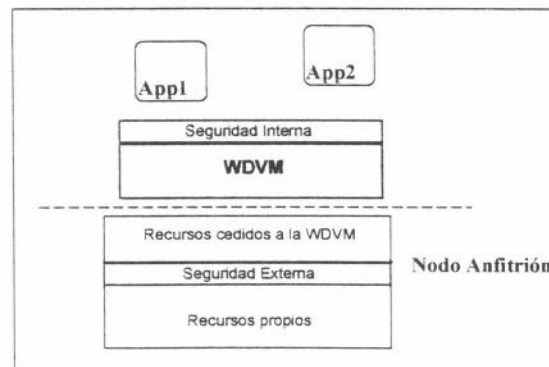


Fig. 5 - Capas de seguridad en la WDVM



4.4 Servicios para la administración de la máquina virtual

En esta sección expondremos los aspectos de diseño que deberían contemplarse en el desarrollo de una máquina virtual global para lograr una adecuada administración de todos los recursos que conforman la máquina.

4.4.1 Balance de carga

Una distribución despareja del uso de los recursos de los procesadores disponibles puede producir una considerable pérdida de la performance obtenida por el uso de paralelismo. Si un procesador está demasiado cargado se pueden producir bloqueos de procesos que podrían ejecutarse sin problemas en otro nodo [LEW/92] [LAD/95].

Para evitar estas ineficiencias se han desarrollado sistemas de scheduling avanzados en los que los procesos pueden cambiar de procesador en medio de su ejecución (process migration), pasando a ejecutar en otro procesador con menos carga. Este proceso se llama balance de carga. Aunque el balance de carga se relaciona directamente con el uso de procesador, el concepto puede ser extendido a otros recursos del sistema como por ejemplo memoria libre [LAD/95].

Los balances de la carga son realizados automáticamente por el sistema operativo y tienen varias consecuencias importantes sobre el diseño :

- Debe haber un mecanismo de migración de procesos transparente, que permita que a los procesos les sea indiferente el procesador en que se hallan [LAN/97]. Esto es relativamente fácil de lograr en sistemas distribuidos fuertemente acoplados [LAD/95], pero en nuestro caso implica diferenciar entre dos partes de cada proceso: la parte independiente de la ubicación y la parte que no puede ser migrada (por ejemplo por hacer uso de los recursos locales de un nodo). Es decir que debe haber una manera de lograr que el proceso siga haciendo uso de los recursos locales de un nodo desde un nodo remoto.
- Es necesario un mecanismo para medir el uso de los recursos en cada procesador. Esta medida debe ser ajustada a la realidad, pero no puede afectar la performance de los nodos, por lo que no se pueden realizar cálculos demasiado complicados. Hay que alcanzar un compromiso entre la exactitud del cálculo que se realiza y el tiempo que lleva realizarlo. Además hay que determinar la frecuencia con que se calculará esa medida. Este



punto es muy importante, ya que estas estimaciones son uno de los factores determinantes en la toma de decisiones sobre la reubicación de procesos [LEW/92] [LAD/95].

- La medida mencionada en el punto anterior debe ser distribuida a otros procesadores para que se puedan tomar decisiones sobre la migración de procesos. Este punto incluye varias decisiones de diseño como por ejemplo si las mediciones se realizan en forma centralizada o distribuida, la forma de distribución de estas mediciones y la periodicidad de las mismas, entre otras. Si se decide hacer la distribución en forma distribuida, hay que elegir también una forma de seleccionar a cuales nodos se les hará llegar la información [LAD/95].
- Otro punto importante es la elección del proceso que se va a migrar. El costo de la migración de un proceso es alto y debe realizarse con procesos que se espera que todavía van a ejecutar durante un periodo largo. Desafortunadamente no existe una manera de predecir el tiempo de ejecución que le queda a un proceso, por lo que es necesario desarrollar heurísticas para seleccionar procesos adecuados. Una heurística existente consiste en pedir a los procesos un "período de residencia" en el nodo para ser elegibles para migrar. De esta manera se evita migrar procesos muy cortos. Además debe tenerse en cuenta que los procesos que se migran no deben hacer uso intensivo de recursos locales, ya que el acceso a dichos recursos cuando el proceso esté en otro nodo exigirá hacer uso de la red. Una buena elección son los procesos CPU bound, que realizan una gran cantidad de cálculos antes de mostrar un resultado.
- En general hay que mantener información sobre cada proceso que permita predecir su comportamiento posterior. Parte de esta información puede ser proporcionada por el programador o también recopilada de anteriores corridas del proceso. Todos los métodos actuales de balance de carga entran en acción demasiado tarde: cuando ya se ha producido un desbalance importante. Sería interesante desarrollar algoritmos que realicen 'balance de carga predictivo', previendo futuros desbalances y evitándolos.

Es importante notar que aunque el balance de carga afecta la performance del sistema, se contradice con otros objetivos que también afectan a ésta, como por ejemplo la reducción de la comunicación entre procesos. La reducción de la comunicación entre procesos es crucial en un sistema distribuido débilmente acoplado porque apunta a minimizar el uso de la red, que es uno de los recursos



que mas penalizaciones impone a la performance de los programas. Ambos objetivos se contraponen porque al distribuirse más la carga, más procesos con necesidades de comunicación quedan en máquinas diferentes, forzando un aumento del uso de la red. Esto nos lleva a ver que los índices de medición antes mencionados deberían ser integrados en índices compuestos por varias medidas. Cada nodo podría mantener tablas de donde pueda extraer información sobre el costo de acceder a una cantidad dada de un recurso en cada nodo. Ese costo está relacionado con el uso del recurso en cada nodo, con la carga del procesador del nodo donde está el recurso y con el costo de migrar un proceso hasta dicho nodo.

En general todos estos algoritmos deben ser cuidadosamente diseñados porque su influencia en la performance del sistema es decisiva [LAD/95].

4.4.2 Tolerancia a fallas

Todo sistema distribuido implica contemplar tolerancia a fallas. Dentro del marco que pretendemos diseñar esta necesidad se incrementa debido al ambiente inestable que utilizaremos (la WEB). Algunas razones son:

- La **separación física** de los nodos conectados a la WEB (que pueden estar en diferentes cuartos, ciudades, países, etc.) implica que estos pueden fallar independientemente, por ello se hace necesario la implementación de estrategias de tolerancia a fallas: *cuando un procesador sufre una falla, la carga puede distribuirse sobre el resto de los procesadores no fallados.*
- En un sistema global como el representado por el marco WDVM se puede utilizar el concepto de **redundancia** (y replicación) debido a que se ejecutará sobre la WEB y potencialmente es posible contar con mas nodos que los estrictamente necesarios. Los protocolos utilizados para implementar este concepto se pueden aplicar fácilmente con objetivos de tolerancia a fallas.
- A medida que la cantidad de nodos conectados aumente, la probabilidad de que alguno sufra una falla también aumentará. Sería ilógico pensar que cada vez que algún componente (de los miles que potencialmente podrían estar ejecutando una determinada aplicación en paralelo) fallé, toda la aplicación sufrirá un fracaso. Esto implicaría que la mayor parte del tiempo el sistema estaría inutilizable.

Dentro del concepto de tolerancia a fallas, se deberán considerar las fallas que puedan afectar a los nodos en el momento de ejecución de una determinada



aplicación (en este punto nos interesa que la máquina virtual pueda sobrevivir a las fallas de un determinado nodo).

Se debería considerar la ocurrencia de distintos tipos de fallas sobre WDVM y el impacto sobre la máquina virtual y las aplicaciones en ejecución, de acuerdo a si las fallas son transitorias (ocurren una vez y luego desaparecen), intermitentes (aparecen y desaparecen continuamente) o permanentes (son las que continúan hasta que es reemplazado el componente) [TAN/96] [BIR/96].

Otro aspecto a contemplar es el envío de mensajes en situaciones de fallas. Que sucede si los nodos se detienen y no responden a los mensajes, que sucede cuando anuncian que ya no están funcionando o en caso de que funcionen erróneamente que sucede cuando proporcionan respuestas incorrectas.

Claramente en un sistema como la WDVM (netamente asíncrono) controlar fallas es mucho más difícil que en un sistema sincrónico. En un sistema sincrónico un procesador envía un mensaje y sabe que la ausencia de respuesta dentro de T segundos significa que el pretendido receptor ha fallado, puede entonces realizar una acción correctiva [BRA/93].

Un método general para la tolerancia de fallas consiste en el uso de redundancia. Con la redundancia física se agrega un equipo adicional para permitir que el sistema como un todo, tolere la pérdida o el mal funcionamiento de algunos componentes. Para lograr esto en la máquina virtual se podrían utilizar más nodos de los necesarios, de modo que, si algunos de ellos fallan, el sistema pueda seguir funcionando de manera correcta.

Se deberían considerar las distintas maneras de organizar nodos adicionales para casos de falla. Una alternativa podría ser que todos los nodos adicionales funcionan en paralelo, en cambio otra sería reemplazar un nodo por otro cuando el primero falla. Para cada una de estas opciones se deberán evaluar el grado de réplica requerido; el desempeño para el caso promedio y el peor caso en ausencia de fallas; y el desempeño para el caso promedio y el peor caso cuando ocurre una falla [TAN/96].

También es necesario aplicar conceptos de tolerancia a fallas ante la necesidad de que los procesos coincidan en algo a través de un acuerdo distribuido (sincronización). El objetivo general de los algoritmos de acuerdo distribuido es lograr que todos los procesadores no defectuosos alcancen el consenso acerca de cierto tema, en un número finito de pasos (el problema de los dos ejércitos es un conocido ejemplo que detalla el problema) [TAN/96] [BRA/93] [BIR/96].



Otro aspecto a considerar es la necesidad de contar con checkpoints distribuidos mediante los cuales las aplicaciones puedan volver a un estado seguro cuando se produzca alguna falla.

4.4.3 Administración de la red de nodos

La red de nodos de la máquina virtual está constituida por máquinas de arquitectura heterogénea conectadas a la WEB que corren una instancia local de la WDVM de manera exclusiva o semi exclusiva. Es decir, en el caso de ser un nodo anfitrión, la instancia local puede ejecutarse junto con otras aplicaciones.

Como dijimos anteriormente, un punto importante de la WDVM es que la escalabilidad se implemente de una manera muy sencilla, así una persona que desee agregar su máquina a la red de nodos pueda hacerlo casi tan fácil y rápidamente como conectarla a la WEB.

La idea es que la persona que quiera conectar la máquina a la red de nodos, decida primero ciertos parámetros de configuración incluyendo:

- Uso exclusivo o semiexclusivo
- Tiempo de uso (por cuanto tiempo cederá su máquina a la red de nodos).

En el caso de ser de uso semiexclusivo deberá especificar además:

- Cantidad de recursos (porcentaje de procesador, memoria y espacio temporal en disco) asignados a la instancia local de la WDVM.

Para el proceso de conexión, desconexión y administración de los nodos un posible modelo sería el de inscripción de la máquina a una lista de nodos disponibles en algún servidor de administrador de recursos. De esta manera, cuando un proceso, ejecutando sobre la red de nodos WDVM, necesita más recursos, se selecciona alguno de los nodos de la lista para compartir la carga.

Otro modelo de administración sería dividir los nodos de la red en regiones, definiendo a la región como un conjunto de nodos que están "cerca" entre sí. La cercanía puede estar basada en la velocidad de la conexión entre ellos, así cuando un nuevo nodo desea conectarse a la red, el administrador de regiones es el responsable por indicar cual es la región que le corresponde. Una ventaja adicional del modelo de regiones se basa en que la ejecución de una determinada



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

aplicación se intenta realizar dentro de una sola región para minimizar de esta manera la latencia de la red. Obviamente la composición de cada región con respecto a cuantos y cuales nodos la integran debe ser flexible y reconfigurable en base a la demanda y disponibilidad de recursos. Por otro lado, una desventaja de este modelo es que la administración de las regiones deberá resolverse dentro de la WDVM ya que Internet no maneja una jerarquía de regiones [TAN/96] [TA2/96].



Un punto muy importante es poder configurar el momento en el cual las máquinas serán utilizadas como nodos. De esta manera, se pueden aprovechar los tiempos muertos de las máquinas, que normalmente están inactivas durante la noche. Como esta red de nodos puede ser tan amplia como la WEB misma, con una distribución de nodos a nivel mundial, es de esperarse que siempre habrá una gran cantidad de los mismos disponibles, ya que la mitad del mundo esta siempre durmiendo. Incluso se podría definir una especie de protector de pantalla que corra una instancia de la WDVM para aprovechar aún más al procesador cuando no se lo esta usando¹.

Otro punto a tener en cuenta es que como la WEB no tiene una administración centralizada, lo que le permite entre otras cosas una gran tolerancia a fallas, sería interesante trasladar esa característica a la administración de la red de nodos, para evitar además el alto tráfico de mensajes de control en las cercanías del nodo administrador.

En el caso de los nodos anfitriones, se deben implementar mecanismos de chequeo para controlar que cantidad real de recursos dispone inicialmente de los configurados. Además se deberían definir políticas de castigos y premios para los dueños de los nodos según la cantidad de recursos que ponen a disposición de la red y si respetan lo configurado. Así, cuando el dueño de la máquina desee usar la red para correr sus propias aplicaciones, se le asignará recursos a la aplicación en base a los antecedentes del usuario. Además se deben definir mecanismos de seguridad para impedir el acceso de otras aplicaciones que corran en el mismo nodo al área de memoria o disco reservada para la instancia local de la WDVM.

Dada la gran cantidad de nodos potencialmente disponibles, se deben considerar mecanismos para definir y administrar nodos redundantes para aumentar la tolerancia a fallas.

4.4.4 Compartición de recursos

Dadas las características de la WDVM, es interesante la posibilidad de definir la compartición de ciertos recursos de los nodos integrantes, como ser el disco, pantalla, impresoras, etc. De manera que se pueda hacer uso de ellos en forma remota. Un ejemplo de aplicación que haga uso de esta característica sería el de pizarra compartida, ya que habría que distribuir el código o el ejecutable a todos

¹ El SETI (Search for Extraterrestrial Intelligence) está implementando un sistema sobre la WEB con una idea similar a la expuesta para usar el poder de cálculo de las PC's inactivas con el fin de analizar las señales del espacio captadas por el radiotelescopio de Arecibo.



los interesados. Luego al correr cada uno de ellos la aplicación, estarían compartiendo su pantalla con los demás usuarios.

La posibilidad de compartir recursos en toda la WEB brinda un potencial muy importante que puede ser aprovechado por cualquier aplicación y debería ser analizado en más.

4.4.5 File system distribuido

El área de almacenamiento de la WDVM debería considerar varias posibilidades de uso. Sería interesante contar con un file system distribuido análogo a los utilizados por los sistemas distribuidos con características similares a la WDVM, donde tanto los usuarios como sus aplicaciones puedan acceder a archivos allí almacenados. La ubicación física no sería importante sino su distribución lógica. Como podemos extender la tolerancia a fallas al área de almacenamiento, se podrían definir áreas resguardadas y áreas inseguras, con lo cual el sistema se encargaría de administrar mecanismos que aseguren la integridad de las áreas seguras, mientras que en las áreas inseguras los usuarios se verían en la necesidad de realizar sus propios backups.

Otro uso relativo al almacenamiento sería la distribución de archivos entre los nodos, con fines diversos (distribución de ejecutables, de código, etc), en este caso se hace necesaria explicitar la lista de distribución física de los nodos, y además, como en este caso la ubicación de los archivos será fuera del área reservada al uso de la WDVM, el dueño de cada nodo debería compartir su disco. Además el área de almacenamiento de la WDVM debe encargarse de administrar el uso de los almacenamientos temporales de cada nodo.

4.4.6 Métricas

En secciones anteriores se ha puesto en evidencia la necesidad de la definición de métricas a fin de establecer parámetros de comparación y control de los recursos del sistema. El Balance de Carga y los profilers son ejemplos de lugares donde dichas métricas son muy importantes. Otro aspecto donde cobra importancia la definición de métricas es en el control de uso de recursos en la WDVM, para evitar un uso excesivo de recursos tanto a nivel de la máquina virtual [LEW/92] como dentro de cada nodo. Estas métricas deberían también utilizarse para detectar la acción de virus o programas fuera de control.



Lo que se quiere recalcar es que un modelo distribuido debilmente acoplado de características del que proponemos, debido a su complejidad y tamaño requiere de un análisis cuidadoso y detallado de todos los puntos donde se hace necesaria la definición de métricas que faciliten la administración y control del mismo.

4.5 Lenguajes sobre la WDVM

El diseño de lenguajes adecuados para la escritura de programas que corran sobre la WDVM es otro de los puntos importantes en nuestro proyecto. Por un lado es deseable brindar al usuario un lenguaje con un nivel de abstracción tal que facilite el desarrollo de aplicaciones y que haga transparente la existencia de la red subyacente. Hay que considerar que la performance de las aplicaciones escritas en un lenguaje con estas características depende mucho de la capacidad del mismo para aprovechar eficientemente los recursos (principalmente el uso de la red, crucial en este entorno) [BAL/92] [BA2/91] [REN/95] [BAL/91].

Por otra parte un lenguaje de bajo nivel permitiría aprovechar todo el conocimiento del programador acerca de la aplicación para lograr una mejor performance. Sin embargo haría mucho mas difícil la generación de aplicaciones y el control de la seguridad [LEW/92].

Una tercera alternativa es utilizar un lenguaje de alto nivel pero con un conjunto de semánticas distribuidas que permita definir las características (desde el punto de vista paralelo) de cada objeto. A partir de esta definición el lenguaje puede controlar el comportamiento de cada objeto de una manera más eficiente [BRA/93] [REN/95]. Al analizar los programas y las aplicaciones que podrían correr en este entorno se ve la necesidad de definir nuevas metodologías de desarrollo orientadas a WEB.

4.6 Entorno de desarrollo

Para el desarrollo de programas que corran sobre esta plataforma (un sistema distribuido sobre la red) es necesario un entorno de desarrollo con capacidades diferentes que la de los entornos tradicionales. Éste debería poder representar la modelización de los distintos tipos de problemas que debe resolver una aplicación que corra en este ambiente, debería permitir además realizar debugging de programas paralelos y distribuidos (no solamente de su comportamiento lógico sino tambien de la interacción con el sistema). Por ejemplo sería interesante que pudiera simular distintas condiciones de carga o distintas condiciones de falla del sistema para permitir un testeo más real del programa.



Además el entorno debería contar con un profiler que entregue distintas medidas sobre la performance y el uso de los recursos del programa, tanto para permitir que el programador mejore la performance de la aplicación (cambiando la semántica de algunos componentes de éste), como para permitir al sistema recoger datos sobre la ejecución a partir de los cuales predecir su comportamiento en posteriores corridas.

La elaboración de métricas sobre el comportamiento del sistema es importante para poder diagnosticar problemas de baja performance.

4.7 Estructura de la WDVM

A fin de clarificar la relación entre los componentes ya individualizados de la WDVM presentamos este gráfico de la estructura de la WDVM.

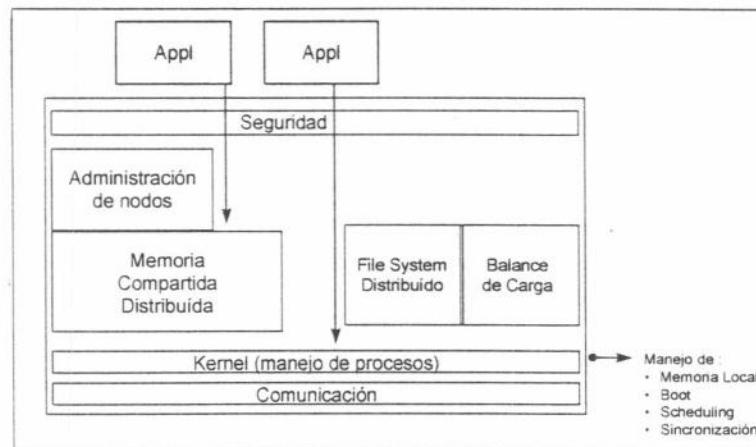


Fig. 6 - Estructura de la WDVM



5 STATE OF THE ART

5.1 Conceptos de view Synchrony

5.1.1 Introducción a View Synchrony

Una máquina global que corra sobre un ambiente inestable como Internet debe ser capaz de tolerar fallas de sus componentes (algunos de sus nodos pueden fallar, la red puede estar sobrecargada, etc.). Nuestro objetivo es definir mecanismos para lograr esta tolerancia a fallas mediante la replicación de los nodos que prestan los distintos servicios de la máquina. Estos mecanismos estarán basados en una abstracción llamada View Synchrony.

A partir de este momento llamaremos **cliente** a un proceso que desea utilizar un servicio de la máquina global. El conjunto de procesos replicados responsables de brindar el servicio se llamarán **servidores** (un servidor en un momento dado puede ser cliente de otro servicio, es decir que los roles pueden cambiar). La única suposición que se hace sobre estos servidores es que si reciben la misma secuencia de pedidos mantendrán la misma percepción del estado del servicio.

Un cliente solicitará un servicio enviando un mensaje a un servidor. Como una solicitud puede cambiar el estado de un servidor para mantener la consistencia entre los servidores es necesario que si alguno de ellos recibe un pedido todos los demás lo reciban también. La propiedad de que un mensaje sea recibido por todos los destinatarios o por ninguno se conoce como **Reliable Broadcast**.

El orden en que lleguen los mensajes a cada servidor también puede alterar el estado del servicio (pensar por ejemplo en el resultado de distintas secuencias de depósitos y calculos de interés sobre cuentas bancarias). Sin embargo, como mantener el orden de entrega entre los mensajes es costoso, se define una jerarquía de ordenes que se utilizarán en cada caso de acuerdo a las necesidades de la aplicación y al costo que uno está dispuesto a pagar.

Como uno quiere que el pedido sea recibido por todos los servidores o por ninguno, debe asegurarse de que todos los destinatarios han recibido la solicitud



antes de autorizar que se la procese. Para lograr esto se divide el envío de pedidos en dos etapas: en la primera etapa se envía el mensaje a los destinatarios y se espera un asentimiento. En la segunda etapa (que comienza cuando se recibieron asentimientos de todos los procesos) se envía un mensaje autorizando el procesamiento del pedido enviado.

Si uno de los servidores falla ², el sistema puede llegar a bloquearse esperando una respuesta. Por otra parte si se decide que un proceso servidor falló y se continúa sin él cuando en realidad solo estaba lento o tenía un problema de comunicación, este puede luego recibir pedidos y procesarlos estando en un estado inconsistente con el de sus pares. Para solucionar este problema cuando se sospecha que un servidor falló (por ejemplo porque tarda en responder) se lo declara fallado y a partir de ese momento el resto de sus pares lo ignoran cortando toda comunicación con él. El problema ahora es como se determina el conjunto de los servidores activos, ya que cada uno de los servidores puede tener diferentes creencias acerca del estado de sus pares. La solución a este problema es implementar una abstracción de grupo (**membership**). La idea es mantener una lista, replicada en cada servidor, del conjunto de servidores activos. Esta lista se conoce como **view** del sistema. Cuando se decide que un servidor falló se actualiza la view acordando su conformación entre los servidores sobrevivientes. Un servidor correcto puede ser expulsado de la view si el resto cree que ha fallado. Es decir que se consideran correctos (no fallados) los servidores que pertenecen a la view.

Un sistema que cuente con reliable broadcast y membership garantiza que el conjunto completo de los servidores reciben una solicitud o ninguno la recibe y además que en cada momento cada servidor sabe si pertenece o no a la view. Sin embargo este sistema tiene todavía problemas con los mensajes entre dos views. Supongamos que un proceso servidor falla (provocando un cambio de view) cuando todavía no se han entregado todos los mensajes iniciados por él. Los servidores sobrevivientes pueden recibir esos mensajes antes o después del cambio de view, lo que puede provocar inconsistencias porque si un servidor los recibe antes del cambio de view no sabe si el resto de los servidores los recibirán antes o después. Por reliable broadcast un servidor que recibe un mensaje puede procesarlo inmediatamente, porque está seguro de que los servidores que queden vivos lo procesarán también, sin embargo al no saber si el resto lo recibirá antes o después del cambio de view no puede procesarlo, porque si a alguno le llega después no lo procesará por provenir de un proceso no perteneciente a la nueva view, lo que dejará a los servidores en estado inconsistente.

² Utilizaremos indistintamente los calificativos "fallado", "inactivo" e "incorrecto" para llamar a los servidores que por una u otra razón ya no pueden brindar el servicio al que están destinados.



La solución a este problema la constituye una abstracción conocida como **View Synchrony** [BIR/96] [BAR/98] [REN/94] [BA2/95] que garantiza que los servidores que sobreviven a un cambio de view recibieron el mismo conjunto de mensajes en la vista previa y que un mensaje enviado en la view actual no puede ser recibido por ningún servidor que todavía no haya recibido el cambio de view .

Con view synchrony, los procesos servidores que sobreviven a un cambio de view:

1. tienen la misma percepción de membership (saben quien está y quien no está en el grupo de servidores activos) .
2. Están seguros de que cada uno de ellos recibió y ejecutó el mismo conjunto de pedidos en la view anterior . Tienen por lo tanto el mismo estado del servicio.
3. Están seguros de que cualquier mensaje enviado a partir de la instalación de la nueva view llegará luego del cambio de view.

Para lograr view synchrony, antes de producirse el cambio de view se inicia un protocolo durante el cual el conjunto de servidores intercambia los mensajes de la view actual que todavía no se hayan entregado, después de lo cual se instala la nueva view. A partir de ese momento se sabe que cualquier mensaje recibido de la view anterior es un duplicado y debe ser descartado.



5.1.2 Técnicas de tolerancia a fallas basadas en la replicación

Utilizaremos para la descripción de los problemas, mecanismos y algoritmos de tolerancia a fallas dos técnicas sistemáticas basadas en la replicación de componentes off-the-shelf: **Primary Backup y State Machine**. Sin embargo, las soluciones propuestas no se limitan a los escenarios en los cuales se basan estas dos técnicas y pueden ser utilizados en forma general. [BAR/98]

Primary-Backup : basados en un servidor primario y N servidores de backup. Sólo el servidor primario responde al cliente a menos que falle, en cuyo caso se elige a otro servidor como primario.

State Machine : N servidores replicados que atienden todos los pedidos del cliente, el cual utiliza la primer respuesta que recibe.

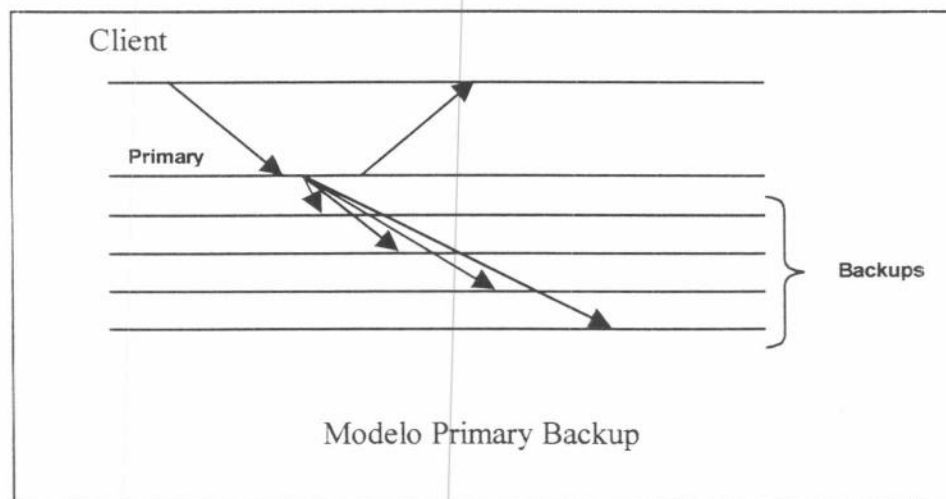


Fig. 7 – Modelo Primary-Backup

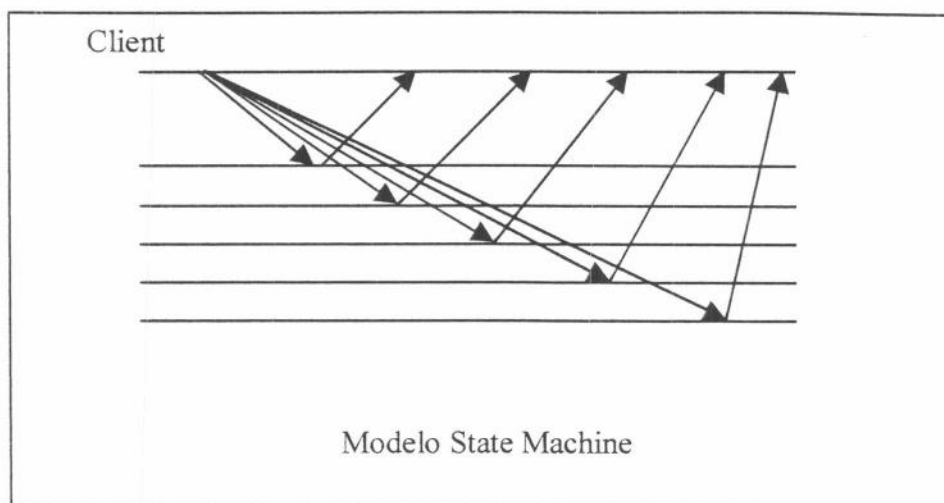


Fig 8 – Modelo State Machine

5.1.3 Primitivas de comunicación de grupos

Una primitiva de comunicación de grupo es un procedimiento para enviar un mensaje a un conjunto de procesos sin necesidad de conocer a los miembros actuales del grupo. Normalmente una primitiva de comunicación será utilizada para transmitir a un subconjunto de los miembros del grupo por lo que utilizaremos el término **multicast** para definir este comportamiento. Un **multicast** es un protocolo para **enviar** un mensaje de un proceso emisor a un conjunto de procesos receptores, los cuales primero lo **reciben** y luego lo **entregan** a la aplicación. Se diferencian en cada proceso servidor dos capas : la que se ocupa de la comunicación entre los procesos del grupo (capa de comunicación) y la que se ocupa del servicio brindado (la **aplicación**). La **recepción** de un mensaje se produce en el momento en que el mensaje llega a la capa de comunicación. La **entrega** del mensaje se produce cuando la capa de comunicación pasa el mensaje recibido a la aplicación. Esta distinción es fundamental para comprender los conceptos y protocolos que veremos a continuación. [BIR/96] [BAR/98]



Supongamos que conocemos la composición actual de un grupo G ¿Qué propiedades debería tener un multicast enviado a G ?

Para responder a esta pregunta es útil analizar qué tipo de problemas pueden surgir al enviar un multicast.

5.1.3.1 Problemas al enviar un multicast

5.1.3.1.1 Problema 1

Puede suceder que un cliente envíe un pedido al servidor primario y este falle mientras está enviando el pedido a los backups. En esta situación algunos servidores habrán recibido y ejecutado el pedido y otros no (ver figura Caída del primario). [BAR/98]

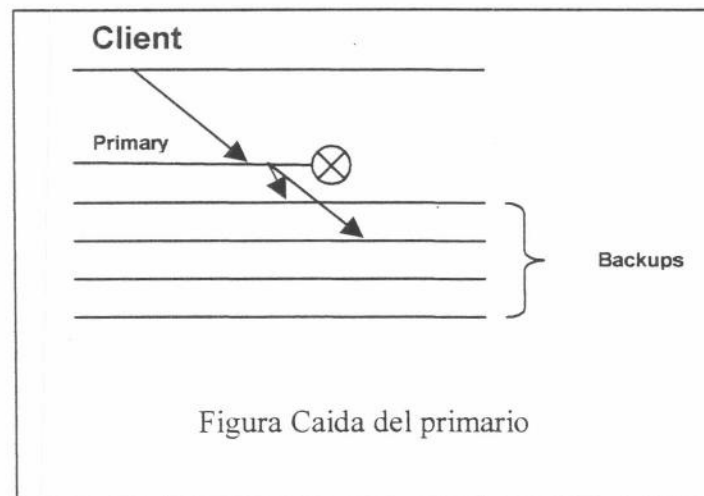


Fig 9 – Figura Caída del primario

5.1.3.1.2 Problema 2

Lo mismo puede suceder en el modelo State Machine si el cliente falla cuando está transmitiendo el pedido a los servidores. (ver figura caída del cliente) [BAR/98]

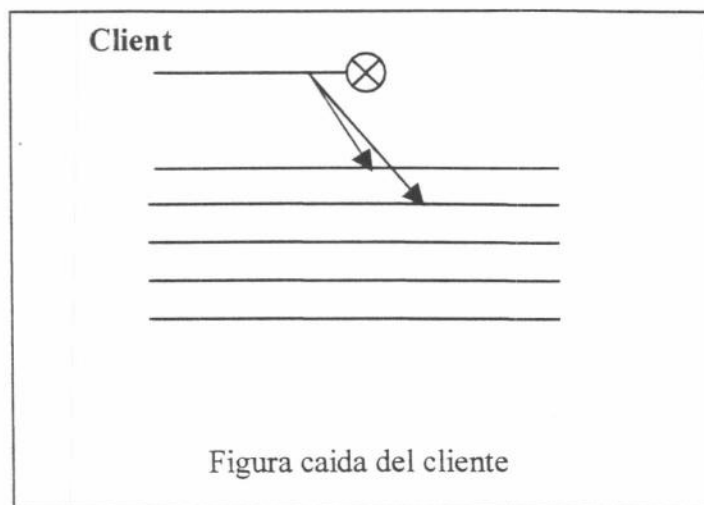


Fig 10 – Caída del cliente

5.1.3.1.3 Problema 3

Por último, tenemos el problema de que los mensajes lleguen en distinto orden a los servidores (ver figura pedidos en distintos ordenes). [BAR/98]

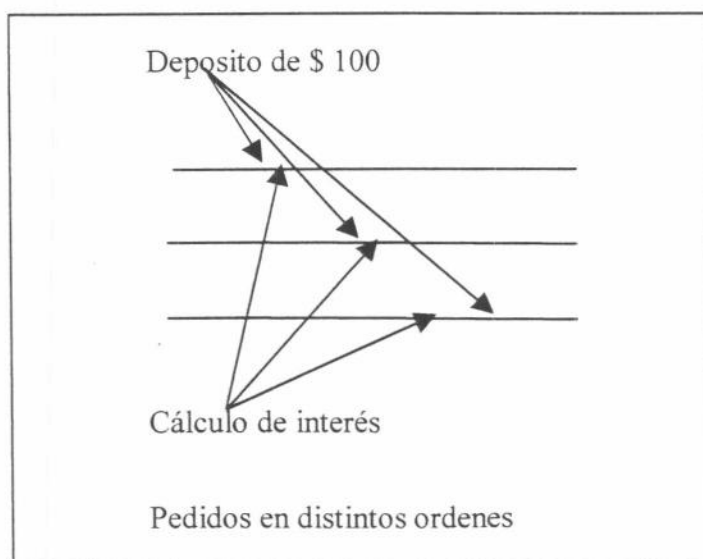


Fig 11 – Pedidos en distintos ordenes



El impacto de estos problemas varía según la aplicación, por ejemplo ninguno de estos problemas afectan a servidores que solo procesen consultas de sus clientes pero son de suma importancia, por ejemplo, para sistemas bancarios o controladores de vuelo.

Analizaremos entonces que propiedades debe cumplir un multicast para resolver los problemas antes descriptos.

5.1.4 Reliable Multicast (Failure Atomic Multicast)

Para algunas aplicaciones que no requieran confiabilidad, ordenamiento ni control de flujo de los mensajes (es decir que no se vean afectadas por los problemas antes descriptos) puede bastar un multicast que simplemente envíe el mensaje a todos los miembros del conjunto de destino (posiblemente utilizando IP-Multicast o UDP-Multicast). [BIR/96]

Para aquellas aplicaciones que se ven afectadas por los dos primeros problemas presentados se necesita un multicast que **garantice atomicidad con respecto a fallas** (reliable o failure atomic multicast), es decir que garantice que para ciertos tipos de falla el multicast será **entregado** en todos sus destinos o en ninguno de ellos.

