TÉCNICAS DE DESARROLLO DE APLICACIONES DISTRIBUIDAS TOLERANTES A FALLOS SOBRE ARQUITECTURAS CORBA Y JAVARMI

Memoria que presenta para optar al grado de

Doctor en Informática

Luis Miguel Peña Cabañas

Dirigida por el doctor

Juan Pavón Mestras

Departamento de Sistemas Informáticos y Programación

Facultad de Informática

Universidad Complutense de Madrid

Febrero, 2002

Si lo puedes soñar, lo puedes hacer.

- Walt Disney

One ring to rule them all,
One ring to find them,
One ring to bring them all
and in the Darkness bind them

- J. R. R. Tolkien

(ejemplo de diseño centralizado,
no tolerante a fallos)

A todos los que os he conocido durante estos largos años de doctorado. Lo largo ha hecho, al menos, que seáis muchos. Gracias por vuestra compañía y apoyo. A todos los sitios donde he vivido y disfrutado mientras Sensei iba evolucionando. En especial, por los recuerdos que me traen en relación con esta Tesis, Brujas, Amberes, Dublin, y un trayecto Frankfurt-Stuttgart tantas veces recorrido.

Resumen

Las técnicas de tolerancia a fallos en arquitecturas distribuidas cliente/servidor (como CORBA, Java RMI o Microsoft DCOM), que a nivel software se implementan principalmente mediante la replicación de servidores para ocultar fallos individuales, presentan una interfaz de muy bajo nivel que hacen difícil la implementación de los servicios tolerantes a fallos. El rendimiento de estos sistemas, que deben actualizar en cada operación dos o más servidores en lugar de sólo uno, y cuya actualización debe realizarse con una sincronización muy cuidada para evitar inconsistencias, es sensiblemente inferior al de los sistemas no tolerantes a fallos. Por esta razón, las principales líneas de investigación en esta área han buscado las formas de optimizar esas comunicaciones para obtener sistemas con un rendimiento *prác*tico.

Sin embargo, otras líneas de investigación se han centrado en el estudio de patrones de implementación que faciliten el empleo de las técnicas subyacentes de tolerancia a fallos. El sistema propuesto en esta tesis, *Sensei*, se encuentra en este campo, estudiando cómo aplicar estas técnicas a grupos de objetos y cómo definir una interfaz de alto nivel que permita un cómodo empleo de los modelos de comunicaciones fiables entre servidores replicados.

Por una parte, *Sensei* trata el problema de la transferencia de estado entre réplicas, un aspecto de la replicación cubierto en la teoría básica de *sincronía virtual* pero con muy poca cobertura en las implementaciones actuales. Definimos un modelo que cubre desde los protocolos de bajo nivel especificando los mensajes y los rendimientos en diversos sistemas, hasta los protocolos de alto nivel especificando su interfaz en lenguaje IDL de CORBA. Más importante, estudiamos las condiciones que deben cumplir las aplicaciones para poder emplear los distintos modelos de transferencia de estado y cómo afectan al modelo de *sincronía virtual* sobre el que se construyen las aplicaciones replicadas.

Estos protocolos se implementan parcialmente sobre uno de los sistemas de comunicaciones fiables más conocidos, *Ensemble*. No obstante, hemos desarrollado también un sistema de comunicaciones fiables propio, *SenseiGMS,* persiguiendo definir una interfaz general, común a los modelos actualmente existentes, sobre la que implementamos totalmente los protocolos desarrollados.

Una vez resuelta la transferencia de estado, *Sensei* se centra en los patrones de comunicación de alto nivel, que permiten a aplicaciones orientadas a objetos mantenerse en el mismo nivel de abstracción al replicar esos objetos. El sistema tradicional de comunicaciones en los sistemas de comunicaciones fiables entre réplicas es el intercambio de mensajes entre esas réplicas, mientras que en *Sensei* hemos desarrollado una aplicación, *SenseiDomains*, que permite factorizar esas comunicaciones a nivel de componentes que pueden definirse dinámicamente en dominios, buscándose de esta manera la abstracción de objetos.

Además, se observan los patrones más empleados al diseñar las aplicaciones con tolerancia a fallos, implementándolos o soportándolos en la medida de lo posible. Una aplicación directa de este soporte de los patrones de implementación es la posibilidad de replicar automáticamente aplicaciones diseñadas inicialmente como entidades autónomas sin replicación. Sensei define la forma de migrar esas aplicaciones para soportar tolerancia a fallos y formas posteriores de optimizar las comunicaciones entre las réplicas resultantes. Para soportarlo se ha desarrollado la herramienta *SenseiUMA*.

Como ejemplo de aplicación de la metodología propuesta, se muestra el diseño de otra de las piezas de la arquitectura, *SenseiGMNS*, servicio de gestión de grupos cuyos componentes están replicados, a su vez, sobre *Sensei*.

Palabras clave

Sistemas distribuidos, CORBA, JavaRMI, Tolerancia a Fallos, Replicación de Objetos, Transferencia de Estado, Comunicaciones en grupo.

Summary

Fault Tolerance software techniques for distributed client/server architectures (like CORBA, JavaRMI or Microsoft DCOM) are usually based on the replication of the servers to hide single failures. However, these techniques work at a low level, making difficult the implementation of fault tolerant services. These systems must update on each request two or more servers instead of just one, and the multiple updates must be carefully synchronized to avoid inconsistencies. As a result, the performance on those systems is considerably worse than the equivalent non fault tolerant case. This is the reason why the main research efforts on this area have been focused on the optimization of the communications between replicas to obtain systems with a *practical* performance.

Nevertheless, other line of work has taken a different approach, focusing on the implementation patterns that facilitate the use of the fault tolerant techniques. The system proposed in this Thesis, *Sensei*, belongs to this second area, studying the application of those techniques to groups of objects and the definition of a high level interface to facilitate the use of the reliable communication models.

On one side, *Sensei* addresses the problem of the state transfer between replicas, an aspect of the replication that, despite being covered by the *virtual synchrony* model, has usually a poor coverage on current implementations of this model. We define a model to solve this issue, which addresses low level protocols, specifying the messages and the performance on a variety of systems, and high level protocols, defining their interface on CORBA IDL. What is more important, we study the conditions that the applications must hold in order to use the different state transfer models, and how do they affect to the virtual *synchrony* model.

These protocols are partially implemented on top of one well-known group communication system, *Ensemble*. Nevertheless, we have developed as well our own reliable group communication system, *SenseiGMS*, with a generic interface, common to the existing ones, and the protocols are completely implemented over it.

Once the state transfer issue is solved, *Sensei* focuses on the communication patterns that happen at high level, which allow object-oriented applications to keep on the same abstraction level when those objects are replicated. The basic traditional communication entity on reliable group systems is the message, and replicas must communicate among themselves using this low level mechanism. In *Sensei*, we have developed an application, *SenseiDomains*, that factorizes those communications into component interactions, components that can be dynamically defined, offering a standard object orientation abstraction.

Additionally, we have studied the design process of fault tolerant applications, catching the most usual implementation patterns, in order to implement or support them. A direct benefit of this support is the possibility to automatically replicate applications that have been initially designed as standalone. *Sensei* defines how to

migrate those applications to have fault tolerance, and how to optimize afterwards the communications between the created replicas. *SenseiUMA* is the tool designed to support this migration.

As an example of the proposed methodology, we show the design and implementation of one of the architectural elements in *Sensei*, called *SenseiGMNS*. This is a group membership service that is itself replicated following the design principles and with the support of *SenseiDomains*.

Keywords

Distributed systems, CORBA, JavaRMI, Fault Tolerance, Object Replication, State Transfer, Group Communications.

ÍNDICE

Capítulo 1 - INTRODUCCIÓN

El enorme crecimiento de Internet durante los últimos años supuso la aparición de numerosas compañías cuya presencia comercial se limitaba a este medio, y la necesidad de las compañías tradicionales de ofrecer también sus servicios en *la red*. Esta modalidad de servicios implica una fuerte dependencia de esas compañías en Internet y los medios empleados, de tal forma que ataques cibernéticos pueden suponer la caída del servicio y conllevar pérdidas económicas millonarias. Este problema es mayor cuando a la caída del servicio se añade la más importante de pérdida de datos, debido, por ejemplo, a la destrucción del medio físico empleado para su almacenamiento. Además de problemas externos, un servidor puede necesitar tolerar errores más comunes, como pueda ser un fallo en el sistema operativo que emplea, o en el ordenador donde se ejecuta, que provocan igualmente la pérdida del servicio. La solución genérica a este problema de tolerancia a fallos es la replicación de recursos: almacenar los datos en diferentes localizaciones, geográficamente distantes, o duplicar los servidores empleados.

El desarrollo de un servicio que se ofrece simultáneamente desde diferentes servidores implica un problema adicional sobre ese servicio, ya que debe resultar consistente a los clientes, ofreciendo en cada momento resultados coherentes desde cualquiera de los servidores. Esta consistencia supone normalmente una necesidad de comunicaciones entre los servidores, que deben sincronizar sus accesos y modificaciones de los datos. El resultado es un mayor empleo de recursos y un peor rendimiento en comparación con el mismo servidor no replicado, de tal forma que la mayor disponibilidad del servicio no justifica en muchas ocasiones la pérdida de

eficiencia. La solución en este caso es el empleo de replicaciones pasivas, donde hay un único servidor activo y uno o más pasivos que sólo se activan en caso de caída del primero. Sin embargo, hay aplicaciones que pueden tolerar esa pérdida de rendimiento y servicios que no pueden soportar la pérdida de disponibilidad, por corta que sea en el caso de replicaciones pasivas, como puede ser el caso de dispositivos esenciales en un avión.

El diseño de un servidor replicado activamente no supone únicamente el empleo de comunicaciones entre todas las réplicas, sino un control de esas comunicaciones que pueden presentar también problemas. Existen abundantes sistemas de comunicaciones en grupo que facilitan esta tarea, primando generalmente la optimización de esas comunicaciones, con un modelo denominado *de sincronía virtual* basado en el envío fiable de mensajes multipunto. Sin embargo, el bajo nivel de esta aproximación implica un diseño de aplicaciones estructurado en mensajes, dificultando el empleo de soluciones de alto nivel, como pueda ser la reutilización de componentes software.