Podemos clasificar a los multicast que garantizan atomicidad con respecto a fallas en **dinámicamente uniformes** o **no dinámicamente uniformes**. Ambos tipos pueden ser útiles dependiendo de las circunstancias. Un multicast es dinámicamente uniforme si garantiza que si un proceso entrega un multicast, entonces todo los procesos que continúen operacionales lo entregarán también, independientemente de si el receptor original continúa vivo despues de entregar el mensaje. Un multicast no dinámicamente uniforme solo garantiza que si uno espera lo suficiente, todos los procesos que continúen operacionales habrán entregado el mensaje o ninguno lo habrá hecho. [BIR/96]

La diferencia clave entre un protocolo dinámicamente uniforme y uno que solo es atómico con respecto a las fallas reside en la obligación que genera el primer evento de entrega. Desde el punto de vista de un proceso p que haya recibido un mensaje m , si m fue enviado utilizando un protocolo dinámicamente uniforme, entonces cuando p entrega m , sabe que tarde o temprano todos los procesos que continúen operacionales también entregarán m . En cambio si un proceso p recibe un multicast no dinámicamente uniforme m , p sabe que si tanto el emisor de m como él fallan o son excluidos del grupo m puede no ser entregado en ningún otro

servidor. Notar que la propiedad de uniformidad dinámica hace referencia, en este caso, a la uniformidad o consistencia del estado entre todos los procesos del sistema, los que permanecieron activos y los que fallaron. [BIR/96]

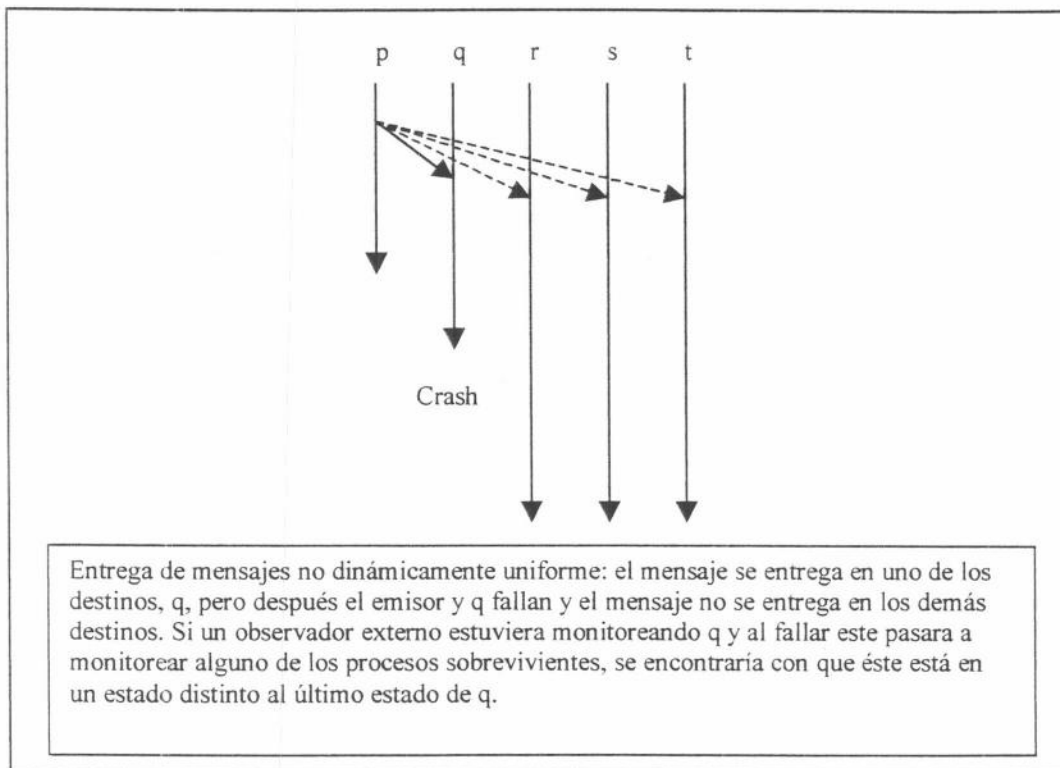


Fig 12 - Entrega de mensajes no dinámicamente uniforme

La uniformidad dinámica es una propiedad costosa (en términos de cantidad de mensajes y sobre todo en tiempo de latencia) pero puede ser necesaria en el caso de mensajes que dejen un efecto externamente visible que el sistema debería conocer como, por ejemplo, redirigir un avión en un sistema de control aéreo o entregar dinero en un cajero automático. En cambio, un multicast no dinámicamente uniforme puede ser suficiente para la mayoría de las acciones internas al sistemas, como actualizar una estructura de datos mantenida por un servidor o incluso para algunas acciones externas como las respuestas a consultas. [BIR/96]



Dado que cada propiedad tiene beneficios y costos distintos, un sistema flexible debe permitir que el desarrollador pueda elegir el multicast con las propiedades más adecuadas para el requerimiento de su aplicación.

Con respecto al tercer problema antes mencionado (mensajes que llegan en distinto orden a los servidores) existen propiedades de los multicast relacionadas con el orden.

5.1.5 Ordenes de entrega de multicasts

Examinaremos primero un multicast que no ofrezca ninguna garantía con respecto al orden en que se entregarán los mensajes. Usando este multicast un proceso que envíe dos mensajes m_0 y m_1 no tiene ninguna seguridad acerca del orden de entrega ni de la atomicidad entre ambos ; no solo puede el mensaje m_1 llegar antes que m_0 en algunos servidores sino que además una falla en el emisor puede causar que m_1 sea entregado atómicamente en todos los destino y m_0 no sea entregado a ningún proceso que continúe operacional. Esto es atómico con respecto a cada multicast pero no es muy útil desde el punto de vista de la aplicación. [BIR/96]

A continuación describiremos una jerarquía creciente de órdenes de entrega de mensajes. [BIR/96]

Orden Fifo : también llamado orden del emisor. Si el mismo proceso envía dos mensajes m_0 y m_1 entonces m_0 será entregado antes que m_1 en todos los destinos que tengan en común.

Orden Causal : si el envío de un mensaje m_0 ocurre "causalmente antes" que el envío de m_1 , entonces m_0 será entregado antes que m_1 en todos los destinos que tengan en común.

Por causalmente antes se entiende en el marco del concepto de **tiempo lógico** [Lamport 1978,1984]. Lamport modela el envío, recepción y entrega de mensajes como eventos y define un orden parcial entre ellos (así por ejemplo el envío de un mensaje por un proceso está causalmente antes que la recepción del mismo en cualquier otro proceso). Si entre dos eventos no hay ninguna relación se dice que son concurrentes. En la figura ordenamiento causal m_0 ocurre causalmente antes que m_1 porque m_0 fue enviado por p que luego envió otro mensaje m_3 que fue recibido por el proceso q , el cual envió el mensaje m_1 después de recibir m_3 .

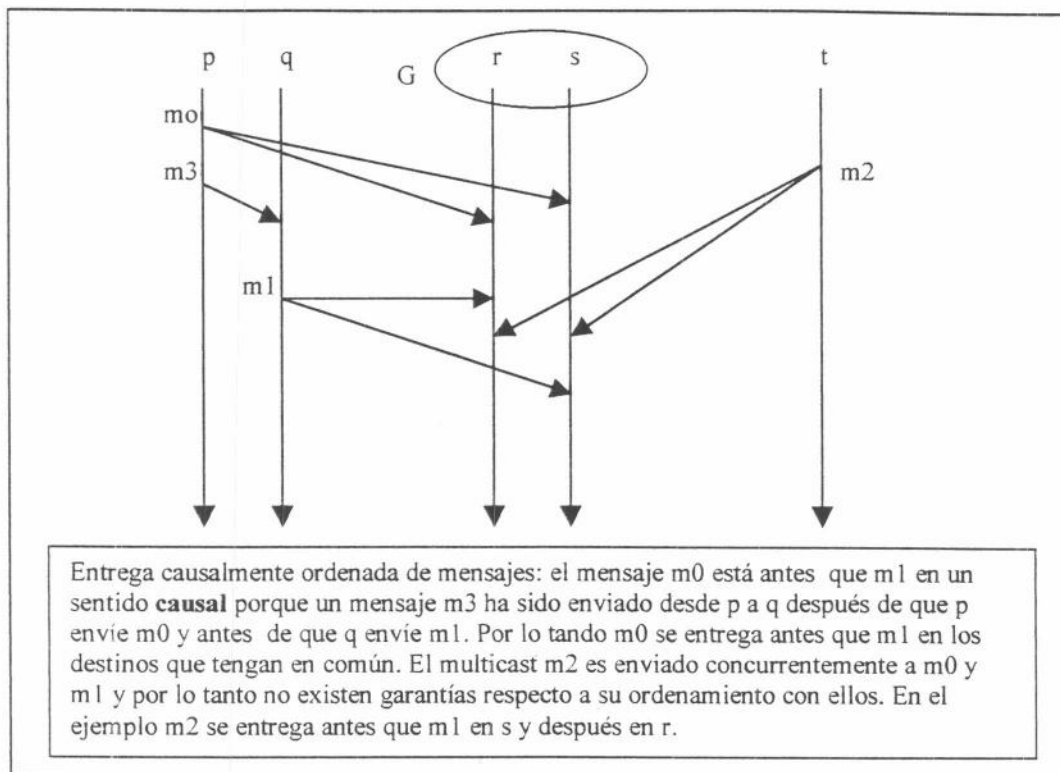


Fig 13 - Entrega causalmente ordenada de mensajes

Orden consensuado o total (Agreed or Total Order) : en este orden si un proceso entrega m0 antes que m1 entonces m0 será entregado antes que m1 en todos los destinos que tengan en común. En realidad este no es un orden total con respecto al sistema, ya que ordena solo los mensajes que tengan algún destino en común. Por eso es más correcto el nombre de orden consensuado porque los destinatarios eligen el orden en que serán entregados los mensajes consensuándolo entre todos.

Orden consensuado causal (causal agreed order) : este orden es equivalente al orden consensuado, pero requiere que si los eventos de envío están relacionados causalmente, entonces este orden sea respetado en la entrega.

Orden consensuado global (Systemwide agreed order) : requiere que haya un solo orden total de entrega entre los mensajes sin importar si tienen destinos en común o no.

Los órdenes consensuados se pueden clasificar además como fuertes y débiles en el mismo sentido en que se define la propiedad de uniformidad dinámica para



la entrega. Un orden total débil solo garantiza que el orden se cumplirá en los procesos correctos, es decir aquellos que continúen operacionales. En cambio un orden total fuerte se cumplirá aún en los procesos fallados, es decir en aquellos que fallaron después de entregar los mensajes pero antes de que el protocolo de ordenamiento terminara. En forma análoga a la uniformidad dinámica, el argumento a favor de tener un orden total fuerte es que el orden de los mensajes puede tener consecuencias visibles externamente que pueden ser notadas por un observador que primero interactúe con un proceso que después falla y luego con otro proceso que se mantenga operacional. Naturalmente esta garantía tiene un costo alto y uno preferirá un protocolo más débil si la garantía no es necesaria.

5.1.6 Relación entre atomicidad de fallas y orden de entrega

Es razonable pensar que el desarrollador de una aplicación confiable querrá extender las propiedades de orden en las de atomicidad de fallas. Esto significa que, si por las propiedades de ordenamiento de un mensaje m_0 este debe ser entregado antes que otro mensaje m_1 , en los destinos que tengan en común, entonces si m_1 es entregado a todos los procesos, m_0 también debe ser entregado. Esto se llama garantía de ausencia de brechas en el ordenamiento (gap-freedom guarantee). [BIR/96]

Si un sistema no tiene esta garantía, se pueden crear procesos huérfanos que quedan técnicamente particionados del grupo.

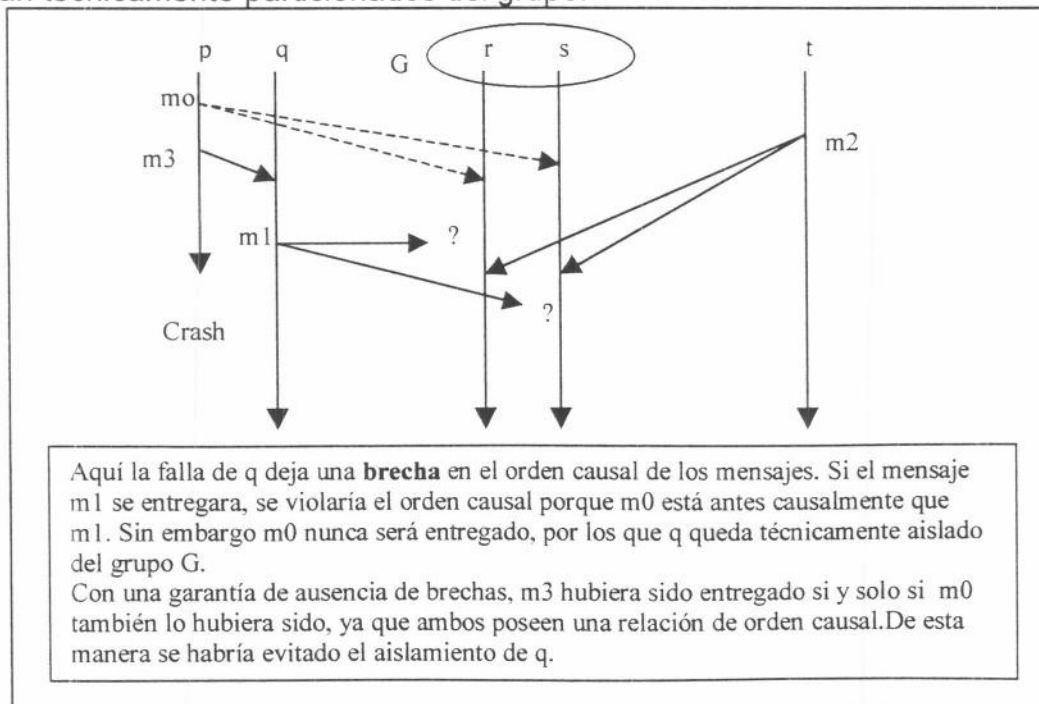


Fig 14 – Atomicidad de fallas y orden de entregas



5.1.7 Grupo de procesos (membership)

Existen dos propiedades que se pueden aplicar para caracterizar a sistemas distribuidos y a algoritmos : **liveness** y **safety**. Liveness es la seguridad de que algo bueno tarde o temprano sucederá. Safety es la seguridad de que nunca va a pasar algo incorrecto. Un sistema distribuido tolerante a fallas es **live** (cumple con la propiedad liveness) si en él nunca se tiene que esperar un tiempo potencialmente ilimitado para tomar una decisión.

Un sistema distribuido tolerante a fallas es **safe** (es decir cumple con la propiedad safety) si en él siempre se toman acciones que garantizan la correctitud y consistencia de su estado. [BAR/98]

Como dijimos anteriormente, para mantener un estado consistente entre un conjunto de procesos que brindan un servicio confiable basado en la replicación, es importante que todos los servidores reciban y procesen el mismo conjunto de pedidos solicitados por los clientes. Para ello, los pedidos pueden enviarse utilizando un multicast que garantice atomicidad con respecto a fallas.

Ahora bien, como vimos, esto implica que un servidor que recibe una solicitud de parte de un cliente no puede procesarla hasta estar seguro de que los otros servidores del servicio la han recibido también. Por lo tanto el servidor debe esperar un asentimiento de cada uno de sus pares en el grupo. La pregunta es ¿ Qué sucede si uno de los servidores falla, mientras se está esperando un asentimiento de él ? En este caso el sistema se bloqueará esperando esa respuesta. Por lo tanto, si un servidor falla podemos decir que el sistema no es live porque se bloqueará. Por otra parte si se decide que falló y se continúa sin él, puede suceder que en realidad no había fallado (sólo estaba lento o con problemas de comunicación) y, luego de un tiempo, se vuelve a conectar con sus servidores pares y toma parte en una nueva computación solicitada por un cliente. Esta situación puede producir inconsistencias en el sistema ya que uno de los procesos servidores tiene un estado del servicio distinto al del resto. La decisión de continuar sin el servidor que se sospecha fallado hace que el sistema no sea safe.

La solución a este problema, es decir a como hacer para que un sistema sea a la vez live y safe, es implementar una abstracción de grupo. La idea es mantener una lista, replicada en cada servidor, del grupo de procesos activos. Llamaremos a esta lista view del sistema. Así, cuando se sospecha que un servidor falló (porque en realidad falló o porque sólo está lento o con algun otro problema), se lo declara muerto y se lo elimina de la view. A partir de ese momento el resto de los



servidores lo ignoran cortando toda comunicación con él. De esta manera la llegada de una nueva view tiene la semántica de un aviso de falla : uno o más servidores de los que estaban en la lista como servidores activos fallaron o se fueron voluntariamente. En consecuencia, ahora el sistema puede reacomodarse a la nueva configuración y seguir brindando el servicio (live) en forma correcta (safe). [BAR/98]

Para mantener y administrar a esta lista que define al grupo de procesos activos se necesita, a su vez, de un servicio de mantenimiento de la conformación del grupo. Llamaremos a este servicio GMS (en inglés, Group Membership Service). El rol de este servicio GMS es mantener la conformación de los miembros del grupo para que, a su vez, estos puedan brindar un servicio distribuido confiable. Así un servidor que desee unirse al grupo debe realizar un pedido de conexión al GMS, el cual lo incluirá en una nueva view y lo enviará a todos los procesos que ya formaban parte de la view, produciendo un cambio de view. También, como ya mencionamos, se producirá un cambio de view cuando se detecte que un proceso falló o el GMS reciba un pedido de desconexión. Al recibir un cambio de view, los procesos servidores pueden reconfigurarse adecuadamente ya que saben que procesos nuevos han entrado al sistema y cuales han salido. Además de la detección de fallas, esto es muy útil si por ejemplo se utiliza a la view para dividir las tareas entre los miembros del grupo. [BIR/96]

Una abstracción de grupos como la antes explicada es muy útil para construir protocolos basados en ella que pueden realizar progresos aún ante fallas siempre que el servicio del GMS continúe operacional.

5.1.7.1 Particionamiento de un grupo de procesos

En el modelo presentado es imposible distinguir entre un proceso que ha fallado y uno que sólo tiene problemas para comunicarse. Por eso, si un grupo de procesos queda aislado del resto por un problema de comunicación, ambos subgrupos creerán que el resto ha fallado creandose así una partición del grupo original. Esto puede ser desastroso si las dos componentes comienzan a tomar decisiones en nombre de todo el servicio. Por ejemplo si nuestro sistema está siendo utilizado para controlar el tráfico aéreo de un aeropuerto, tal partición puede crear un escenario donde cada componente, creyendo ser el único activo, comience a controlar las mismas porciones del espacio aéreo sin tener en cuenta las decisiones del otro. [BAB/96] [BIR/96] [BAB/97] [BA2/94]

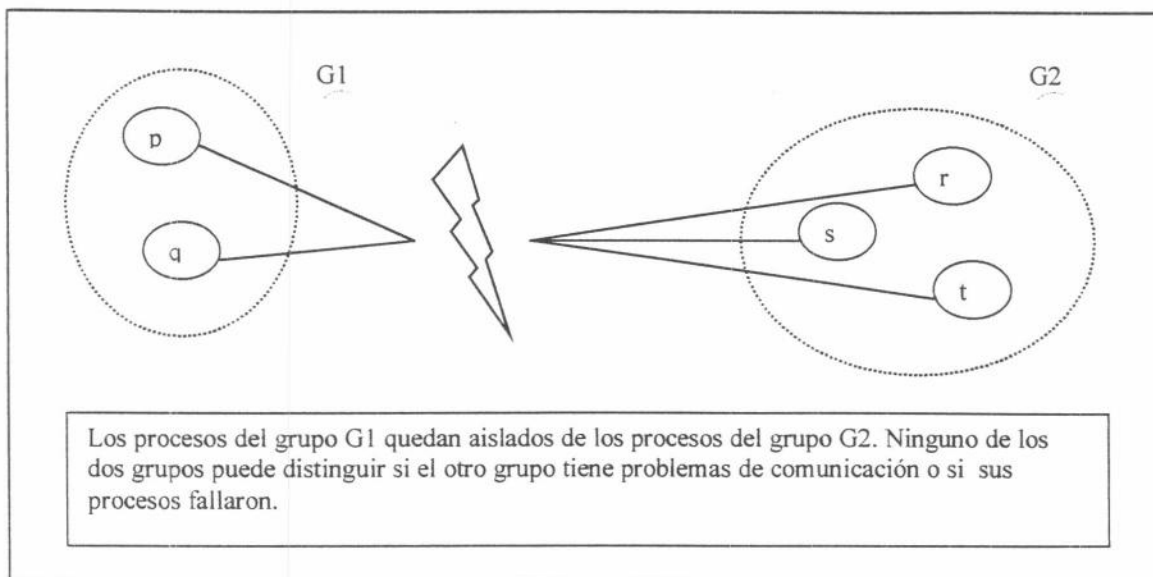


Fig 15 – Particionamiento del grupo de procesos

Aunque es deseable que el sistema continúe operacional aún durante una partición, es claro que sólo se le debe permitir a una de las particiones resultantes iniciar nuevas acciones. Tal partición será conocida como **partición primaria** y se necesita una forma de elegirla entre los componentes del sistema particionado. En general, los **componentes no primarios** pueden seguir reportando información basándose en el estado en que se encontraban en el momento de producirse la partición pero no deben permitir acciones potencialmente conflictivas. Esta regla puede ser demasiado restrictiva para algunas aplicaciones, que pueden manejar el progreso en paralelo de las distintas particiones. Por ejemplo una aplicación distribuida de ventas de pasajes puede tener una asignación estática de los pasajes que puede vender cada servidor, en el caso de una partición. Así, al producirse esta, cada partición sabe que pasajes puede seguir vendiendo sin peligro de sobrevender un pasaje. Esto trae aparejado el problema de la reconciliación de los estados de los distintos servidores si después es posible unir las particiones.

La propiedad fundamental que debe tener una partición para ser considerada primaria es que debe contener una mayoría de los miembros de la partición primaria anterior. De esta manera uno se asegura que la partición tiene la misma percepción del estado del sistema que la partición primaria anterior. [BIR/96]



5.1.8 View Synchrony

Hemos visto hasta aquí un conjunto de técnicas y conceptos utilizados para construir un servicio replicado tolerante a fallas. En primer lugar vimos los problemas con que nos enfrentamos cuando un conjunto de procesos proveedores de un servicio deben mantenerse consistentes. Como vimos para solucionar estos problemas son necesarias ciertas garantías sobre la entrega efectiva de los mensajes (reliable broadcast) y sobre el orden en que esta se realiza. Así llegamos a un punto donde los procesos tienen la garantía de que el un mensaje es recibido por el conjunto completo de los servidores o por ninguno de ellos, y además pueden especificar el orden en que desean que los mensajes sean entregados. En un contexto donde pueden producirse fallas de los procesos, se necesita además el concepto de pertenencia a un grupo para evitar que el sistema se bloquee ante alguna falla (no cumpla con liveness) o llegue a un estado inconsistente (no cumpla con safety). De esta manera agregamos a las garantías anteriores la de que en cada momento cada proceso sabe si pertenece o no al grupo.

Un sistema que cuente con todas las propiedades antedichas tiene sin embargo problemas con los mensajes entre dos views. Supongamos que un proceso P falla (provocando un cambio de view) cuando todavía no se han entregado todos los mensajes enviados por el. Los procesos sobrevivientes pueden recibir esos mensajes antes o después del cambio de view, lo que puede provocar inconsistencias porque si un proceso lo recibe antes del cambio de view no sabe si el resto de los procesos lo recibirán antes o después. Si a alguno le llega después no lo procesará por provenir de un proceso no perteneciente a la nueva view, lo que dejara a los servidores en estado inconsistente (ver figura situación indistinguible). [BAR/98]

Tiene sentido asumir que el desarrollador querrá que la notificación de la falla de un proceso (la llegada de la nueva view) constituya un estado final del mismo. Esto significa que no puedan aparecer mensajes del proceso fallado con posterioridad al anuncio de su falla.

Veamos la siguiente figura :

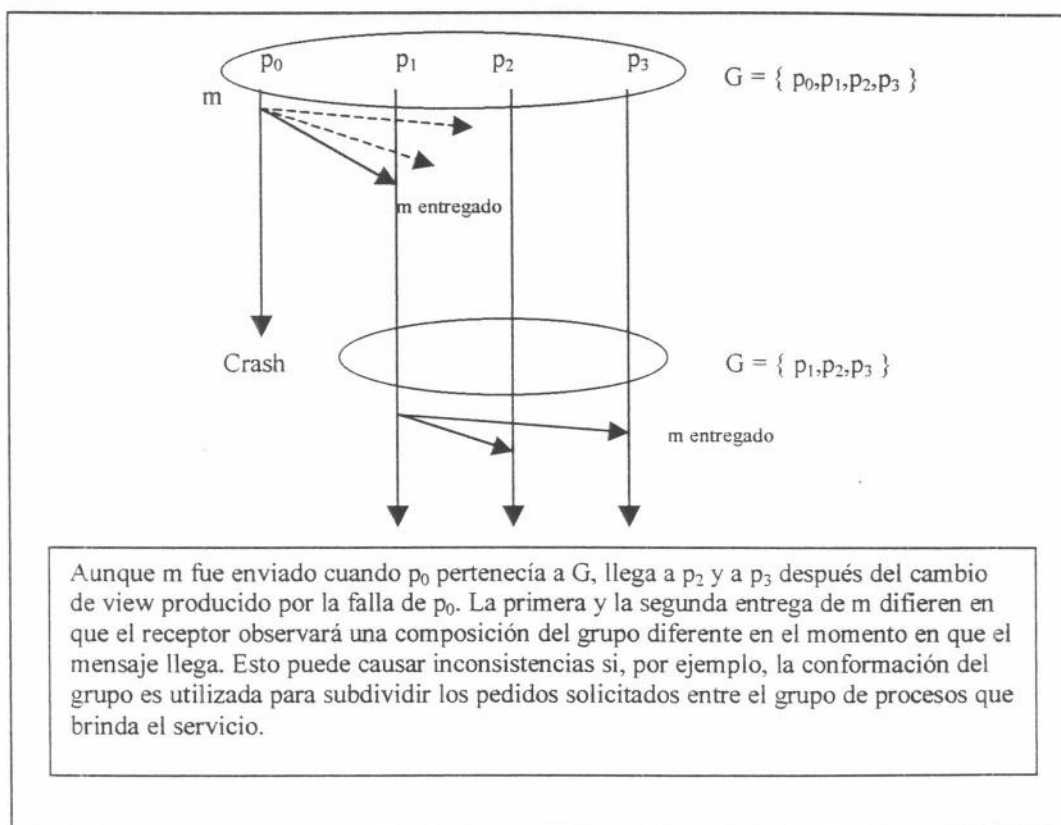


Fig 16 – Relación entre entrega de mensajes y cambio de view (1)

Podemos ver otra inconsistencia en el siguiente ejemplo. En el momento del cambio de view ni B1 ni B2 pueden determinar en base a sus datos locales cual de las tres situaciones se produjo. Ningún proceso sobreviviente sabe qué mensajes recibieron el resto de los procesos en la view anterior. Si por ejemplo B1 recibió un mensaje que B2 descartó por haberlo recibido después del cambio de view, B1 y B2 quedarán en estados inconsistentes.

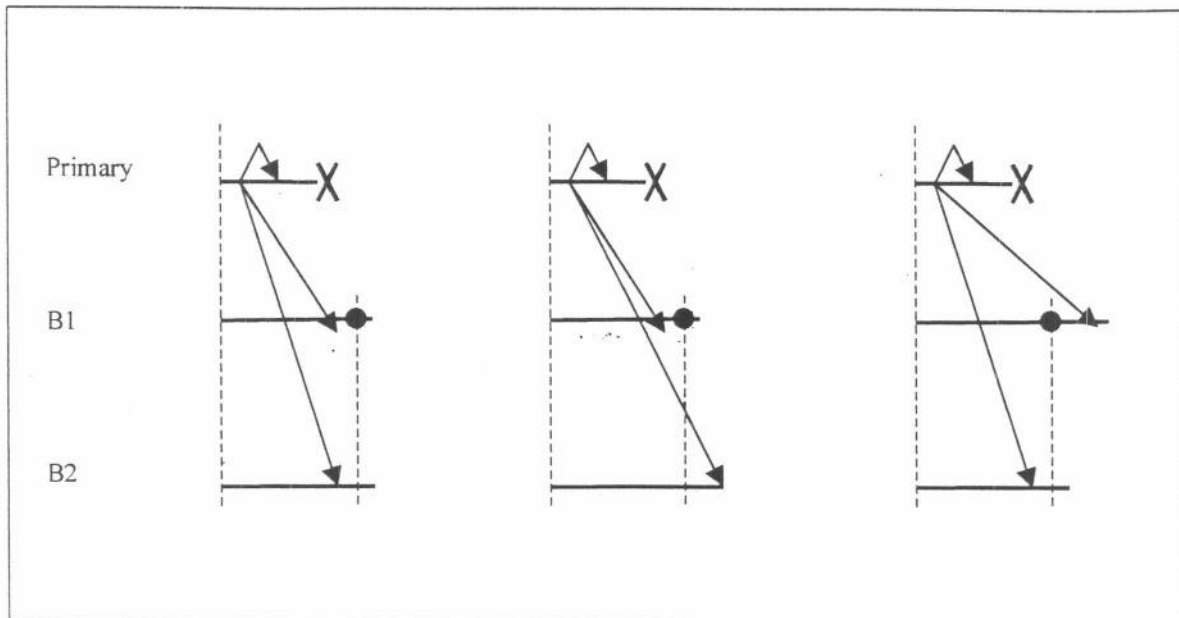


Fig 17 – Relación entre entrega de mensajes y cambio de view (2)

Se ve así la necesidad de un mecanismo que nos garantice que el envío de mensajes sea atómico con respecto al cambio de view, es decir que garantice que los procesos servidores que sobrevivan a un cambio de view recibieron el mismo conjunto de mensajes en la vista previa y que un mensaje enviado en la view actual no puede ser recibido por ningún servidor que todavía no haya recibido el cambio de view. Esta propiedad es conocida como **View Synchrony**. [BIR/96] [BAR/98] [BAB/96]

Con view Synchrony, los procesos servidores que sobreviven a un cambio de view :

1. Tienen la misma percepción de membership (saben quien está y quien no está en el grupo de servidores activos).
2. Están seguros de que cada uno de ellos recibió y ejecutó el mismo conjunto de pedidos en el orden correcto en la view anterior. Tienen por lo tanto el mismo estado del servicio.
3. Están seguros de que cualquier mensaje enviado a partir de la instalación de la nueva view llegará luego del cambio de view.



5.1.8.1 Propiedades del servicio de mantenimiento de conformación del grupo (GMS) [BIR/96]

GMS-1 : La conformación del sistema tiene la estructura de una lista llamada view del sistema. Existe una view inicial predeterminada en el momento en que el sistema comienza. Las view subsiguientes difieren sólo por el agregado o borrado de procesos.

GMS-2 : Sólo los procesos que solicitan ser agregados al grupo son agregado a la view. Sólo los procesos que se sospecha que han fallado o que solicitan su desconexión son borrados de la view.

GMS-3 : Una mayoría de los procesos de la view i debe asentir y confirmar la composición de la view $i + 1$.

GMS-4 : Comenzando con la view inicial del sistema, subsecuencias de una misma secuencia de la view del sistema son reportadas a los servidores miembros del grupo. Cada miembro observa dicha subsecuencia comenzando con la view en el cual él fue agregado al grupo y continúa observandolas hasta que es borrado, cuando o bien solicitó su desconexión o fue excluido del grupo porque se sospecha que falló.

GMS-5 : Si un miembro del grupo p sospecha que otro miembro del grupo q falló, y si el GMS continúa operacional y puede reportar nuevas views, entonces q será borrado de la view o p lo será o ambos serán borrados.

GMS-6 : En un sistema con relojes sincronizados y tiempos de latencia de mensajes acotados, cualquier miembro excluido de un grupo se dará cuenta de que esto ha ocurrido dentro de un lapso acotado.

5.2 Protocolos de view synchrony

5.2.1 Protocolos de multicast atómico con respecto a fallas

Veremos a continuación una serie de protocolos para implementar un multicast atómico con respecto a fallas partiendo desde una implementación simple aunque ineficiente hasta implementaciones mas complejas y eficientes. Se verán protocolos tanto para multicast dinámicamente uniforme como para el no uniforme. [BIR/96]



5.2.1.1 Multicast no dinámicamente uniforme

El primer protocolo que estudiaremos es el más simple e ineficiente de todos. El emisor agrega al mensaje un encabezado con la lista de miembros del grupo en el momento que envía el mensaje. Transmite luego el mensaje a todos los miembros del grupo a través de una capa de transporte que sólo falla si uno de los extremos es expulsado del grupo. Al recibir el mensaje, cada destinatario lo entrega inmediatamente a la aplicación y lo reenvía al resto. Cada participante recibirá por lo tanto un mensaje del emisor original y uno de cada uno del resto de los receptores que no hayan fallado. Una vez que un proceso entregó un mensaje descarta el resto de las copias como duplicados.

Para ver que el protocolo es atómico con respecto a fallas, supongamos que un proceso p_i recibe y entrega un mensaje y continúa operacional. Por atomicidad de fallas todo proceso que continúe funcionando debe recibir y entregar el mensaje también. La única manera de que p_i no reenvíe el mensaje a otro proceso p_j es que p_j o p_i hayan sido expulsados del grupo. Pero asumimos antes que p_i había continuado funcionando correctamente por lo que el protocolo es atómico con respecto a fallas.

Es además no dinámicamente uniforme porque si el emisor original del mensaje envía una copia sólo al proceso p_i y después ambos fallan, puede haber sucedido que p_i haya entregado el mensaje a su aplicación antes de fallar. Sin embargo el resto de los procesos que continuaron funcionando no recibirán ni entregarán el mensaje.

Este protocolo es costoso porque requiere que cada proceso reciba y envíe cada mensaje aproximadamente n veces (donde n es la cantidad de miembros del grupo) por lo que tiene un costo del orden $O(n^2)$.

Una mejora al protocolo anterior es esperar antes de retransmitir el mensaje y hacerlo solo en el caso de que llegue una nueva view informando que el emisor original ha fallado. Con este cambio el costo del algoritmo para el caso más común en que no falle el emisor baja a $O(n)$ pero cada participante debe ahora mantener una copia de cada mensaje que reciba indefinidamente.

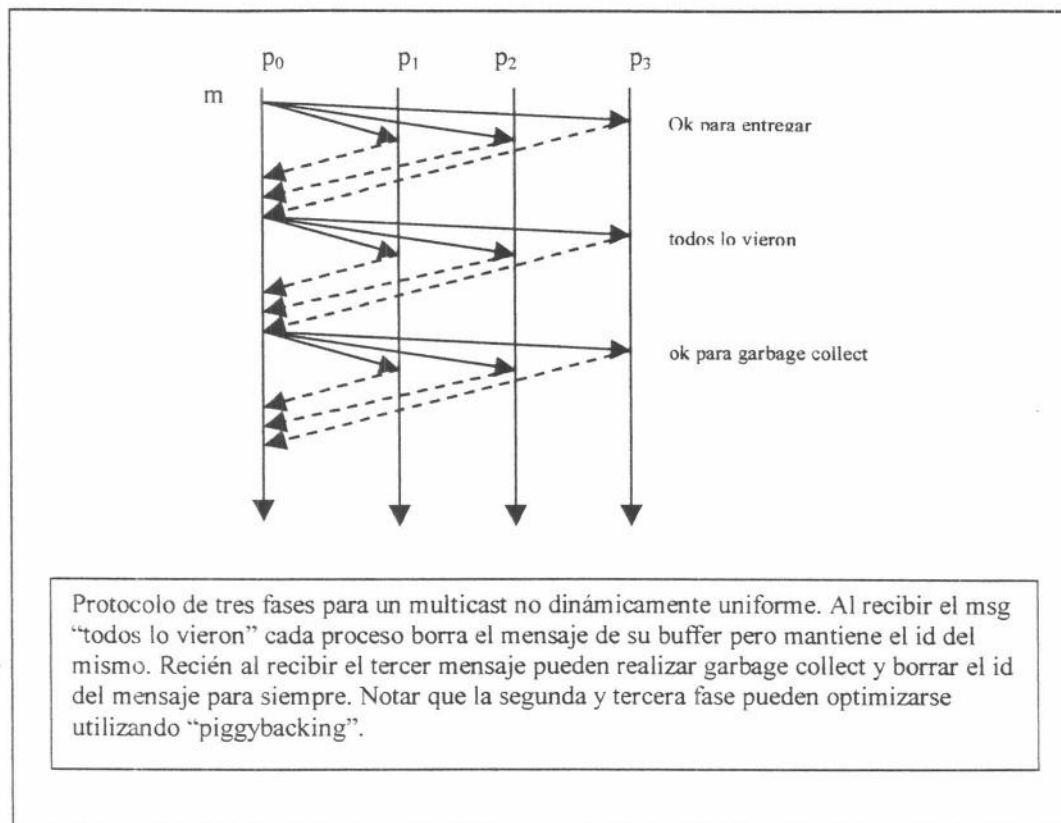


Fig 18 – Protocolo de tres fases para un multicast no dinámicamente uniforme

Como estamos transmitiendo los mensajes sobre un capa de comunicación confiable, existen asentimientos en los niveles más bajos de comunicación que van de cada participante hacia el emisor original. El emisor puede utilizar estos asentimientos para detectar el momento en que el mensaje llegó a todos los destinos. En ese momento puede enviar un segundo mensaje informando a los participantes que es seguro borrar la copia del mensaje aunque todavía debe mantenerse el identificador del mensaje para detectar duplicados si el emisor original falla en el medio de esta segunda etapa. Mas tarde el emisor puede iniciar una tercera etapa en la que anuncia a los participantes que pueden borrar incluso esta identificación del mensaje.

Para obtener una mayor optimización, puede ser útil que el emisor retarde el envío de los mensajes de la segunda y tercera etapa con la esperanza de que un nuevo multicast sea iniciado hacia el grupo. De esta manera la información que se iba a enviar por separado puede adjuntarse en los nuevos multicast salientes utilizando la técnica conocida como piggybacking.



Finalmente llegamos a un protocolo que tiene un costo promedio de n mensajes por multicast. Cuando suceda una falla cualquier multicast pendiente generará n^2 mensajes adicionales. Podemos incluso mitigar este problema ya que todos los miembros del grupo conocen su composición y además cada mensaje lleva como encabezamiento la lista de destinatarios. Entonces los participantes pueden esperar antes de comenzar la retransmisión a que un miembro de mayor rango (por ejemplo, el de mayor tiempo de pertenencia al grupo) complete el protocolo en lugar del emisor original.

5.2.1.2 Multicast dinámicamente uniforme

Para convertir al protocolo anterior en uno dinámicamente uniforme se requiere que ningún proceso entregue el mensaje hasta saber que todos los destinatarios en el grupo tengan una copia.

Para lograr esto se necesita agregar una nueva etapa al protocolo antes de que el multicast pueda ser entregado. Desafortunadamente, esta etapa extra incrementa la latencia del multicast ya que en el protocolo anterior un mensaje podía ser entregado tan pronto como llegaba a un destino, mientras que, si se requiere uniformidad dinámica, se deberá esperar a que el mensaje llegue al destinatario más lento, que vuelva el asentimiento al emisor y además el tiempo que tarde el emisor en enviar el mensaje "listo para entregar".

Así aunque algunas aplicaciones necesitan garantías de uniformidad dinámica, el desarrollador de una aplicación distribuida debe solicitarla sólo cuando sea realmente necesaria o la performance se verá severamente afectada.

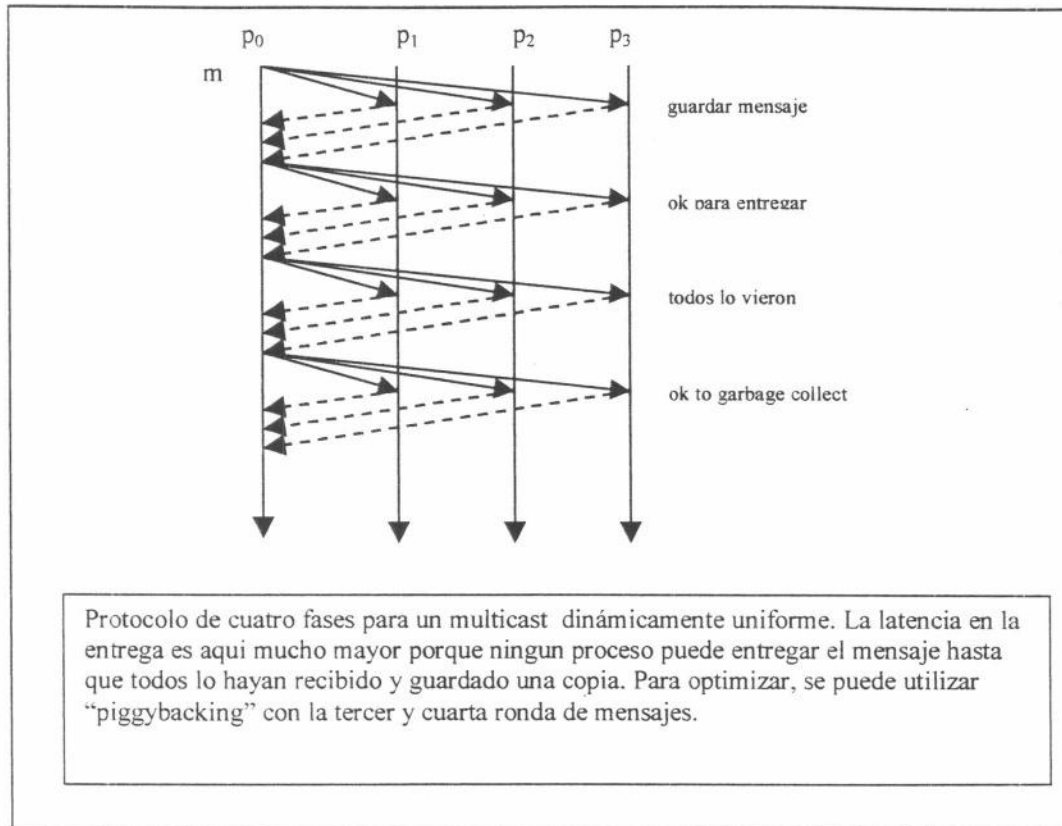


Fig 19 - Protocolo de cuatro fases para un multicast dinámicamente uniforme

5.2.2 Orden de mensajes

5.2.2.1 Protocolo de Orden Fifo

Existen dos variantes en el ordenamiento FIFO de mensajes : el ordenamiento FIFO normal y el dinámicamente uniforme.

La implementación de estos órdenes es sencilla . En efecto manteniendo un contador local en cada server con el que se van etiquetando los mensajes que salen de cada server , es posible para los destinatarios ordenar los mensajes que le llegan para entregarlos de acuerdo al orden en que fueron enviados. Para lograr el ordenamiento FIFO dinámicamente uniforme , basta con agregar otra vuelta de



mensajes , para confirmar a todos los destinatarios que no hayan fallado que pueden entregar el mensaje. [BIR/96]

Para medir el costo de un protocolo normalmente se utilizan los criterios de latencia antes de que un mensaje pueda ser entregado , la carga impuesta a cada participante y el overhead causado por los headers del protocolo en el mensaje. Con estos protocolos , la cantidad de mensajes no se incrementa si se considera que deben implementarse junto con los protocolos de reliable broadcast antes mencionados y la sobrecarga en los headers es mínima , ya que se trata solo de unos pocos bytes para ordenar el mensaje. Por lo tanto el costo que realmente aumenta el ordenamiento FIFO es el de la latencia. En el caso del ordenamiento FIFO normal , ni siquiera este costo aumenta ya que , como un mensaje puede ser entregado apenas llega a su destino , la latencia del protocolo es la misma que de la red . En cambio en el ordenamiento FIFO dinámicamente uniforme la latencia se aumenta a aproximadamente dos veces la latencia de la red , por la necesidad de una segunda vuelta de mensajes. [BIR/96]

5.2.2.2 Protocolo de Orden Causal

Una pregunta que surge al estudiar el ordenamiento FIFO es cuál es el protocolo que provee el orden más fuerte con el mismo costo que el protocolo FIFO. Ha sido demostrado que este protocolo es el orden causal , que es la clausura transitiva de la relación entre envío y recepción de mensajes. Es decir que de acuerdo a este orden si un mensaje m_0 llega a un servidor s , todos los mensajes que salgan de s a partir de ese momento estarán después de m_0 para el orden. [BIR/96]

Mostraremos a continuación dos posibles algoritmos para lograr causal order .

La primera utilizará una marca de tiempo (timestamp) lógica . Esa marca se mantendrá de acuerdo al reloj lógico de Lamport , de manera que si un mensaje m_0 está causalmente antes que otro m_1 , la marca de m_0 será menor que la de m_1 . En un ambiente donde los miembros del grupo envían continuamente mensajes , podemos estar seguros de que todos los miembros del grupo irán incrementando las marcas de sus mensajes. Cuando un server tiene un mensaje m_0 , marcado con una timestamp $LT(m_0)$ puede esperar hasta que reciba de todos los procesos miembros al menos un mensaje con timestamp mayor que $LT(m_0)$. Como los canales de comunicación son FIFO , el proceso sabe que todo mensaje que le llegue a continuación será mayor causalmente que m_0 y puede entregarlo sin problemas. Si dos mensajes poseen la misma timestamp (es decir



son concurrentes) pueden entregarse en un orden arbitrario o se puede utilizar alguna regla (por ejemplo la antigüedad del emisor dentro del grupo) , para ordenarlos y conseguir así un orden causal total.

Por supuesto si alguno de los participantes no envía mensajes durante un período largo de tiempo , todos los mensajes quedarán bloqueados , esperando en vano un mensaje de este proceso para saber en que punto se encuentra su reloj lógico. Esto se puede solucionar obligando a los procesos a enviar mensajes cada un determinado período de tiempo , solo para mantener los canales activos o permitiendo a un proceso que necesita conocer el estado del reloj lógico de otro realizar un ping sobre este.

Este protocolo es muy costoso en terminos de latencia , ya que para entregar un mensaje se necesita haber recibido mensajes de todos los participantes con timestamps mayores o iguales que las del que se quiere entregar. Además el número de mensajes que se deben almacenar puede crecer mucho . generando problemas de buffering.

La segunda implementación utiliza vectores de timestamps , formados por la cantidad de mensajes recibidos por el emisor separados por fuente. Es decir que si un mensaje m_0 enviado por el proceso p_1 tiene un timestamp $[10,4,3,2]$ y el grupo de procesos está formado por los procesos $[p_0, p_1, p_2, p_3]$ eso significa que cuando el mensaje fue enviado , p_1 había recibido 10 mensajes de p_0 , 3 de p_2 y 2 de p_3 y además que el mensaje es el cuarto enviado por p_1 . Presumiblemente los mensajes anteriores hayan creado un contexto en el que el mensaje m_0 tiene un significado particular , por lo que tiene sentido que los mensajes anteriores deben recibirse antes que m_0 .

La regla para entregar un mensaje m_0 en este algoritmo es la siguiente :

1. El mensaje m_0 es el siguiente en la secuencia de mensajes provenientes de su emisor
2. Todo mensaje causalmente anterior a m_0 ha sido ya entregado a la aplicación.

Es decir que en el ejemplo anterior , m_0 no será entregado por un proceso antes de haber recibido 10 mensajes de p_0 , 3 de p_1 , 3 de p_2 y 2 de p_3 . Como los canales son FIFO , al haber recibido estos mensajes , sabe que se encuentra en el mismo estado que el proceso que envió m_0 , que es lo que se busca preservar con el orden causal.



El impacto del principal problema de este algoritmo , es decir el overhead que le impone a los mensajes , puede ser disminuido si se utilizan técnicas de compresión con los vectores de timestamps.

5.2.2.3 Protocolo de Orden Total

Una manera de obtener protocolos de orden total es combinando protocolos de orden causal con mecanismos que permiten lograr orden total. Es decir que se extiende el orden causal con una manera de 'desempatar' los mensajes concurrentes de una manera única . Un ejemplo de estas técnicas de desempate es entregar los mensajes concurrentes ordenados por la antigüedad del emisor dentro del grupo. Sin embargo estos protocolos (que son a la vez ordenados causal y totalmente) son muy caros en términos de latencia , ya que para poder entregar un mensaje m , un proceso debe haber recibido mensajes con tiempo lógico posterior al de m de todos los procesos con los que podría llegar a tener que desempatar (en el ejemplo , los más antiguos dentro del grupo) con el emisor de m . [BIR/96] [AGA/95] [AMI/94]

Si el ordenamiento causal no es un requerimiento específico , existen algunas maneras muy sencillas de obtener orden total. La más comun es la de tener un proceso secuenciador , que se encarga de publicar un ordenamiento de los mensajes que conoce . Todos los demás procesos guardan los multicast hasta que se enteran de este orden y recién entonces los entregan. Un token es una manera de rotar el secuenciador dentro del grupo. El token se va pasando de un miembro del grupo a otro y el poseedor del token puede asignar números de secuencia a los multicast que envíe. Mientras el grupo solo posea un token , se dará un orden total a los multicast del grupo.

Otro algoritmo para obtener orden total es un algoritmo de voto , en el que los procesos votan un ordenamiento de los mensajes , expresando su voto a través de una timestamp propuesta para cada mensaje. El algoritmo tiene dos fases : en la primera el originador del mensaje lo envía a todos los miembros del grupo de destino de este. Estos procesos guardan el mensaje pero no lo entregan y contestan proponiendo un timestamp para el mensaje. El coordinador recolecta las timestamps propuestas , elige la máxima y envía un mensaje de commit de vuelta a los participantes. Estos actualizan sus relojes (por lo tanto nunca van a proponer una timestamp menor) y reordenan los mensajes pendientes. Los mensajes que tengan una timestamp de commit que además sea menor que todas las demás timestamp propuestas por el proceso y de commit , pueden entregarse. Puede verse que este algoritmo provee orden total porque antes de que un proceso



entregue un mensaje se asegura de que toda timestamp propuesta por el en el pasado es posterior y además actualiza su reloj lógico a la timestamp de este mensaje. De esta manera su voto para cualquier mensaje que se entregue en el futuro será mayor que el del mensaje entregado, proveyendo un orden total. Es interesante ver que el algoritmo no es dinámicamente uniforme con respecto al orden. Para ver esto consideremos el caso en que el emisor y uno de los receptores fallan, y el receptor que fallo es también el que propuso la máxima timestamp. Esa timestamp puede haber sido devuelta como timestamp de commit al receptor que falla luego de entregar el mensaje, dejando al resto de los receptores con la libertad de elegir otro orden de entrega.

Una diferencia entre los dos algoritmos presentados es que el orden total basado en el algoritmo de voto es global aún entre grupos distintos, mientras que dos mensajes de grupos distintos que tengan miembros en común pueden llegar a algunos de esos miembros compartidos en distinto orden con el algoritmo basado en el token.

5.2.3 Protocolo de pertenencia al grupos de procesos

El protocolo que presentaremos en esta sección es el que se ocupa de manejar los cambios en la lista de miembros de un grupo de procesos (view). [BIR/96]

El protocolo está basado en la idea de que uno de los procesos del grupo (por ejemplo, el más antiguo de sus miembros) cumple funciones de coordinador para los cambios de pertenencia al grupo. Así, si un proceso desea unirse al grupo o solicita irse, el coordinador es el encargado de actualizar y distribuir la nueva lista de miembros. El coordinador es, además, el encargado de monitorear el estado de los miembros del grupo (utilizando un servicio como el del GMS, si está disponible, o directamente utilizando un ping de la capa de red) para detectar y excluir a cualquier miembro fallado tan pronto como sea posible. A su vez, los miembros del grupo también deben monitorear la "salud" del coordinador, para asegurar la propiedad de liveness al protocolo. Esto es así porque dependen para continuar efectuando acciones de las views enviadas por el coordinador.

Todos los procesos miembros del grupo mantienen una copia local de la lista de miembros llamada view de grupo. El protocolo asegura que cada vez que se produce un cambio en la conformación del grupo una nueva view será reportada a sus miembros por el coordinador. El protocolo además establece que todos los miembros del grupo ven la misma secuencia de views dentro de un componente de un sistema particionado (es decir, en el caso de que se produzca una partición,



la propiedad se mantiene entre los miembros de cada componente). En el caso en que sólo permitamos a la partición primaria realizar progresos, entonces, o bien todos los procesos reciben la misma secuencia de views, o bien dejan de recibir nuevas views; en cuyo caso detectan que han sido excluidos del grupo. Al detectarlo, pueden terminar su ejecución o reintentar conectarse.

El protocolo consta de 2 fases, en el caso en que no falle el coordinador. Si éste falla, se necesita de una etapa extra para elegir al nuevo coordinador.

En el protocolo de 2 fases, cuando algún proceso detecta una falla (debido a un multicast fallido o a un ping no contestado) avisa al coordinador a través de un mensaje msuspect indicando que sospecha la caída de un proceso P_i . También puede avisar a los otros miembros del grupo para que dejen de aceptar cualquier mensaje proveniente de P_i tan pronto como sea posible. Cuando el coordinador recibe este mensaje comienza la primera fase del protocolo enviando un multicast vchg de cambio de view a todos los procesos. Ante este mensaje, todo proceso vivo debe contestar con un multicast en el cual especifica su percepción de la conformación del grupo (que pedidos de ingreso y desconexión recibió, con que procesos se puede comunicar y con cuales no). Al final de esta fase, todo proceso no fallado sabe qué procesos deben ser echados del grupo y cuales desean unirse al mismo (si algún proceso no recibe esta información es porque falló, lo que será detectado por el emisor del mensaje que modificará su percepción del grupo y enviará un nuevo multicast).

El coordinador espera un tiempo determinado, recolectando todos estos multicasts para lograr un acuerdo en la conformación de la nueva vista. La segunda fase del protocolo comienza cuando, terminado este lapso, el coordinador envía un mensaje de commit de estos cambios a todos los procesos autorizando la instalación de la nueva view. Cada proceso, al recibir este mensaje actualiza su vista con los cambios que recibió en la etapa anterior. Si algún proceso falla en este momento será detectado en el primer multicast que se le intente enviar en la nueva view por lo que comenzará una nueva instancia del protocolo.

Si durante la ejecución del protocolo algún proceso detecta que el coordinador falló se agrega una etapa extra al principio del protocolo para elegir a un nuevo coordinador. En la primera fase, el nuevo coordinador (el más antiguo en la vista después del viejo coordinador, por ejemplo) envía un mensaje a todos los procesos anunciando un cambio de view y buscando consenso para ser nombrado nuevo coordinador. Todos los procesos asienten su pedido y además reenvían información de procesos fallados y agregados que había enviado su antecesor. A partir de ese momento se continúa con el protocolo de 2 fases antes explicado.



Dado que la acción de distribuir nuevas views sólo trae aparejados cambios internos al estado del sistema y no deja acciones visibles desde afuera del mismo, podemos utilizar para reportar las nuevas views un multicast no dinámicamente uniforme.

5.2.4 Protocolo de view synchrony

Con la implementación de los protocolos de reliable broadcast, orden y membership, hemos creado un ambiente en el cual un proceso que se una a un grupo recibirá la view actual en el momento de la conexión y en adelante recibirá las sucesivas views reflejando los cambios producidos en la conformación del grupo hasta que se desconecte o falle. [BIR/96]

Pero supongamos que un proceso perteneciente al grupo falla mientras un multicast suyo está todavía pendiente. ¿Cuándo podrán los otros miembros estar seguros de que han recibido todos sus mensajes, para poder continuar sin el ?. Hasta ahora nuestros protocolos no proveen esta información. Necesitamos entonces un protocolo que provoque la terminación de los multicast parcialmente completados de una view antes de pasar a la siguiente, es decir que asegure la atomicidad de los mensajes con respecto a los cambios de view. Llamaremos a este protocolo protocolo de flush de view synchrony.

Este protocolo se agrega como una nueva etapa del protocolo de cambio de view explicado anteriormente (al final del mismo pero antes de que el cambio de view se entregue a la aplicación). El coordinador solicita a los miembros del grupo que envíen cualquier multicast pendiente de entrega (no solamente los provenientes del o de los procesos que causaron el cambio de view porque nuevos procesos pueden fallar durante esta fase) y el protocolo de cambio de view se completa solo cuando todos los mensajes han sido finalmente entregados.

Para asegurar que los multicast iniciados posteriormente al inicio del cambio de view sean entregados en la nueva view hay dos alternativas posibles. Una es prohibir la iniciación de multicasts durante el protocolo de cambio de view. Esta solución generaría un extenso período de tiempo durante el cual no se pueden iniciar nuevos multicasts. La otra solución consiste en que se tome de la lista de destinos de los multicast iniciados durante el protocolo del cambio de view solo a aquellos procesos que continúen siendo miembros del grupo en la futura view. El multicast puede entonces ser iniciado mientras el protocolo de cambio de view aún



está corriendo y solo será demorado si cuando el sistema está listo para entregar el mensaje a algún miembro, la view no ha sido reportada todavía a ese miembro.

5.2.4.1 Protocolo de View Synchrony

Este protocolo es una modificación al protocolo de pertenencia al grupo de procesos (protocolo de membership) ya que se realizan en el mismo momento: ante el aviso de detección de falla de algún proceso. [BIR/96] [BAB/96]

El protocolo funciona de la siguiente manera: Ante una sospecha de falla uno de los miembros del grupo es elegido como coordinador del protocolo (generalmente el miembro más antiguo). El coordinador envía entonces un mensaje de FLUSH a todos los miembros de la view indicando que ha comenzado un protocolo de cambio de view y que, por lo tanto, deben enviarle el conjunto de mensajes inestables provenientes de miembros que han fallado así como también los eventos de add (procesos que desean agregarse a la view) y delete (fallas detectadas o pedidos de desconexión) que han recibido. Luego de esto, cada miembro termina esta fase enviando un mensaje de FLUSH_OK y a continuación comienza a ignorar mensajes provenientes de miembros que, se cree, han fallado esperando la instalación de la nueva view.

Al recibir todas las respuestas de FLUSH_OK, el coordinador comienza a enviar multicast de todos los mensajes inestables provenientes de miembros fallados. Cuando todos los mensajes se estabilizan, una nueva view puede ser instalada así que el coordinador envía un mensaje de INSTALL_NEW_VIEW con la nueva view calculada a partir de la información de pertenencia enviada por todos los miembros en la etapa de FLUSH.

5.3 View Synchrony Estructurado

5.3.1 Modelo del sistema

Hasta ahora hemos explicado los conceptos y protocolos utilizados para implementar view synchrony sobre un conjunto de miembros sin estructura, es decir que todos los miembros tienen la misma función y jerarquía adentro del grupo. Este esquema es adecuado para la implementación de view synchrony sobre una red de area local (LAN) pero presenta limitaciones y dificultades cuando se trata de escalar la solución para un conjunto de redes de area local (es decir un grupo de intranets). Es bien sabido que la implementación de los protocolos de multicast para asegurar uniformidad dinámica así como también la utilización de un ping periódico como detector de fallas débil generan una sobrecarga de mensajes que crece con la cantidad de miembros. Se ha demostrado [GOU/94] que los protocolos de view synchrony funcionan bien para una red de hasta 40 máquinas, pero más allá de eso los protocolos colapsan. [BAB/95] [MAL/96]

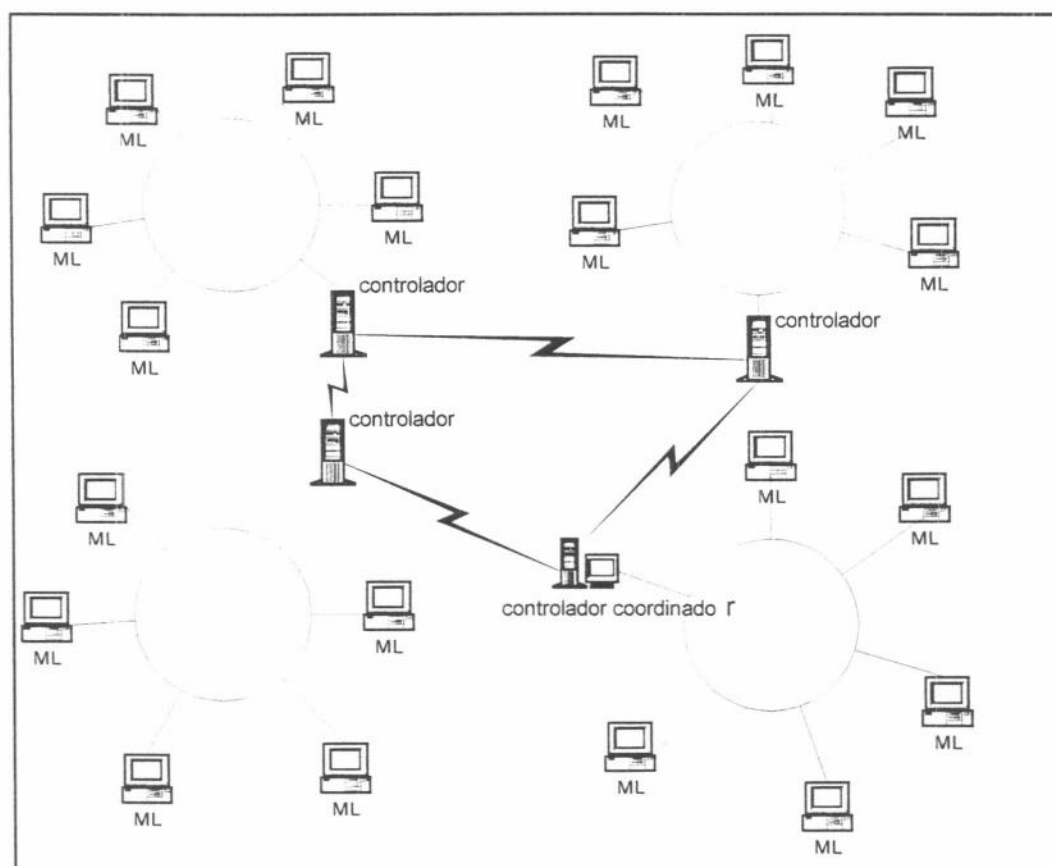


Fig 20 – View Synchrony Estructurado



Por esto se propone un modelo de 2 niveles en el cual los miembros son divididos en 2 categorías de acuerdo a su función: **Controladores y miembros locales**. Cada controlador maneja a un subconjunto de los miembros de la view total quienes constituyen sus miembros locales. Los controladores interactúan entre si y sus miembros locales, mientras que estos solo interactúan con otros miembros a través de su controlador. Además, uno de los controladores tiene una función especial durante el protocolo de cambio de view por lo que se lo distingue como **controlador coordinador**. [GOU/94]

La idea es mapear esta estructura sobre una red de intranets o LANs y así uno elegiría un controlador por intranet (idealmente la elección debería realizarse sobre la máquina más poderosa y estable de la red) mientras el resto serían miembros locales.

De esta manera el nuevo límite para los protocolos es de alrededor de 20 LANs de 40 miembros cada una, lo que da un total de 800 miembros posibles. [GOU/94]

5.3.2 Protocolos de Multicast

En esta sección explicaremos las modificaciones de los protocolos para enviar multicast dinámica y no dinámicamente uniformes (con sus variaciones de ordenes posibles: Fifo y causal) necesarias para su implementación en view synchrony estructurado.

Los protocolos originales de multicast para view synchrony no estructurado no serán explicados aquí. Para recordarlos ver 5.2

5.3.2.1 Multicasts para view synchrony estructurado

Como se comentó anteriormente, la solución propuesta consiste en utilizar una estructura de árbol en la cual los miembros son divididos en 2 categorías: Controladores y miembros locales. Cada controlador maneja a un subconjunto de los miembros de toda la View que constituyen sus miembros locales. Los controladores interactúan entre si y con sus miembros locales a través de una modificación a los protocolos de view synchrony, membership y multicast comunes.

Cuando se debe enviar un mensaje multicast, un controlador comienza enviándolo a los miembros del grupo de controladores. Luego cada controlador lo reenvía a los miembros de su grupo local. Cuando un miembro local desea enviar un



mensaje, primero lo envía a su controlador y es éste el que se encarga de realizar el multicast como si fuera el emisor original (adueñándose del mensaje).

Cada miembro mantiene todos los mensajes que ha enviado y recibido. Un miembro local envía los ACK sólo a su controlador. Después de recibir un mensaje *m* del controlador *j*, el controlador *i* no lo marca como RECIBIDO hasta que haya sido asentido con ACKs por todos sus miembros locales. Cuando un controlador detecta la estabilidad del mensaje *m* en el grupo de controladores, lo borra de su buffer. Cada controlador envía un multicast avisando la estabilidad de *m* a sus miembros locales para que lo borren de sus respectivos buffers.

5.3.2.1.1 Multicasts no dinámicamente uniforme

El controlador que desea enviarlo (ya sea porque es un mensaje originado en él o porque se lo adueñó de un miembro local) comienza enviando el mensaje OK_PARA_ENTREGAR al grupo de controladores y a sus miembros locales. Cada controlador lo reenvía a sus miembros locales como un mensaje OK_PARA_ENTREGAR_LOCAL. Como en el protocolo original para el modelo no estructurado cada miembro puede entregarlo apenas lo recibe por lo que aquí también es así. Cada miembro local responde a sus controladores con un ACK OK_PARA_ENTREGAR_LOCAL_ACK. Al recibir todos los ACKS de todos sus miembros locales cada controlador responde al controlador adonde se originó el mensaje con un OK_PARA_ENTREGAR_ACK. El controlador originador colecta todos los ack tanto del grupo de controladores como de sus propios miembros locales y luego pasa a la segunda fase de protocolo.

En la segunda fase envía el mensaje TODOS_LO_VIERON que es reenviado de la misma manera antes descrita por cada controlador a sus miembros locales. Y estos a su vez responden también con un TODOS_LO_VIERON_LOCAL_ACK lo cual genera TODOS_LO_VIERON_ACK de los controladores al controlador original. Al recibir el mensaje TODOS_LO_VIERON cada miembro puede borrar el mensaje pero debe todavía retener el message-id para el caso de que le llegue un duplicado. La tercera fase es igual pero con el mensaje OK_PARA_GARBAGE_COLLECT.

5.3.2.1.2 Multicasts dinámicamente uniforme

Las modificaciones al protocolo original son las mismas que para el no dinámicamente uniforme con la única diferencia (en realidad causada por la uniformidad dinámica y no por la estructura de view synchrony) de que la entrega del mensaje solo es posible al final de la segunda fase del protocolo. Los mensaje



enviado en cada etapa son : `SAVE_MESSAGE` (que se reenvia como un `SAVE_MESSAGE_LOCAL`) y las otras tres etapas son las mismas que en el protocolo anterior.

5.3.3 Protocolos de View Change

Explicaremos a continuación el protocolo de view change estructurado. Este es el protocolo ejecutado cuando se producen cambios en la composición del grupo (la falla o la conexión de algún miembro) para resincronizarse y pasar a una nueva view.

Primero esbozamos el protocolo utilizado para cambios de view en view synchrony no estructurado y luego explicaremos la adaptación para el cambio de view bajo view synchrony estructurado. [GOU/94] [REN/94]

5.3.3.1 Protocolo de cambio de view de view synchrony no estructurado

Al producirse un cambio en la composición del grupo , el coordinador (elegido entre los miembros del mismo) realiza un multicast del mensaje `FLUSH` (el nombre es tomado de la bibliografía, significa a la vez que se está comenzando un cambio de view y que le envíen los mensajes inestables y la percepción de pertenencia al grupo). Al recibir el `FLUSH`, todos los miembros sobrevivientes suspenden la emisión de nuevos mensajes y envían al coordinador todos los mensajes de los miembros que fallaron que no se sepa que han sido recibidos por todos (mensajes inestables) y la lista de quienes ellos creen están vivos y quienes no. Después de esto cada miembro sobreviviente envía un `FLUSH-OK` al coordinador. Al recibir este mensaje , el coordinador realiza un multicast de los mensajes inestables a todos los miembros que le contestaron el primer mensaje. Cuando todos los mensajes se han vuelto estables, el coordinador envía, por último un mensaje `VIEW` conteniendo la nueva composición de los miembros del grupo.

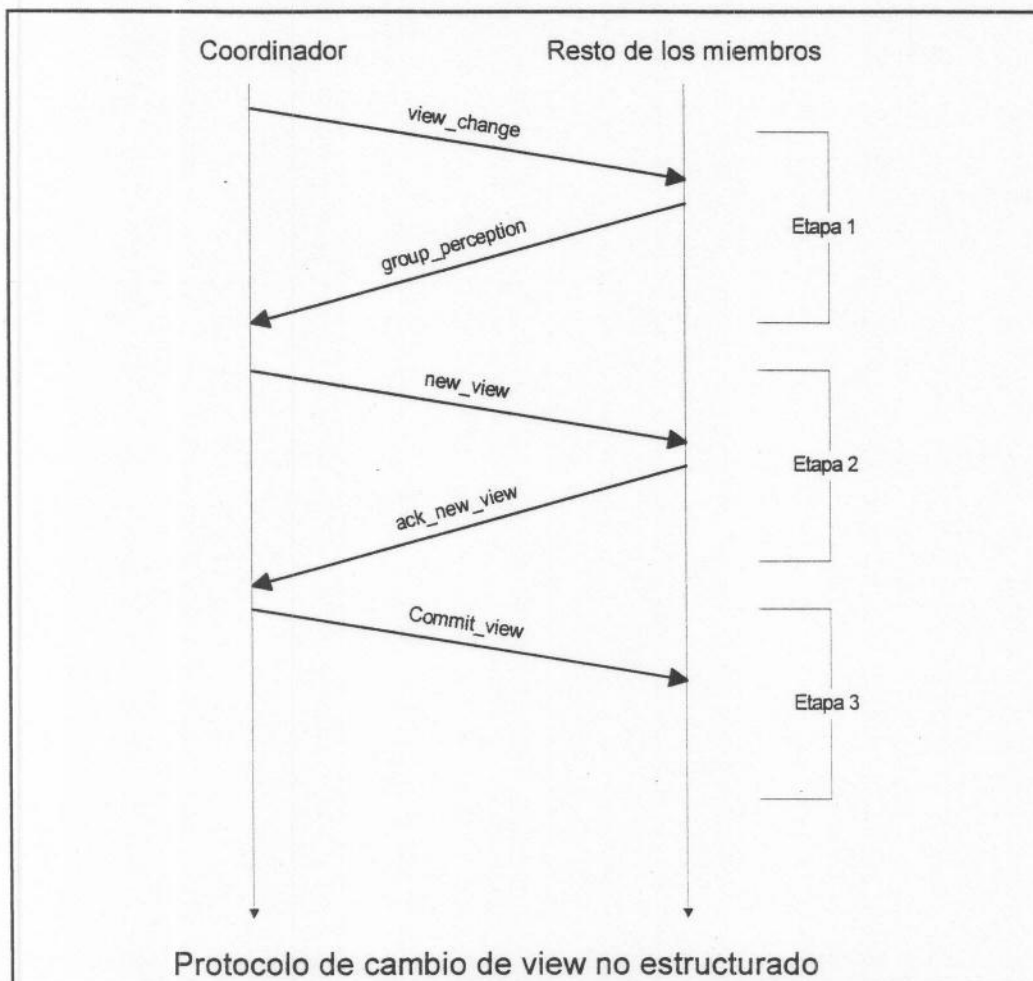


Fig 21 –Protocolo de cambio de view de View Synchrony Estructurado

5.3.3.2 Protocolo de cambio de view de View Synchrony estructurado

Cuando se detecta un cambio en la composición del grupo (una falla o una solicitud de conexión), el coordinador del grupo de controladores envía un mensaje de FLUSH (que significa: estoy comenzando un cambio de view, envíenme los mensajes inestables y su percepción de quien falló y quien desea conectarse) al grupo de controladores. Al recibir el FLUSH, cada controlador lo reenvía como un FLUSH-LOCAL en su grupo de miembros locales. Al recibir un FLUSH o un FLUSH-LOCAL, cada miembro para de enviar nuevos mensajes.



Los miembros locales, entonces, envían a sus respectivos controladores todo mensaje inestable así como también la percepción de pertenencia al grupo y luego un FLUSH-LOCAL-ACK.

Cada controlador, después de recibir FLUSH-LOCAL-ACK de cada uno de sus miembros locales, envía el conjunto de mensajes inestables (unión de los mensajes enviados por sus miembros y los propios) y la percepción de pertenencia al coordinador seguido de un FLUSH-ACK. El coordinador posee así el conjunto de mensajes inestables de todo el grupo y la lista de miembros fallados y no fallados.

Al comenzar el segundo round el coordinador envía el conjunto de mensajes inestables y la nueva vista (mensaje VIEW) a los controladores, los cuales lo reenvían a sus miembros locales como un mensaje VIEW_LOCAL.

Cuando cada controlador, por último, le hace llegar al coordinador el VIEW_ACK (luego de esperar los VIEW_LOCAL_ACK de sus miembros locales) comienza la tercera fase adonde el coordinador envía un mensaje de COMMIT. Mensaje que como los anteriores es reenviado por los controladores a sus miembros locales. Cada miembro ante la llegada de este mensaje instala la nueva view.

Con esta implementación se posibilita un aumento en la cantidad de miembros lográndose un incremento dramático en la escalabilidad de View Synchrony producto de :

- 1) Reducir el tráfico de mensajes de control
- 2) procesamiento en paralelo de ACKs de datos y FLUSH-LOCAL-ACKS
- 3) manejar las retransmisiones en forma local por cada controlador.



5.4 Otros sistemas

5.4.1 Relacs

5.4.1.1 Introduccion

Relacs es una infraestructura de comunicación para la construcción de aplicaciones confiables en sistemas distribuidos de gran escala. Fue diseñado para ser altamente confiable e implementa un número de abstracciones y primitivas para desarrollar sistemas tolerantes a fallas sobre sistemas distribuidos de gran escala basados en mecanismos de grupos de procesos. [BA2/94] [BAB/95]

Esta construido como una arquitectura de capas sobre la cual se desarrolla el conjunto de los mecanismos de grupos. Implementa un pequeño conjuntos de primitivas que permiten a una aplicación usuaria conectarse, desconectarse y enviar mensajes de multicast entre los miembros del grupo. Las garantías de consistencia de los estados de las aplicaciones miembros del grupo están basadas en la noción de view synchrony. Relacs, sin embargo, no implementa otras abstracciones ni servicios de mayor nivel como ser: orden total, causal o la propiedad de uniformidad dinámica. Todas estas son construidas por encima de Relacs en base a las facilidades que provee.

A diferencia de Isis/Horus que están orientados a trabajar en redes de área local (LANs), Relacs está diseñado para sistemas distribuidos de mayor escala.

5.4.1.2 Arquitectura del sistema

Cada sitio corre una instancia de un proceso llamado **Servidor Relacs** que se encarga de recolectar información sobre accesibilidad (de determinar si un sitio es alcanzable o no), multicasts de mensajes, entrega de mensajes, cambios de views y facilitar la comunicación con otros sitios a otros procesos locales al sitio que necesiten de estos servicios.



Por ejemplo, un request para enviar un multicast de un mensaje se realiza de la siguiente manera:

Primero, el proceso emisor del multicast realiza el request al servidor relacs invocando la primitiva `v_cast()`. El servidor Relacs local envía el mensaje a todos los otros miembros locales del grupo. El mensaje es enviado también a otros servidores Relacs en sitios remotos que tengan por lo menos a un miembro del grupo para que lo distribuyan entre estos.