Un servidor replicado es un caso específico de aplicación distribuida, para las que existen modelos y arquitecturas que soportan paradigmas de programación distribuida más avanzados que ese intercambio directo de mensajes. Ejemplos de estas arquitecturas son CORBA, Microsoft DCOM o JavaRMI. Precisamente la arquitectura CORBA se ha visto enriquecida recientemente con la especificación de un servicio de tolerancia a fallos soportando replicaciones activa y pasiva. Este servicio permite la creación de servidores replicados empleando técnicas de alto nivel como separación de interfaz e implementación, orientación a objetos, compatibilidad de implementaciones sobre plataformas y lenguajes, y soporte de otros servicios avanzados que permiten emplear transacciones, entre otros.

El trabajo sobre el que versa esta tesis comparte dominio con este servicio CORBA de tolerancia a fallos, ya que soporta un modelo de replicación activa de alto nivel, especificado a nivel interfaz de objetos. Sin embargo, los modelos de replicación son muy diferentes. *Sensei* se centra en la replicación activa e implementa un sistema que soporta JavaRMI además de CORBA. Por otra parte, la granularidad en la replicación es también muy diferente; mientras CORBA, con gran soporte para replicación pasiva, debe tomar el servidor completo como unidad de replicación, nuestro trabajo se basa en la factorización del servidor en varios componentes que pueden ser replicados.

El principal objetivo de esta tesis es la integración del modelo de bajo nivel tradicionalmente empleado para el desarrollo de aplicaciones replicadas con los modelos de alto nivel empleados actualmente en el diseño e implementación de aplicaciones distribuidas, aportando una metodología que facilite la replicación activa de servidores. El resultado que se pretende es simplificar el proceso necesario para esa replicación, facilitando por lo tanto el desarrollo de servidores tolerantes a fallos. Como objetivo secundario, pero totalmente necesario para la consecución del anterior, Sensei extiende los mecanismos de transferencia de estado definidos en el

modelo de sincronía virtual, esenciales para la consistencia entre servidores replicados.

Para alcanzar estos objetivos, hemos considerado componentes individuales (definidos por las interfaces que implementan) como la unidad de replicación, lo que ha supuesto dos líneas de investigación.

Por un lado, hemos analizado la creación de esos componentes, lo que supone no sólo imponer un modelo orientado a objetos sobre una interfaz de menor nivel basada en el paso de mensajes, sino también la necesidad de protocolos que garanticen la consistencia de esos componentes, ya que son instanciados sobre diferentes réplicas. La abstracción así conseguida es la de un único componente compartido por esas réplicas.

Por otro lado, hemos considerado la integración de esos componentes para construir aplicaciones tolerantes a fallos. Un problema asociado a esta integración es el ciclo de vida especial de un componente replicado, que pretendemos equiparar al de un componente *normal* para facilitar su empleo. Otro problema importante es la concurrencia de acceso asociada a esa abstracción de componentes compartidos, lo que implica la necesidad de soluciones a los problemas de concurrencia derivados.

El planteamiento empleado para realizar estas tareas ha sido:

- Diseño de protocolos de transferencia de estado, directamente especificados sobre el modelo de sincronía virtual e implementables por lo tanto en los sistemas de comunicaciones en grupo ya existentes.
- Especificación de una interfaz de transferencia de estado a alto nivel para los protocolos desarrollados. En especial, esta interfaz se ha integrado con la especificación del servicio CORBA de tolerancia a fallos.
- Trasladar el nivel de la interfaz empleada en el diseño de aplicaciones replicadas desde un modelo basado en intercambio de mensajes a un modelo basado en objetos e interacciones entre objetos.
- Desarrollo de un sistema propio de comunicaciones fiables en grupo, bajo el modelo de sincronía virtual, con soporte de las arquitecturas CORBA y JavaRMI para la especificación de interfaces y su implementación.
- Captura de patrones de diseño comúnmente empleados en aplicaciones replicables, con el objetivo de definir un entorno de implementación que los soporte.
- Desarrollo de una metodología de diseño de aplicaciones tolerantes a fallos basada en su descomposición en componentes, facilitando su implementación y la reutilización de software.
- Implementación de un sistema de soporte de la metodología desarrollada, permitiendo el diseño de aplicaciones basadas en componentes replicados.

- Definición de herramientas para la generación automática de componentes replicados a partir de su definición de interfaz y una implementación de esa interfaz sin soporte de replicación.

Esta memoria se ha organizado en doce capítulos, incluyendo esta introducción, y cuatro apéndices.

El capítulo 2 trata los sistemas distribuidos, explicando conceptos como *sockets*, arquitectura CORBA, o modelos de mensajería y cliente/servidor. Su nivel es sólo introductorio, siendo su lectura innecesaria en caso de poseer ya esos conocimientos básicos.

El capítulo 3 es también introductorio, esta vez sobre sistemas distribuidos fiables, enfocado en los conceptos fundamentales de fiabilidad y tolerancia a fallos. Estudia los problemas asociados a las comunicaciones en grupo, y detalla las bases para obtener comunicaciones multipunto fiables. Expone a continuación el modelo de sincronía virtual, sobre el que se ha desarrollado este trabajo, y los problemas de observabilidad sobre aplicaciones replicadas.

En el capítulo 4 se describen los sistemas existentes soportando comunicaciones fiables en grupo. Esta descripción es detallada en los mecanismos de transferencia de estado entre miembros del grupo que esos sistemas soportan, al ser estos mecanismos una parte importante de esta memoria. Se estudian dos sistemas en mayor profundidad: el servicio de tolerancia a fallos de CORBA, por su importancia actual, y *Ensemble*, desarrollado por el creador del modelo de sincronía virtual y base inicial de este proyecto.

El capítulo 5 es el primero de tres dedicado al estudio de la transferencia de estado. Trata, desde una perspectiva formal, de los problemas de la transferencia de estado entre miembros de un grupo bajo el modelo de sincronía virtual. Estudia las condiciones que debe verificar una aplicación replicada para poder emplear un determinado tipo de transferencia, poniendo especial interés en el caso de transferencias en varios pasos. Finalmente, propone algoritmos genéricos que solucionan los problemas expuestos.

En el capítulo 6 se desarrollan los protocolos de bajo nivel necesarios para soportar transferencias de estado flexibles y eficientes. Se estudian diferentes tipos de protocolos y se realizan comparativas entre esos tipos que permitan elegir el protocolo adecuado a una determinada aplicación o configuración topológica.

En el capítulo 7 se estudia de nuevo el soporte de transferencias de estado flexibles y eficientes, pero ahora desde el punto de vista de la interfaz de aplicación. Esta interfaz contempla problemas tales como la concurrencia de transferencias, la elección del miembro que las coordina o el bloqueo de la aplicación durante las transferencias. También se estudia la implementación de las interfaces propuestas sobre los protocolos de bajo nivel desarrollados en el capítulo 6.

En el capítulo 8 se describe *SenseiGMS*, un sistema de comunicaciones fiables desarrollado específicamente para este proyecto, que soporta, mediante una interfaz

CORBA o JavaRMI, el desarrollo de aplicaciones replicadas sobre el modelo de sincronía virtual. Describimos esta interfaz, que define una funcionalidad estrictamente limitada al modelo de sincronía virtual, y los algoritmos con que se implementan, comparándolos con otros algoritmos existentes. Explicamos a continuación cómo emplear y configurar SenseiGMS e introducimos finalmente *VirtualNet*, un sistema desarrollado para probar aplicaciones replicadas, enfocado en los problemas de comunicaciones entre réplicas.

El capítulo 9 parte del modelo orientado a objetos soportado por SenseiGMS para desarrollar una metodología de desarrollo de aplicaciones tolerantes a fallos. Esta metodología cubre patrones de diseño, como la sincronización requerida entre solicitud y respuesta en los accesos a servidores replicados, o la transformación de operaciones sobre estos servidores en mensajes al grupo, y estudia la automatización del proceso de replicación y sus limitaciones, enfocándose entonces en el empleo de componentes. La metodología estudia la integración de estos componentes y los problemas de concurrencia que aparecen, incorporando soluciones como monitores o transacciones. Por último, comparamos el modelo desarrollado con el del servicio CORBA de tolerancia a fallos.

El capítulo 10 describe *SenseiDomains*, una implementación de la anterior metodología desarrollada sobre SenseiGMS. Describimos su interfaz y soporte, y se discuten los aspectos y decisiones de implementación que afectan a la metodología.

En el capítulo 11 se describe una aplicación integral de Sensei, denominada *SenseiGMNS* y que completa la implementación del modelo de sincronía virtual. Su diseño se realiza sobre SenseiDomains, con lo que sirve además para mostrar un ejemplo de aplicación de la metodología desarrollada.

Finalmente, el capítulo 12 muestra las conclusiones de nuestro proyecto y sus actuales líneas de investigación.

Esta memoria se completa con cuatro apéndices. Tres listan las interfaces de alto nivel de las diferentes partes del proyecto: SenseiGMS, SenseiDomains, SenseiGMNS, y el último muestra ejemplos de componentes replicados.

Capítulo 2 - SISTEMAS DISTRIBUIDOS

El tema de sistemas distribuidos abarca un amplio conjunto de conceptos cuya explicación difícilmente puede reducirse a un capítulo introductorio. Nos limitamos aquí a exponer una serie de términos y conceptos que empleamos a lo largo de esta memoria, detallando igualmente otros por su posible asociación con los expuestos.

La forma más elemental de interpretar un sistema distribuido es la de un sistema computacional compuesto por diferentes procesadores interconectados. Normalmente, esta interconexión estará soportada por una red abierta, basada en un conjunto de protocolos estándar que permita [Schmidt95a] la colaboración entre aplicaciones, escalabilidad y portabilidad.

Esta interpretación no es, sin embargo, completa; los componentes de una aplicación distribuida pueden residir en la misma máquina o en distintos nodos de la red y, por lo tanto, al hablar de las interconexiones no se trata tanto de que se produzcan a través de enlaces hardware, sino de comunicaciones entre procesos.

Las siguientes secciones muestran los conceptos en los que nos centramos en esta introducción: modelos distribuidos, *sockets*, procedimientos remotos, objetos distribuidos y tendencias actuales.

2.1. Modelos distribuidos

En todo servicio pueden identificarse dos papeles: el servidor, que procesa una solicitud de servicio, y el cliente, que la envía. La arquitectura cliente/servidor [Berson96] es un modelo distribuido donde estos papeles se encuentran claramente diferenciados. Un ejemplo es un navegador de Internet: el navegador es el cliente que accede a una página Web, suministrada por un servidor *http*.

En general, el término cliente/servidor se asocia a arquitecturas centralizadas, donde una determinada máquina presta un servicio específico. Sin embargo, un servidor puede actuar a la vez como cliente de otro servidor. En especial, dos componentes cualesquiera de una determinada aplicación pueden actuar simultáneamente como cliente y servidor, mutuamente entre sí. En este caso, tenemos una aplicación P2P (par a par) [Oram01], cuya principal característica es su descentralización. Un ejemplo muy conocido es *Napster*, un sistema de intercambio de ficheros[1] donde éstos no residen en un servidor central, sino en cada una de las máquinas conectadas al servicio. En este caso concreto, existe todavía un servidor central que mantiene las listas de ficheros, pero la carga de trabajo sobre este servidor es mucho menor que si tuviera que albergar y suministrar esos ficheros.

Otra alternativa es el modelo de mensajería o eventos, un modelo distribuido donde, aunque existen servidores y clientes, éstos no se encuentran directamente acoplados, es decir, el cliente no accede directamente al servidor para obtener la información. En su lugar, se emplean entidades intermedias, generalmente denominadas canales o colas de mensajes o eventos [Schmidt97]. En este caso, el servidor envía su información en un mensaje, que se mantiene en una cola donde puede ser accedido por el cliente o clientes. Este desacoplamiento entre servidor y cliente implica que un determinado canal de información puede ser accedido por múltiples clientes, pero también puede ser suplido por múltiples servidores. Y resulta especialmente útil para aplicaciones cuyos componentes no están permanentemente conectados. Este paradigma se emplea en el caso de correo electrónico, donde entre el emisor y el receptor de un correo existe una cola de mensajes.

En este último caso la arquitectura sigue siendo cliente/servidor. Simplemente se introduce una nueva entidad, la cola de mensajes o canal de eventos, que actúa como cliente para los productores de eventos y de servidor para los consumidores.

Existen otros modelos distribuidos, menos empleados o de dominio más restringido. Por ejemplo, *JavaSpaces* [Freeman99] define una estructura de datos

[1] Los ficheros contienen música codificada en formato MP3; el intercambio indiscriminado de esta música no respetaba los derechos de autor provocó demandas judiciales y el cierre del servicio, por lo que no nos es posible indicar en este momento una referencia a su página Web.

accesible por las distintas aplicaciones distribuidas que no es mas que una colección de datos con propiedades transaccionables. Aunque se desarrolla sobre JavaRMI (un sistema que explicamos a continuación), la abstracción que ve el desarrollador es simplemente un diccionario (*HashMap*) donde puede escribir o leer datos, datos que son entonces disponibles por las demás aplicaciones, sin necesidad de acceder al modelo de mensajes de JavaRMI. Este paradigma de programación distribuida no es original de *JavaSpaces*, está fuertemente influenciado por los sistemas *Linda* [Gelernter85].

Otro ejemplo lo ofrece *JavaParty* [Philippsen97]. En este caso, la máquina virtual de Java se implementa de forma distribuida, de tal forma que los objetos declarados remotos pueden migrar sobre las distintas máquinas que integran el entorno distribuido, para lo que hace falta modificar y extender la semántica de las operaciones. Aspectos inherentes a JavaRMI como el registro y publicación de las identidades de los objetos remotos quedan ocultos al desarrollador.

2.2. Sockets

La comunicación entre clientes y servidores, bajo cualquiera de los modelos distribuidos, se realiza a bajo nivel sobre un determinado protocolo de red, siendo TCP/IP el más empleado. Cada dispositivo en red recibe una dirección IP única (al menos en el ámbito de esa red) que la identifica para sus comunicaciones. Un punto final de comunicaciones queda definido por la dirección IP y un número de puerto, es decir, un mismo dispositivo puede tener múltiples líneas de comunicación abiertas, al menos una por cada puerto empleado.

La popularidad actual de TCP/IP se debe a Internet, que lo emplea como protocolo de red, y ha sido el protocolo empleado en máquinas Unix desde su inicio. A principios de los ’80 la distribución Unix de *Berkeley* introdujo el modelo de *sockets* como un mecanismo de comunicación entre procesos [Leffler89], que se ha convertido en el estándar *de facto* para programación en red sobre TCP/IP. Sin embargo, su API (interfaz de programación de aplicaciones) puede en principio usarse con otros protocolos de red.

Un *socket* [Stevens90] es un punto final de comunicación, identificado en TCP/IP mediante la dirección IP y un puerto. Existen dos tipos de sockets: orientados a conexión (TCP) y sin conexión, también llamados datagramas (UDP). TCP crea un circuito virtual entre los procesos que comunica, por lo que los sockets sobre TCP se consideran fiables. Los datagramas no son fiables ni se asegura el orden o no duplicación de los datos enviados, pero permiten el envío de mensajes *broadcast*, a más de un destino final.

Un servidor que emplea *sockets* debe asociarse a una determinada dirección, donde espera continuamente la llegada de datos. Un cliente debe conocer cuál es esa dirección específica para enviarle datos. La información que se transmite no tiene

ningún significado para los *sockets*, que actúan únicamente como puntos de entrada y salida para las comunicaciones. Es la aplicación la que debe interpretar esos datos y producir, posiblemente, una respuesta.

2.3. RPC – Llamadas a procedimientos remotos

Un ejemplo sencillo de aplicación cliente/servidor es el de un servicio de autentificación: un cliente suministra un nombre (*login*) y una clave (*password*), y el servicio comprueba su validez. Empleando *sockets*, el cliente debe conectarse al servidor y enviar en una cadena de bytes la información necesaria, el nombre y la clave, que suponemos que son cadenas de caracteres. No existe ningún requisito sobre cómo enviar esa información y el servidor debe especificar qué formato espera. Por ejemplo, un primer byte que indique la longitud del nombre, seguido por tantos bytes como caracteres tenga el nombre, y luego otro byte que indique la longitud de la clave y tantos bytes como caracteres tenga esa clave. Es responsabilidad del cliente el aplicar correctamente el formato esperado. Y este formato debe especificarse con mayor detalle: qué orden de bytes se espera, *big-endian* o *little-endian*, qué codificación de caracteres, ASCII o EBCDIC, etc. Además, la aplicación debe gestionar los errores de comunicaciones. Desde el punto de vista del servidor [Schmidt95b], si precisa soportar varios clientes simultáneamente, es también la aplicación la que debe incluir toda la lógica de concurrencia y de gestión de múltiples clientes.

Una librería puede soportar el formateo/deformateo de determinados tipos de datos, definiendo cómo transferir cadenas de caracteres, tipos enteros, etc. Si tanto el servidor como el cliente emplean la misma librería, parte de los anteriores problemas se solucionan. RPC [White75] suministra este soporte de librería, a la vez que realiza una abstracción de llamadas a procedimientos [Birrel84]. Siguiendo el ejemplo anterior, el servidor puede especificarse como una función definida de la siguiente manera:

```
int validate (in char* login, in char *password);
```

Al emplear un compilador RPC sobre esta definición, se generan dos porciones de código. Una se llama *stub* del cliente, y lo que hace es proporcionar un procedimiento con la misma definición dada. El cliente invoca este procedimiento [figura 2.1], y éste automáticamente prepara la cadena de bytes a enviar al servidor, formateando los datos y enviándolos a través del *socket*. La segunda porción de código se denomina *stub* del servidor, y verifica continuamente el socket donde recibe la información, que deformatea y envía al servidor. Cuando se elabora la respuesta, ésta se envía por el camino inverso.

De esta forma, se accede al servidor como si fuera local, al que le invoca mediante procedimientos, tal como si fuera una librería del sistema.

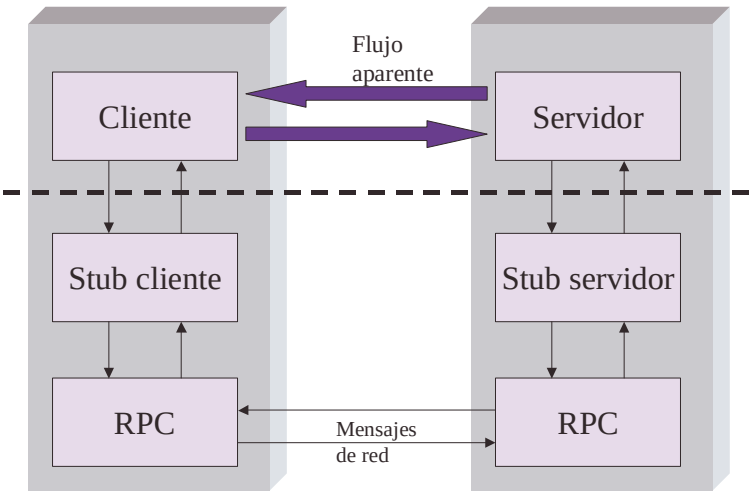

Figura 2.1. Filosofía de RPC

2.4. Objetos distribuidos

La evolución de los lenguajes de programación hacia el paradigma de orientación a objetos supone que el procedimiento ya no es la estructura básica de un programa, que pasa a estructurarse en objetos. Como extensión, los sistemas RPC evolucionan para realizar la abstracción de objetos remotos.

JavaRMI (*Remote Method Invocation*) [Dwoning98] soporta esta abstracción sobre objetos Java. Un objeto distribuido es accesible de forma remota a través de su interfaz, definida en Java. El compilador RMI genera a partir de esta interfaz, al igual que en RPC, porciones de código que simulan el acceso local de los objetos remotos. En este caso, el cliente emplea *stubs* o *proxies*, y al servidor se le asocian los *skeletons*.