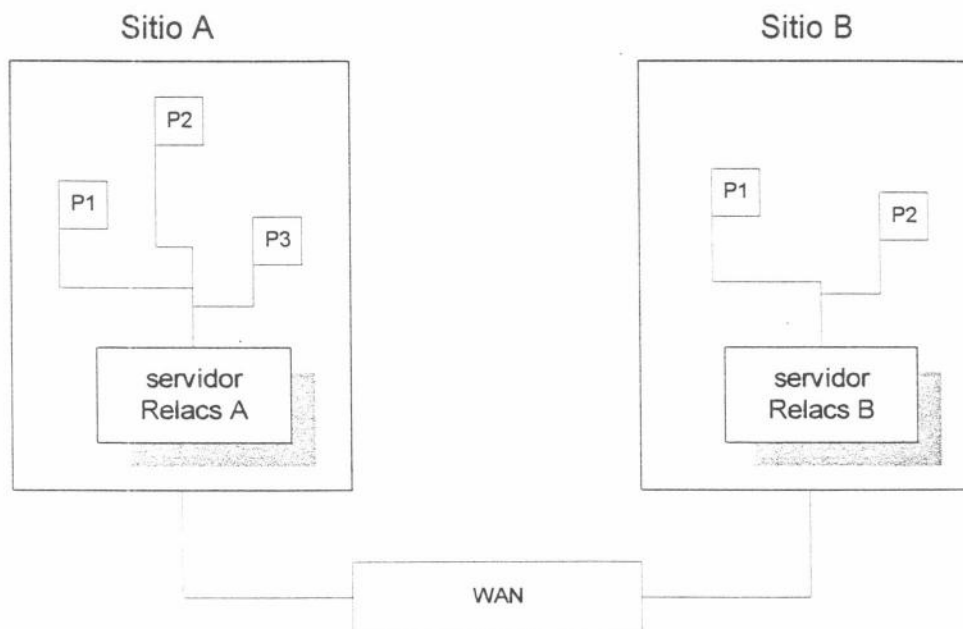


Fig 22 –Arquitectura de red del sistema Relacs

Un servidor Relacs consta de la siguiente pila de capas:

Network: Es la capa de más bajo nivel, interface para los servicio de comunicación de red para sitios remotos.

Local communication: Similar a la capa Network pero para comunicación con los procesos locales.

Transport: Capa que implementa una comunicación punto a punto confiable por sobre la capa network.



Failure Suspector: Utiliza mensajes de “ping” periódicos para construir la información de accesibilidad de otros sitios remotos.

View change/Multicast: Maneja los cambios de view y el envío y entrega de mensajes.

Gossip: Recolecta información de que procesos locales están vivos o no y la comparte con otros sitios remotos.

En un servidor relacs típico existe una única instancia de las capas de comunicación (Network, Local y transport) y el failure suspector. En cuanto al resto de las capas se instancia una por cada grupo que haya en ese sitio. Esto se hace para limitar los costos de la solución a un orden de sitios en lugar de procesos. Un ejemplo de esto es el failure suspector que provee información de accesibilidad a nivel de sitios. Esta información sólo se traduce a fallas de procesos cuando se la combina con la que proviene de la capa Gossip.

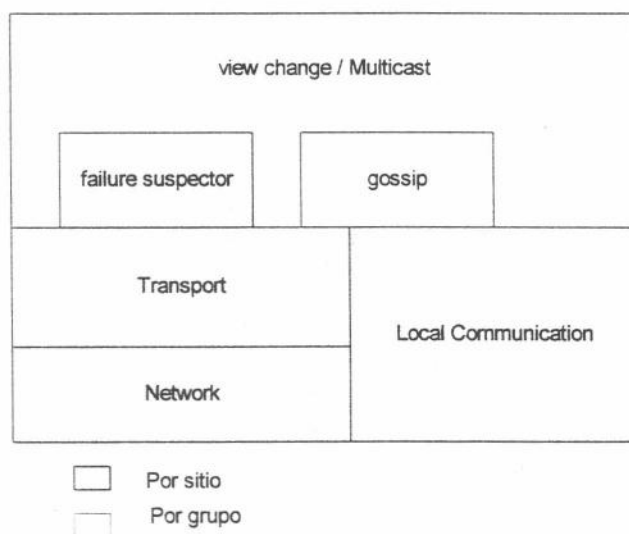


Fig 23 –Arquitectura de capas del sistema Relacs

5.4.1.3 Descripción detallada de las capas que componen la arquitectura



Network: es la capa que aísla a las superiores de los detalles de implementación del sistema operativo. Está construida encima del servicio de datagramas de UDP/IP. Permite a la capa de transporte enviar y recibir mensajes punto a punto no confiables de sitios remotos.

Transport: Esta capa provee un servicio de envío de mensajes confiable a las capas superiores. Los mensajes enviados son asentidos al ser recibidos por los receptores o retransmitidos si el timeout vence. La falta de asentimientos se informa al failure suspector para que éste modifique la información de accesibilidad de sitios.

Para descartar mensajes viejos provenientes de miembros fallados, utiliza un número de encarnación para descartar todos los mensajes con números de encarnación menores al actual.

Esta capa, sin embargo, no provee un ordenamiento FIFO de los mensajes que envía. La propiedad de view synchrony no especifica el orden de los mensajes, sino el conjunto de mensajes que se deben entregar en una view determinada.

La interfaz entre esta capa y la capa de View change /Multicast se realiza a través de un conjunto de funciones que son registradas en la capa de transporte. Cuando llega un mensaje dirigido a la capa de Multicast, un bit en la cabecera del mensaje sirve para identificar a cual de las funciones registradas de la capa de View change /Multicast se debe invocar.

Failure Suspector: Supervisa todo el tráfico de mensajes a fin de detectar cambios en la información de accesibilidad de sitios. Al comienzo todos los sitios son marcados como "NO ALCANZABLE" y sólo se cambian al estado "ALCANZABLE" cuando se recibe un mensaje proveniente del sitio en cuestión, ya sea de datos o un asentimiento a un mensaje. Si falla un envío, después de varias retransmisiones fallidas se marca al sitio como "NO ALCANZABLE".

Una vez marcado un sitio como "NO ALCANZABLE" el failure suspector trata de reincorporarlo enviando periódicos mensajes de echo en espera de un asentimiento.

Local communication: Esta capa se encarga de manejar la comunicación entre los procesos locales y el servidor Relac. Utiliza para ello envío de mensajes UDP que activan en los procesos locales una invocación a una función que ha sido preparada al efecto (callback function).



Además esta capa mantiene información acerca de si los procesos locales continúan operativos o no a través de un mecanismo de lock. Cuando un proceso local se une al grupo, solicita un área de memoria compartida para su uso. La capa de Local communication trata periódicamente de obtener esas áreas de memoria compartida. Si lo logra significa que o bien el proceso terminó su ejecución o bien falló.

Gossip: Esta capa brinda el servicio de recolectar cambios en la información de pertenencia al grupo (membership changes) y enviarlos a los otros servidores a través de la capa de transporte. Se dice que esta capa envía el commset, es decir el conjunto de todos los sitios alcanzables desde este server, a todos los miembros del grupo. Ahora bien, esto implicaría que cada servidor Relacs envíe un mensaje extenso por cada cambio en la pertenencia al grupo que detecte. En lugar de esto se envía un número de versión de commset que es utilizado por el algoritmo de view synchrony que se explicará en la siguiente capa.

View Synchrony/Multicast: Esta capa, como se mencionó con anterioridad, se ocupa de manejar los cambios de views y la entrega de los mensajes de multicast. Para ello, la capa puede trabajar en dos modos mutuamente excluyos: modo view change y modo multicast.

La view de cada grupo, mantenida por el servidor, consta de la siguiente información:

- el nombre de la view.
- el commset y su número.
- Una tabla para cada server que contenga al menos a un miembro del grupo. Para cada uno de estos guarda su número de versión de commset y un flag indicando si el sitio es alcanzable o no. A esta información de un servidor A, se la conoce como CV(A).
- Una tabla con los mensajes entregados (aquellos mensajes de multicast que han sido acordados por todos los miembros). Los mensajes recibidos pero no entregados se guardan en otra tabla hasta que finalice el algoritmo de view change.

Durante el algoritmo de view change uno de los servidores funciona como coordinador. Este miembro es el encargado de construir la nueva view una vez que todos los otros servidores acordaron con él la conformación del nuevo grupo



(se lo elige en forma determinística entre los miembros de la view, por ejemplo el primero de la lista).

Cuando un servidor A entra en modo view change realiza dos acciones:

1. Selecciona a un coordinador de la lista (como la función es la misma en todos los sitios se elige al mismo miembro).
2. Envía a este coordinador la siguiente información: el nombre de la view actual, su CV(A) y el conjunto de mensajes entregados.

El coordinador, una vez que ha recibido toda esta información de todos los sitios que él considera alcanzables, construye con ella un mensaje conteniendo el CV acordado, el conjunto de mensajes entregados por todos acordado y el commset completo (la concatenación de todos los procesos locales de cada sitio).

Cuando un servidor recibe el mensaje de nueva view del coordinador, entrega los mensajes provenientes en el mensaje que todavía no haya entregado y luego instala la nueva view.

En modo multicast, cada vez que el servidor Relacs recibe una solicitud de envío de mensajes de un proceso local, envía el mensaje a través de la capa de transporte a todos los sitios adonde halla por lo menos un proceso perteneciente al grupo. Cuando se recibe, por esta misma capa un multicast proveniente de otros sitios, se lo pasa a la capa de local communication que se encarga de entregarlo a los procesos locales pertenecientes al grupo.

El cambio de mode multicast a view change se produce cuando o bien la capa de transporte informa al servidor de un cambio en la percepción de alcanzabilidad de sitios o bien cuando la capa de comunicación local informa que un proceso quiere unirse o desconectarse del grupo.



5.4.2 Horus

5.4.2.1 Introduccion

Horus es una arquitectura de comunicación para la construcción de servicios replicados confiables en sistemas distribuidos de gran escala. Fue diseñado como una pila de capas de protocolos que se interconectan a través de una interfaz bien definida a la manera de componentes de hardware (o componentes de software). Dichas capas pueden ser apiladas y conectadas de varias maneras, incluso en tiempo de ejecución. [REN/94] [REN/95] [BIR/96]

Horus ofrece una gran variedad de capas de protocolos que van desde aquellas que implementa una comunicación de red no confiable (envío de paquetes de mejor esfuerzo) hasta aquellas que proveen recuperación ante pérdida de paquetes, control de flujo y distintos ordenes entre mensajes. Las garantías de consistencia de los estados de las aplicaciones miembros del grupo la proveen unas capas de protocolo basadas en el concepto de view synchrony.

Como dijimos antes, en Horus las capas de protocolos son tratadas como un objeto de software con una interfaz de entrada y salida (por debajo y por encima de la capa) estandarizada y bien conocida. Encima de un módulo del protocolo hay otros módulos o aplicaciones que realizan requerimientos y esperan una respuesta.

El sistema Horus provee así un framework de composición de protocolos orientado a objetos. El sistema brinda objetos para realizar comunicación punto a punto y multipuntos de mensajes entre un conjunto de procesos. La facilidad para componer distintas pilas de protocolos de acuerdo a los requerimientos de la aplicación hace que sea un sistema flexible y cada aplicación puede evitar sobrecarga de capas no útiles en un contexto dado.

Fue diseñado originalmente para trabajar en redes de área local de alta velocidad (NOWs: clusters de workstation), pero se le han ido agregando las últimas modificaciones en técnicas y teorías referidas a la tolerancia a fallas.



5.4.2.2 Arquitectura del sistema

El modelo de proceso para Horus es el de un proceso del sistema operativo construido por sobre una pila de los protocolos elegidos para esa aplicación en particular.

Existen alrededor de una veintena de capas de protocolos que se pueden apilar para una determinada aplicación. Entre ellas podemos nombrar a las siguientes capas de protocolos:

- **Routing:** rutea fragmentos de un mensaje más largo sobre Internet.
- **Fragment/Assembling:** Fragmentación de un mensaje largo en otros más cortos y su ensamblado en el receptor.
- **Flow control:** Control de congestionamiento y flujo entre miembros.
- **Ordering:** Provee un envío de mensajes con distintos órdenes: fifo, causal y total.
- **Mbrship:** Servicio de pertenencia a un grupo de procesos, caída y agregado de miembros y protocolo de cambio de view.

Horus también provee objetos de tipo thread que realizan cálculos, compartiendo un estado en común (entre todas las threads).

5.4.2.3 Detalle de las capas de protocolos

En esta sección describimos algunas de las capas de protocolos más importantes que ofrece Horus para la composición de la pila de protocolos.

Un ejemplo de composición es la siguiente disposición desde la capa superior a la inferior:

- ❖ TOTAL
- ❖ MBRSHIP
- ❖ FRAG
- ❖ NAK
- ❖ COM



Sin embargo, la describiremos a continuación de abajo hacia arriba:

5.4.2.4 Protocolo COM

Provee una capa de comunicación no confiable basada sobre servicios de red de bajo nivel. Mantiene el origen de cada mensaje y descarta mensajes

provenientes de miembros no pertenecientes al grupo.

5.4.2.5 Protocolo NAK

Esta capa ofrece un servicio de envío de mensajes con orden FIFO. Númera para ello a todo mensaje saliente con un número de secuencia único por cada miembro que el receptor chequea para ver si no ha recibido algún mensaje. El miembro que detecta un agujero en la secuencia de numeración de los mensajes envía un mensaje de NAK (reconocimiento negativo) indicando el último mensaje que ha recibido. Se guardan los mensajes enviados durante un cierto tiempo. Cada cierto tiempo se envía un reporte de los mensajes recibidos entre todos los miembros (mensajes estables). La falta de un reporte de este estilo de un miembro c genera la sospecha en el resto de los miembros de que c ha fallado o se ha desconectado.

5.4.2.6 Protocolo FRAG

Provee un servicio de fragmentación y reensamblado de mensajes largos. Cuando un usuario de la capa intenta enviar un mensaje más largo que el permitido por la red subyacente, FRAG lo divide en múltiples fragmentos que luego envía utilizando el servicio de envío FIFO provisto por la capa NAK.

5.4.2.7 Protocolo TOTAL

Esta capa subyace sobre una capa de comunicación que provea una garantía de sincronía de vistas (View Synchrony). Provee un servicio de envío de mensajes



con orden TOTAL. En modo normal utiliza para numerar los mensajes un token. Sólo el miembro que posee el token en un determinado momento puede enviar un mensaje con orden total numerado con el número de secuencia que viene en el token. Un algoritmo en esta capa elige al siguiente miembro al que se le pasará el token.

Durante la fase de FLUSH del protocolo de cambio de view, todo miembro que todavía no envió sus mensajes TOTAL los multicastea a todos los otros miembros de la vista. Estos mensajes no son entregados inmediatamente y se guardan en buffers especiales. Cuando la nueva vista se instala estos mensajes se ordenan con un orden total global (por ejemplo el orden del miembro emisor en la view) y se entregan finalmente.

5.4.3 Transis

5.4.3.1 Introducción

Transis es un sistema de comunicación basado en multicasts que facilita el desarrollo de aplicaciones tolerante a fallas distribuidas sobre una red de máquinas. Soporta comunicación de grupos de procesos confiable para la implementación de aplicaciones de alta disponibilidad. Contiene un protocolo para entrega de mensajes confiable que optimiza la performance para el hardware algunas redes existentes. Provee además mecanismos para reconciliar los estados de los miembros del grupo luego de particiones de la red basados en Extended View Synchrony. [AMI/95] [BIR/96]

El sistema de comunicación Transis provee primitivas de comunicación a grupos de procesos. Los grupos son identificados por un nombre seleccionado por el usuario. Los mensajes son así direccionados a un grupo entero especificado por el nombre. Usando la abstracción de grupos el subsistema de comunicación evita que el usuario debe ocuparse de los destinos de los mensajes o de la ruta que deben seguir. Garantiza además una semántica de entrega "a Todos" o "a ninguno" y maneja pérdida de mensajes y fallas transitorias de la red transparentemente.

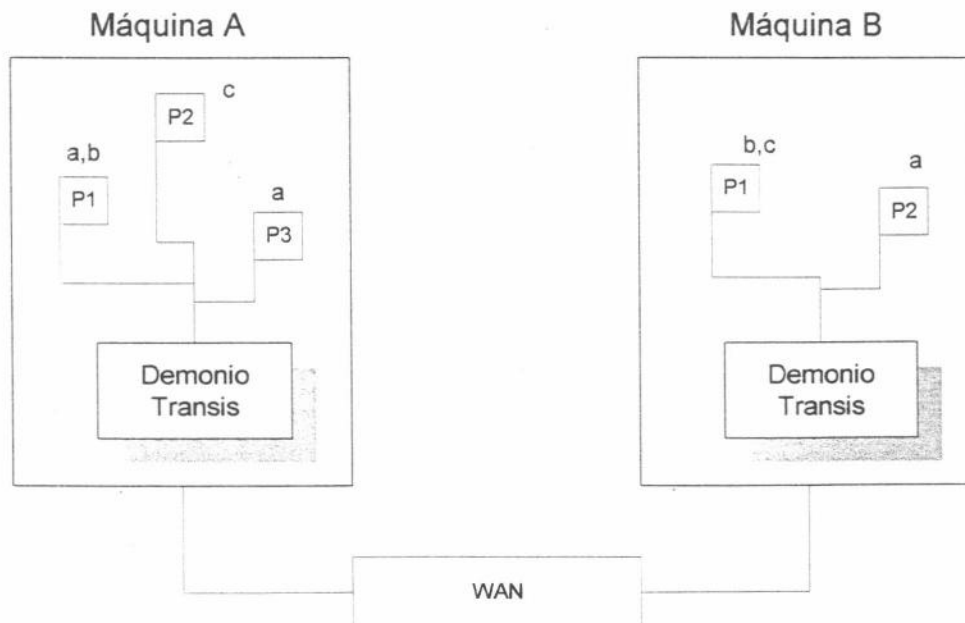
Provee varios tipos de primitivas de multicasts: con orden FIFO, con orden Causal, con orden total y con orden seguro (safe order). Los distintos tipos difieren en su semántica y en su costo (básicamente, la latencia). El multicast de orden total es el de mayor latencia ya que la entrega del mensaje ocurre luego que todas las



maquinas de la red luego de recibir la copia del mensaje, acuerdan el orden de entrega.

5.4.3.2 Arquitectura

La arquitectura del sistema de comunicación Transis está basada en unos procesos conocidos como "Transis Daemons". Cada máquina que pueda albergar procesos participantes de la comunicación en grupo posee un demonio Transis corriendo. La comunicación entre las máquinas que contienen miembros del grupo se realiza a través de estos demonios. Cada demonio Transis mantiene una lista de los procesos residentes en esa máquina que forman parte de alguno de los grupos de comunicación. Lo que se mantiene como lista de pertenencia al grupo son las máquinas que contengan miembros participantes en lugar de los procesos a diferencia de otros sistemas de comunicación basados en grupos como Horus. los mecanismos de comunicación de grupo (las primitivas de multicast y detección de fallas) son implementados como librerías que pueden ser linkeadas con los procesos de aplicación (del servicio que se está replicando).



a,b,c: Nombre de los grupos

Fig 24 –Arquitectura de red del sistema Transis



Las ventajas que ofrece esta arquitectura son muy importantes:

El algoritmo de cambio de view es invocados sólo cuando hay un cambio en la composición de pertenencia del grupo, en este caso de la máquina adonde reside el demonio Transis (un acontecimiento más improbable que la falla de un proceso). Cuando un proceso voluntariamente se agrega o deja el grupo el demonio envía un mensaje de notificación a los otros demonios. Cuando este mensaje es ordenado (con los otros multicast que lleguen) el demonio entrega un mensaje de cambio de view conteniendo la nueva composición del grupo a los miembros sobrevivientes.

El control de flujo se mantiene al nivel de los demonios Transis en lugar de al nivel de procesos individuales. Esto produce una mejora global de la performance.

El orden de los mensajes se consigue y mantiene a nivel de los demonios Transis en lugar de por cada grupo. Por lo tanto el ordenamiento de mensajes es más eficiente en términos de latencia y excesos de mensajes.

El ordenamiento de mensajes a nivel global es trivial ya que se mantiene sólo un orden global al nivel de las máquinas que forman parte del grupo.

Los procesos que no forman parte de ningún grupo pueden enviar multicast al grupo sólo con comunicarse con un demonio Transis.

La arquitectura puede ser cambiada de ser necesario de minimizar la comunicación inter-procesos. El código del demonio Transis puede ser linkeado junto con el código del programa de usuario para crear un único proceso.

5.4.3.3 Interfaz de programación (API)

La interfaz de programación de Transis consta de las siguientes funciones:



connect: Un proceso inicia una conexión con Transis. Esto crea un manejador de comunicación entre procesos en el proceso usuario similar a un socket TCP. Un proceso puede mantener varias conexiones a Transis.

disconnect: El proceso termina la conexión al sistema.

join: Un proceso voluntariamente se une a un grupo especificado (por el nombre) en una conexión Transis dada. El primer mensaje del grupo será una notificación de los miembros actualmente conectados en el grupo (la view del grupo).

leave: Un proceso voluntariamente deja un grupo especificado de una conexión especificada.

multicast: Un proceso genera un mensaje a ser enviado como un multicast por Transis a un conjunto de grupos destino. Se especifica además el orden con que deben ser enviados.

receive: Utilizado por un proceso para recibir un mensaje entregado por Transis en una conexión específica. El mensaje puede ser un conjunto de mensajes regulares enviados por procesos o una notificación de cambio de view generada por Transis indicando un cambio en la composición de algunos de los grupos de los que este proceso forma parte.

5.4.3.4 Protocolo de Multicast confiable basado en anillo (Ring)

El protocolo de multicast confiables basado en anillo provee entrega de mensajes y avisos de cambios de views de acuerdo al modelo de view synchrony extendido. El protocolo se basa en la existencia de algún mecanismo de multicast no confiable en el sistema (para mayor performance se utiliza el multicast por hardware de algunas redes). La mayoría de las redes de area local poseen mecanismos de multicast. De no existir puede ser simulado con múltiples envíos punto-a-punto sin afectar la correctitud del protocolo.

El principio fundamental del protocolo es conseguir ordenamiento de mensajes a través de la circulación de un token alrededor de un anillo lógico configurado por sobre las máquinas que componen el sistema. Sólo la máquina en posesión del token puede enviar multicast a los otros miembros del anillo. Se tiene en el protocolo especial cuidado en el caso de pérdidas del token o caídas o particiones del anillo.



El ordenamiento de mensajes se logra a través de un único número de secuencia para todas las máquinas que forman parte del anillo que se incluye en el último mensaje enviado como multicast en el token. Estampillando a cada mensaje saliente con el número de secuencia produce un ordenamiento total de los mensajes antes de ser enviados. Por lo tanto, cada proceso sabe al recibir los mensajes en que orden deben ser entregados.

5.4.4 Totem

5.4.4.1 Introduccion

El sistema Totem es un sistema para la comunicación entre grupos de procesos desarrollado en la Universidad de California. Provee un servicio de multicast confiable y totalmente ordenado entre grupos de procesos pertenecientes a una misma a Lan o a un grupo de Lans conectadas mediante gateways. Garantiza este servicio en la presencia de fallas como pérdida de mensajes , caída de procesadores , particion de la red y problemas de desincronización de relojes , pero no ante fallas arbitrarias (fallas bizantinas). [AGA/95] [AMI/94] [MOS/95] [MOS/96]

La característica más destacada de Totem es su capacidad de ofrecer a las aplicaciones un ordenamiento total entre sus mensajes , lo que produce una drástica simplificación en la complejidad en el desarrollo de sistemas que requieran el mantenimiento de la consistencia entre conjuntos distribuidos de datos. Efectivamente , lograr que dos servidores que reciben el mismo conjunto de mensajes en el mismo orden mantengan un mismo estado es una tarea bastante sencilla.



5.4.4.2 Arquitectura

Totem está estructurado como una jerarquía de capas :

Aplicación
Process Group Interface
Multiple-Ring Protocol
Single-Ring Protocol
Lan

Fig 25 –Arquitectura de capas del sistema Totem

La capa de más bajo nivel (Single-Ring Protocol) provee entrega y ordenamiento confiable de mensajes dentro de cada Lan y detecta cambios en la configuración local del grupo.

La capa siguiente (Multiple-Ring Protocol) provee el servicio de envío de mensajes totalmente ordenados en las comunicaciones entre procesos de Lans distintas. Además detecta los cambios en la configuración de la red (por ejemplo la desconexión de una Lan de la red global)

La capa de Process-Group Interface se encarga del mantenimiento de la composición de los grupos de procesos.

5.4.4.3 Descripción detallada de las capas que componen la arquitectura

Single Ring Protocol

Este protocolo utiliza un anillo lógico sobreimpuesto en cada Lan para proveer un servicio de entrega de mensajes confiable y totalmente ordenado dentro de un



dominio de broadcast. El anillo se construye mediante un token que circula entre los miembros de la Lan como un mensaje punto a punto.

Un proceso debe poseer el token para poder enviar mensajes al resto de los miembros del grupo. Cada mensaje contiene un número de secuencia que se deriva de un campo del token. Este campo del token se incrementa cada vez que se envía un mensaje y como el token circula por el anillo lógico, se obtiene un ordenamiento total entre todos los mensajes que se envían en la Lan.

Los procesos descubren pérdidas de mensajes cuando detectan discontinuidades en los números de secuencia de los mensajes recibidos. Para solicitar retransmisión de estos mensajes, incluyen sus números de secuencia en el token. Cuando un proceso recibe el token y este contiene pedidos de retransmisión de mensajes que el proceso posee, reenvía los mensajes y elimina los pedidos de retransmisión del token.

En el anillo local, un mensaje puede ser entregado tan pronto como todos los mensajes con número de secuencia menor lo hayan sido. Un mensaje puede eliminarse cuando todos los procesos lo hayan recibido.

Para proveer tolerancia a fallas, el protocolo está integrado con un protocolo de mantenimiento de la composición del grupo que permite la recuperación ante pérdida del token, falla de procesos o particiones de la red y el agregado de nuevos miembros al grupo. La falla o desconexión de procesos, así como la pérdida del token, se detectan por timeouts, mientras que los nuevos miembros se detectan cuando se reciben mensajes de procesos no pertenecientes al grupo.

El algoritmo de mantenimiento de la composición del grupo contiene cuatro estados. En el primer estado (llamado Gather) los procesos acuerdan la composición del nuevo grupo. En el segundo (llamado Commit) establecen el identificador del nuevo grupo y el orden en que el token visitará los distintos miembros. Después de que el token a empezado a circular comienza el estado Recover, durante el cual se entregan los mensajes provenientes de la view anterior, cumpliendo por lo tanto con la propiedad de View Synchrony. Una vez que se ha completado la entrega de estos mensajes los procesos pueden pasar al estado normal.

Multiple Ring Protocol

Este protocolo es ejecutado por cada proceso y gateway del sistema y es responsable de reenviar mensajes entre distintos anillos, mantener la topología de la red, y diseminar información local sobre fallas y uniones de miembros.



Cada mensaje enviado por este protocolo es marcado con una timestamp obtenida mediante los relojes lógicos de Lamport [BIR/96]. Los mensajes son enviados dentro de cada anillo mediante el protocolo de Single-Ring. Cuando un este protocolo entrega un mensaje a una gateway , se le asigna al mensaje un nuevo número de secuencia , adecuado para el nuevo anillo , pero mantiene su timestamp. Los gateway entregan un mensaje de un anillo a otro en el orden dado por su numero de secuencia local.

La condición para poder entregar un mensaje en este protocolo es que el mensaje tenga la timestamp más chica de todos los mensajes recibidos y que además se haya recibido al menos un mensaje con timestamp más grande originado en cada uno de los anillos que componen el sistema , respetándose de esta manera el orden global .

Cada gateway mantiene una estructura de datos llamada topología , con los anillos con los que está conectada y las gateway que los conectan. La topología de la red puede ser totalmente arbitraria, ya que como los gateways conocen esta topología pueden adaptar el ruteo de los mensajes.

Las fallas de procesos y particiones de la red son detectadas en forma local por el protocolo de Single-Ring , que genera un mensaje reportando el cambio. Cada gateway analiza el efecto del cambio en la topología y el protocolo Multiple-Ring envía un mensaje que refleja este cambio.

Process Group Interface

Una aplicación construida sobre Totem , esta estructurada como una colección de grupos de procesos. Cada grupo de procesos es un conjunto de procesos que cooperan para llevar a cabo una de las tareas de la aplicación. Un proceso puede ser miembro de varios grupos y cada mensaje puede ser enviado a uno o más grupos.

La interfase de grupos de procesos provee servicios para enviar mensajes a grupos de procesos , recibir estos mensajes , crear nuevos grupos y unirse o dejar un grupo de procesos. La interfase provee a cada proceso de la aplicación un socket mediante el cual este puede interactuar con Totem y recibir mensajes de las otras procesos del grupo.

En cada procesador , la interfase mantiene también el estado de todos los grupos a los cuales al menos uno de los procesos del procesador pertenece . Cuando un



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

proceso se une o deja un grupo , esta información es diseminada por toda la red hacia todos los miembros del grupo mediante el protocolo de Group Membership.



6 Definición de la arquitectura de software

6.1 Introducción

El objetivo del presente trabajo es diseñar una arquitectura de capas que ofrezca los mecanismos de tolerancia a fallas necesarios para el desarrollo de aplicaciones distribuidas de gran escala basados en la abstracción de View Synchrony Estructurado. [GOU/94] [BIR/96] [BAR/98]

Comenzaremos describiendo el modelo del sistema que tuvimos en cuenta al diseñar nuestra arquitectura. De esta manera se podrán apreciar las razones de las decisiones de diseño que hemos tomado así como también porque hemos hecho hincapié en solucionar algunos problemas y otros no. Luego presentaremos la arquitectura de capas propuesta y las ventajas y desventajas que tiene respecto a otros sistemas. Junto a la arquitectura se describirá el framework de aplicación que presenta una abstracción de mayor nivel para programar aplicaciones distribuidas de gran escala. Por último indicaremos como se puede utilizar nuestra arquitectura para brindar mecanismos de tolerancia a fallas para una máquina global distribuida sobre internet definida por nosotros y cuya arquitectura de distribución de cómputo ya fue presentada en la tesis [EST/00] .

6.2 El Modelo del sistema

Las características y los servicios del sistema sobre el cual se basa nuestra arquitectura son los típicos de un sistema distribuido débilmente acoplado como Internet. De forma abstracta el sistema puede ser modelado como una colección de procesos ejecutando en máquinas distintas en sitios potencialmente remotos. Los procesos deben comunicarse a través de un servicio de intercambio de mensajes provisto por una capa de red. Tanto los procesos como la red de comunicación pueden fallar.

El modelo de red al que nosotros nos dirigimos es el de una red imperfecta, no totalmente conectada, en la cual la comunicación se puede interrumpir parcial o totalmente. La red puede perder o duplicar paquetes y además no provee ninguna garantía en cuanto al orden de los mensajes enviados entre procesos ejecutando en sitios distintos. En cuanto a la carga del sistema, los recursos de comunicación y de computación pueden ser compartidos por una gran cantidad de procesos por lo cual la carga del sistema es altamente impredecible y variable. Por todos estos



motivos es imposible acotar el delay de comunicación entre procesos o la velocidad de procesamiento relativa entre ellos. A un sistema de estas características se lo denomina **asincrónico**. [BIR/96] [BAR/98]

Los sistemas asincrónicos hacen que los razonamientos de un proceso antes fallas de otros procesos o de la red de comunicación sean complejos. El motivo por el cual un proceso p no puede comunicarse con otro proceso q no puede ser establecido con exactitud: q puede haber fallado, o estar lento o el link de comunicación puede haber fallado o con retardos. Desde el punto de vista de p, cualquiera de esos escenarios de fallas provoca que q no sea alcanzable por p. [BAB/97]

En nuestro sistema utilizaremos para inferir si un proceso ha fallado ó no a un **detector de fallas débil** [BIR/96] que se ha demostrado es suficiente para resolver view synchrony ante un escenario de fallas como el descrito. El detector genera sospechas de fallas a través de la utilización de timeouts para detectar caídas de procesos cuando no han enviado mensajes en un intervalo de tiempo dado. Se generan "sospechas" en lugar de "avisos" de fallas debido a que el sistema es asincrónico por lo cual el proceso puede no haber fallado y sólo estar lento. El efecto en nuestra arquitectura es el mismo: se lo considera como fallado y todos los procesos dejan de comunicarse con él.

El manejo de fallas no excluye la posibilidad de que se produzca una partición lógica de la red, en la cual un conjunto de procesos es particionado en varias islas mutuamente excluyentes no alcanzables entre si. Para manejar este problema la literatura ofrece varias alternativas que van desde elegir a una de las particiones para seguir operando (primary partitions) hasta las de intentar continuar con todas y luego proveer mecanismos de recuperación y reconciliación de los estados disjuntos (**Extended View Synchrony** [BIR/96] [BAB/97], **Enriched View Synchrony** [BAB/96] [BAR/98]). En nuestra arquitectura nos centraremos en el problema de la escalabilidad así que la resolución de particiones se encuentra fuera del alcance de esta tesis. Aun así, puede, posteriormente, ser incorporada por encima de nuestra solución de view synchrony estructurado. Así, en nuestro modelo del sistema simplificado asumimos que no se producen particiones lógicas de los grupos de procesos.

Por último, la visión de internet que utilizamos para fundamentar nuestra arquitectura es la de una unión de intranets corporativas, un conjunto de LANs interconectadas con links de comunicación típicamente más lentos que los que se dan entre las computadores pertenecientes a una misma LAN.

Una nota final, la arquitectura no fue diseñada para sistemas en los que se producen fallas aleatorias que hacen que un proceso responda a veces



correctamente y otras veces no. Por ello no tendremos en cuenta fallas bizantinas en nuestro modelo del sistema.

6.3 La arquitectura del sistema

En esta sección describiremos la jerarquía de capas que conforman la arquitectura de nuestro sistema; vale decir la funcionalidad de cada capa y la forma en que cooperan para ofrecer los mecanismos de tolerancia a fallas de la abstracción de view synchrony estructurado.

Como explicamos anteriormente, la arquitectura de red, sobre la cual correrá nuestro sistema es la de un conjunto de LANs interconectadas por links de comunicación cuya velocidad y ancho de banda son órdenes de magnitud menores a los que se dan entre los computadores de una misma red de área local. Esto hace que el costo de la comunicación local sea mucho menor al costo de comunicación entre miembros pertenecientes a distintas redes de área local. En cada máquina de la Lan corre un y sólo un proceso llamado **servidor** que es el encargado de brindar el servicio que se está replicando a los clientes que lo soliciten. Llamaremos **Subgrupo** al conjunto de máquinas de una misma Lan que participan activamente de nuestro sistema. **Grupo** es el conjunto de todos los subgrupos que forman a la view del sistema, es decir a todos los miembros del mismo. A las máquinas pertenecientes a un subgrupo se las denomina **miembros locales (ML)**. Para nosotros un miembro local y el proceso servidor que corre en él son indistinguibles por lo que se los mencionará de una forma u otra indistintamente en esta tesis. Uno de los miembros locales es el que se encarga de la comunicación con otros subgrupos de la view, por lo que se lo conoce como **controlador (contr)**. Entre todos los controladores se distingue a uno que cumple una función especial durante el protocolo de cambio de view, lo llamaremos de acuerdo a la bibliografía de view synchrony, **coordinador (coord)** de la view.

La estructura de jerarquía de capas de nuestra arquitectura consta de los siguientes componentes lógicos:

- **Netcomm:** Network communication, es la capa de comunicación de red de más bajo nivel. Actúa como una interface de comunicación de bajo nivel para comunicarse con sitios remotos.



- **RSS:** Reliable Send Service o servicio de envío confiable. Implementa un servicio de envío punto a punto confiable (con recuperación de paquetes y descarte de duplicados) por encima de la capa de red. Además permite aplicar filtros a los mensajes que llegan y avisa al servicio de detección de fallas cuando llega algún mensaje de algún miembro, indicando que aún está vivo.
- **MSS:** Multicast Send Service ó servicio de envío de multicasts. Esta capa implementa los distintos tipos de multicast de acuerdo a distintos órdenes (Fifo, Causal, Agreed, etc.) y con distintas propiedades de uniformidad dinámica (DU o NDU). Ofrece además un multicast especial de control no bloqueante a ser utilizado durante el protocolo de cambio de view.
- **FD:** Failure Detector o detector de fallas. Implementa a través de timeouts un detector de fallas débil, avisando cuando hace un tiempo que no llegan mensajes de otro miembro de que se sospecha que ha fallado. Además de encarga de enviar a otros FDs un mensaje de que el miembro al que pertenece aún no ha fallado.
- **VSS:** View Synchrony Service o servicio de view synchrony. Es el que ofrece la interfaz de alto nivel para el framework de aplicación de los multicast con distintos órdenes y propiedades que luego envía utilizando la capa MSS. Su función principal es realizar al detectar una falla de un miembro el protocolo de cambio de view de view synchrony estructurado.
- **FEVS:** Es el framework de aplicación por sobre el cual se debe programar la aplicación particular que brinda el servicio que se está replicando.



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

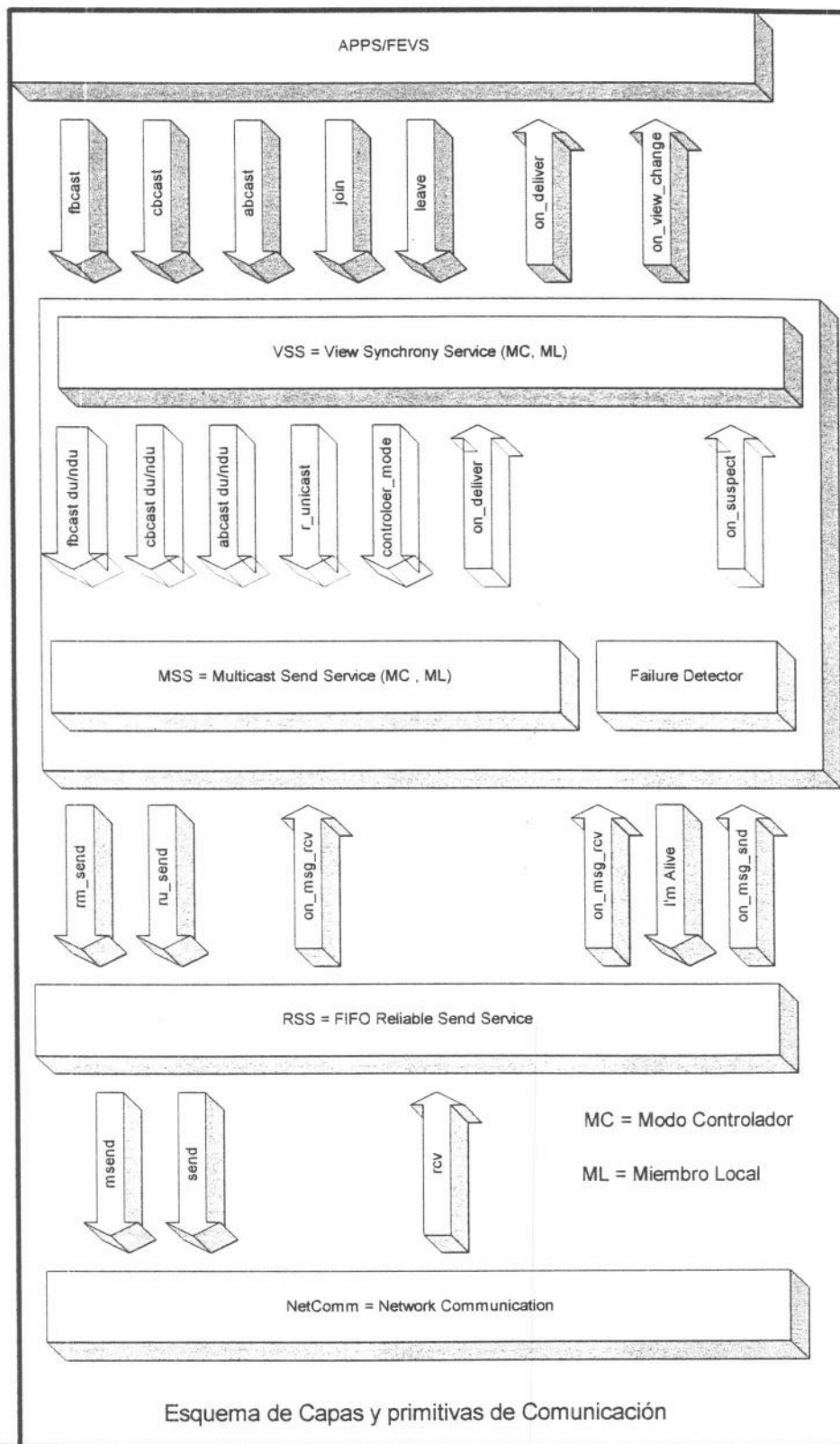




Fig 26 –Arquitectura de capas de la solución

6.4 Descripción detallada de las capas de la arquitectura

Se verá ahora en más detalle el diseño de cada una de las capas de nuestra arquitectura. Junto con él se describirán cuando sean necesario los algoritmos desarrollados.

6.4.1 La capa Netcomm

Esta capa se implementa por sobre un protocolo de red UDP no confiable que puede perder o duplicar paquetes. Brinda una interfase para enviar un paquete a un sólo destino (unisend) ó a varios a la vez (multisend). La idea al encapsular la interfase con la capa de red en esta capa es que se pueda reemplazar por otra con un protocolo de red que soporte, por ejemplo, envíos broadcast directamente.

6.4.2 La capa Rss

El objetivo fundamental de esta capa es aislar a las capas superiores de los problemas en la confiabilidad de los mensajes o en su orden. La capa Rss corresponde en nuestra arquitectura a la capa de transporte. Implementa un servicio de envío de mensajes punto a punto, no orientado a conexión, confiable y ordenado. El orden que se provee es FIFO, es decir que los mensajes provenientes del mismo emisor son recibidos en el orden en que fueron enviados. El hecho de que esta capa mantenga este orden facilita la implementación de los algoritmos de las capas superiores.

Los servicios que brinda a las capas superiores son :

- ReliableMSend: Envío punto a punto confiable con orden fifo de un emisor a múltiples receptores
- ReliableUSend: Envío punto a punto confiable con orden fifo de un emisor a un receptor.



La capa Rss es la que brinda un servicio de transporte confiable por sobre la capa NetComm a las capas superiores. Estas dos capas serían el equivalente lógico al protocolo TCP/IP, que no fue utilizado por motivos de optimización, ya que nuestros algoritmos son básicamente no orientados a conexión. De utilizar TCP, deberíamos abrir una conexión TCP, enviar un mensaje y cerrar la conexión por cada mensaje a enviar a cada miembro o mantener una conexión con cada miembro del grupo.

Para implementar un servicio confiable se utilizan asentimientos negativos que son enviados por la capa Rss de un miembro cuando detecta mensajes faltantes. Los mensajes faltantes se detectan fácilmente porque cada emisor numera los mensajes para que los receptores conserven el orden de emisión.

Esta capa tiene una relación muy fuerte con el Failure Detector (FD). Como se verá en su descripción, el FD determina que otro miembro del grupo ha sufrido una falla si se ha sobrepasado un tiempo dado sin recibir mensajes de este. Por lo tanto el FD de cada miembro debe ser notificado cada vez que se recibe o se envía un mensaje a un miembro. La recepción de un mensaje se notifica al FD porque de esta manera éste sabe que el miembro emisor está todavía operacional, mientras que el envío de mensajes se notifica para minimizar la cantidad de mensajes enviados por el FD, ya que cuando éste detecta que ha pasado un tiempo dado sin enviarle un mensaje a algún miembro, envía un mensaje sin datos solo para avisarle que sigue operacional. Por otro lado el hecho de que si las capas superiores no envían mensajes durante un tiempo dado el FD enviará uno, garantiza que el esquema de recuperación de mensajes perdidos mediante NAKs funciona correctamente, sin que la capa RSS se tenga que ocupar de enviar mensajes vacíos o de mantener timers.

La relación con el FD es otro de los motivos que llevaron a la no utilización de una capa de transporte standard, ya que en ese caso no se podrían utilizar las retransmisiones y avisos internos de TCP como señales de que un miembro continúa operacional.

Esta capa no necesita conocer la estructura del grupo y envía y recibe mensajes independientemente de la view en que se encuentra. Las capas superiores se comportan según la clase de miembro (miembro local, controlador o coordinador) a la que pertenezcan, y descartan los mensajes pertenecientes a views anteriores.



6.4.3 La capa FD

Failure detector o detector de fallas, es la capa que se ocupa de implementar un detector de fallas débil del estilo de los definidos por Chandra y Toueg []. Su función es enviar mensajes de control a ciertos miembros cada un tiempo dado (como forma de avisarles que el miembro al que pertenece continúa operacional) y verificar que le lleguen esos mismos mensajes de los miembros que está controlando. De detectar que no le llega algún mensaje de alguno de sus controlados durante un cierto tiempo, lo reporta a la capa superior como sospechado de falla. Utiliza para enviar sus mensajes la capa de transporte punto a punto confiable proporcionada por el RSS.

Cada FD tiene la responsabilidad de verificar a un conjunto de miembros y a su vez es verificado por otro conjunto. A los miembros que debe verificar se los conoce como verificados, ya a los que lo controlan a él, verificadores. De los primeros espera por lo menos un mensaje cada cierto tiempo $T1$. La capa Rss le avisa cada vez que llega un mensaje proveniente de algún miembro del grupo para que resetee el timer asociado a ese miembro. Comienza a correr así otro lapso $T1$ antes de darlo por fallado.

A cada miembro perteneciente a la lista de Verificadores le envía un mensaje "estoy vivo" cada cierto tiempo $T2$ salvo que el RSS le haya avisado de que el MSS acaba de enviar un mensaje a ese mismo destino (en cuyo caso, resetea el timer para esperar otro lapso $T2$ antes de enviar un mensaje).

Para evitar que la demora de un mensaje cause una falsa sospecha $T2$ es bastante menor a $T1$. Ver referencia en pruebas.

Los mensajes de control sólo se envían en ausencia de mensajes de datos con los mismos destinos. Por lo tanto la detección de fallas sólo tiene un costo en mensajes cuando el tráfico de datos es bajo.



Como ya vimos antes, dentro del modelo de view synchrony estructurado existen 3 tipos de miembros: los locales, los controladores y el coordinador. El modelo de detección de fallas que hemos elegido trata de respetar y aprovechar la arquitectura de red subyacente para no verificar fallas de miembros lejanos cuando no sea necesario con el fin de no sobrecargar a la red de mensajes innecesarios (ya que son mensajes de control, no de datos). Por ello se eligió un modelo en el cual:

- Cada miembro local verifica a su controlador y cada controlador verifica a todos sus miembros locales. Sin embargo, los miembros locales no se verifican entre sí.
- Cada controlador, además, verifica a los otros controladores y al coordinador de la view.
- El coordinador de la view verifica a los controladores.

Así se minimizan la cantidad de mensajes necesarios gracias a la estructura jerárquica de la view y no se pierde la propiedad de detectar la falla de sus miembros (o al menos la sospecha de falla).

La capa que utiliza estos servicios de detección de fallas es la VSS.

6.4.4 La capa MSS

Multicast Send Service o servicio de envío de Multicast. es la capa que implementa los distintos tipos de multicast confiables de acuerdo al protocolo y al orden de los mensajes. Ofrece a la capa superior (en este caso el VSS) los siguientes tipos de multicasts:

- FbcastDU: Multicast con orden fifo dinámicamente uniforme.
- FbcastNDU: Multicast con orden fifo no dinámicamente uniforme.
- CbcastDU: Multicast con orden causal dinámicamente uniforme.
- CbcastNDU: Multicast con orden causal no dinámicamente uniforme.
- AbcastDU: Multicast con orden agreed o consensuado dinámicamente uniforme
- AbcastNDU: Multicast con orden agreed o consensuado no dinámicamente uniforme



Además se agrega un nuevo tipo de multicast similar al FbcastNDU pero de control llamado ctrlFbcastNDU, porque necesitamos durante el protocolo de cambio de view distinguir a los mensajes del protocolo (mensajes de control) de cualquier mensaje de aplicación enviado con FbcastNDU.

Esta capa no tiene en cuenta si el miembro al que pertenece es un coordinador o no, pero se comporta de manera muy diferente si pertenece a un miembro local o a un controlador.

6.4.4.1 Componentes del Mss

El Mss está compuesto por: dos ProtocolManager, por un OrdenManager por cada tipo de orden y por un Filtrador - Clasificador de mensajes.

El Filtrador y Clasificador de mensajes se ocupa de leer los mensajes entregados por el Rss y clasificarlos de acuerdo al protocolo al que pertenecen para ubicarlos en la cola NDU o DU. Además realiza la función de filtrado de mensajes. Puede filtrar mensajes provenientes de una view anterior, de algún miembro fallado, etc.

Los Protocol Manager son los encargados de implementar los distintos tipos de multicast de manera de asegurar la propiedad de uniformidad dinámica o no en el envío de esos mensajes. Existen 2 protocol manager, uno que implementa el envío de mensajes dinámicamente uniformes (DU protocol manager) y otro de mensajes no dinámicamente uniformes (NDU protocol manager).

Los Orden Manager son los encargados de numerar los mensajes a enviar y de ordenar y almacenar los mensajes recibidos. Cuando uno de los multicast de un protocol manager alcanza la etapa "Listo para entregar" (ok_to_deliver: después de la primera etapa para el NDU y después de la segunda para el DU) es pasado al orden manager correspondiente de acuerdo al tipo de orden del mensaje. El orden manager entonces chequea si el mensaje puede ser entregado de acuerdo a este orden y si es así lo pasa a la cola de mensajes entregados. Por otra parte, cuando se va a comenzar el protocolo (DU o NDU) para enviar un multicast, el protocol manager correspondiente solicita al orden manager adecuado para el tipo de orden del mensaje, el id de orden que debe tener para que los receptores puedan ordenarlo (es el siguiente id definido en ese orden). Existen 2 Orden Manager que implementan los ordenes causal y fifo (Causal Order Manager y Fifo Order Manager respectivamente). Además, por la necesidad de contar



durante el protocolo de cambio de view con un servicio de envío de mensajes con orden fifo independiente, tenemos un tercer Order Manager, Control Fifo Order Manager que es un FiFo Orden Manager que solo se usa durante el protocolo de cambio de view.

Asi, por ejemplo, si el Vss desea enviar un mensaje con orden causal y dinámicamente uniforme invocará el metodo del Mss cbcasDU. El Mss entonces solicitará al CausalOrderManager un identificador de orden causal (el siguiente en el orden) y luego requerirá al DU Protocol Manager que envíe el mensaje en su nombre.

Como cada miembro de nuestra máquina global puede actuar como controlador o como miembro local, cada protocol manager sabe a que tipo de miembro pertenece y de acuerdo a esto funciona como un protocol manager de un controlador o de un miembro local (de acuerdo a structured view synchrony). La implementación de los metodos de modo miembro local constan de una sola fase: Se envia un mensaje al controlador para que se apropie del mensaje y lo envíe en su nombre y se espera un ACK antes de borrarlo. La implementación de los metodos de modo controlador, por otra parte, consta de 4 fases para el protocol manager dinámicamente uniforme: `save_message`, `ok_to_deliver`, `all_have_seen_it` y `garbage_collect` y 3 para el no dinámicamente uniforme (se unifican el `save_message` con el `ok_to_deliver`).

La división entre Order Managers y Protocolo Managers facilita el agregado de nuevos tipos de ordenes , ya que solo se debe implementar el Orden Manager correspondiente al orden que se quiere agregar y conectarlo con los Protocol Manager.

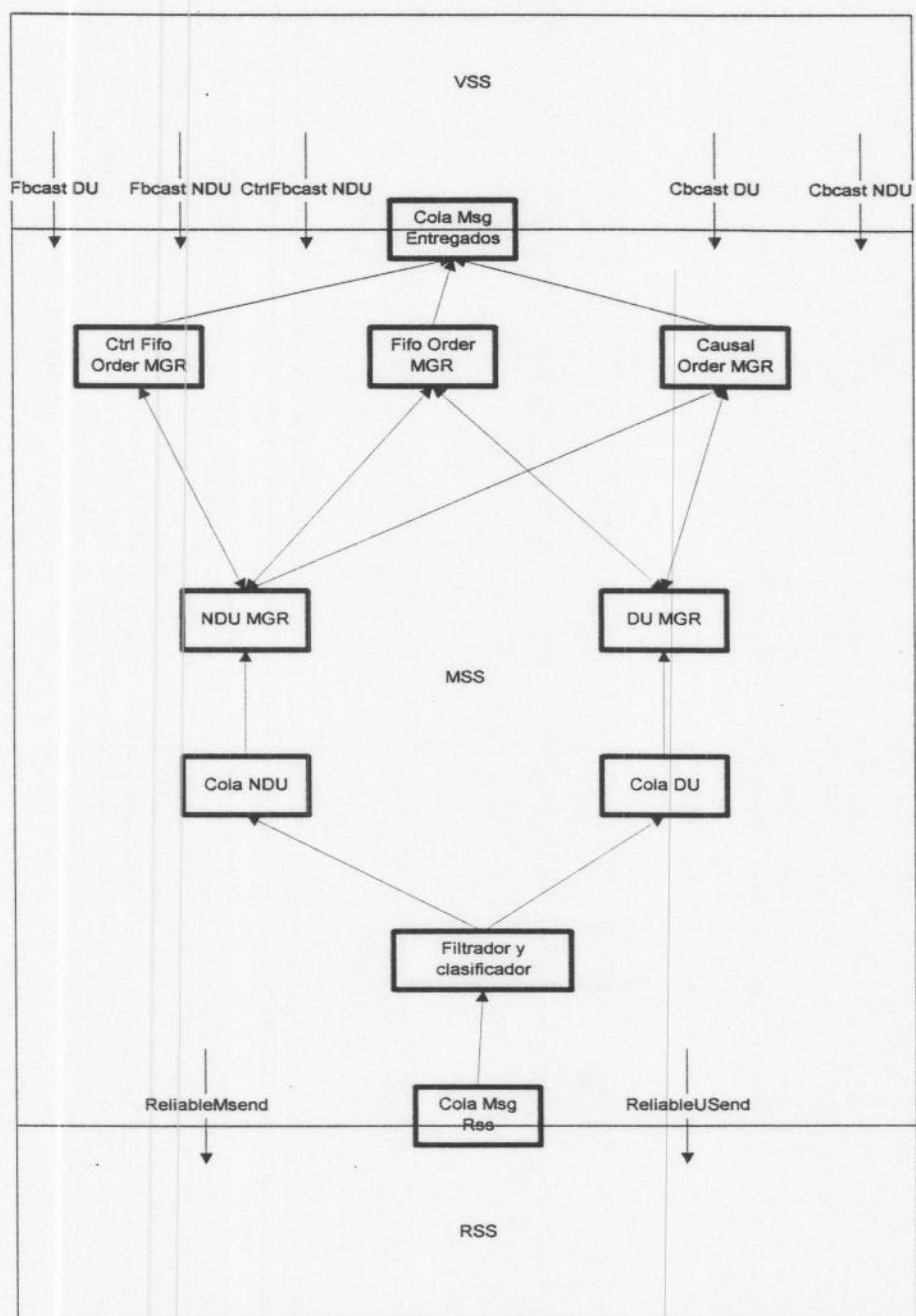




Fig 27 –Estructura del Mss

6.4.5 La capa VSS

6.4.5.1 Introducción

Vss (View Synchrony Service) es la capa que ofrece a la aplicación un servicio de view synchrony. Es la que mantiene la propiedad de view synchrony entre distintas views del sistema cuando alguno de sus miembros falla a través de la ejecución de un protocolo de cambio de view. Utiliza para ello los servicios provistos por las capas MSS y FD.

La capa Vss cumple una doble función : implementa el protocolo de cambio de view y da acceso a las capas superiores a los servicios de multicast. Su función principal es la implementación del cambio de view y casi no agrega funcionalidad a los servicios de multicast , excepto la de restringir el envío de mensajes durante el protocolo de cambio de view.

El comportamiento de esta capa depende del tipo de miembro al que pertenezca , ya que los miembros locales , los controladores y el coordinador deben realizar tareas distintas durante el protocolo de cambio de view. Su estructura es bastante similar en los miembros locales y en los controladores pero difiere un poco en el coordinador. Esto es así porque el coordinador debe cumplir también con las funciones de controlador .

6.4.5.2 Componentes del Vss

- **FiltradorClasificadorMensajes** : este componente se ocupa de tomar los mensajes provenientes de las capas inferiores y según sean mensajes pertenecientes a la aplicación o mensajes del protocolo de cambio de view los entrega a la capa superior o los pasa al VssEventTranslator . También se ocupa de desechar los mensajes pertenecientes a views anteriores
- **VssEventTranslator** : la función de este componente es la de homogeneizar los distintos eventos que se producen durante un cambio de view para que los manejadores del protocolo de cambio de view puedan manejarlos uniformemente.
- **VssProtocoloManager** : es el componente que implementa la lógica del algoritmo de cambio de view. Según el tipo de miembro debe comportarse de



distinta manera por lo que existen tres subtipos de este componente : **MIVssProtocolMgr** (en los miembros locales) , **CtrlVssProtocolMgr** (en los controladores) y **CoordVssProtocolMgr** (en el coordinador). Como el coordinador es a la vez un controlador , posee un CtrlVssProtocolMgr y un CoordVssProtocolMgr al mismo tiempo. Estos componentes se comunican como si estuvieran en miembros distintos por lo que para la lógica del protocolo de cambio de view el coordinador es un miembro del grupo que solo cumple esa función (por ejemplo cuando envía una nueva view no debe ocuparse de reenviarla a los miembros locales porque esa función la cumple el controlador que se encuentra en el mismo miembro).

El View Synchrony Protocol Manager implementa una máquina de estado que se ocupa de llevar a cabo el protocolo de cambio de view. Para eso a partir del estado que mantiene y los eventos que recibe del traductor de eventos , ejecuta el protocolo de la sección 6.4.5.3

En el caso que la capa pertenezca a un miembro coordinador (que es siempre también un controlador de su subgrupo local) , la capa tendrá dos view synchrony protocol manager , uno para el rol de controlador y otro para el rol de coordinador. Cuando sea necesario los dos protocol manager se comunicarán a través del Mss , de la misma manera que si estuvieran en dos máquinas distintas.

6.4.5.3 Algoritmo de cambio de view

6.4.5.3.1 Introducción

El algoritmo de cambio de view que hemos diseñado se puede modelar como una máquina de estados , en la que los eventos que producen cambios de estado pueden ser mensajes de otros miembros del grupo , avisos de la capa de detección de fallas (FD) sobre la caída de algún miembro o pedidos de miembros nuevos para pasar a formar parte del grupo. En la descripción del protocolo no distinguiremos entre la caída de un miembro y el pedido de un miembro nuevo para ser incorporado al grupo , porque son esencialmente el mismo evento : un cambio en la percepción del grupo (con la salvedad , que no es relevante en este nivel de abstracción , de que los pedidos de incorporación se podrían manejar con otra política para disminuir la cantidad de cambios de view).

En el protocolo intervienen tres tipos diferentes de actores : los miembros locales, los controladores y un controlador distinguido que llamaremos coordinador. La máquina de estados de cada uno de ellos es diferente. Cabe aclarar que aunque



en realidad el miembro que cumple el rol de coordinador también cumple el rol de controlador de un subgrupo, dichos roles se manejan en forma separada, lo que facilita la descripción de los algoritmos.

Para comprender porque ciertos eventos no se producen en algunos actores del protocolo, será útil recordar el esquema de verificación que se realiza en nuestro modelo de Structured View Synchrony (Ver 6.4.3). Cada miembro verifica a y es verificado por otros miembros, según su tipo:

Tipo	Verifica a	Es verificado por
Miembro Local	Su controlador	Su controlador
Controlador	Los miembros locales de su subgrupo y a los otros controladores	Los miembros locales de su subgrupo y a los otros controladores
Coordinador	Nadie	Los controladores

Así se ve por ejemplo que un miembro local no puede recibir un evento de detección de falla de otro miembro local de su grupo.

Otro punto importante a aclarar es que durante la ejecución del protocolo de cambio de view no se permite a las aplicaciones iniciar nuevos mensajes.

6.4.5.3.2 Protocolo de cambio de view en un miembro local

Inicialmente cada miembro local se encuentra en el estado que llamamos normal. Si estando en ese estado recibe un pedido de otro proceso para pasar a ser miembro del grupo notifica ese evento a su controlador y espera a que se inicie el protocolo de cambio de view, manteniéndose en el mismo estado. Cuando recibe el mensaje de su controlador que le indica que se inicia el algoritmo de cambio de view responde enviando su percepción del grupo (membership perception, **MP**) y los mensajes inestables (**MI**) que conoce. y pasa al estado siguiente, en donde espera la llegada de la nueva view y de los mensajes inestables de todo el sistema.

Si mientras espera la llegada de la nueva view recibe otro pedido de un proceso para ser parte del grupo, reenvía su **MP** al controlador. Cuando veamos el protocolo de cambio de view en el controlador veremos que hace con los distintos **MP** recibidos. Cuando recibe la nueva view y los **MI**, procesa los mensajes inestables, envía un asentimiento al controlador y pasa al tercer y ultimo estado

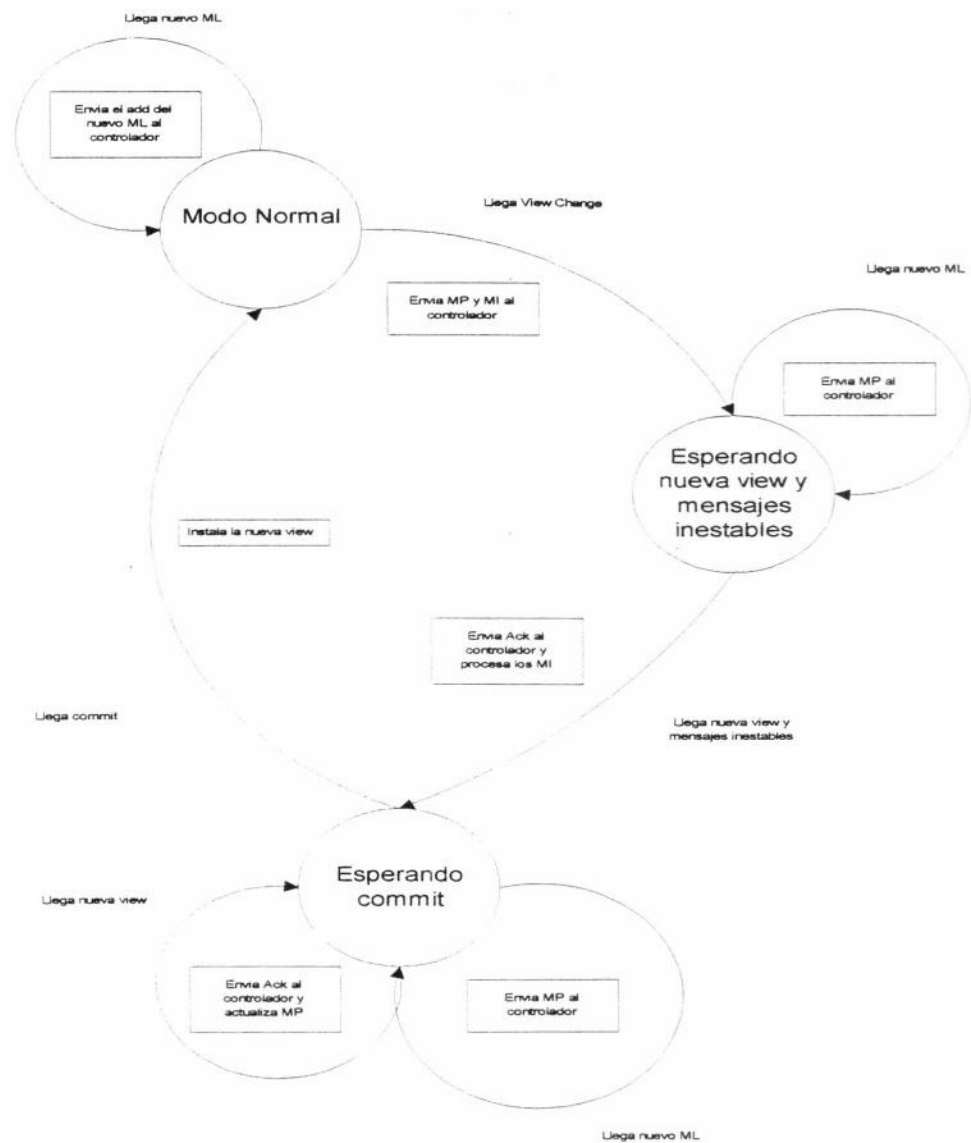


Fig 28 --Protocolo de cambio de view en un miembro local



También puede suceder que el miembro local detecte que su controlador a fallado. Cuando esto sucede , automáticamente todos los miembros locales pasan a considerar como su controlador al miembro local más antiguo del grupo (como todos comparten la misma percepción original del grupo esto se puede hacer sin necesidad de mensajes) y el miembro local que se considera a si mismo el más antiguo del grupo envía un mensaje al coordinador notificándole que el es el nuevo controlador del grupo. En realidad el resultado final depende de que el subgrupo sea lo suficientemente rápido en reorganizarse , porque si los otros controladores no reciben mensajes del subgrupo durante un tiempo dado , daran por caído al subgrupo entero. Dado el modelo sobre el que nos basamos (cada grupo en una LAN, conectadas entre si vía Internet) , las detecciones de fallas dentro de cada LAN deberían ser mucho más rápidas que entre LANs distintas.

6.4.5.3.3 Protocolo de cambio de view en un controlador

A continuación explicaremos el algoritmo de cambio de view en un miembro que cumple el rol de coordinador. Como desde el punto de vista del protocolo son eventos indistinguibles , no diferenciaremos el pedido de un proceso para pasar a ser miembro del grupo de la detección de la falla de un miembro

Inicialmente el controlador esta en un estado normal. Si detecta la caída de un miembro , envía un mensaje al coordinador para que este inicie el protocolo de cambio de view y se mantiene en el mismo estado. Si recibe un mensaje de comienzo de cambio de view , lo reenvía a sus miembros locales y espera a que todos ellos le envíen su percepción del grupo (**Membership perception** , **MP**) y sus mensajes inestables (**MI**).

Si mientras está esperando la llegada del **MP** y los **MI** de los miembros locales detecta la caída de algún otro miembro del grupo , actualiza su percepción del grupo y continúa en el mismo estado. Cuando finalmente recibe el **MP** y **MI** de todos los miembros locales combina la percepción del grupo y los mensajes inestables de todos los miembros locales con las suyas , envía un mensaje con estos datos al coordinador y pasa a esperar la nueva view.

Mientras está esperando la nueva view puede llegar a detectar la caída de otro miembro del grupo. En ese caso reenvía su **MP** al coordinador y se mantiene en el mismo estado. Cuando finalmente llega la nueva view junto con los mensajes inestables , reenvía a ambos sus miembros locales , procesa los mensajes inestables y espera a que todos ellos asientan la recepción de la nueva view.



En el intervalo hasta recibir los asentimientos de la nueva view puede suceder que el controlador detecte la caída de algún otro miembro. En ese caso actualiza su **MP** y se lo reenvía al coordinador. También puede suceder que el coordinador sea notificado de alguna modificación en el grupo y que envíe una nueva view. En ese caso el controlador reenvía la nueva view a sus miembros locales y espera que todos ellos asientan la recepción de la nueva view (obviamente no cuentan los asentimientos de la view anterior). Cuando finalmente llegan todos los asentimientos a la última view enviada el controlador reenvía el asentimiento de la view al coordinador y pasa a esperar el commit de la view.

Si mientras el coordinador está esperando el commit de la view detecta la caída de otro miembro local le envía la novedad al coordinador. Si este mensaje llega antes de que el coordinador envíe el commit, la caída del miembro local se notificará durante el protocolo de cambio de view actual. Si el mensaje llega después del envío del commit, se producirá una nueva ejecución del protocolo de cambio de view. También puede suceder que llegue una nueva view desde el coordinador (esto se produce por la caída de algún miembro durante el protocolo de cambio de view), en cuyo caso el controlador vuelve al estado anterior.

Cuando llega el commit, el controlador lo reenvía a sus miembros locales e instala la nueva view.

Si un controlador A detecta la caída del coordinador comienza una etapa de elección de un nuevo coordinador. El controlador que detectó la falla anuncia al resto de los controladores la falla del coordinador. Los controladores que reciben el anuncio envían un ack al controlador A. Cuando este recibe asentimientos de la mayoría de los controladores del grupo se elige como coordinador al controlador más antiguo de la view y se continúa con el protocolo normal de cambio de view.

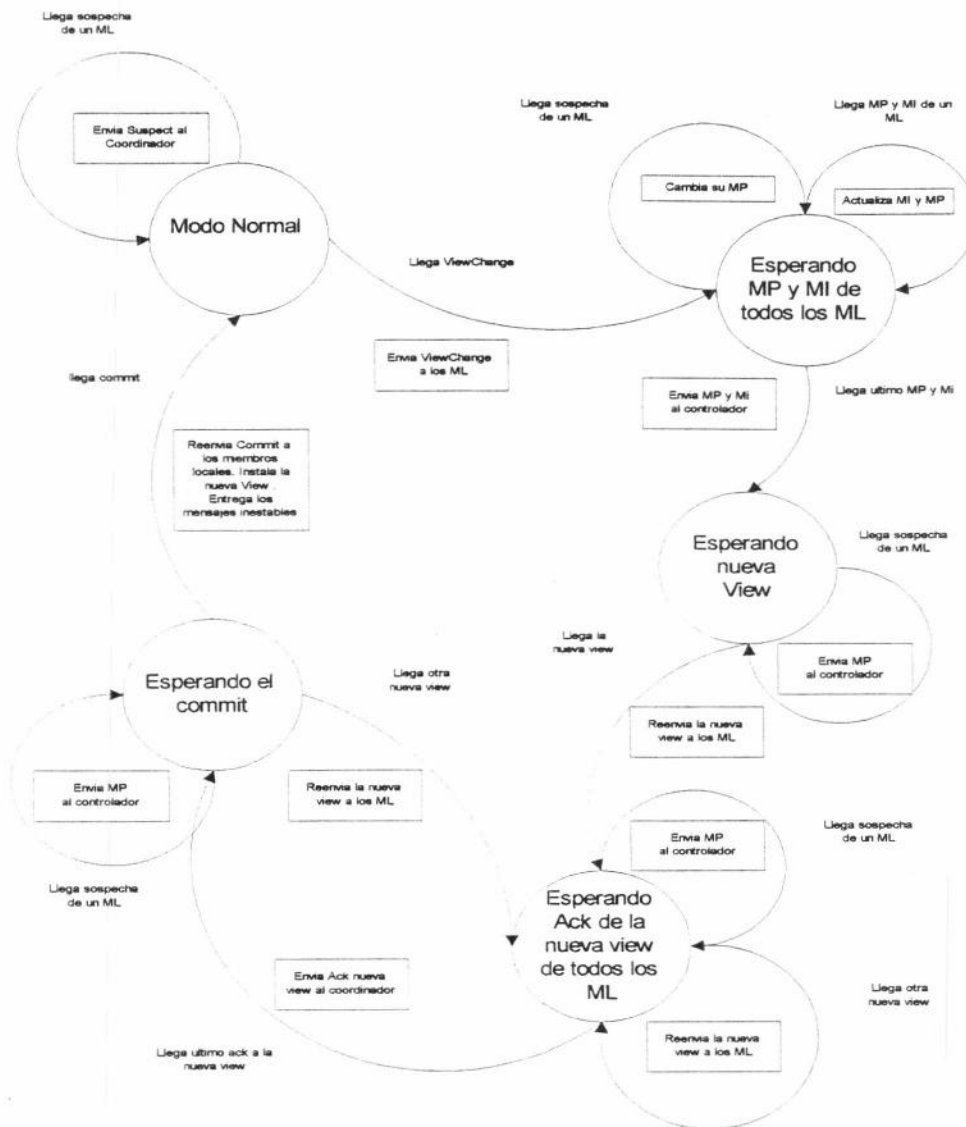


Fig 29 –Protocolo de cambio de view en un miembro controlador

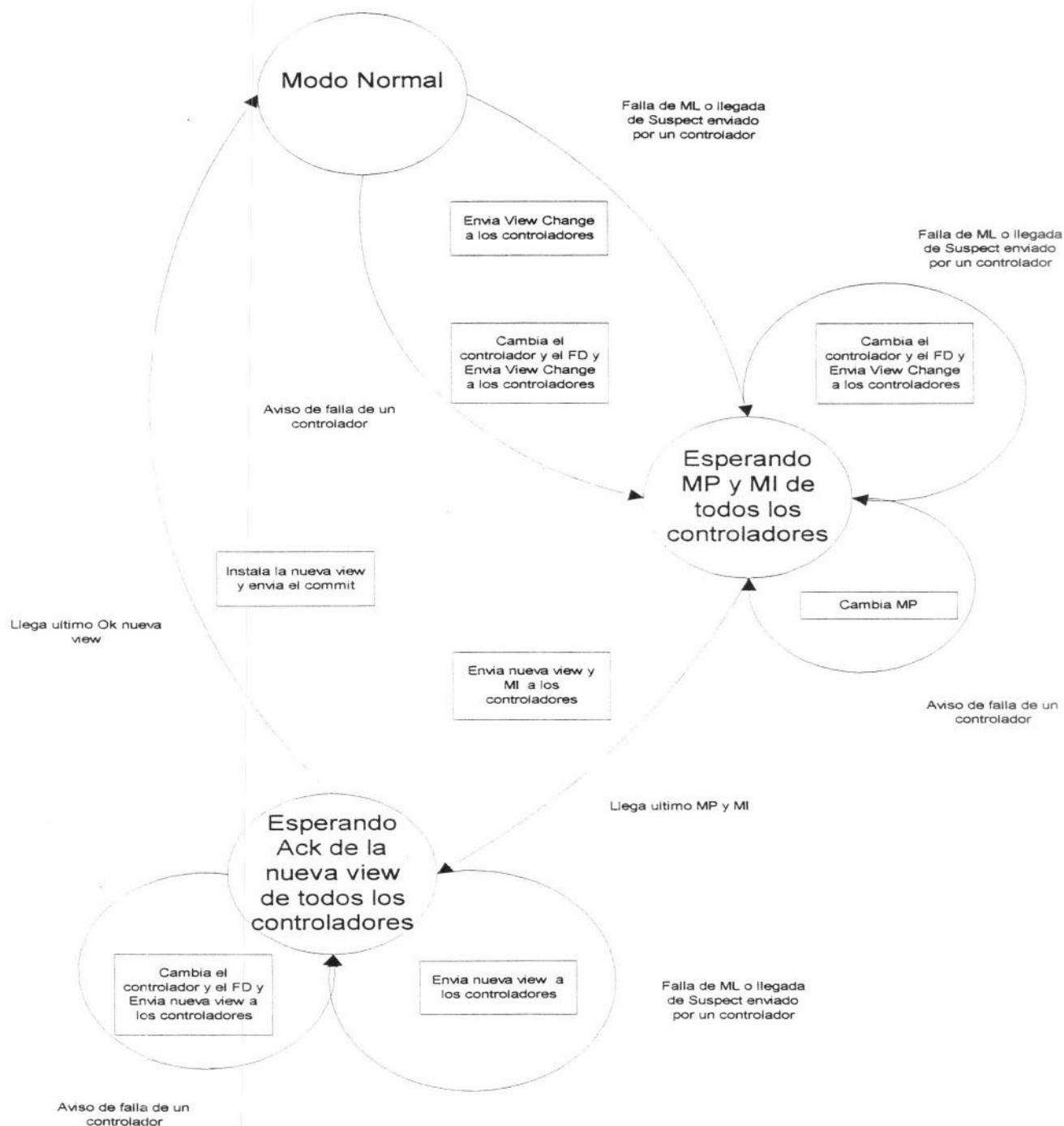


6.4.5.3.4 Protocolo de cambio de view en un Coordinador

Como el coordinador no verifica a ningún miembro del grupo , el comienzo del protocolo de cambio de view siempre es disparado por la llegada de un mensaje de un controlador indicando la falla de algún miembro . Cuando el coordinador recibe este mensaje envía un mensaje anunciando que comienza el protocolo de cambio de view y espera a que todos los controladores le envíen la percepción del grupo (Membership perception , **MP**) y los mensajes inestables (**MI**) de sus respectivos subgrupos.