Además de realizar una abstracción de objetos remotos, JavaRMI proporciona otras facilidades a la aplicación. Por ejemplo, dispone de un servicio de directorio, denominado *registro*, donde las aplicaciones pueden publicar sus servidores, asociándolos a un nombre. Un cliente no necesita ahora conocer la localización exacta del servidor, basta con que acceda a este servicio para obtener una referencia al objeto remoto. Otra facilidad es el concepto de objeto remoto persistente, que no está activo continuamente y se activa cuando un cliente lo invoca, lo que facilita el escalado de las aplicaciones.

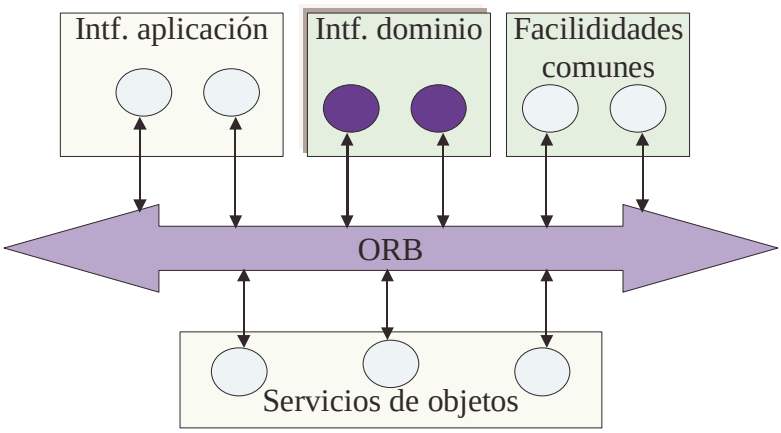

Figura 2.2. Modelo de arquitectura OMA

Aunque hemos descrito primero por su mayor simplicidad JavaRMI, CORBA (*Common Object Request Broker Architecture*) [OMG98] es una arquitectura anterior, diseñada por el OMG (*Object Management Group*) con el mismo objetivo básico: manejar objetos distribuidos de forma transparente.

En JavaRMI, todo el sistema se programa con un único lenguaje, mientras que CORBA soporta actualmente (como estándar) nueve: *C*, *C++*, *Java*, *Cobol*, *Smalltalk*, *Ada*, *Lisp*, *Python* y *CORBAScript*. Por esta razón, la definición de la interfaz de los objetos remotos o componentes se hace en un lenguaje propio, OMG/IDL. Un compilador de IDL, específico para cada lenguaje, genera de nuevo los *stubs* y *skeletons*.

La mayor complejidad no se debe sólo a la variedad de lenguajes soportado, sino a la funcionalidad que implementa [Orfali97]. Parte de esta funcionalidad se incluye en el ORB, que es el gestor de objetos distribuidos, pero la arquitectura [figura 2.2] emplea elementos opcionales, como los servicios CORBA [OMG97], que implementan funcionalidad genérica, o las facilidades CORBA, que implementan funcionalidad asociada a determinados tipos de aplicaciones. La figura 2.3 muestra las interacciones en el cliente y el servidor; sus comunicaciones se realizan a través del ORB y generalmente empleando los *stub* y *skeleton* generados. Sin embargo, existen otras opciones; por ejemplo, el cliente puede desconocer en tiempo de compilación cuál es la interfaz del servidor al que accede, en cuyo caso debe emplear la interfaz de invocación dinámica.

Un elemento importante en esta arquitectura es el adaptador de objetos, cuya finalidad es aislar al ORB del lenguaje empleado para implementar los servidores.

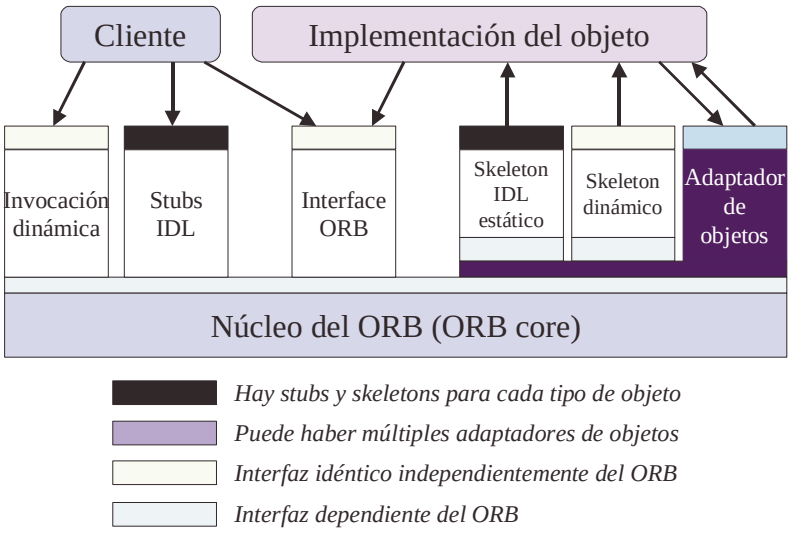

Figura 2.3. Interfaces ORB

Existen diferentes adaptadores, y el más habitual es el POA (*Portable Object Adapter*), que maximiza la portabilidad de las aplicaciones, a nivel de código, sobre diferentes implementaciones de ORB.

Empleando la notación de CORBA, se distingue entre *servant*, que es la implementación en un lenguaje específico de la interfaz que define un servicio, y *servidor* propiamente dicho, que es el objeto activo capaz de procesar las peticiones de servicio. Normalmente, existe un *servant* por cada servidor, pero una aplicación puede tomar otras alternativas, como que un *servant* encarne múltiples servidores. Los servidores pueden ser persistentes, en cuyo caso el ORB los debe activar si llegan peticiones de servicio, y la aplicación puede incluso emplear un *servant* para cada petición a un mismo servidor. Otra utilidad de esta distinción es que un servidor puede migrar a una máquina diferente (donde emplea evidentemente un *servant* diferente) de forma totalmente transparente para el cliente.

Una de las principales ventajas de CORBA es su interoperabilidad, que permite que objetos implementados en lenguajes, sistemas operativos y tipos de máquina diferente se comuniquen entre sí. No es preciso que los objetos empleen un mismo ORB, con tal que los empleados sean compatibles con la misma versión de CORBA. Es incluso posible que un objeto CORBA se comunique con un objeto JavaRMI, a partir de la especificación [OMG99] de RMI sobre IIOP, el protocolo estándar para comunicación entre ORBs sobre redes TCP/IP.

Existen otros gestores de objetos remotos. Por ejemplo, el sistema operativo Windows emplea uno propio, con un modelo de objetos distribuidos llamado DCOM *(Distributed Component Object Model*) [Sessions97]. Permite también la interacción de

componentes escritos en distintos lenguajes, pero siempre sobre el mismo tipo de máquina y sistema operativo. No es posible realizar a este nivel de introducción una comparativa entre los gestores expuestos; así, aunque a nivel arquitectural, CORBA está más maduro, existen otras consideraciones que pueden inclinar la elección de un gestor de objetos hacia DCOM o JavaRMI [Chung98, Gopalan98, Curtis97, Juric00].

2.5. Tendencias actuales

Empleando *sockets*, un servidor espera las solicitudes de servicio en una dirección IP determinada y un puerto específico. Ese servidor puede desarrollarse empleando JavaRMI o CORBA, pero aun así está disponible en un puerto específico. La tendencia actual a ofrecer servicios a través de Internet implica que las compañías instalan máquinas que son accesibles públicamente, lo que supone la posibilidad de ataques maliciosos donde un *hacker* es capaz de acceder a la máquina en su totalidad y leer información privada. Una respuesta a este problema es ocultar las máquinas tras *firewalls*, que controlan el flujo de datos entrante y saliente; la seguridad que se obtiene es mayor cuanto menos puntos de acceso tenga una máquina, ya que cada puerto *abierto* (esto es, donde un servidor espera solicitudes de servicio) es un punto débil del sistema.

Un sistema seguro deja el mínimo número de puertos abiertos, y sólo aquellos cuyo protocolo se considere seguro. Por ejemplo, un servidor Web debe ser accesible de forma estándar en el puerto 80. Una implicación directa es la dificultad de ofrecer servicios públicos mediante servidores CORBA o JavaRMI que esperen directamente las solicitudes de servicio. Una alternativa factible recibe el nombre de *http tunneling*. Significa que las solicitudes se reciben a través del puerto 80 (*http*), como si fueran accesos Web normales; una vez recibidas, deben propagarse al servidor específico, resultando en su conjunto en una pérdida de rendimiento y funcionalidad.

J2EE, *Java Enterprise* [Allaramaju01], ofrece un amplio abanico de posibilidades en este sentido. Por ejemplo, con JSP (*Java Server Pages*), se accede a una página Web cuyo contenido es dinámico, dependiente del estado del servidor; puede recibir solicitudes de servicio que se propagan automáticamente a unos componentes Java denominados *Enterprise Java Beans* que procesan el servicio. Las comunicaciones entre estos componentes se realizan mediante JavaRMI (la última especificación emplea RMI sobre IIOP).

El contenido de una página Web ha sido normalmente HTML, un lenguaje basado en etiquetas que permite especificar el formato de los datos en el navegador. Un lenguaje *similar* es XML [W3C00], en cuanto a que está también basado en etiquetas, pero no tiene una estructura fija. Es posible definir esta estructura, lo que lo hace muy conveniente para el envío de datos. Como éstos se transfieren en

formato texto ASCII, pueden compartirse los datos con independencia de la plataforma software y hardware. Es decir, dos componentes pueden enviarse información en formato XML sin preocuparse por lenguajes de programación o el soporte hardware. Al definir su descripción XML, definen las estructuras de los datos que se intercambian, que pueden perfectamente identificar solicitudes de servicio.

XML no define protocolos o mecanismos para la transmisión de los datos. SOAP [W3C01] es un protocolo ligero basado en XML que permite la invocación de servicios remotos sobre *http*, y es la base de los servicios Web que están empezando a popularizarse. Al definirse sobre *http*, se pueden emplear todos los mecanismos actuales empleados para el manejo de páginas Web.

Todas las formas de comunicaciones mostradas están basadas a nivel de transporte en el empleo de *sockets*, pero el modo de acceder al servidor y a sus servicios se realiza ahora a un nivel muy superior, definiendo de alguna forma la interfaz de ese servidor. Con procedimientos y objetos remotos, la interfaz se enfoca en la forma de acceder al servidor, definiéndose las operaciones a realizar para invocar sus servicios, mientras que con XML, la interfaz se enfoca en la información a obtener de ese servidor.