Cuando llegan el **MP** y los **MI** de todos los controladores , el coordinador arma la nueva view y la lista de mensajes inestables , la envía a los controladores y espera a que todos los controladores asientan la llegada de la nueva view. Si antes de que se completen los asentimientos llegara un mensaje modificando la percepción de la view , se envía un nuevo mensaje conteniendo la nueva view y se espera asentimientos de todos los controladores a esta nueva composición del grupo.

Cuando se reciben asentimientos de todos los controladores a la última view propuesta , el coordinador envía el mensaje de commit e instala la nueva view .





7 Framework de aplicación

7.1 Introducción

Con el fin de proveer un marco adecuado para el desarrollo de aplicaciones distribuidas confiables utilizando la abstracción de view synchrony estructurado hemos desarrollado un framework de aplicación. Dicho framework permite acceder a las primitivas de nuestra arquitectura para enviar los distintos tipos de multicasts, recibir peticiones de clientes y eventos de view synchrony de una manera simple, estructurada y poderosa. Utilizando estas facilidades se simplifica grandemente el desarrollo de servicios replicados distribuidos tolerantes a fallas.

7.2 Framework

Veamos a continuación el esquema de utilización de nuestra arquitectura y como se relaciona ésta con el framework de aplicación.

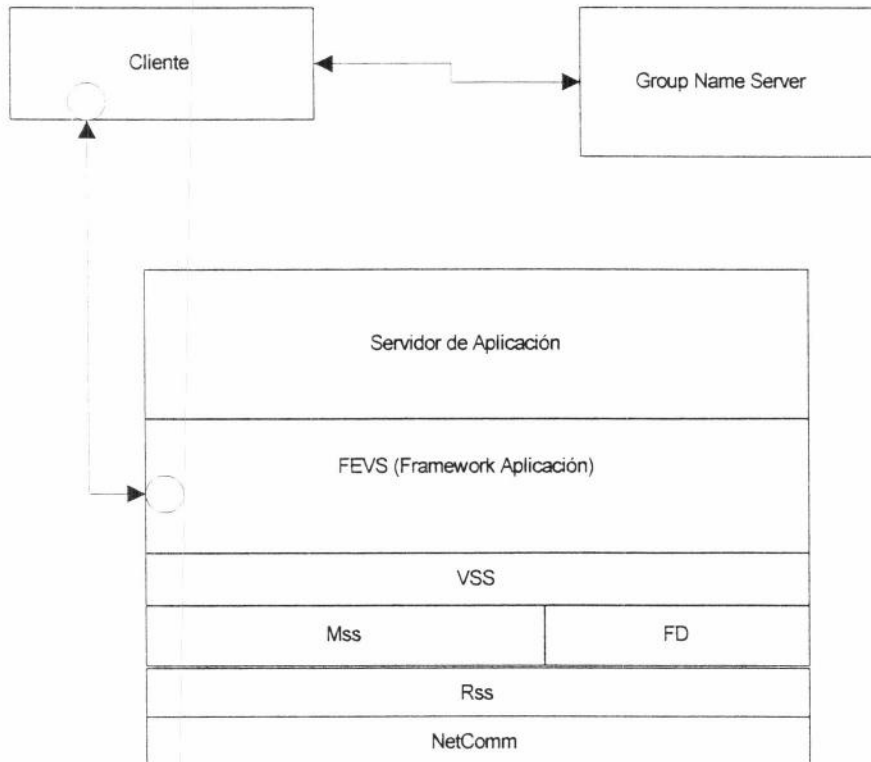


Fig 30 –Framework de aplicación



El framework **FEVS** (framework para structured view synchrony) se sitúa entre la arquitectura de capas de nuestra solución y el código del servidor de aplicación que implementa la lógica específica al servicio replicado distribuido que se está desarrollando. Existe también un framework para uso de los clientes de las aplicaciones, que llamaremos **FEVS Client** (FevsClient) que proporciona a los clientes un entorno para localizar, conectarse y realizar pedidos a los miembros del grupo.

Cuando un cliente del servicio replicado quiere realizar algún requerimiento se comunica primero con un GNS (group name server) que se encuentra en una dirección de red bien conocida en un port conocido. El GNS se encarga de mantener la información de composición de cada grupo de servidores que se haya registrado con él a la manera de un DNS. Así mantiene para cada grupo su nombre y una lista de los miembros (la view) junto con sus direcciones de red. Notar que si bien el GNS es un punto de corte en la confiabilidad del sistema, se puede adoptar medidas para disminuir este afecto a través de un esquema de redundancia. En este caso si el GNS falla el cliente se comunica con otro GNS que esté disponible.

De esta manera el cliente obtiene a través del GNS al conjunto de miembros (la view) del grupo de procesos que implementa el servicio al cual quiere realizar un requerimiento.

El cliente luego elige a uno de los integrantes de la view (presumiblemente a algún miembro cercano o que forme parte de su misma LAN) y le envía el requerimiento a un port bien conocido. En este port se encuentra escuchando el FEVS que registra la solicitud y se encarga de que el servicio de aplicación le de respuesta. Por último el framework rutea la respuesta de vuelta al cliente que la solicitó.

Una alternativa que puede utilizar el cliente es enviar el requerimiento a la view total. En este caso el framework se encarga de enviar el requerimiento uno a uno a los miembros de la view. Si alguno no contesta en un tiempo determinado, la reenvía al siguiente hasta que o bien se obtiene una respuesta o bien se acaban los miembros y da un error.

El FEVS mantiene además una lista de todos los requerimientos de clientes que se han recibido en el grupo y de la respuesta que se obtuvo. Si un cliente ha realizado una solicitud a un miembro y éste falla, el cliente puede repetir la misma solicitud (utilizando para ello el mismo request-id enviado, único para cada miembro) a otro miembro. El FEVS se encarga entonces de reenviar la respuesta al cliente, ahora desde otro miembro del grupo.



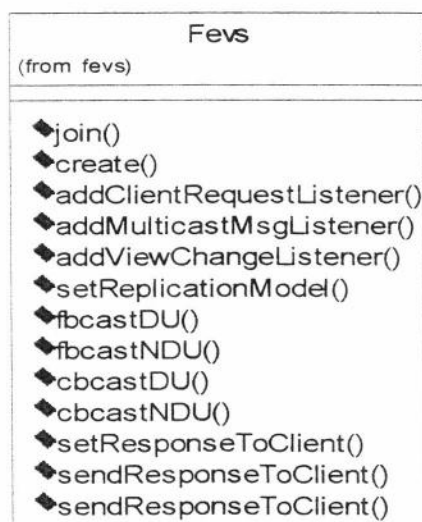
La función primordial del FEVS es enviar al proceso servidor de aplicación (construido por encima de él) los distintos eventos que se generan en las capas inferiores de la arquitectura o en el cliente y que son relevantes para la aplicación. Estos eventos se clasifican en tres tipos :

- Pedidos del cliente (***ClientRequestEvent***)
- Mensajes provenientes de otros servidores (***MulticastMsgEvent***)
- Cambios en la composición del grupo de servidores (***ViewChangeEvent***)

El modelo del framework permite al proceso servidor de aplicación especificar objetos que atienden a cada uno de los tipos de eventos antes mencionados. Se Proveen objetos que implementan la atención de dichos eventos de acuerdo a modelos de replicación ya existentes (proveen un comportamiento por defecto).

Existen dos modelos principales de replicación : *Primary Backup* y *State Machine* ; en el primero solo el servidor que recibe el pedido del cliente es el que le contesta, mientras que en el segundo todos los servidores contestan al cliente , que se queda con la primera respuesta que recibe. El FEVS contiene el comportamiento necesario para implementar aplicaciones basadas en ambos modelos . El servidor de aplicación puede elegir uno de estos modelos o implementar su propio modelo de replicación.

7.3 Interfaz del framework Servidor de Aplicación





Primitivas de administración de grupos

Join(groupName : String, subgroupId : String, miembro : MemberId) : este método se utiliza para unirse al grupo de servidores de aplicación cuyo nombre es **groupName**. El servidor pasará a formar parte del subgrupo identificado por **subgroupId** (si ese subgrupo no existe será creado, pasando el servidor a ser el controlador del subgrupo). Este método solicita al **GNS** la lista de miembros del grupo de servidores y envía a alguno de ellos un mensaje de **AddMember** solicitando el ingreso al grupo.

Create(groupName : String) : Crea un grupo de servidores identificado por el nombre **groupName**. El servidor que invoca el método se transforma en el coordinador del grupo. Este método se comunica con el **GNS**, en el que registra al nuevo grupo.

Manejo de Eventos

AddClientRequestListener(listener : ClientRequestListener) : El método registra al objeto de la aplicación que se encargará de atender los pedidos de los clientes.

AddMulticastMsgListener (listener : MulticastMsgListener) : Este método registra al objeto de la aplicación que se encargará de atender los mensajes provenientes de otros miembros del grupo.

AddViewChangeListener (listener : ViewChangeListener) : Se registra al objeto de la aplicación que se encargará de atender el evento enviado por las capas inferiores cuando se producen cambios en la composición del grupo. Ese objeto tendrá, entre otras, la tarea de inicializar el estado de los nuevos miembros que se agregan a la view (transferencia de estado).

SetReplicationModel(model : ReplicationModel) : Permite a la aplicación elegir el modelo en base al cual se enviarán las respuestas a los pedidos de los clientes. El modelo puede ser **PrimaryBackup**, **StateMachine** o **Free_Model** (en este caso la aplicación debe encargarse de manejar el modelo de las respuestas).

Primitivas de comunicación con los miembros del grupo



FbcastDU,cbcastDU, FbcastNDU,cbcastNDU : Permiten acceder a los métodos de la capa Vss para enviar distintos tipos de multicasts a los miembros del grupo. El tipo de multicast elegido dependerá de las necesidades de la aplicación que se esté desarrollando e impacta sobre la performance de la misma (de acuerdo al orden y a la propiedad de uniformidad dinámica que se elija).

Primitivas de manejo de requerimientos de clientes

addClientRequest(req: ClientRequest): Agrega a la lista de solicitudes de clientes el nuevo requerimiento req.

existsClientRequest(idRequest: ClientRequestId): Devuelve verdadero si ya se ha registrado con anterioridad una solicitud de cliente con el identificador idRequest).

deleteClientRequest(idRequest: ClientRequestId): Remueve toda la información asociada a la solicitud de un cliente identificada por idRequest.

deleteOlderThan(timeStamp: Date): Remueve todas las solicitudes de un cliente registradas con anterioridad al tiempo timeStamp.

setResponseToClient(idRequest : ClientRequestId, respuesta : serializable) : Almacena la respuesta al pedido de cliente identificado por idRequest para ser enviada posteriormente de acuerdo al modelo de replicación. Si el miembro es el dueño del requerimiento, es enviada a continuación con el método sendResponse. También es enviada si el cliente reenvía el pedido porque la aplicación a quien se la solicitó en primer lugar no contestó.

sendResponseToClient(idRequest : ClientRequestId, respuesta : serializable) : Especifica la respuesta al pedido de cliente identificado por idRequest. Dependiendo del tipo de modelo de replicación elegido el FEVS envía realmente la respuesta al cliente ó sólo la almacena para enviarla con posterioridad.

sendResponseToClient(idRequest : ClientRequestId) : Envía la respuesta al pedido del cliente identificado por idRequest.



7.4 Esquema de uso del framework para un proceso servidor

Class ServidorAplicacion {

```
Public ServidorAplicacionEcho()
{
    Fevs frame = new Fevs ();
    // especifica los objetos que manejaran los eventos del framework
    frame.addClientRequestListener(new MyClientRequestListener());
    frame.addMulticastMsgListener (new MyMulticastMsgListener())
    frame.addViewChangeListener (new MyViewChangeListener())

    // el servidor intenta conectarse a un grupo. Si no existe lo crea
    if(!frame.join("grupo servidores echo",lan1Id,memberId))
        create("grupo servidores echo");
}

// Clase que maneja los pedidos del cliente. La respuesta a cualquier pedido del cliente
// es devolverle lo mismo que se recibio
Class MyClientRequestListener Implements ClientRequestListener {

    // Constructor de la clase que maneja los pedidos del cliente.
    Public MyClientRequestListener(Fevs frame)
    {
        this.frame = frame;
    }

    Public void processClientRequest(ClientRequestEvent evt)
    {
        // setea como respuesta al pedido el mismo mensaje que recibió y envía la respuesta
        frame.setResponseToClient(crEvent.getRequestId(),evt.getClientMsg());
        frame.sendResponseToClient(crEvent.getRequestId());
    }
}
```



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

```
// Clase que maneja los cambios en la composicion del grupo (para un servidor simple
// como este no es necesario tenerlos en cuenta)
Class MyViewChangeListener Implements ViewChangeListener {

    public void processViewChangeEvent(fevs.ViewChangeEvent vcEvent) {
        System.out.println("Se ha producido el temido cambio de view");
    }
}
```

7.5 Interfaz del framework para un proceso cliente

FevsClient	
(from fevs)	
◆getServerView(groupName : String) : return ◆nextRequestId() : ClientRequestId ◆sendRequest(id : ClientRequestId, msg : Message, server : MemberId) : return ◆sendRequest(id : ClientRequestId, msg : Message) : return ◆getServerView(groupName : String) : View	

getServerView(groupName : String) : obtiene la lista de miembros del grupo identificado por el nombre groupName. Se comunica con el **GNS** , que le devuelve dicha lista de miembros. Los pedidos posteriores del cliente se hacen a alguno de los servidores que devuelve este método.

nextRequestId() : obtiene un identificador único de pedido para el cliente que lo invoca. Con este identificador se realiza la solicitud a un servidor .

sendRequest(id: ClientRequestId, msg Message ,server MemberId) :Object
Envía el pedido **msg** identificado por **id** a un servidor **server** perteneciente a la **view**. En caso de que el servidor falle y no se obtenga una respuesta, el pedido puede volver a realizarse con el mismo **id** a otro miembro, con la garantía de que la operación se realizará a lo suma una sola vez. El proceso puede quedar bloqueado esperando la respuesta o especificar un objeto que procesará la respuesta cuando llegue.



sendRequest(id: ClientRequestId, msg Message) :Object Envía el pedido **msg** identificado por **id** al primer miembro de la view. Si falla o el servidor no responde despues de un lapso determinado de tiempo, el pedido es reenviado a los otros servidores de la view hasta que o bien todos fallen o alguno devuelva la respuesta.

7.6 Esquema de uso del framework para un proceso cliente

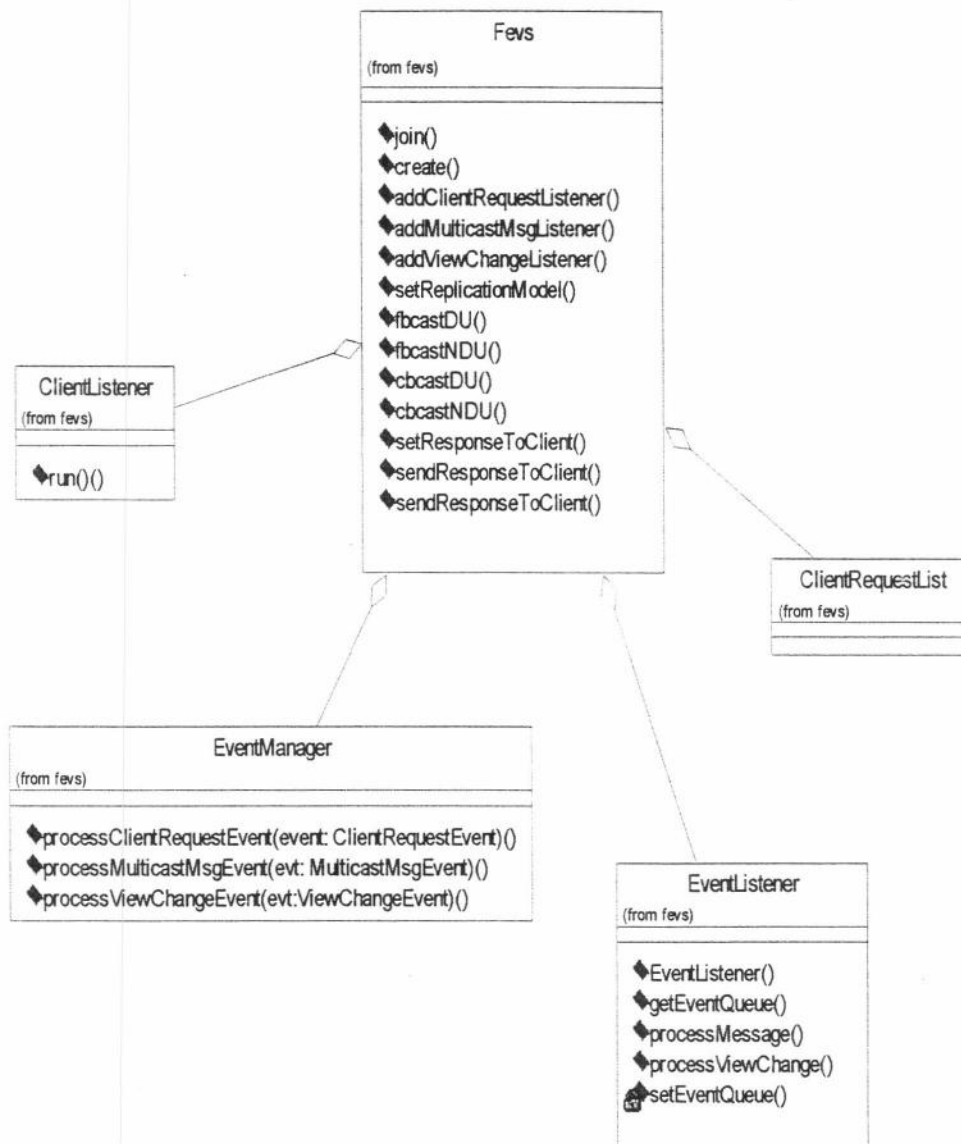
Class ClienteAplicacion implements Runneable{

```
Public run()
{
    FevsClient frame = new FevsClient(GNSDIR) ;
    // obtiene los miembros del grupo de servidores a los que les hará pedidos
    MemberId []servidores = frame.GetGroupServers("grupo servidores");

    While(true)
    {
        // obtiene un id unico y hace un pedido a uno de los servidores de la lista
        RequestId id = frame.GetRequestId();
        Object response = SendRequest(new MsgApp("pedido de
prueba"),id,servidores[0])
    }
}
```



7.7 Diagrama de clases del framework.





8 Implementación del Framework

8.1 Introducción

Describiremos ahora la arquitectura de implementación del framework de utilización de view synchrony estructurado, tanto el del servidor, **Fevs** como el del cliente **FevsClient**.

8.2 Composición del Fevs (proceso servidor)

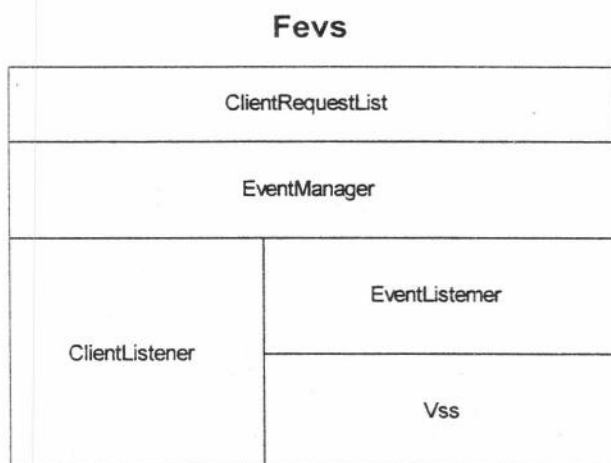


Fig 31 –Composición del Servidor FEVS

Vemos así que el Fevs está compuesto por:

Vss: Objeto que se encarga de mantener la propiedad de View Synchrony entre el grupo de procesos servidores aún ante fallas de los miembros. Ejecuta el protocolo de cambio de view e implementa las primitivas de multicasts (fbcastDU, fbcastDNU, etc.) que el Fevs exporta para ser utilizadas por la capa de aplicación.



EventListener: Transforma la actividad del objeto Vss en eventos que pueden ser procesados por el manejador de eventos del Fevs (EventManager). Principalmente transforma los multicast enviados a la aplicación servidor en MulticastMsgEvent y el aviso de un cambio de view en un ViewChangeEvent.

ClientListener: Escucha solicitudes de servicios enviados a este servidor por los clientes (a través del FevsClient a un puerto TCP). Transforma esas solicitudes en eventos clientRequestEvent para que sean procesados por el eventManager.

EventManager: Es el que contiene la lógica de atención y despacho de los eventos recibidos MulticastMsgEvent, ViewChangeEvent y ClientRequestEvent. Recibe de la aplicación a los objetos que están interesados en recibir esos eventos cuando se produzcan (MulticastMsgListener, ViewChangeListener y el ClientRequestListener. Invoca a esos objetos al producirse alguno de los eventos. Además maneja el modelo de replicación, así, si por ejemplo se recibe un requerimiento de un cliente y el modelo de replicación es primary-backup lo reenvía automáticamente a los otros servidores del grupo.

ClientRequestList : Lista de manejo de requerimientos de clientes (ClientRequest), contienen las solicitudes originales, el identificador del pedido (ClientRequestId), si el pedido fue originalmente enviado a este servidor (isOwner) y la respuesta que se ha de devolver.

El Fevs recibe las solicitudes de un cliente a través del ClientListener, se almacena la solicitud en la ClientRequestList y se la pasa al objeto ClientRequestListener del eventManager (objeto de la aplicación encargado de procesar el evento ClientRequestEvent). Este puede decidir contestarla directamente (si es una consulta que no modifica el estado) o enviarla como un multicast a los otros procesos servidores (utilizando para ello las primitivas proporcionadas por el Vss). Cuando se obtiene una respuesta, esta se almacena también en la ClientRequestList para ser enviada al cliente (si corresponde porque él es el dueño del requerimiento o porque el cliente le ha reenviado el pedido a él).

Si el servidor ha reenviado el requerimiento de un cliente con un multicast (porque implicaba una modificación del estado compartido) este llega a través del Vss-EventListener al EventManager que avisa al objeto de la aplicación MulticastMsgListener que es el encargado de procesar el requerimiento y eventualmente enviar la respuesta al cliente.

En cualquier momento si uno de los procesos servidores falla, el Vss producirá un cambio de view. Luego de ejecutado el protocolo de cambio de view se avisará del evento al EventManager que se encargará de invocar al objeto de la aplicación



ViewChangeListener que debe consensuar el estado de la aplicación con los nuevos integrantes de la view

8.3 Composición del FevsClient (cliente)

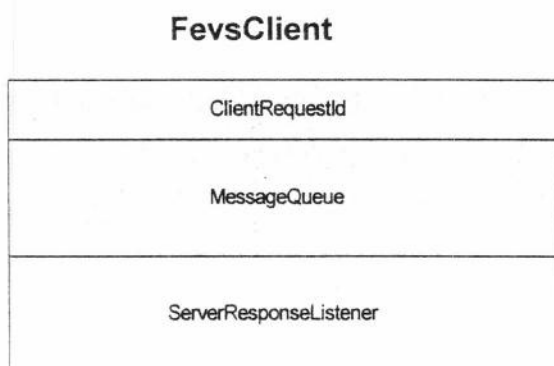


Fig 32 –Composición del Cliente FEVS

El FevsClient, como explicamos anteriormente, se encarga de obtener a la lista de miembros del grupo, de armar y enviar los requerimientos y de obtener la respuesta. Para esto, envía los requerimientos a través de TCP/IP al puerto de servicio (servicePort) especificado para cada miembro del grupo (información que es obtenida de la view del grupo). Una vez enviado se bloquea esperando la respuesta (que espera leyendo en la MessageQueue). Esta compuesto por:

ServerResponseListener: Se encarga de escuchar en un port TCP/IP (responsePort) la respuesta enviada por alguno de los servidores a los que le envió el requerimiento. Una vez llegada la encola en la MessageQueue. Esto permite tanto una solicitud de requerimientos sincrónica bloqueante (adonde el cliente quede bloqueado hasta que llegue la respuesta) como una no bloqueante adonde el cliente simplemente especifique un objeto que atenderà la respuesta cuando llegue.

ServerResponseListener: Almacena las respuestas que le han llegado al ServerResponseListener para ser leídas por el FevsClient.

ClientRequestId: Es el último requestId enviado a un proceso servidor. Se incrementa cada vez que se envía un requerimiento para que cada solicitud sea única. Esto permite que si el pedido se reenvía a otro proceso servidor porque el



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

primero no contestó que el segundo devuelva la respuesta sin reprocesar el pedido (lo que causaría que la operación no fuera idempotente).



9 Pruebas

9.1 Prueba 1 - Tiempo de cambio de view

El objetivo de esta prueba es comparar el tiempo que lleva realizar un cambio de view en un esquema de View Synchrony (VS) y en View Synchrony estructurado (SVS).

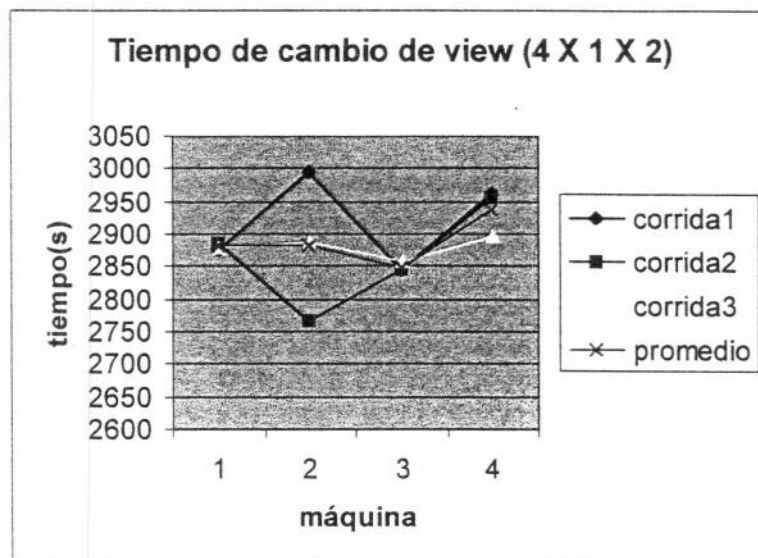
Se probarán cambios de view con distinta cantidad de nodos (con 4 cantidades distintas, n_1 , n_2 , n_3 y n_4) y con mensajes inestables durante el protocolo de cambio de view.

9.1.1 Cambio de view en Structured view synchrony

9.1.1.1 Configuración : 4 Maq X 1 Miembros X 2 grupos

Total Miembros: 4

miembro numero	tiempo para view change			promedio
	corrida1	corrida2	corrida3	
1	2877	2886	2879	2880.66667
2	2996	2765	2887	2882.66667
3	2845	2844	2856	2848.33333
4	2960	2956	2896	2937.33333
cambio de view				2887.25000



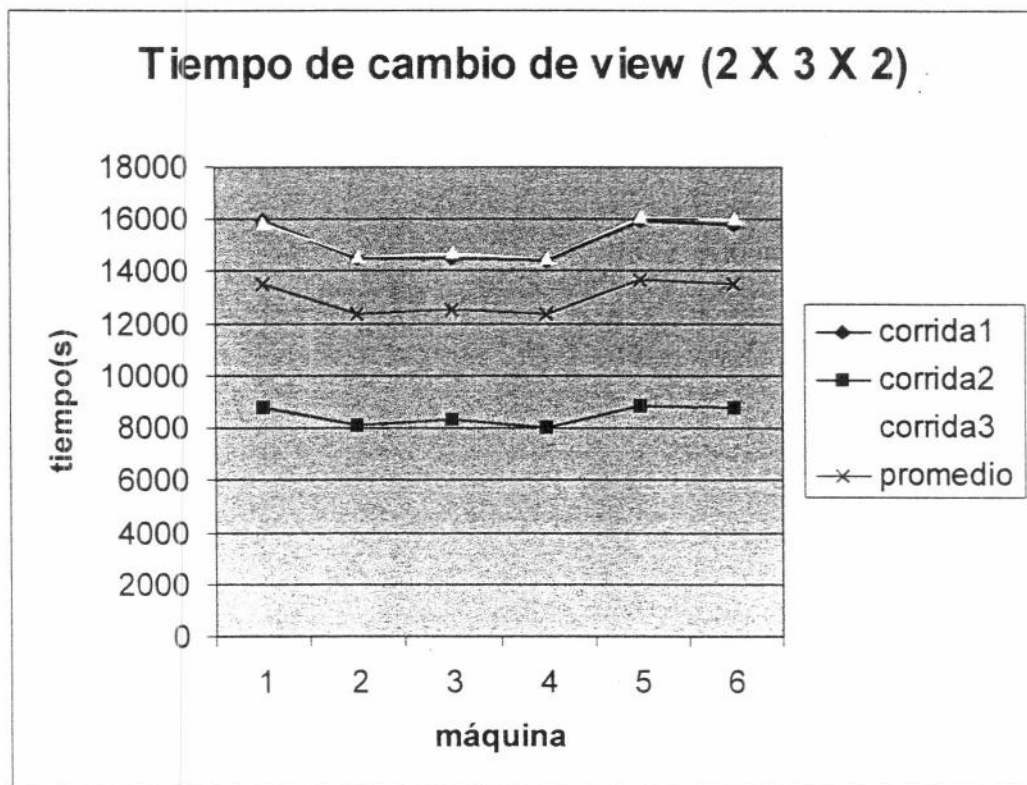
9.1.1.2 Configuración 2 Maq X 3Miembros X 2 grupos

Total Miembros: 6

miembro numero	tiempo para view change			promedio
	corrida1	corrida2	corrida3	
1	15915	8801	15869	13528.33333
2	14487	8051	14539	12359.00000
3	14487	8323	14724	12511.33333
4	14453	8016	14491	12320.00000
5	15969	8838	16098	13635.00000
6	15809	8735	16000	13514.66667
cambio de view				12995.25000



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.



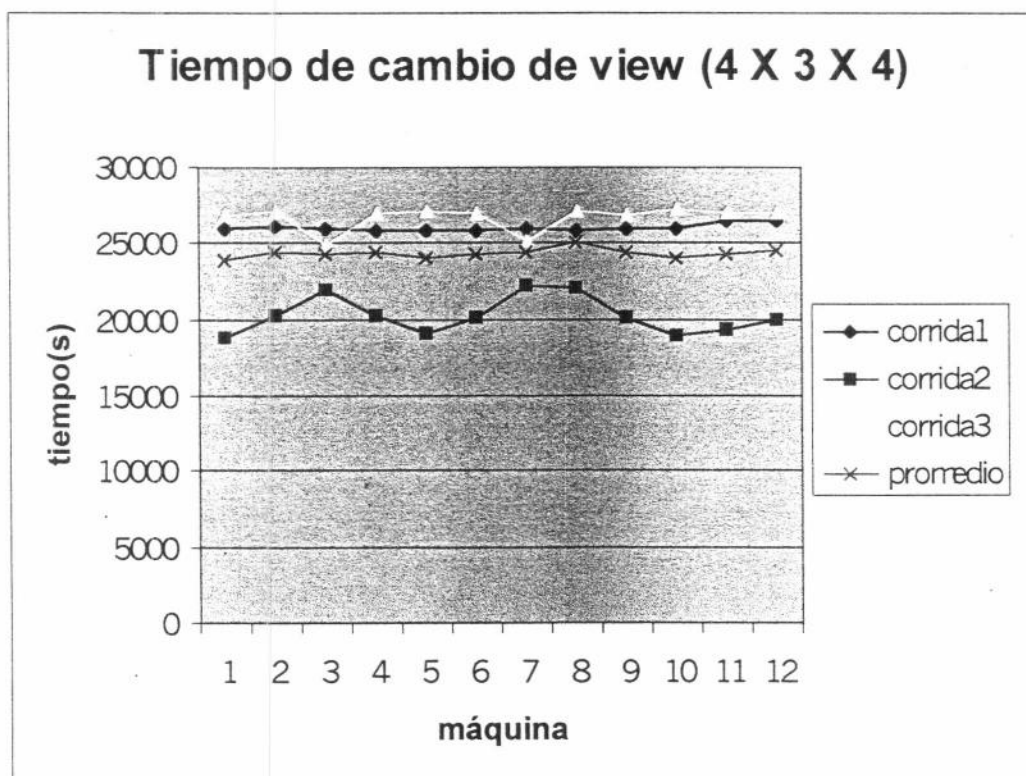
9.1.1.3 Configuración 4 Maq X 3Miembros X 4 grupos

Total Miembros: 12



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

miembro numero	tiempo para view change			promedio
	corrida1	corrida2	corrida3	
1	25902	18807	26853	23854.00000
2	26062	20177	27069	24436.00000
3	25991	21955	24869	24271.66667
4	25891	20183	27047	24373.66667
5	25770	18983	27166	23973.00000
6	25822	20037	26963	24274.00000
7	25957	22174	25157	24429.33333
8	25860	22076	27153	25029.66667
9	25958	20134	26926	24339.33333
10	25916	18861	27246	24007.66667
11	26515	19264	27043	24274.00000
12	26520	19935	27038	24497.66667
cambio de view				24313.33333

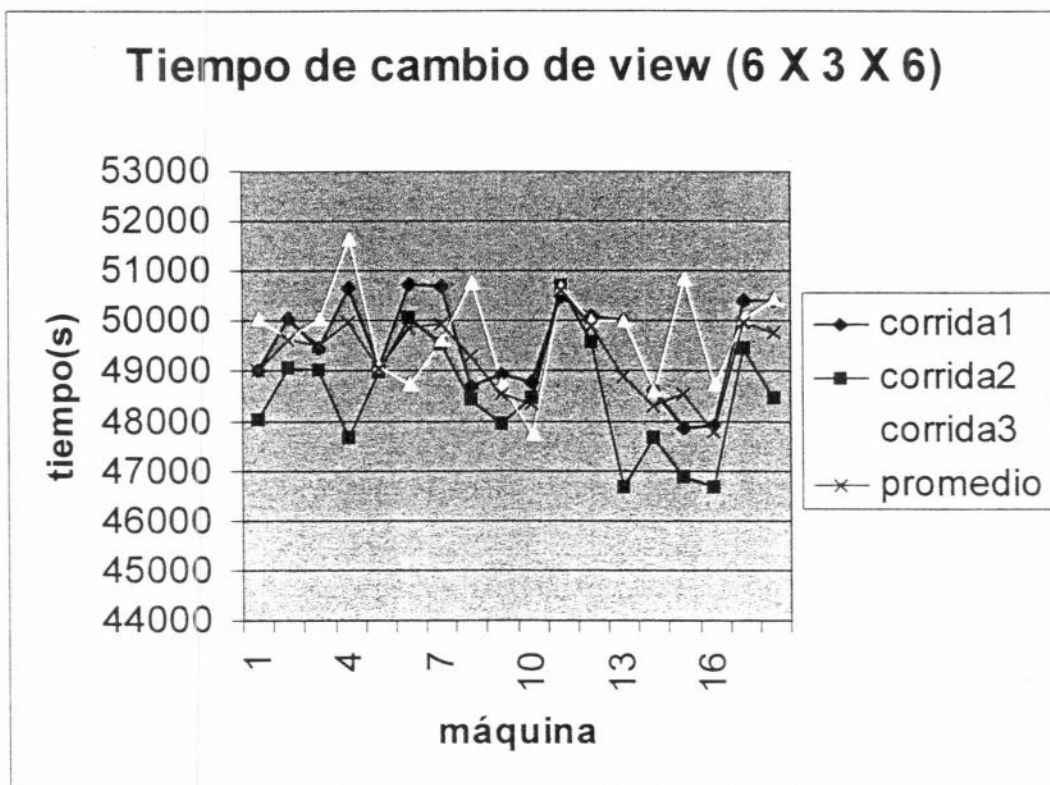




9.1.1.4 Configuración 6 Maq X 3Miembros X 6 grupos

Total Miembros: 18

miembro numero	tiempo para view change			promedio
	corrida1	corrida2	corrida3	
1	49011	48011	50051	49024.33333
2	50064	49065	49667	49598.66667
3	49449	49033	50064	49515.33333
4	50634	47678	51644	49985.33333
5	49091	48976	49091	49052.66667
6	50728	50068	48749	49848.33333
7	50697	49516	49596	49936.33333
8	48719	48419	50764	49300.66667
9	48946	47946	48749	48547.00000
10	48778	48478	47760	48338.66667
11	50482	50687	50667	50612.00000
12	50084	49584	50031	49899.66667
13	50002	46670	49999	48890.33333
14	48630	47652	48607	48296.33333
15	47867	46866	50851	48528.00000
16	47887	46678	48749	47771.33333
17	50410	49458	50068	49978.66667
18	50424	48467	50454	49781.66667
cambio de view				49272.51852