2.6. Conclusiones

La elección de la arquitectura CORBA como marco de trabajo para la realización de esta tesis se debe esencialmente a que en este momento ofrece el modelo de programación distribuida más estándar, de amplia difusión en la industria, y que soporta el paradigma de orientación a objetos integrado en las prácticas de ingeniería de software actuales. Asimismo, los resultados obtenidos en este modelo son fácilmente trasladables a otras tecnologías similares. Prueba de ello es que también se han aplicado en JavaRMI con mínimos cambios (reutilizando gran parte del código de los sistemas implementados).

Capítulo 3 - Sistemas Distribuidos Fiables

Este capítulo realiza primero una introducción a los sistemas fiables y a continuación, se centra en los grupos dinámicos de réplicas como método para hacer fiable un sistema. Los grupos dinámicos se basan en el modelo de sincronía virtual, que emplea comunicaciones multipunto fiables, y en un servicio que maneja la lista dinámica de miembros del grupo. Este servicio y el modelo de sincronía virtual son descritos en detalle, así como la observabilidad del sistema: cómo es observado el grupo de componentes al actuar como un único servidor.

3.1. Fiabilidad

Esta sección introduce los principales conceptos asociados a sistemas fiables: tolerancia a fallos, fiabilidad y descripción de esos fallos. A continuación describe los sistemas distribuidos y los problemas adicionales que plantean respecto a aplicaciones monoprocesador, así como la forma de detectar y simplificar esos problemas añadidos. Finalmente, se describen los tipos de grupos de componentes y las bases para obtener comunicaciones fiables entre componentes distribuidos.

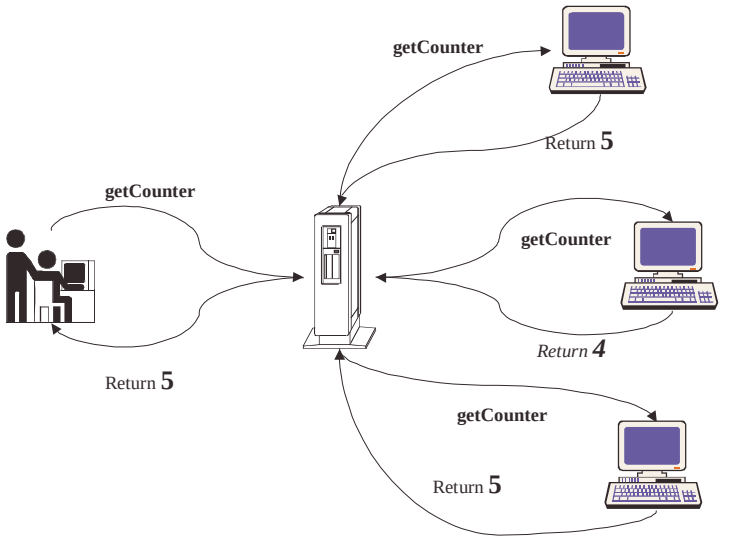

Figura 3.1. Soporte de errores en la respuesta usando replicación

3.1.1. Tolerancia a fallos

Todo servicio puede definirse a partir del conjunto de salidas que se producen a partir de unas entradas, definición que debe complementarse con la semántica de fallos del servicio, que especifica los errores posibles. Existen varias clasificaciones de estos errores [Christian91, Birman96]

- Fallo por omisión: el servicio no responde a una entrada o mensaje. Puede ser a su vez un fallo en la emisión del mensaje o en su recepción.
- Fallo de temporización: la respuesta llega fuera de un margen de tiempo especificado (márgenes mínimo y máximo). Este fallo puede darse por problemas ajenos a los componentes mismos como, por ejemplo, fallos en la red.
- Fallo en la respuesta: el servicio ofrece una respuesta incorrecta. En este grupo de errores entran los fallos bizantinos [Lamport82].
- Fallo por caída: el servicio se cae. Este error viene acompañado evidentemente por uno de los anteriores.

Estos fallos pueden ocurrir tanto en componentes hardware como en componentes software. En este último caso, los errores que provocan los fallos se clasifican en las dos categorías siguientes:

- *Bohrbugs*: fácilmente reproducibles, resulta fácil encontrarlos y fijarlos.

- *Heisenbugs*: difíciles de reproducir, resulta complicado encontrarlos y, consecuentemente, fijarlos.

La tolerancia a fallos de un sistema es su habilidad para recuperarse tras el fallo de algún componente. Otros dos conceptos relacionados son la fiabilidad y la disponibilidad del sistema: la fiabilidad de un servicio da la probabilidad de comportamiento correcto de ese servicio, y su disponibilidad es el porcentaje de tiempo en que el servicio puede proporcionarse.

Una forma de incrementar la tolerancia a fallos de un sistema es la replicación de sus componentes [Neumann56]. Si un componente se cae, el sistema puede aún emplear su réplica o réplicas. La forma en que estas réplicas deben trabajar depende de la semántica de fallos de los componentes. Por ejemplo, si pueden dar eventualmente respuestas erróneas (fallo en la respuesta), el servicio puede acceder a todas las réplicas y dar la respuesta mayoritaria [figura 3.1].

3.1.2. Sistemas distribuidos

Una aplicación puede tolerar fallos en la implementación de un algoritmo [Maguire93] empleando una implementación alternativa independiente, posiblemente no optimizada, que pueda comprobar los resultados dados por el primer algoritmo. Es decir, se emplean distintas implementaciones y posiblemente distintos algoritmos para evitar errores de codificación o de lógica en el algoritmo. Si esa aplicación debe tolerar fallos en el ordenador sobre el que se ejecuta, deberá procesarse en dos o más ordenadores y sincronizar de alguna manera las distintas instancias de esa aplicación, lo que lleva al concepto de computación distribuida. Este concepto implica un conjunto de programas fuertemente acoplados, ejecutándose en dos o más ordenadores y coordinando sus acciones.

Para conseguir que una determinada arquitectura tolere fallos en sus componentes, debe ser capaz de detectar cuándo esos componentes fallan, y esta detección está relacionada con el modelo distribuido:

- Sistemas síncronos: fuertemente dependientes de la temporización, todos sus componentes comparten una medida de tiempo; esta sincronización de los componentes permite dividir el tiempo en periodos (que todos los componentes ven iguales), siendo los mensajes enviados al principio de cada periodo y procesados al inicio del siguiente. Asume, por consiguiente, una perfecta coordinación de los componentes y sus acciones, coordinación que no existe en la inmensa mayoría de los sistemas reales. Detectar la caída de un componente es inmediato.
- Sistemas asíncronos: el concepto del tiempo desaparece, lo que impide realizar suposiciones sobre velocidades de la red, de los procesos, etc. No hay límites en los retrasos de los mensajes o en el tiempo necesario para ejecutar cualquier paso y es imposible detectar si un componente se ha caído o es simplemente muy

lento. Es un modelo irreal, que permite realizar simples abstracciones del sistema al eliminar el concepto de tiempo.

- Sistemas intermedios, denominados asíncronos reales o asíncronos parciales: existe el concepto de tiempo, y los mensajes entre componentes se realizan dentro de unos límites de tiempo. Esta definición es muy amplia, pues esos límites no tienen por qué ser conocidos o estables; según se definan, se obtienen distintos sistemas asíncronos parciales [Dwork88].

La detección de fallos está, consecuentemente, ligada a la sincronización del sistema, y cuanto más sincronizado sea éste, más factible será realizar una detección de fallos fiable. En un sistema real, cuando un componente envía un mensaje a otro, espera una respuesta en un tiempo máximo. Vencido este tiempo, el componente debe suponer un fallo, pero no puede decidir dónde se ha producido éste, pues el problema puede deberse a:

- Problemas en las comunicaciones: el mensaje no ha llegado al otro componente que, por lo tanto, no lo ha procesado, o bien ha llegado y lo ha procesado, pero no se ha recibido la respuesta.
- El componente se ha caído.
- El componente es muy lento, o está procesando muy lentamente el mensaje por otras causas.

En estas condiciones, el primer componente no puede saber si el segundo se ha caído o no y, en este último caso, si ha procesado el mensaje o no; por esta razón, los algoritmos para las comunicaciones entre estos componentes resultan muy complicados.

Una forma de simplificar los algoritmos en aplicaciones distribuidas es disponer de un sistema que gestione estos problemas y libere a esos algoritmos de la lógica asociada; ese sistema son los detectores de fallos.

3.1.3. Detección de fallos

Un detector de fallos comprueba que los componentes de un sistema no presentan errores, y si detectan un fallo, lo comunican a los demás componentes. Pero, al igual que un componente no puede determinar si otro ha fallado o no, el detector de fallos tampoco puede considerarse seguro y el sistema debe soportar una detección incorrecta de fallos de los componentes, para lo que hay dos opciones:

- La aplicación considera que el detector de fallos es perfecto; cualquier elemento que el detector de fallos considera sospechoso lo es para siempre, siendo excluido del grupo.
- La aplicación considera que el detector de fallos es imperfecto; sus algoritmos deben entonces ser capaces de progresar sabiendo que estas detecciones son

imperfectas, y no considerar excluidos a los componentes sospechosos. La lógica del grupo resulta mucho más complicada que en el primer caso.

La implementación de un detector de fallos se realiza generalmente mediante temporizadores. Cada componente emite mensajes, cuya recepción permite a los demás componentes comprobar que no se ha caído. Si la aplicación considera que el detector es perfecto, estos temporizadores deben ser lo suficientemente largos para que el número de fallos sea mínimo, puesto que sus fallos excluyen directamente a los miembros sospechosos. Existen otros métodos para detectar los fallos, basándose en características determinadas de un sistema o arquitectura, pero no pueden considerarse genéricas.

Un detector de fallos permite comprobar si un determinado componente se ha caído o no. Y un fallo en las comunicaciones con un componente puede verse como un fallo de ese componente. El problema surge en que el otro componente no se ha caído, pero observará recíprocamente un fallo en el primer componente, con lo que hay dos componentes que sospechan mutuamente uno del otro. Extendiendo este razonamiento, un problema real en los enlaces con los que un grupo de réplicas se comunica puede provocar la creación de dos o más subgrupos, sospechando cada uno que los demás están fallando: la partición de la red ha provocado una partición del grupo. Como además el detector de fallos es imperfecto, un enlace lento puede provocar el mismo resultado; en este caso, una partición virtual de la red puede suponer una partición del grupo.

3.1.4. Grupos de componentes

Una forma de incrementar la tolerancia a fallos de un sistema es la replicación de sus componentes: este grupo de componentes es visto como una entidad única, y un cliente que deba acceder al servicio prestado por ese componente accederá al grupo como un ente, sin conocer el número, identidad o localizaciones de los miembros individuales.

Estos grupos pueden ser estáticos o dinámicos; los grupos estáticos están formados por un conjunto predeterminado de elementos. Incluso cuando estos elementos se caen, siguen perteneciendo al grupo; por ello, si un elemento toma una decisión, todos los demás, incluidos los elementos caídos, están obligados a seguir esa decisión. Se supone, por lo tanto, que el elemento caído puede volver a levantarse, y debe haber guardado la información necesaria para completar esa decisión.

En los grupos dinámicos, la lista de elementos del grupo se considera dinámica, y la misma lógica del grupo contempla la inclusión y exclusión de miembros. Un elemento caído queda directamente excluido del grupo. Si un elemento toma una decisión, sólo los demás elementos que permanezcan funcionales deberán tomarla igualmente. Hay dos tipos de grupos dinámicos:

- Grupos dinámicos no uniformes: si un elemento toma una decisión pero se cae antes de comunicarla al grupo, no es preciso que los elementos activos restantes tomen esa misma decisión. Adicionalmente, si toma una decisión y se la comunica a una parte del grupo, pero tanto ese elemento como los que han recibido la comunicación se caen, los demás elementos activos no están obligados a tomar esa decisión. Por ejemplo, un cajero automático, en la extracción de pequeñas cantidades de dinero, no necesita comunicar esa extracción y luego procesarla. Si se cae, un registro contiene esa información, con lo que la transacción no se pierde.
- Grupos dinámicos uniformes: incluso si el elemento que tomó la decisión se cae, los demás elementos activos deben tomar la misma decisión. En el ejemplo anterior, para grandes cantidades de dinero, un cliente podría crear un gran descubierto si accede a varios cajeros que entregan el dinero y, antes de comunicar la transacción, el cajero correspondiente se cae.

Estos distintos tipos suponen distintos grados de consistencia, siendo la menor consistencia la alcanzada en los grupos dinámicos no uniformes. Los grupos dinámicos son más potentes y tolerantes a errores que los estáticos, pero resultan también más difíciles de implementar.

3.1.5. Comunicaciones fiables

Si un servicio se implementa mediante un grupo de componentes distribuidos para aumentar su tolerancia a fallos, esos componentes deberán estar en comunicación para preservar la consistencia del sistema, y que cada componente presente una visión coherente de ese servicio. Esas comunicaciones pueden ser:

- Comunicaciones punto a punto: un componente se comunica con el resto mediante mensajes individuales a cada uno de los miembros del grupo.
- Comunicaciones multipunto (*multicast*) no fiables, como IP *multicast*. Estas comunicaciones pueden resultar apropiadas para aplicaciones no críticas en la recepción de mensajes, donde la pérdida de un mensaje no afecta al estado de la aplicación. Por ejemplo, aplicaciones multimedia, donde la pérdida del mensaje puede simplemente suponer la falta de actualización en uno o varios *frames* de vídeo.
- Comunicaciones multipunto atómicas: se garantiza que o todos o ninguno de los componentes procesan la comunicación. Puede ser dinámicamente uniforme o no [Malki94]; si es uniforme, cuando un proceso procesa un mensaje, todos los demás procesos en estado operacional lo deben procesar
- Comunicaciones multipunto *best-effort*, que no se consideran atómicos (todos o ninguno), pues su objetivo es 'casi todos o casi ninguno'; a cambio, resultan muy escalables y más apropiados, consecuentemente, para grandes grupos. Por ejemplo, en el protocolo *multicast bimodal* [Birman98], los componentes se

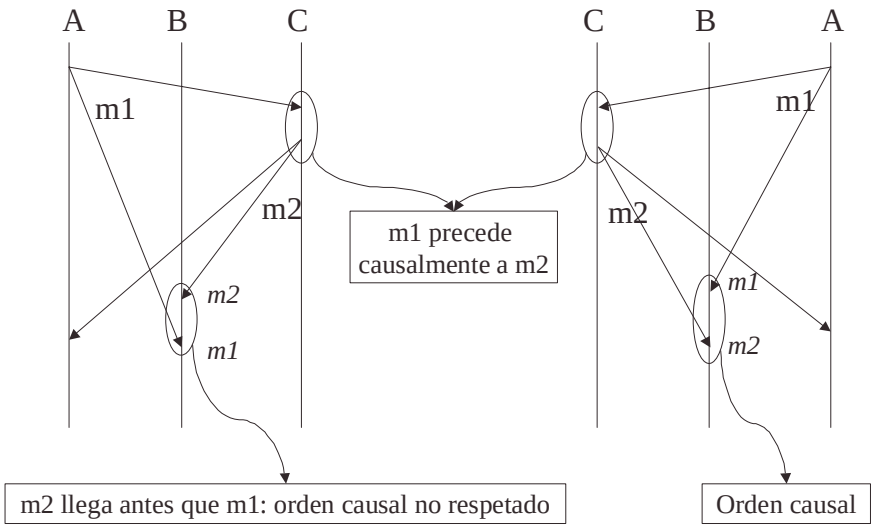

Figura 3.2. Orden causal

comunican periódicamente con otros para comprobar si han recibido los mismos mensajes; si existe algún desfase, se transmiten los mensajes que faltan. Tras un periodo de tiempo determinado tras la recepción de un mensaje, pueden procesarlo asumiendo que si hubiera habido algún desfase, ya se habría detectado; si la detección de un mensaje no recibido se realiza después de procesar algún mensaje posterior, la aplicación debe ser capaz de retroceder al punto inicial. Puesto que para detectar si faltan mensajes un componente se comunica aleatoriamente con otros, pero no con todos, la detección no es segura, y precisa de una mayor calidad en las comunicaciones: no es aplicable de forma general.

Además de las condiciones de atomicidad, las comunicaciones se caracterizan por el orden de procesado, es decir, el orden con que distintos componentes van a procesar los mismos mensajes:

- Procesar los mensajes sin orden, tal como llegan.
- Orden *fifo*: los mensajes enviados por un componente se ejecutan en el orden en que son enviados.
- Orden causal: los mensajes enviados por uno o varios componentes, que puedan relacionarse causalmente, se procesan de acuerdo a esta relación. Por ejemplo [figura 3.2], si un componente A envía un mensaje y un componente B, al recibirlo, envía un segundo mensaje, el primero precede causalmente al segundo, y donde ambos mensajes deban ser procesados, lo harán en el orden debido. Es, por lo tanto, un orden *fifo* incluyendo a varios componentes.

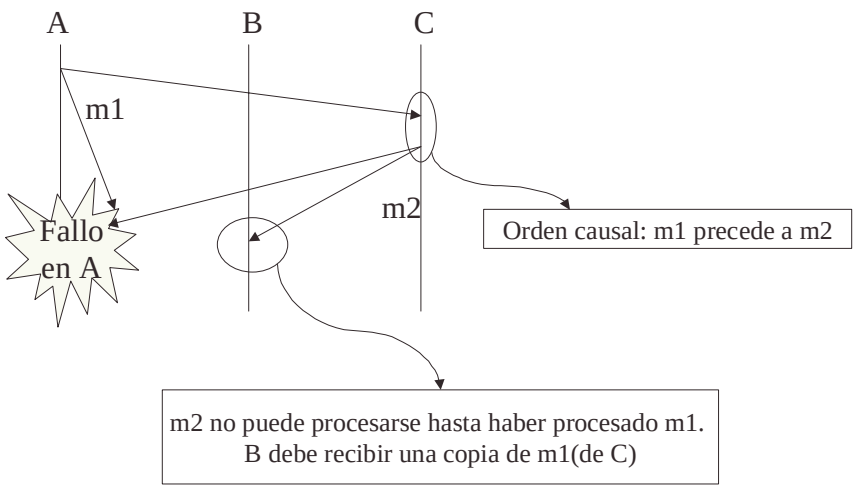

Figura 3.3. *Gap freedom*

- Orden total: todos los componentes del sistema que procesen un conjunto igual de mensajes, los procesarán en el mismo orden.
- Orden total causal: orden total preservando la causalidad. Si un componente envía dos mensajes, el orden total no implica que el primer mensaje enviado debe ejecutarse antes que el segundo, sino que ese orden de procesado será constante en los demás miembros del grupo: es decir, si todos los miembros procesan primero el último mensaje y luego el primero, el orden total queda todavía preservado. El orden total causal elimina esta posibilidad.

Según sea el tipo de orden asociado con el multicast, se producirán determinados errores de observación de los mensajes o eventos [Fidge96]:

- Observadores diferentes viendo diferentes ordenaciones: dos eventos concurrentes puedan ser vistos en distinto orden por dos procesos.
- Ordenaciones incorrectas: incluso cuando un mensaje precede causalmente a otro, este orden no es visto en un proceso dado (generalmente por retrasos en el envío de mensajes).
- Indeterminismo en la observación de los eventos: el mismo proceso en las mismas condiciones generales ve los eventos en distinto orden cuando se ejecuta varias veces.
- Ordenaciones arbitrarias: un proceso que recibe un evento antes que otro deduce que lo precede causalmente (cuando pueden ser concurrentes).