9.1.2 Cambio de view con View Synchrony

9.1.2.1 Configuración 4 Maq X 1 Miembros X 4 grupos

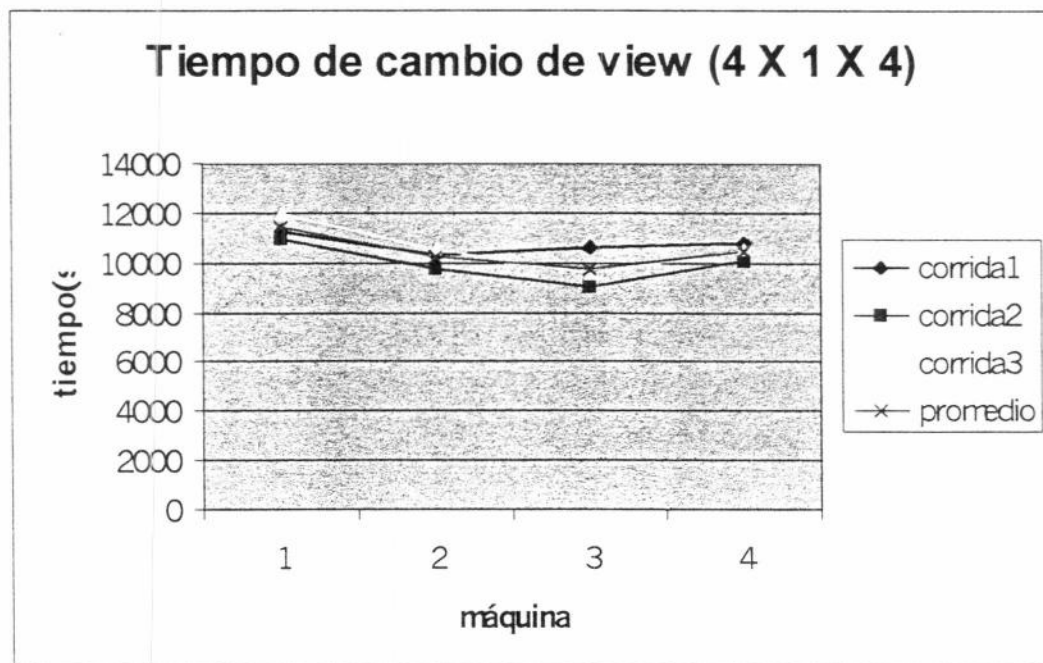
Total Miembros : 4



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

miembro numero	tiempo para view change			promedio
	corrida1	corrida2	corrida3	
1	11296	10964	11984	11414.66667
2	10319	9743	10567	10209.66667
3	10603	8997	9567	9722.33333
4	10816	10048	10556	10473.33333

cambio de view 10455.00000



9.1.2.2 Configuración 6 Maq X 1 Miembros X 6 grupos

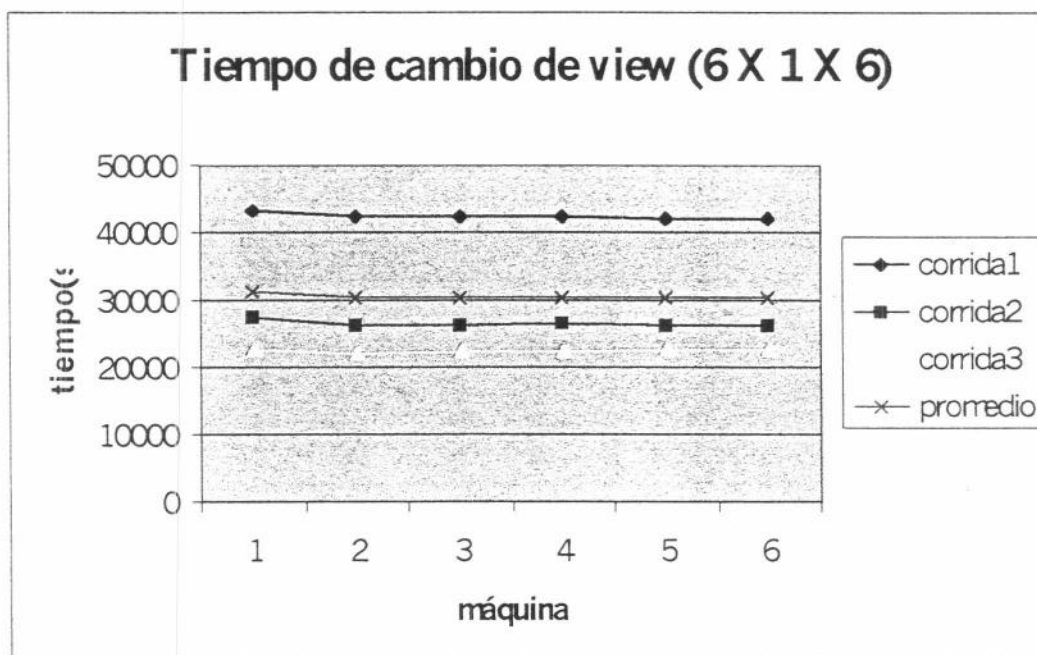
Total miembros: 6



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

miembro numero	tiempo para view change			promedio
	corrida1	corrida2	corrida3	
1	43122	27450	22729	31100.33333
2	42339	26194	22054	30195.66667
3	42485	26232	22366	30361.00000
4	42276	26355	22372	30334.33333
5	41960	26302	22794	30352.00000
6	41936	26052	22787	30258.33333

cambio de view 30326.41667

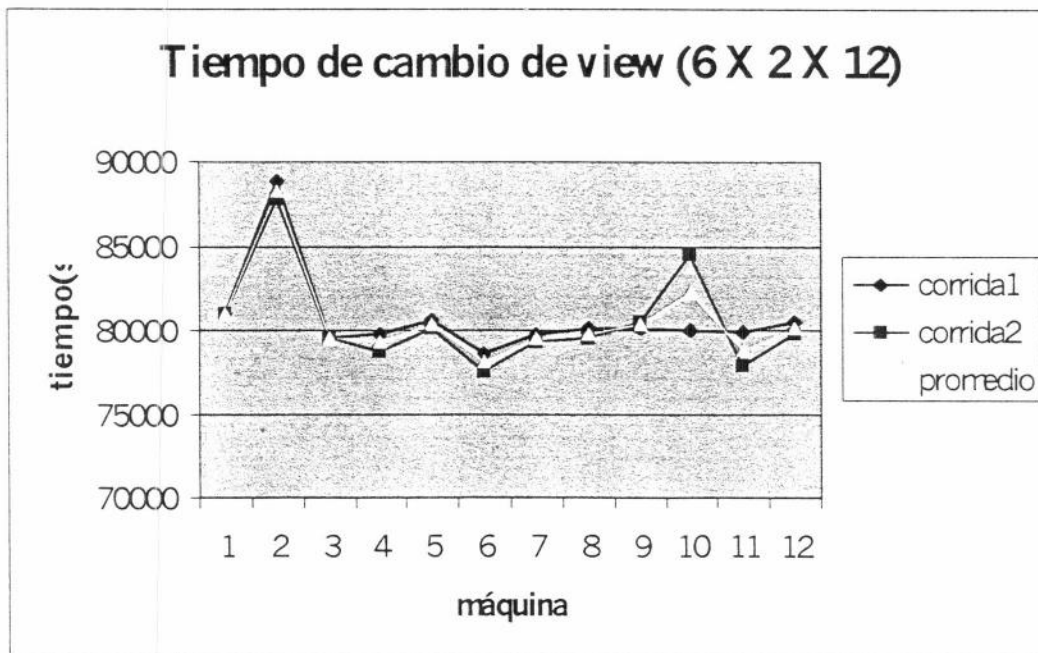


9.1.2.3 Configuración 6 Maq X 2 Miembros X 12 grupos

Total miembros:12



miembro numero	tiempo para view change			promedio
	corrida1	corrida2	corrida3	
1	80965	80972	fallo	80968.50000
2	88765	87755	fallo	88260.00000
3	79500	79567	fallo	79533.50000
4	79789	78749	fallo	79269.00000
5	80541	80076	fallo	80308.50000
6	78548	77586	fallo	78067.00000
7	79790	79345	fallo	79567.50000
8	80073	79564	fallo	79818.50000
9	80098	80495	fallo	80296.50000
10	80031	84509	fallo	82270.00000
11	79865	77838	fallo	78851.50000
12	80469	79823	fallo	80146.00000
cambio de view				80391.00000





9.1.2.4 Configuración 6 Maq X 3 Miembros X 18 grupos

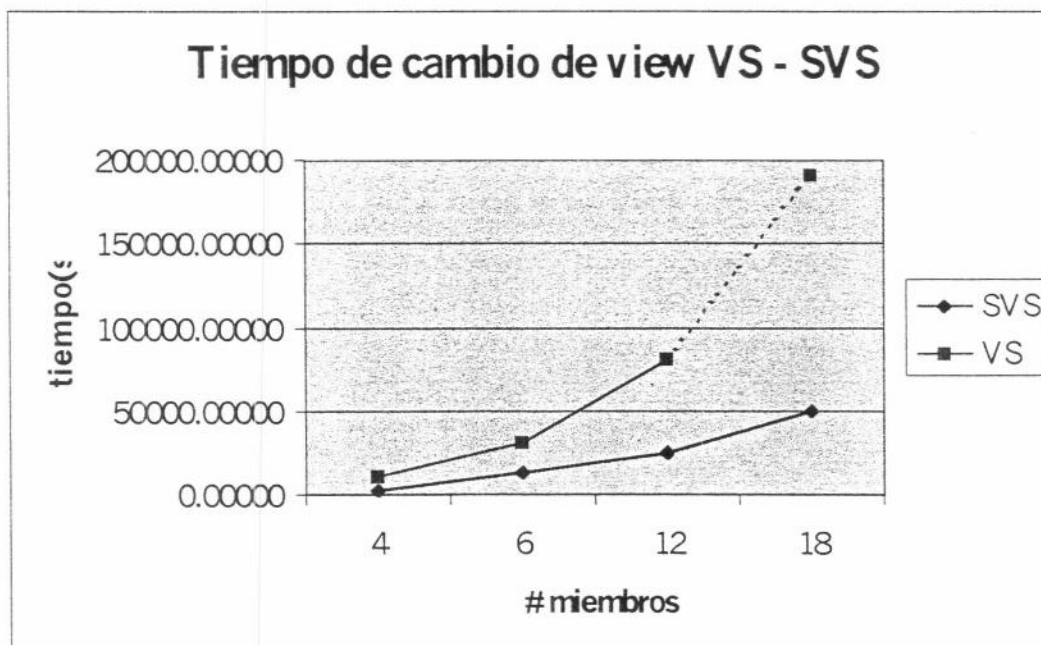
Total miembros:18

Esta configuración no funcionó. Al comenzar a correr el sistema comenzaban a producirse timeouts entre los distintos failure detectors debido a la sobrecarga de mensajes en la red. El tiempo del cambio de view de 190000 ms se calculó como una extrapolación de los tiempos obtenidos con menor cantidad de nodos.

9.1.3 Comparación tiempo de cambio de view entre view synchrony y structured view synchrony

Tiempo promedio de cambio de view

# miembros	SVS	VS
4	2887.25000	10455.00000
6	12995.25000	30326.41667
12	24313.33333	80391.00000
18	49272.51852	190000.00000





Podemos observar como los tiempos de cambio de view crecen en ambas abstracciones a medida que aumenta la cantidad de miembros que deben ponerse de acuerdo en la conformación de la nueva view. A medida que crece la cantidad de miembros aumentan la cantidad de mensajes que se deben intercambiar tanto los mensajes del protocolo como los del failure detector. Sin embargo, se comprueba que el esquema de structured view synchrony permite una mayor escalabilidad:

- En un esquema jerárquico de 2 niveles como es SVS la cantidad de mensajes de detección de fallas que debe procesar cada miembro se mantiene baja ya que por cada miembro local adicional sólo se agrega un par de mensajes más (entre el controlador y el nuevo miembro). En cambio en el esquema plano de VS, al agregar un nuevo miembro se necesitan $n-1$ mensajes más (entre el nuevo miembro y todo el resto de los miembros de la view) siendo n la cantidad total de miembros del grupo. La sobrecarga de mensajes generados hace que en nuestra prueba en el esquema de VS con número de miembros mayor o igual 18 se produzca una saturación que provoca falsas detecciones de fallas en todos los nodos. El sistema de esta manera, colapsa.
- Aún si no se están enviando mensajes de detección de fallas (ya que si la red se encuentra suficientemente cargada, los mensajes de los protocolos de multicasts bastan como avisos de que los procesos continúan operacionales) el esquema de SVS presenta ventajas en cuanto a la escalabilidad debido a la distribución de la carga del procesamiento de los mensajes. En VS, un miembro que envía un multicast debe hacerse cargo del procesamiento de los mensajes del protocolo de reliable broadcast de todos los miembros de la view (es decir debe procesar $2 * \# \text{miembros} * \# \text{etapas del protocolo}$ mensajes). Con un número relativamente pequeño de miembros la sobrecarga de procesamiento y la saturación de la red en el emisor producen el colapso del protocolo. En cambio en SVS el emisor solo procesa los mensajes entre él y el resto de los controladores por lo que mientras el número de estos se mantenga bajo no se producen sobrecargas en los miembros.

9.2 Prueba 2 – Latencia de envío de mensaje (One way latency) con mensajes dinámicamente uniformes



La latencia de envío (one way latency) mide el tiempo transcurrido entre el envío del mensaje por el emisor hasta la llegada a los miembros receptores. Se define para Structured View Synchrony, la latencia de envío como el tiempo transcurrido entre el envío del mensaje por el miembro controlador emisor hasta que llega a los miembros locales que no son propios del emisor.

El objetivo de esta prueba es comparar la diferencia en la latencia de envío de mensajes entre view synchrony y view synchrony estructurado. Se probarán ambas abstracciones con la misma carga de envío y con mensajes dinámicamente uniformes. Se espera que la latencia de mensajes sea mayor en el esquema de Structured View Synchrony ya que cada mensaje debe dar 2 o 3 saltos (hops): del ML emisor a su controlador, del controlador a los otros controladores y del controlador al ML receptor contra un sólo salto en el caso de View Synchrony.

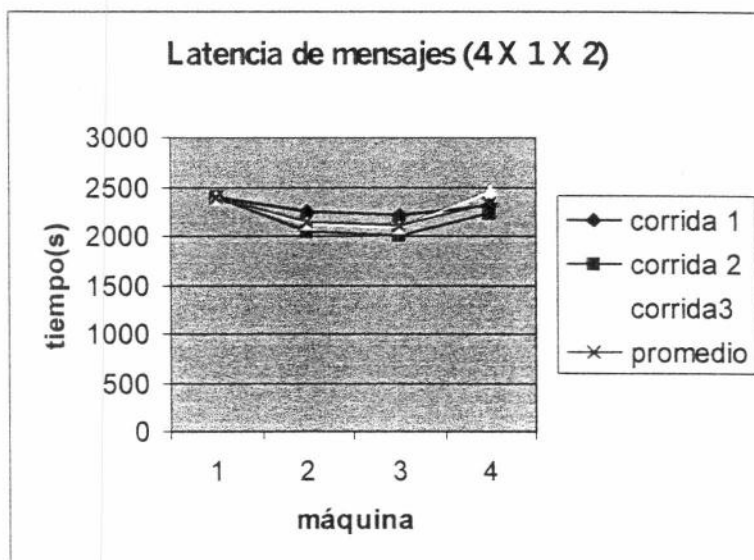
9.2.1 Structured View Synchrony

9.2.1.1 Configuración 4 Maq X 1 Miembros X 2 grupos

Total miembros: 4

miembro numero	tiempo de latencia de mensajes dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	2389	2386	2391	2388.66667
2	2239	2043	2131	2137.66667
3	2198	2004	2081	2094.33333
4	2306	2231	2456	2331.00000

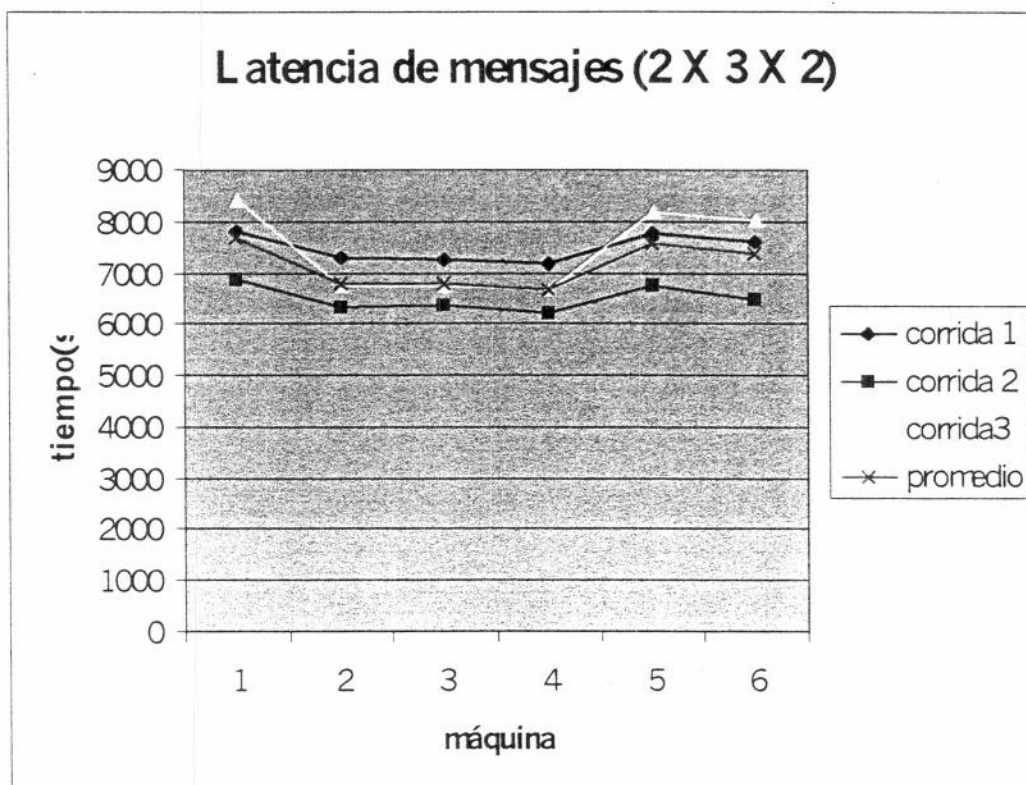
latencia de mensajes 2237.91667



9.2.1.2 Configuración 2 Maq X 3Miembros X 2 grupos

Total miembros: 6

miembro numero	tiempo de latencia de mensajes dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	7803	6859	8435	7699.00000
2	7298	6314	6764	6792.00000
3	7261	6348	6737	6782.00000
4	7158	6219	6672	6683.00000
5	7771	6736	8185	7564.00000
6	7621	6466	8011	7366.00000
latencia de mensajes				7098.75000



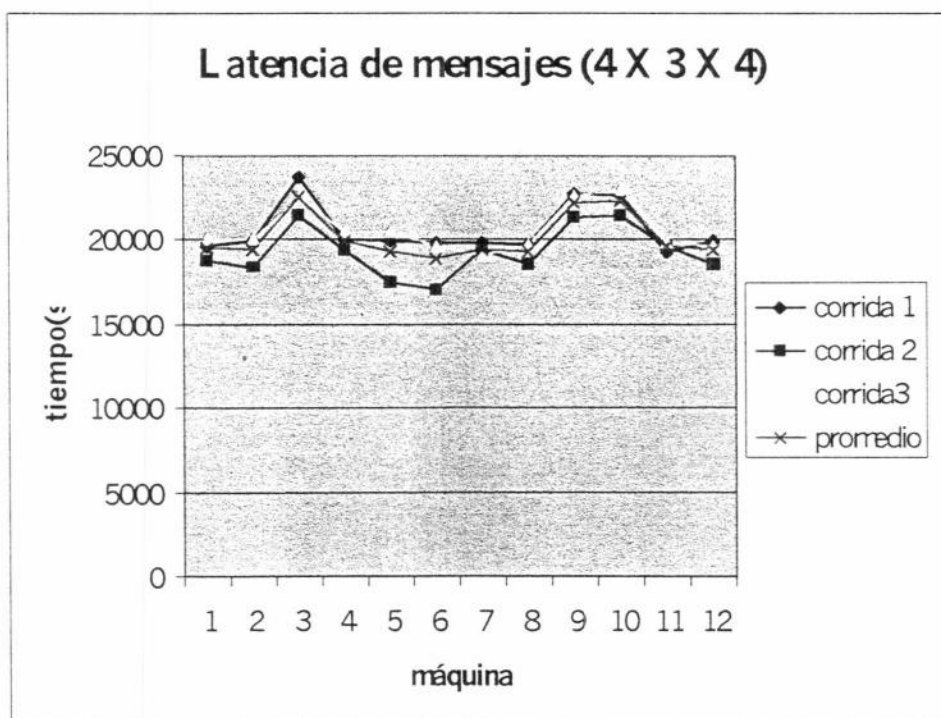


9.2.1.3 Configuración 4 Maq X 3Miembros X 4 grupos

Total miembros:12

miembro numero	tiempo de latencia de mensajes dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	19652	18731	20076	19486.33333
2	19940	18348	19998	19428.66667
3	23665	21459	22569	22564.33333
4	20048	19445	20462	19985.00000
5	19915	17456	20515	19295.33333
6	19777	16998	19654	18809.66667
7	19812	19346	19098	19418.66667
8	19696	18543	19696	19311.66667
9	22726	21352	22586	22221.33333
10	22616	21458	22986	22353.33333
11	19341	19569	19876	19595.33333
12	19907	18526	19632	19355.00000

latencia de mensajes 20152.05556

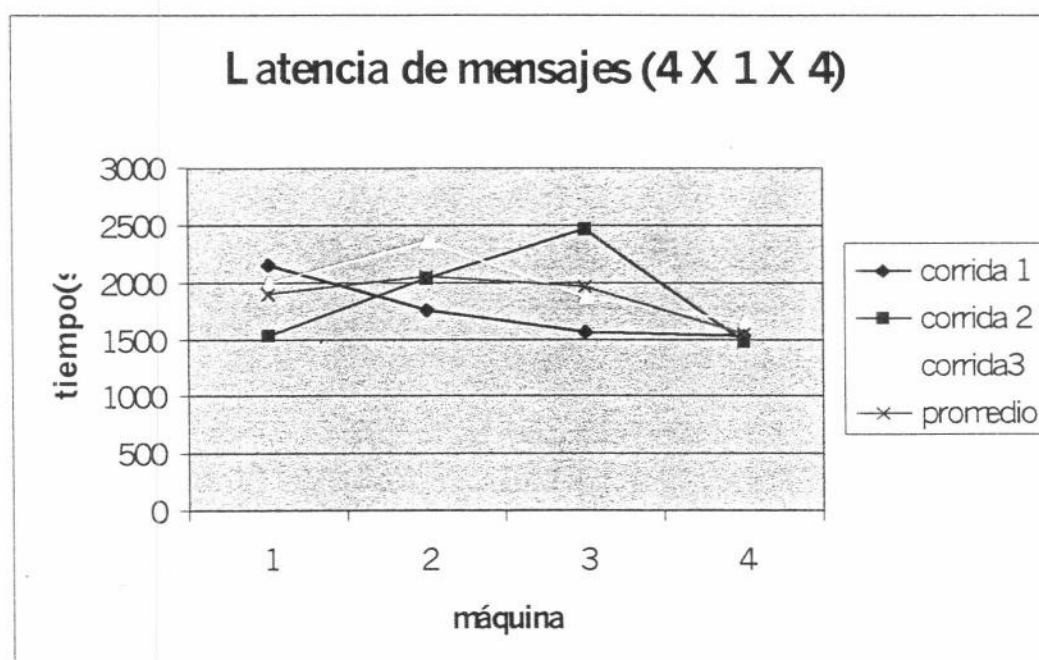




9.2.2 View Synchrony

9.2.2.1 Configuración 4 Maq X 1 Miembros X 4 grupos

miembro numero	latencia de mensajes dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	2148	1533	1987	1889.33333
2	1758	2021	2354	2044.33333
3	1564	2457	1876	1965.66667
4	1525	1472	1654	1550.33333
latencia de msgs				1862.41667

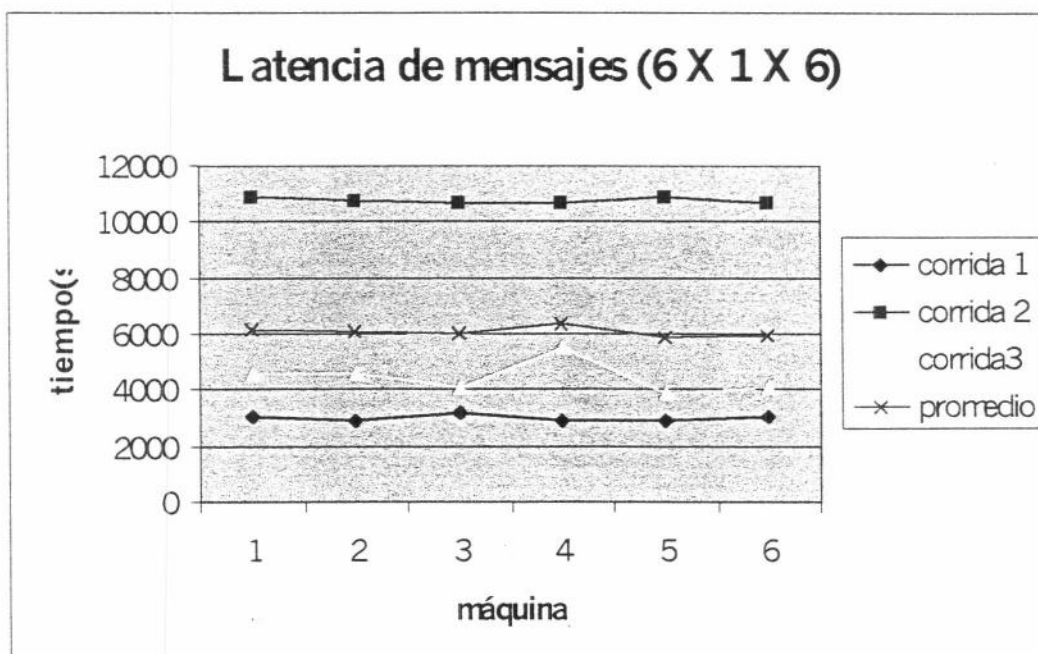




9.2.2.2 Configuración 6 Maq X 1 Miembros X 6 grupos

Total miembros : 6

miembro numero	latencia de mensajes dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	3070	10870	4564	6168.00000
2	2883	10764	4578	6075.00000
3	3201	10680	4123	6001.33333
4	2905	10652	5487	6348.00000
5	2923	10837	3904	5888.00000
6	3028	10636	4098	5920.66667
latencia de msgs				6039.50000

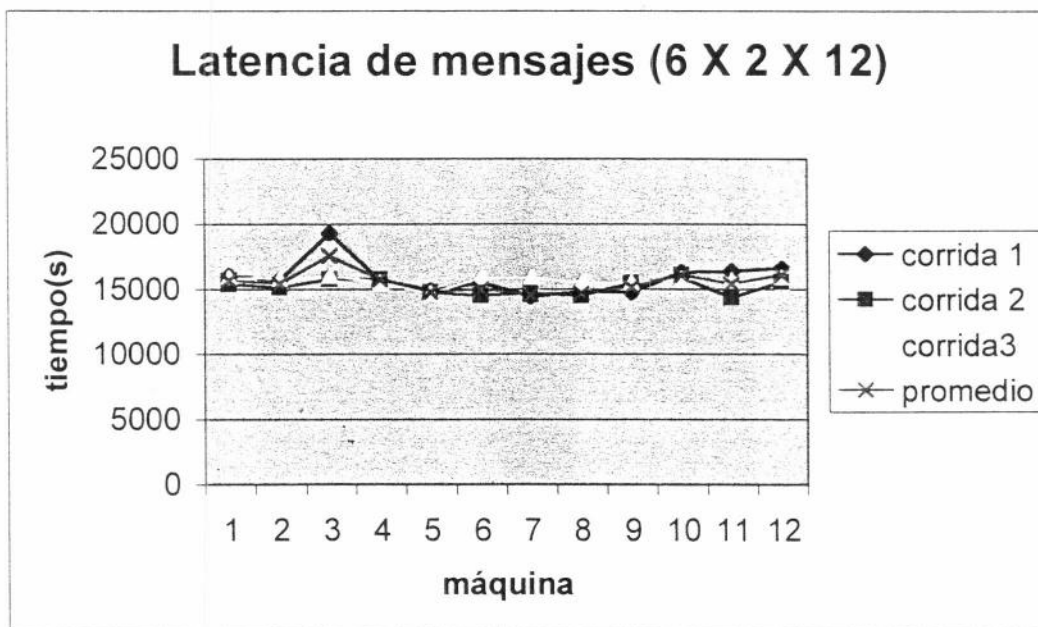




9.2.2.3 Configuración 6 Maq X 2 Miembros X 12 grupos

Total miembros: 12

miembro numero	latencia de mensajes dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	15947	15326	15876	15636.50000
2	15667	15076	15765	15371.50000
3	19290	15702	15987	17496.00000
4	15683	15727	15487	15705.00000
5	14822	14714	14678	14768.00000
6	15498	14445	15876	14971.50000
7	14392	14678	15874	14535.00000
8	14751	14472	15467	14611.50000
9	14754	15444	15467	15099.00000
10	16243	15874	15764	16058.50000
11	16362	14316	15678	15339.00000
12	16578	15508	15876	16043.00000
latencia de msgs				15634.87500

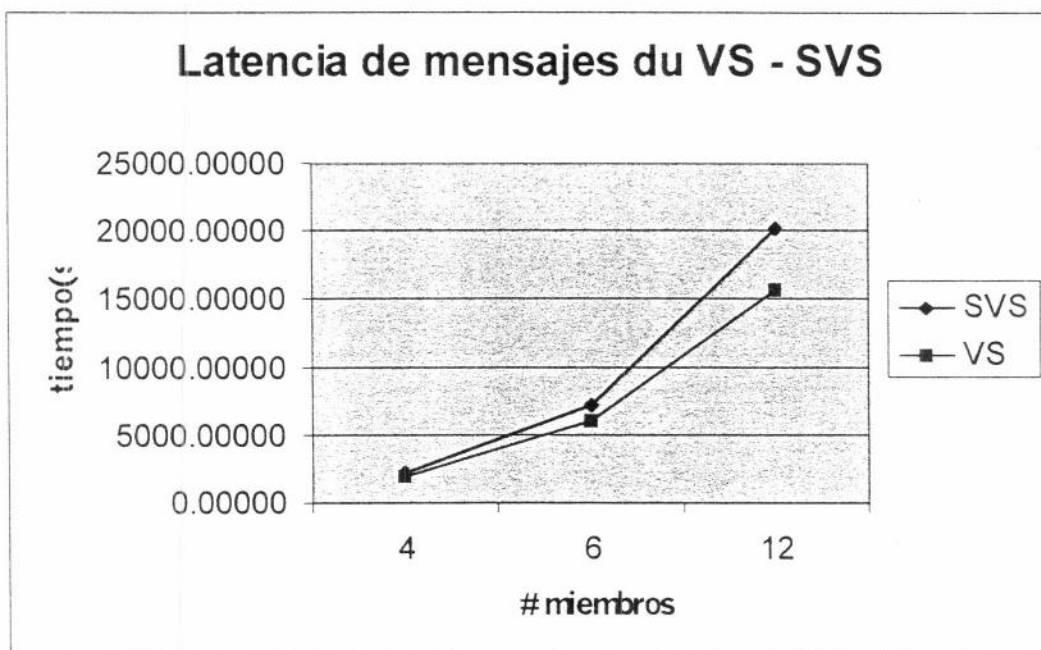




9.2.3 Comparación latencia de mensajes du entre view synchrony y structured view synchrony

latencia promedio de mensajes
dinámicamente uniformes

# miembros	SVS	VS
4	2237.91667	1862.41667
6	7098.75000	6039.50000
12	20152.05556	15634.87500



El incremento en la latencia de mensajes es el costo a pagar por obtener una mayor escalabilidad en Structured View Synchrony. En ambos esquemas a medida que aumenta la cantidad de miembros crece la latencia de los mensajes. A una misma cantidad de miembros la latencia siempre es mayor en SVS que en VS, sin embargo la tasa de crecimiento es similar en ambos esquemas.



9.3 Prueba 3 – Latencia de envío de mensaje (One way latency) con mensajes no dinámicamente uniformes

La latencia de envío (one way latency) mide el tiempo transcurrido entre el envío del mensaje por el emisor hasta la llegada a los miembros receptores. Se define para Structured View Synchrony, la latencia de envío como el tiempo transcurrido entre el envío del mensaje por el miembro controlador emisor hasta que llega a los miembros locales que no son propios del emisor.

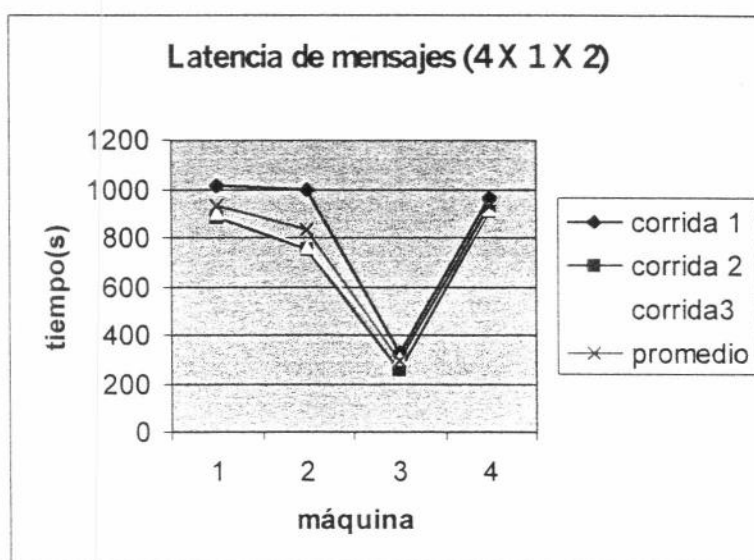
El objetivo de esta prueba es comparar la diferencia en la latencia de envío de mensajes entre view synchrony y view synchrony estructurado. Se probarán ambas abstracciones con la misma carga de envío y con mensajes no dinámicamente uniformes. Se espera que la latencia de mensajes sea mayor en el esquema de Structured View Synchrony ya que cada mensaje debe dar 2 o 3 saltos (hops): del ML emisor a su controlador, del controlador a los otros controladores y del controlador al ML receptor contra un sólo salto en el caso de View Synchrony.