Las opciones de ordenación se denominan débiles o fuertes según sean dinámicamente uniformes o no (según involucren o no a los componentes que

fallan). Por último, hay una propiedad adicional relacionada con el orden de mensajes, denominada *gap freedom*: si el orden exige que un mensaje m_1 sea procesado antes que m_2, que m_2 se procese implica que m_1 también debe haberse procesado. La figura 3.3 muestra el resultado de aplicar esta propiedad; un miembro *A* envía antes de caerse un mensaje m_1, que alcanza al miembro *C* pero no a *B*. El miembro *C* envía antes de detectar la caída de *A* un mensaje m_2 que, consecuentemente, sucede causalmente a m_1. Cuando *B* lo recibe, no puede procesarlo; aplicando la propiedad de *gap freedom*, sólo puede procesarlo tras m_1, por lo que *C*, o cualquier otro miembro del grupo que lo hubiera recibido, debe enviarle primero una copia de ese mensaje.

3.2. Servicio de miembros de grupo (GMS)

Los miembros de un grupo dinámico deben permitir la inclusión de nuevos miembros o la exclusión de miembros actuales de la lista. Además de los problemas naturales de consistencia del grupo, se añade el manejo de las posibles inconsistencias que se pueden producir con el tratamiento de la lista de miembros. Por esta razón, los grupos dinámicos se apoyan en un servicio externo, el GMS (*Global Membership Service*), que realiza la gestión de la lista de miembros del grupo.

La interfaz básica de este servicio permite la inclusión de los componentes que soliciten incorporarse al grupo y la exclusión de miembros, bien porque lo soliciten o porque se sospeche que fallen. Para ello, debe apoyarse en un detector de fallos y hacer las funciones del mismo de cara al grupo. Según la lista de miembros va cambiando, el GMS envía esta lista en forma de vistas a los componentes que quedan en el grupo. El contenido de la vista podrá ser la lista completa o la lista de cambios respecto a la anterior vista.

Mediante este servicio, el grupo sólo debe encargarse de su propia consistencia; como además el GMS actúa como detector de fallos, se crea un entorno donde los únicos fallos que ven los miembros son los de caída de los otros miembros. Los errores en las comunicaciones son absorbidos por el GMS, que los transforma en nuevas vistas; efectivamente, ante cualquier error o sospecha de error de un componente, el GMS lo elimina del grupo, temporal o permanentemente, enviando una nueva vista a los demás componentes. La forma de instalar estas nuevas vistas debe garantizar la propiedad denominada sincronía de vistas (*view synchrony*) [Babaoglu96]: todos los componentes que sobreviven dos vistas consecutivas deben haber procesado el mismo juego de mensajes en la primera vista. Esta propiedad es fundamental para mantener la consistencia del grupo entre las vistas; si un miembro no ha procesado alguno de los mensajes que los demás han procesado, no se podría garantizar su consistencia.

Los miembros de un grupo van recibiendo diferentes vistas del grupo según nuevos miembros se incluyen o quedan excluidos. Cuando un miembro intenta

comunicarse con otro y se produce un problema en esa comunicación, transmite el error al GMS, que considerará que uno o los dos miembros falla, excluyéndole consecuentemente de la lista. El empleo del GMS permite simular un entorno donde los únicos fallos son por caída del componente [Sabel94], ocultando todo problema de comunicaciones y particionado de la red, en tanto ese GMS permanezca activo. Puesto que este grupo pasa a depender completamente de este servicio, un fallo en éste provocará la parada del grupo. Por esta razón, el GMS mismo debe estar replicado, es asimismo un grupo dinámico que se autogestiona mediante un protocolo denominado *GMP* (*Global Membership Protocol*). Al igual que el GMS permite incluir y excluir miembros de sus grupos, el GMP permite incluir y excluir miembros del GMS.

Un particionado de la red provocará un particionado no sólo en el grupo de componentes de una aplicación, sino también en el grupo que implementa el GMS. En caso de una partición, éste reportará una nueva vista a los componentes en la partición; pero paralelamente la otra partición o particiones del GMS podrían reportar vistas complementarias a los otros componentes, lo que supondría la existencia de varios grupos aislados que consideran que los demás han caído. El GMS controla su propia lista de miembros y las listas de los grupos asociados; si se particiona, tiene las siguientes opciones respecto a su propia lista de miembros [Ricciardi93]:

- No realizar ningún progreso. Lo que supone no permitir que nuevos miembros se incluyan en el GMS o alguno de los miembros actuales lo abandone.
- Una partición se considera primaria y puede seguir haciendo progresos. Las demás dejan de prestar servicio, deberán incluirse de nuevo en el GMS. La partición primaria puede determinarse de varias formas; por ejemplo, por mayoría de miembros en el momento de la partición.
- Varias particiones pueden progresar, pudiendo dar vistas concurrentes a diferentes miembros del grupo. Si el GMS mismo puede seguir haciendo progreso en caso de partición, los distintos *subGMSs* que se crean podrían evolucionar de tal forma que, al recuperarse la partición, ninguno de los componentes de una partición supiera de la existencia de los componentes de otra partición. En ese caso no podrían comunicarse, haciendo definitiva la partición, por lo que esta situación debe, claramente, evitarse.

En un sistema normal de sincronía de vistas, el GMS puede ir evolucionando en caso de partición, pero sólo en una partición. La forma de asegurar su consistencia es precisando que una mayoría de los componentes de una vista del GMS aprueben la siguiente vista. Ésto significa que si un sistema dinámico, por sucesivos fallos o particiones pierde más de la mitad de sus miembros, podría progresar siempre que cada fallo no implicara a la vez a una mayoría de miembros. Por ejemplo, si hay diez miembros, y una primera partición supone la caída de cuatro miembros y una segunda partición la caída de dos miembros, el GMS podría en cada paso asegurar una mayoría para progresar. Los miembros del grupo que

soporte, como consecuencia, siguen recibiendo vistas si se encuentran en la partición primaria; si no, dejan de recibir vistas y de prestar ninguna funcionalidad.

La partición primaria no tiene porqué estar definida en función de una mayoría de componentes. Esa mayoría puede estar balanceada de tal forma que determinados componentes tengan un mayor peso que otro, con lo que la partición primaria ya no sería la que ha quedado con un mayor número de componentes, sino con un mayor peso de éstos.

Aunque una aplicación que pueda funcionar en modo particionado presenta una mayor disponibilidad, su diseño es también más complicado. Además, mientras que la sincronía de vistas es un modelo muy elegante, donde el GMS gestiona todos los cambios en las vistas, los sistemas llamados de sincronía de vistas extendida (*extended view synchrony*) [Babaoglu96], soportando vistas concurrentes, no lo son tanto y requieren la cooperación de la aplicación. Por otro lado, cuanto mayor (geográficamente) sea un sistema, más frecuentes y prolongadas podrán ser las particiones, lo que puede implicar la necesidad de soportar particiones en estos sistemas.

3.3. Sincronía virtual

A partir de las primitivas de comunicaciones fiables multipunto y del servicio GMS, que hace a su vez uso de esas primitivas, es posible describir el modelo de sincronía virtual [Birman87], usado en todos los sistemas distribuidos asíncronos actuales que soportan grupos dinámicos y propiedades de consistencia no triviales.

Este modelo se basa en el siguiente principio: si todos los miembros procesaran los mismos mensajes en el mismo orden (orden total), no habría posibilidades de inconsistencia entre ellos. El modelo de sincronía virtual se basa además en la posibilidad de que, aunque dos miembros procesen dos mensajes en distinto orden, el resultado puede ser aún el mismo: de esta forma, al permitir emplear primitivas de comunicaciones menos estrictas en el orden, como orden *fifo* o causal, el rendimiento resulta mayor.

El servicio de miembros de grupo requerido por este modelo debe cumplir las siguientes propiedades:

- Soporte dinámico de grupos: exclusión e inclusión de miembros dinámicamente.
- Envío de vistas a los miembros del grupo, donde todos ven la misma secuencia de vistas.
- Sincronía de vistas: dos miembros que sobreviven a dos vistas consecutivas procesan los mismos mensajes. Los sistemas que soportan sincronía extendida de vistas relajan esta propiedad, pues miembros de vistas concurrentes no están obligados a haber procesado los mismos mensajes.

- *Gap-freedom*: si el grupo procesa un mensaje, que es sucesivo en el orden establecido a otro mensaje dado, el grupo ha procesado también este último mensaje.

Además, el modelo precisa de primitivas de comunicaciones multipunto, con uso preferente de orden *fifo* y causal en las comunicaciones. Los algoritmos de comunicaciones del grupo [Birman96] pueden desarrollarse basándose en comunicaciones totalmente ordenadas, reemplazando luego las comunicaciones con primitivas con menor orden, y pasando de entornos dinámicamente uniformes a no uniformes, mejorando notablemente el rendimiento ofrecido. Además, al emplear estas primitivas de comunicaciones que no emplean orden total, los distintos miembros del grupo podrán procesar los mensajes en distinto orden, tolerando así errores software (*Heisenbugs*): si todas las réplicas procesaran los mismos mensajes en el mismo orden, y hubiera un error software, todas las réplicas fallarían por igual. Hay estudios que prueban que la mayoría de los errores en un sistema son errores software [Chou97], luego tolerar estos errores resulta ser una consecuencia muy interesante del empleo de primitivas de comunicaciones sin orden total.

3.4. Observabilidad del grupo

Los puntos anteriores han estudiado los métodos para mantener la consistencia dentro del grupo, usando un servicio GMS y mediante unas comunicaciones atómicas preservando el orden necesario. Sin embargo, hay otras comunicaciones a tener en cuenta, externas al grupo y que no pueden diseñarse de la misma manera. Hay que considerar también el tipo de replicación del sistema: si todas las réplicas tienen la misma funcionalidad o no.

3.4.1. Comunicaciones externas del grupo

Cuando un componente individual accede al grupo, tiene varias opciones:

- Acceder a un miembro del grupo, que propaga la acción a realizar [figura 3.4]. Si el miembro se cae o es expulsado del grupo, el resultado es indeterminado. El miembro al que se accede puede elegirse de tal forma que la carga sobre cada miembro esté balanceada.
- Acceder a todos los miembros del grupo [figura 3.5]. La carga de comunicaciones es mayor y el componente debe conocer la composición exacta del grupo, aun siendo dinámica. Se multiplican aquí las opciones, pues puede responder un sólo miembro o todos o un número determinado de ellos. Si el cliente no conoce la composición exacta, pueden originarse problemas al no acceder a los componentes adecuados.