9.3.1 Structured View Synchrony

9.3.1.1 Configuración 4 Maq X 1 Miembros X 2 grupos

Total miembros: 4

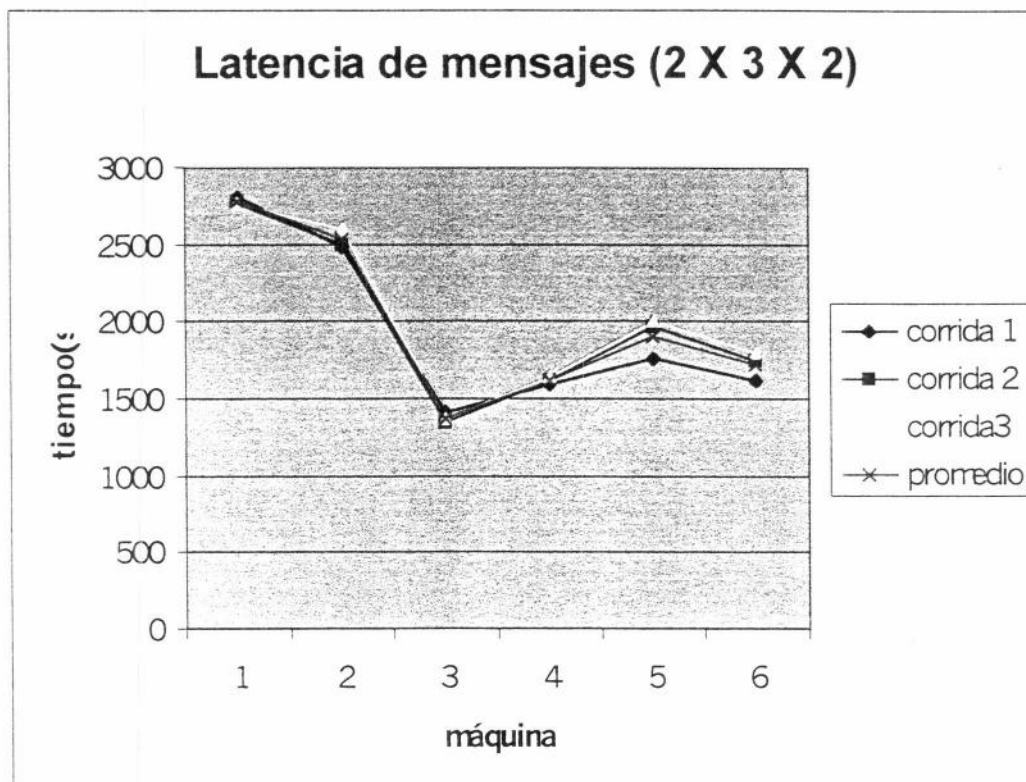
miembro numero	tiempo de latencia de mensajes no dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	1014	882	894	930.00000
2	994	750	756	833.33333
3	330	255	301	295.33333
4	961	910	911	927.33333
latencia de mensajes				746.50000



9.3.1.2 Configuración 2 Maq X 3Miembros X 2 grupos

Total miembros: 6

miembro numero	tiempo de latencia de mensajes no dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida 3	
1	2806	2772	2765	2781.00000
2	2486	2501	2598	2528.33333
3	1404	1344	1366	1371.33333
4	1589	1613	1645	1615.66667
5	1758	1961	1998	1905.66667
6	1619	1743	1802	1721.33333
latencia de mensajes				1653.50000

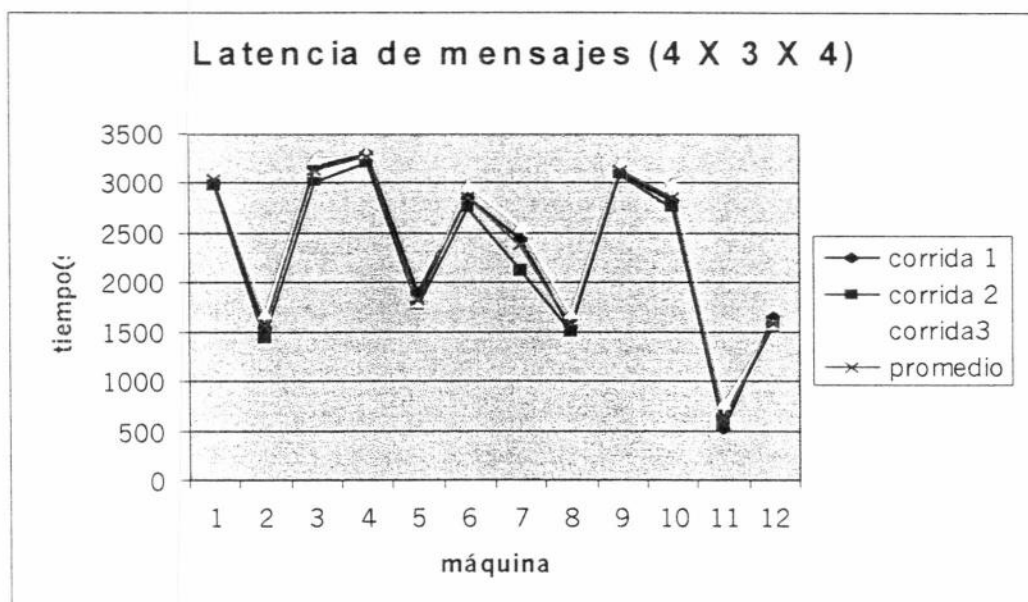




9.3.1.3 Configuración 4 Maq X 3Miembros X 4 grupos

Total miembros:12

miembro numero	tiempo de latencia de mensajes no dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	3005	2987	3103	3031.66667
2	1543	1433	1687	1554.33333
3	3154	3011	3245	3136.66667
4	3304	3208	3307	3273.00000
5	1898	1756	1788	1814.00000
6	2848	2742	2976	2855.33333
7	2436	2131	2567	2378.00000
8	1537	1508	1678	1574.33333
9	3098	3091	3187	3125.33333
10	2825	2745	2987	2852.33333
11	530	601	765	632.00000
12	1644	1564	1567	1591.66667
latencia de mensajes			2318.22222	



9.3.2 View synchrony

9.3.2.1 Configuración 4 Maq X 1 Miembros X 4 grupos

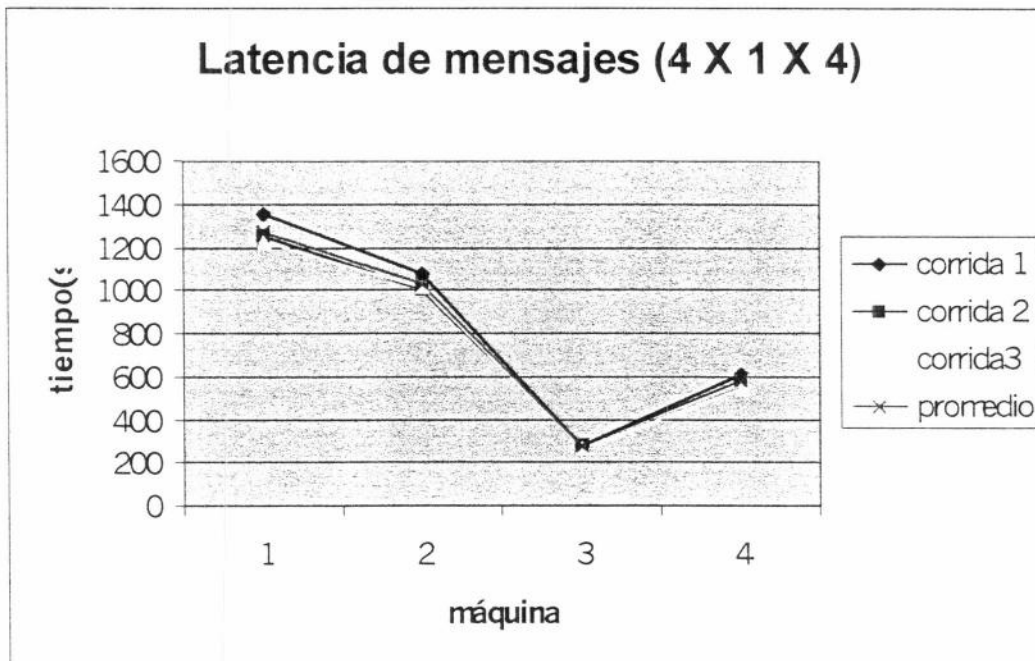
Total miembros: 4



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

miembro numero	tiempo de latencia de mensajes no dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	1352	1247	1221	1273.33333
2	1075	1005	1008	1029.33333
3	277	275	269	273.66667
4	604	583	541	576.00000

latencia de msgs 788.08333



9.3.2.2 Configuración 6 Maq X 1 Miembros X 6 grupos

Total miembros: 6

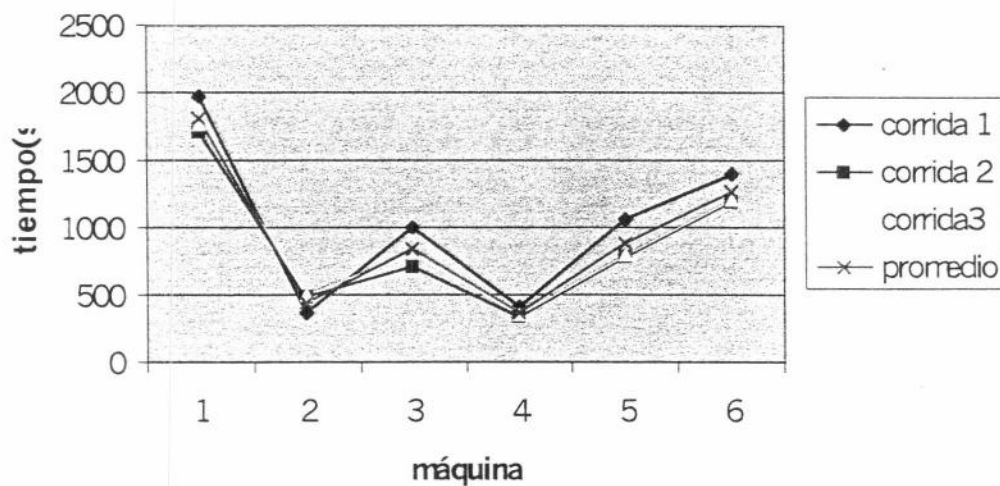


Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

miembro numero	tiempo de latencia de mensajes no dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	1968	1700	1765	1811.00000
2	362	481	501	448.00000
3	1004	706	809	839.66667
4	411	331	354	365.33333
5	1064	775	789	876.00000
6	1394	1188	1203	1261.66667

latencia de msgs 835.66667

Latencia de mensajes (6 X 1 X 6)

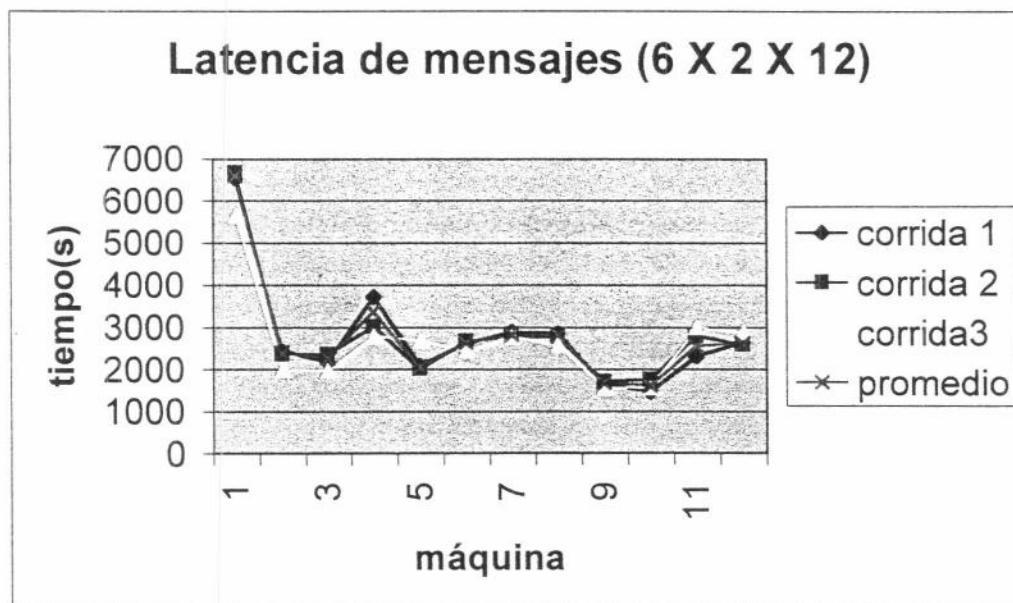




9.3.2.3 Configuración 6 Maq X 2 Miembros X 12 grupos

Total miembros:12

miembro numero	tiempo de latencia de mensajes no dinámicamente uniformes			promedio
	corrida 1	corrida 2	corrida3	
1	6529	6655	5646	6592.00000
2	2422	2344	1987	2383.00000
3	2169	2344	1976	2256.50000
4	3709	3002	2784	3355.50000
5	2080	2003	2677	2041.50000
6	2611	2656	2456	2633.50000
7	2878	2771	2777	2824.50000
8	2843	2765	2567	2804.00000
9	1639	1698	1567	1668.50000
10	1452	1745	1544	1598.50000
11	2305	2783	2987	2544.00000
12	2634	2567	2865	2600.50000
			latencia de msgs	2102.87500

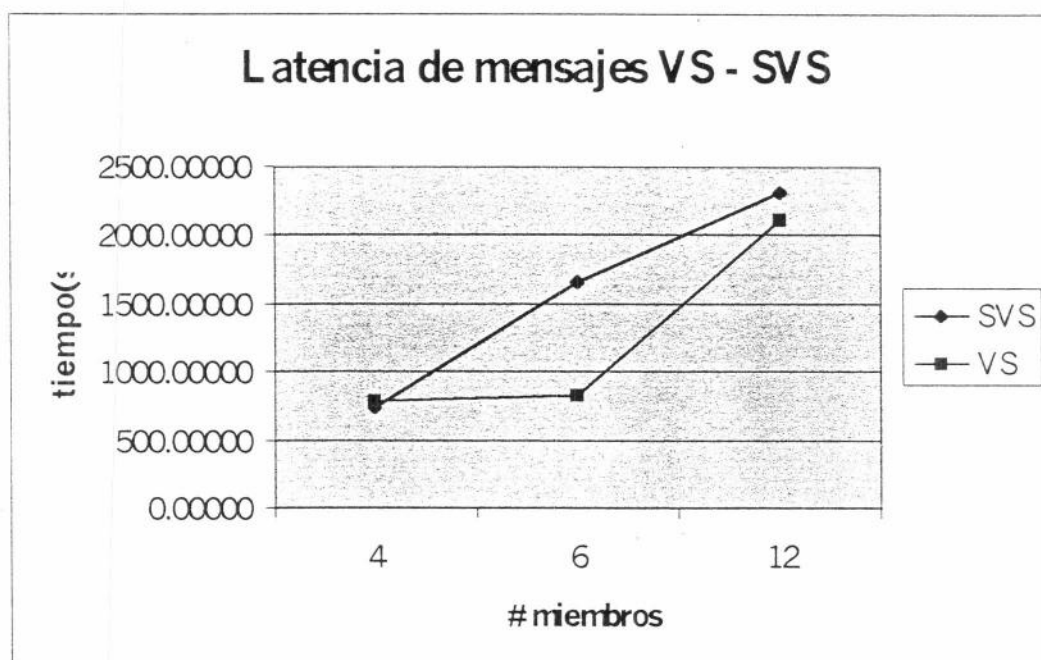




9.3.3 Comparación latencia de mensajes ndu entre view synchrony y structured view synchrony

latencia promedio de mensajes
dinámicamente uniformes

# miembros	SVS	VS
4	746.50000	788.08333
6	1653.50000	835.66667
12	2318.22222	2102.87500





El incremento en la latencia de mensajes es el costo a pagar por obtener una mayor escalabilidad en Structured View Synchrony. En ambos esquemas a medida que aumenta la cantidad de miembros crece la latencia de los mensajes. A una misma cantidad de miembros la latencia siempre es mayor en SVS que en VS, sin embargo la tasa de crecimiento en ambos casos (mensajes no dinámicamente uniformes) es similar en ambos esquemas. Los resultados obtenidos se asemejan a las pruebas realizadas con mensajes dinámicamente uniformes solo que en este caso la latencia en promedio es mucho menor debido a la propiedad de no uniformidad dinámica de los mensajes.



10 Modelo de tolerancia a fallas de la WDVM

10.1 Introduccion

El origen de la presente tesis fue la intención de desarrollar una arquitectura para la implementación de una super computadora virtual que permitiera administrar los recursos combinados de las computadoras conectadas a Internet, a fin de ponerlos a disposición de cualquier usuario interesado en la ejecución de aplicaciones distribuidas y/o paralelas con altas necesidades de poder de cálculo. La arquitectura general de dicha supercomputadora (WDVM) fue diseñada por Javier Esteban y Marcelo Weinbinder en [EST/00], mientras que nuestro grupo se enfocó hacia la investigación y desarrollo de los mecanismos de tolerancia a fallas necesarios en un ambiente inestable como internet.

A continuacion describiremos un esquema para aplicar a la arquitectura de [EST/00] los mecanismos de tolerancia a fallas diseñados en la presente tesis

10.2 Arquitectura de la WDVM

En la arquitectura de [EST/00], existen los siguientes tipos de entidades

Cliente: Aplicación que requiere de una gran cantidad de recursos de CPU para ejecutar un conjunto de tareas.

Worker: Proceso que ejecuta sobre una máquina voluntaria y ofrece recursos de CPU para que un cliente ejecute sus tareas.

Broker: Proceso que coordina el pedido y la entrega de recursos de CPU. Se comunica con otros brokers, por propósitos de balance de carga, se encarga de monitorear el correcto funcionamiento de los workers a su cargo y administra estadísticas para poder decidir que worker tomará el puesto de broker en caso de que se necesite.

Server: Proceso que se ejecuta sobre un server con dirección bien conocida y se encarga de la asignación inicial de nuevos workers y clientes a los brokers.



Un cliente se comunica con el server a través de la dirección bien conocida y este le asigna un broker , que se encargará de enviar a ejecutar las tareas que el cliente le solicite.

Las workers son máquinas que se presentan como voluntarias para formar parte de la supercomputadora global. Se conectan con el server a través de una dirección bien conocida y este les asigna un broker , que se encargará de asignarles tareas pedidas por los clientes.

Cada broker administra un conjunto de workers , encargándose de la asignación de tareas y de la comunicación entre los clientes y los workers. Cada broker tiene un límite en la cantidad de workers que puede supervisar y cuando ese límite es superado se encarga de elegir entre sus workers un nuevo broker , al que asignará parte de sus brokers.

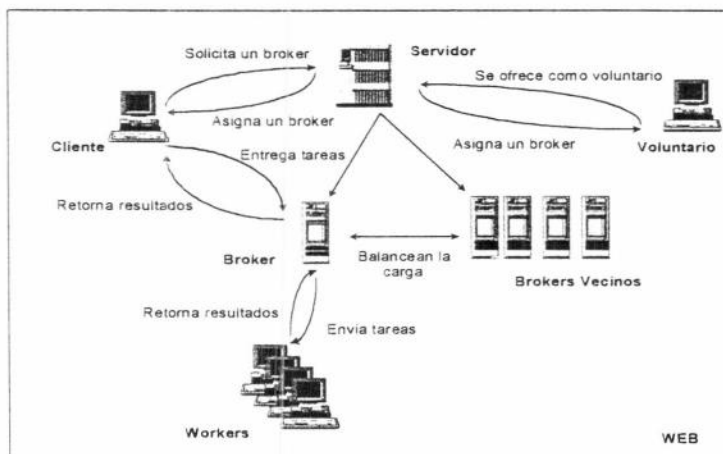


Fig 32 –Arquitectura de la WDVM

En esta arquitectura se pueden observar distintos puntos de corte , con consecuencias disímiles para el sistema :

Falla o desconexión del Server: Posiblemente es la falla con peores consecuencias. Produce el aislamiento del sistema , ya que no se podrán volver a conectar nuevos clientes ni nuevas máquinas voluntarias.

Falla o desconexión de un Worker : produce la pérdida de la tarea en que se estaba ocupando el worker. Si el broker que lo supervisa tuviera alguna manera de detectar la caída del worker , podría encargarle a otro la tarea , pero aun en ese caso debería tener cuidado con la posible "reaparicion" de un worker que no haya fallado completamente si no que esté muy lento



Falla o desconexión de un Broker : produce la pérdida de los todas las tareas de las que se encargaban sus workers , así como la desconexión entre estos y sus clientes. Si los clientes y workers detectan de alguna manera la caída del broker , pueden volver a pedir al Server que les asigne un nuevo broker , pero las tareas tendrán que volver a empezarse desde cero.

10.3 Modelo de tolerancia a fallas en la WDVM

Una aplicación construida sobre nuestro esquema de tolerancia a fallas , esta estructurada como una colección de grupos de procesos. Cada grupo de procesos es un conjunto de procesos que cooperan para llevar a cabo una de las tareas de la aplicación. Basándonos en este esquema , la solución a los problemas del punto anterior consiste en la replicación de los componentes de la WDVM para que puedan así sobreponerse a las fallas.

De esta manera se reemplazara el Server por un conjunto de procesos que se ocupen de la misma tarea. Todos los miembros del grupo deberán atender los requerimientos de clientes y voluntarios en direcciones bien conocidas. Los pedidos de clientes o workers recibidos por uno de los miembros del grupo serán reenviados al resto de los miembros del grupo Server , para mantener la coherencia entre la visión del estado de la máquina global de los diferentes miembros.

Asimismo se reemplazarán los workers por grupos de workers . Cuando una máquina voluntaria se ofrezca a uno de los miembros del grupo de servers para formar parte de la máquina global , el server la asignará a uno de los grupos de workers o bien creará un nuevo grupo de workers al que pertenecerá la nueva máquina. Los grupos de workers serán a su vez supervisados por grupos de brokers . Cuando un grupo de brokers detecte que tiene demasiados grupos de workers a su cargo , elegirá un grupo de workers para ser promovido a grupo de brokers.

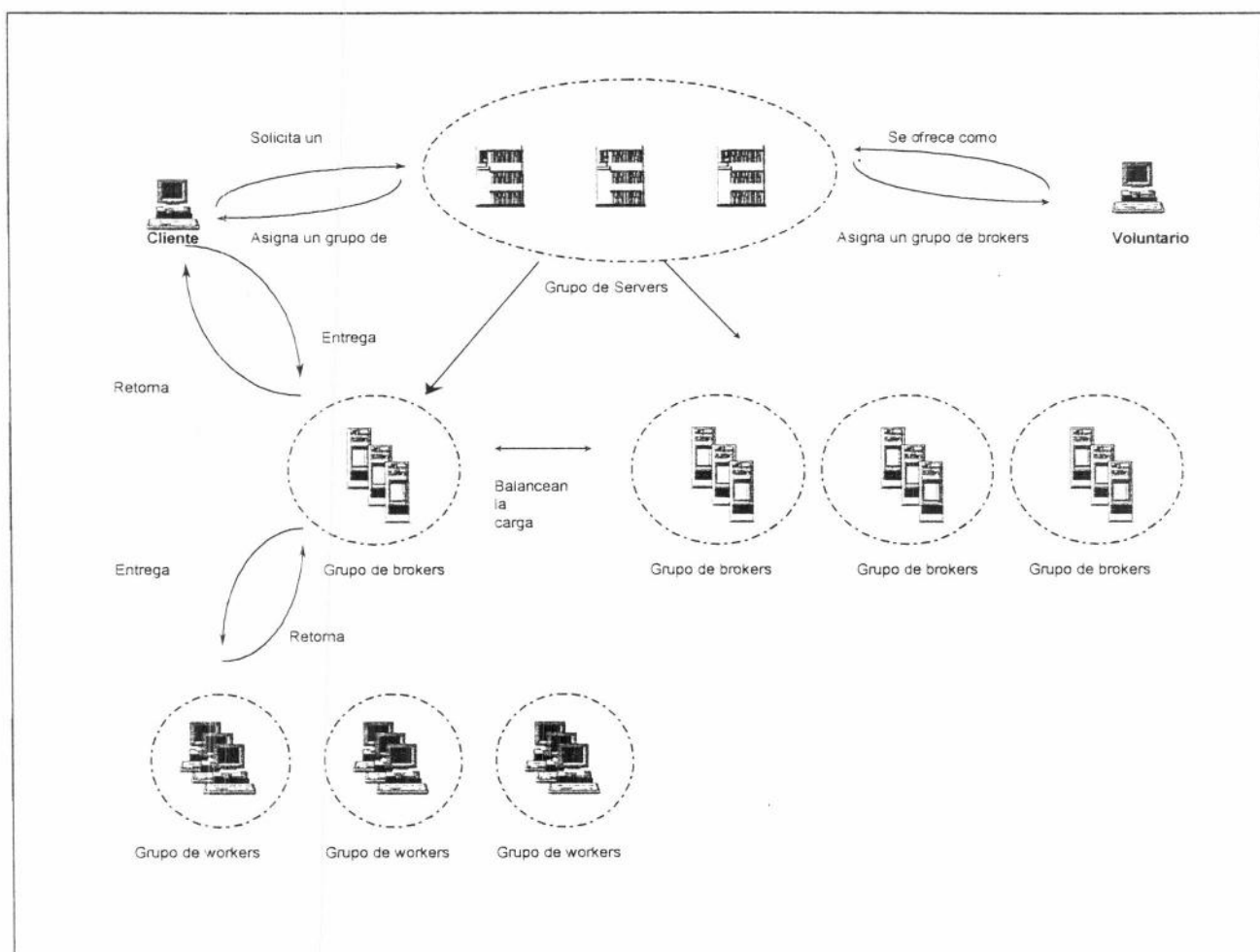
Con este esquema , examinemos como reacciona el sistema ante la falla o desconexión de alguno de sus componentes.

Falla o desconexión de uno de los servers: cuando un cliente o máquina voluntaria no se puede comunicar con alguno de los miembros del grupo de servers para solicitar un servicio o unirse a la máquina global , elige otro de los miembros del grupo de servers (todos tienen direcciones bien conocidas) y se comunica con este.



Falla o desconexión de un miembro de un grupo de workers: La tarea de la que está encargado el grupo de workers es llevada a cabo por el resto de los miembros del grupo. Esto es transparente tanto para el grupo de brokers que supervisa la tarea como para el cliente.

Falla o desconexión de un miembro del grupo de brokers : el resto de los miembros del grupo se ocupa de seguir supervisando a los grupos de workers que dependen de él. Ni los workers ni los clientes se ven afectados por esto. En caso de que la cantidad de miembros de un grupo de workers descienda peligrosamente , el grupo puede solicitar al grupo de servers que le asigne algunos de los nuevos voluntarios.





Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

Fig 33 –Modelo de tolerancia a fallas de la WVDM



11 Conclusiones

Nuestro objetivo primario fue diseñar una arquitectura de software que soportara el modelo de Structured View Synchrony con el fin de incrementar la escalabilidad de un sistema distribuido tolerante a fallas.

Implementamos nuestro diseño en forma completa en cuanto a funcionalidad con el fin de verificar su aplicación para una máquina distribuida global sobre internet y realizar pruebas comparativas con respecto al modelo plano de View Synchrony. Durante la etapa de implementación dejamos de lado algunas funcionalidades u optimizaciones cuando las mismas no formaban parte del objetivo esencial que nos habíamos trazado. Como ejemplo más importante, la implementación de nuestro algoritmo de cambio de view no toma en cuenta la caída de miembros controladores o coordinadores aunque nuestro algoritmo sí lo hace. La razón para no haber realizado la implementación completa del algoritmo es que ese protocolo de cambio de view es exactamente igual al de la caída de un miembro local, con el agregado de que se debe ejecutar como primer paso un algoritmo de "elección" para promover a uno de los miembros locales como nuevo controlador o coordinador. En cuanto a la performance, si bien en la literatura encontramos que el límite para el colapso en View Synchrony era de unos 40 miembros, en nuestro caso el mismo se produjo con una cantidad mucho menor (18 miembros) creemos que esto se debe al entorno en que ejecutamos las pruebas, ya que debido a la escasa cantidad de máquinas disponibles tuvimos que juntar varios miembros en cada máquina.

Consideramos, no obstante, como totalmente cumplido el objetivo. Creemos que nuestro diseño de la arquitectura de software presenta un modelo de calidad para la construcción de un sistema distribuido tolerante a fallas adonde se encuentra perfectamente delimitadas las tareas de cada capa de abstracción. Además proveemos un framework de aplicación para programación de soluciones tolerante a fallas sencillo y orientado a eventos como otros frameworks disponibles en Java (Swing por ejemplo).

También consideramos exitosa nuestra implementación ya que nos permitió validar y comparar las soluciones de Structured View Synchrony y View Synchrony a semejanza de las pruebas realizadas en [GOU/94].

El segundo objetivo propuesto fue unir nuestro trabajo de investigación y desarrollo con el realizado por [EST/00], como una forma de agregar a la infraestructura para una máquina global propuesta en el citado trabajo



mecanismos de tolerancia a fallas. Creemos que este objetivo ha sido completado en general. Hemos presentado un modelo de tolerancia a fallas adonde proponemos como aplicar nuestro diseño de arquitectura a la infraestructura de la máquina global pero a alto nivel. El marco de situación en el cual se desarrollo esta tesis (alumnos trabajando a tiempo completo, distintos objetivos y tiempos de realización de las mismas, etc) impidió que se lograra un modelo de aplicación de la tolerancia a fallas con una mayor integración entre ambas tesis. Lo ideal hubiera sido agregar a la implementación de [EST/00] nuestro modelo de tolerancia a fallas. Lo que hacemos aqui es solo esquematizar como se realizaría dicho proceso.

La realización de los objetivos antes enumerados nos permitió llegar a las siguientes conclusiones:

- Una arquitectura de software basada en Structured View Synchrony permite una mayor escalabilidad que es fundamental para sistemas distribuidos tolerantes a fallas sobre internet.
- De cualquier manera todavía existen límites a la escalabilidad debido fundamentalmente a la capacidad de las redes de comunicación, aunque, como es de preverse, este límite irá desapareciendo con el correr del tiempo.
- Es fundamental un framework de aplicación sencillo y de alto nivel que aisle al desarrollador de aplicaciones distribuidas de los problemas inherentes a la detección y el tratamiento de fallas. Problemas como la determinación de si un miembro falló o no, la distribución de los requerimientos de un cliente, la generación de identificadores únicos de requerimientos, el orden y propiedades de los multicast se simplifican enormemente al contar con un framework por encima de structured view synchrony.
- Un tema que afecta grandemente a la performance de la arquitectura es la correcta utilización del orden de multicast apropiado para cada aplicación. El elegir un orden incorrecto puede implicar desde inconsistencias en el estado (si el orden es menor del necesario) hasta penalizaciones en la performance (si es mayor). Los ordenes, recordamos, van desde uno que solo asegure entrega de acuerdo al orden de envío (FIFO order) hasta aquel que asegura un orden total de entrega entre todos los procesos intervinientes (TOTAL order).

Logramos en definitiva realizar un estudio profundo de la abstracción conocida como View Synchrony y de su aplicación para sistemas distribuidos tolerantes a fallas. Creemos que dicha aplicación además puede ser implantada en "pequeñas" máquinas globales compuestas por varias LANs siguiendo nuestro



diseño propuesto para Structured View Synchrony. De acuerdo a [GOU/94], se podría crear una máquina global de unas 20 LANS de 40 máquinas cada una.

Como tareas que han quedado afuera del alcance de nuestra tesis podemos enumerar:

- Aumentar la cantidad de órdenes de multicast disponibles, implementando, por ejemplo, alguno de los algoritmos de orden total.
- Tener en cuenta para el modelo del sistema, la posibilidad de particiones de la red, utilizando para ello el modelo de Enriched View Synchrony, propuesto por Alberto Bartoli et al en [BAB/96].
- Implementar sobre la WDVM de [EST/00] el modelo de tolerancia a fallas propuesto por nosotros.
- Cuestiones de optimización en nuestra implementación de la arquitectura.

Todas estas cuestiones pueden ser encaradas, ahora, tomando como base el trabajo realizado y descrito en esta tesis.



12 Bibliografía

[AGA/95] Deborah A. Agarwal, **Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks**, University of California Santa Barbara 1995.

[AMI/94] .Y Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, P. Ciarfella, **The Totem Single-Ring Ordering and Membership Protocol**, University of California, Santa Barbara, 1994.

[AMI/95] Yair Amir, **Replication Using Group Communication Over a Partitioned Network**, Thesis submitted for the degree "Doctor of Philosophy" Submitted to the Senate of the Hebrew University of Jerusalem, 1995.

[BAB/94] Ozalp Babaoglu Andre Schiper, **On Group Communication in Large-Scale Distributed Systems**, Technical Report UBLCS-94-19 July 1994 Laboratory for Computer Science University of Bologna.

[BA2/94] Ozalp Babaoglu, Renzo Davoli, Luigi-Alberto Giachini, Mary Baker, **Relacs: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems**, Technical Report UBLCS-94-15 June 1994.

[BAB/95] Ozalp Babaoglu Renzo Davoli Luigi-Alberto Giachini Mary Baker, **Relacs: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems**, Technical Report UBLCS-94-15 June 1994 Revised January 1995.

[BA2/95] Ozalp Babaoglu, Alberto Bartoli, Gianluca Dinni, **On programming with view synchrony**, Technical Report UBLCS -95-15, September 1995.

[BAB/96] Ozalp Babaoglu Alberto Bartoli Gianluca Dini, **Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems**, Technical Report UBLCS-96-3 February 1996 Revised November 1996 Department of Computer Science University of Bologna.

[BAB/97] Ozalp Babaoglu Renzo Davoli Alberto Montresor, **Partitionable Group Membership: Specification and Algorithms**, Technical Report UBLCS-97-1 January 1997 Revised May 1997 Department of Computer Science University of Bologna.



[BA2/97] *Ozalp Babaoglu, Renzo Davoli, Alberto Montessor,* **Partitionable Group Membership: Specification and algorithms,** Technical Report UBLCS -97-1, January 1997.

[BAL/91] *Henri E. Bal, M. Frans Kaashoek Andrew S. Tanenbaum,* **The Amoeba distributed operating system – A status report,** Computer Communications, Vol. 14 pág. 324 – 335, Ago. 1991.

[BAL/92] *Henri E. Bal, M. Frans Kaashoek Andrew S. Tanenbaum,* **Orca, a language for parallel programming of distributed systems,** Dept of Mathematics and Computer sciences Vrije Univ Amsterdam, The Netherlands, Mar. 1992.

[BAR/96] *Baratloo A y otros, Charlotte:* **Metacomputing on the Web,** Proc of the 9th International Conference on Parallel and Distributed Computing Systems, Sep 1996.

[BAR/98] *Dr. Alberto Bartoli* Universidad de Pisa, **Sistemas Distribuidos Asincrónicos,** Apuntes ECI N2 1998.

[BIR/96] *Kenneth P. Birman,* **Building Secure and Reliable Network Applications,** Harbound, 1996.

[BRA/93] *Thomas Bräun,* **Parallel Programming,** Prentice Hall, 1993.

[CHR/97] *Bernd O. Christiansen y otros,* **Javelin: Internet Based Parallel Computing Using Java,** HYPERLINK <http://www.cs.ucsb.edu/~schauser/papers> <http://www.cs.ucsb.edu/~schauser/papers>, Jun. 1997.

[EST/00] *Javier Esteban, Marcelo Weinbinder,* **Infraestructura para una computadora virtual en un sistema distribuido global débilmente acoplado -** Universidad de Buenos Aires Facultad de Ciencias Exactas y Naturales. 2000.

[GEI/94] *Al Geist y otros,* **PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing,** MIT Press, 1994.

[GOU/94] *Katherine Guo, Werner Vogels and Robbert van Renesse,* **Structured Virtual Synchrony: Exploring the Bounds of Virtual Synchronous Group Communication,** Dept. of Computer Science, Upson Hall, Cornell University, Ithaca, NY 14853, U.S.A., 1994.



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

[GRI/94] Andrew S. Grimshaw, William A. Wulf, James C. French, Alfred C. Weaver, Paul F. Reynolds Jr, **Legion: A world wide virtual computig**, UVa CS Technical Report CS-94-20, Jun 1994.

[GRI/97] Andrew S. Grimshaw, Mark Lewis, **The core Legion Object Model**, Departament of Computer Science University of Virginia, 1997.

[GUE/96] Rachid Guerraoui Andre Schiper, **Fault-Tolerance by Replication in Distributed Systems**, Departement d'Informatique Ecole Polytechnique Federale de Lausanne 1015 Lausanne, Switzerland, 1996.

[LAD/95] Oren Laden & others **The MOSIX Multicomputer Operating System for Scalable NOW and its Dynamic Resource Sharing Algorithms**, The Hebrew University of Jerusalem, Mar. 1995.

[LAN/97] Danny B Lange & Misuru Oshima **Programming Mobile Agents in Java with the Java Aglet API - Chapter 2 Anatomy of an Aglet**, HYPERLINK <http://www.trl.ibm.co.jp/aglets/aglet-book/anatomy.html>
<http://www.trl.ibm.co.jp/aglets/aglet-book/anatomy.html>, 1997.

[LAW/96] George Lawton, **Agents to roam the Internet**, HYPERLINK <http://www.sun.com/sunworldonline/swol-10-1996/swol-10-agent.html>
<http://www.sun.com/sunworldonline/swol-10-1996/swol-10-agent.html>, 1996.

[LEW/92] Ted G. Lewis, Hesham El-Rewini, **Introduction to Parallel Computing**, Prentice Hall, 1992.

[MAL/96] Christoph Malloth Andre Schiper, **View Synchronous Communication in Large Scale Networks**, Lambda Departement d'Informatique Ecole Polytechnique F'ed'erale de Lausanne CH - 1015 Lausannne (Switzerland), 1996.

[MOS/95] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, T. P. Archambault, **The Totem System**, Department of Electrical and Computer Engineering University of California, Santa Barbara, CA 93106, 1995.

[MOS/96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, **Totem: A Fault-Tolerant Multicast Group Communication System**, Department of Electrical and Computer Engineering University of California, Santa Barbara, CA 93106, 1996.



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.

[OSF/96] Open Software Foundation, **Introduction to OSF DCE Release 1.2.2**,
HYPERLINK <http://www.opengroup.org/dce> <http://www.opengroup.org/dce>,
1996.

[REN/94] *Robbert Van Renesse* *Kenneth P. Birman* *Takako Hickey*, **Design and Performance of Horus: a lightweight group communication system**, Cornell University, Agosto 1994.

[REN/95] *Robbert van Renesse*, *Kenneth P. Birman*, **Protocol Composition in Horus**, Dept. of Computer Science Cornell University \Lambda March 29, 1995.

[SAR/98] *Luis F. G. Sarmenta*, **Bayanihan: Web-Based Volunteer Computing Using Java**, MIT, HYPERLINK <http://www.cag.lcs.mit.edu/bayanihan.html>
<http://www.cag.lcs.mit.edu/bayanihan.html>, 1998.

[SCH/96] *Andre Schiper* *Lambda Alain Sandoz*, **Uniform Reliable Multicast in a Virtually Synchronous Environment**, Departement d'Informatique Ecole Polytechnique Federale de Lausanne 1015 Lausanne, Switzerland, 1996.

[SET/99] HYPERLINK <http://www.seti.org/science/setiathome.html>
<http://www.seti.org/science/setiathome.html>, 1999.

[TA2/96] *Andrew S. Tanenbaum*, **Computer Networks Third Edition**, Prentice Hall, 1996.

[TAN/96] *Andrew S. Tanenbaum*, **Sistemas Operativos Distribuidos**, Prentice Hall, 1996.

[VER/97] *Paulo Vêrissimo* *Luis Rodrigues* *Werner Vogels*, **Group Orientation: a Paradigm for Modern Distributed Systems**, Tech. Univ. of Lisboa - IST - INESC Broadcast Technical Report Esprit Basic Research Project 6360, 1997.



Diseño de una nueva arquitectura de software para view synchrony estructurado y definición de un framework para su utilización.