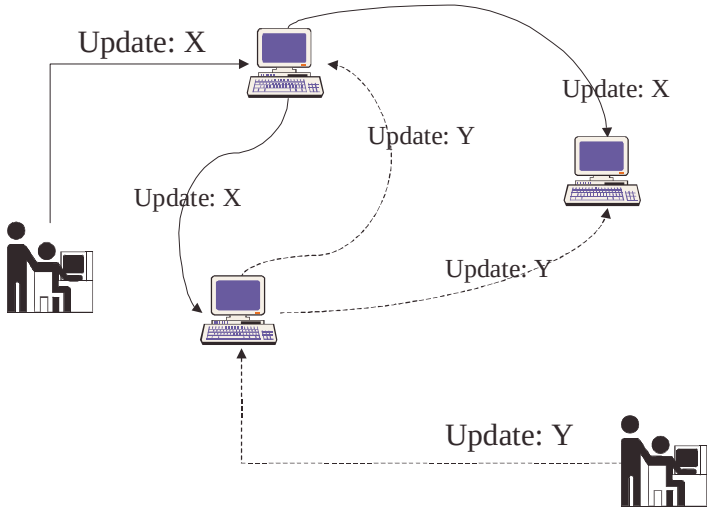

Figura 3.4. Propagación de acciones en el servidor

Además, deben contemplarse las opciones de orden de sus mensajes. En el caso de un componente que accede síncronamente a un grupo, cuando éste devuelve la respuesta, no es necesario que todo el grupo sea aún participe de la acción realizada. Por consiguiente, si ese componente accede a otro miembro del grupo, pueden crearse situaciones de inconsistencia al no respetarse el orden de los mensajes. Una solución posible es no devolver el resultado al componente hasta que todo el grupo conozca la acción.

Estas comunicaciones hacia el grupo no tienen por qué partir exclusivamente de un componente individual, también pueden provenir de otro grupo, y debe resolverse si sólo un miembro del grupo que solicita el servicio o bien todo el grupo debe recibir la respuesta. De la misma manera, si un componente individual accede a un grupo, debe decidirse si recibirá una única respuesta (acceso transparente) o la respuesta de todos los miembros. En este último caso, pueden fijarse varias variables, como el número mínimo de respuestas a recibir, o el intervalo máximo de espera de las respuestas.

Existen varias soluciones para el problema de orden de mensajes entre diferentes grupos:

- No establecer ningún orden en los mensajes entre grupos, lo que puede resultar perfectamente adecuado si estos grupos están poco acoplados.
- Establecer un orden global en el sistema, de tal forma que todas las comunicaciones preserven su causalidad. Además de ser una solución muy cara, impone un coste a las comunicaciones que no necesiten preservar ningún orden.

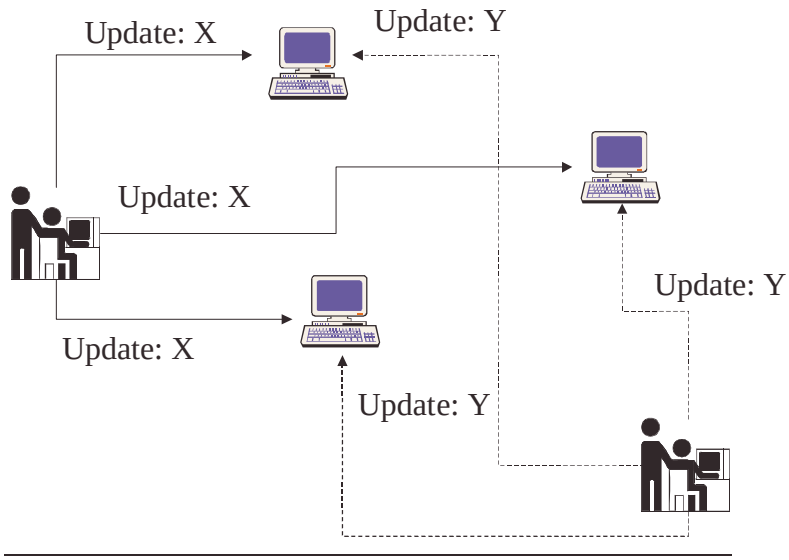

Figura 3.5. Propagación de acciones por el cliente

- Cada grupo controla la causalidad de los mensajes que envía y recibe; de esta manera, no enviará ningún mensaje hasta que todo el grupo conozca el mensaje entrante que lo originó. No se respeta sólo el orden de los mensajes salientes respecto a los entrantes, sino también de los internos; si un mensaje se produce como consecuencia de uno o más mensajes internos, el mensaje sólo saldrá cuando todo el grupo conozca todos sus mensajes precedentes causalmente. Este método se denomina conservativo [Birman96] y permite conservar el orden de los mensajes concernientes a un grupo, pero no preserva el orden global del sistema.
- Una solución similar a la anterior es el empleo de protocolos *flush*: completar todas las comunicaciones internas del grupo antes de enviar un determinado mensaje fuera del grupo. Es un caso concreto de la anterior solución, donde se desea que sólo algunos mensajes respeten la causalidad, y en esos casos se realiza un *flush* antes de enviar el mensaje.
- Cuando varios grupos están fuertemente acoplados, otra solución es crear un *supergrupo* o dominio que los incluya, permitiéndose entonces emplear las primitivas de comunicaciones de un grupo: orden *fifo*, causal, etc.

3.4.2. Replicación

Al definir los grupos de objetos, se ha presupuesto que todos estaban en el mismo nivel. Sin embargo, la replicación puede realizarse de varias maneras: todos los miembros activos [Schneider93], un solo miembro activo y los demás en pasivo

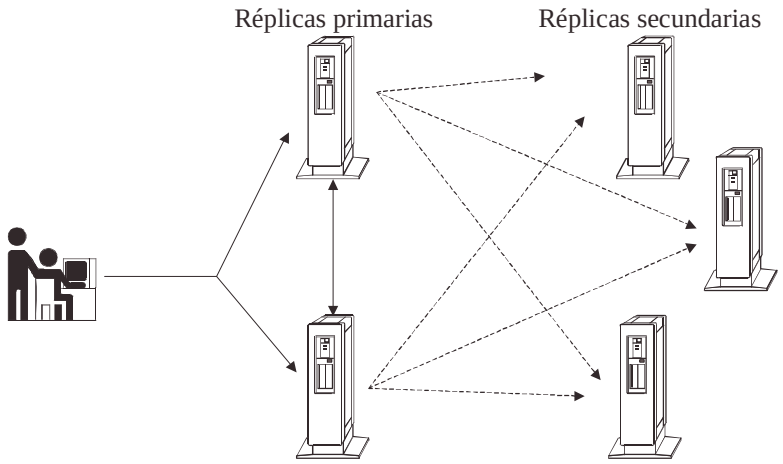

Figura 3.6. Modelo mixto de replicación

[Budhiraja93], o el estado intermedio, donde varios miembros están activos y los demás son pasivos [figura 3.6].

Esta replicación activa/pasiva puede entenderse como dos grupos de las mismas réplicas donde un grupo, el activo, procesa todas las peticiones de servicio y actualiza periódicamente al segundo grupo; un componente del grupo pasivo puede incluirse en el grupo activo si se produce un fallo en éste, abandonando a su vez el grupo pasivo. Éste es en un proceso monitorizado externamente, mediante un sistema que observa los fallos en los grupos y mueve miembros de un grupo al otro.

Relacionado con el modelo de replicación está el concepto de balanza de carga, que permite promediar la carga de servicio que cada componente del grupo debe soportar. Este concepto se introdujo brevemente en el apartado anterior al discutir los métodos con que un cliente puede acceder al grupo. Si el grupo se diseña para hacer balanza de carga, el cliente puede elegir el miembro al que accede –confinado entonces en un acceso promediado- o el acceso puede realizarse sobre todos los elementos del grupo, decidiendo éste cuál debe procesar el servicio.

Capítulo 4 - Sistemas de Comunicaciones en Grupo

Los anteriores capítulos han mostrado las opciones de comunicaciones punto a punto disponibles actualmente y las formas de comunicaciones en grupo necesarias para crear aplicaciones distribuidas fiables mediante la replicación de sus componentes.

Los sistemas de comunicaciones en grupo existentes se basan en el modelo de sincronía virtual. Una de las propiedades que este modelo precisa [Birman96] es el soporte de la transferencia de estado, pero las aproximaciones empleadas para este soporte varían enormemente entre los distintos sistemas. La transferencia de estado es uno de los objetivos cubiertos por *Sensei*, pero antes de presentar el modelo desarrollado, es necesario mostrar los sistemas existentes y su modelo de transferencia de estado.

4.1. Sistemas de comunicaciones en grupo

Sobre el modelo de sincronía virtual se han desarrollado varios sistemas de comunicaciones de grupos. A continuación se muestra una lista con los principales sistemas (restringidos a interfaces Java, C y C++) que soportan tolerancia a fallos y excluyendo sistemas comerciales de dominio restringido. Dos sistemas se han postergado a las secciones finales, entrando en mayor detalle en su implementación,

por su influencia en la especificación y el desarrollo de *Sensei*: *Maestro* (sobre *Ensemble*) y la especificación de tolerancia a fallos de CORBA.

4.1.1. Amoeba

Amoeba [Amoeba] es un sistema operativo basado en *microkernel* que transforma un grupo de estaciones de trabajo en un sistema distribuido transparente [Tanembaum91]. La idea básica es presentar al usuario la ilusión de un único y potente sistema que, sin embargo, se implementa en un grupo de ordenadores, potencialmente dispersos en diversos países. Desarrollado a partir de 1981 en la Universidad *Vrije* de Amsterdam, es un sistema de libre distribución. Está implementado en *Orca*, un lenguaje para programación paralela en sistemas distribuidos. No obstante, *Amoeba* es un completo sistema operativo (presenta incluso una librería de emulación *POSIX*), no simplemente un sistema de comunicaciones de grupo. Una idea similar es la soportada por el servicio de clusters de *Microsoft* (*MSCS*) [Vogels98], pero en este caso la funcionalidad se escribe sobre su omnipresente sistema operativo. Sin embargo, *MSCS* sólo soporta inicialmente un conjunto limitado de aplicaciones (servidores de ficheros, de mail electrónico, de páginas Web y bases de datos) y únicamente sobre dos nodos.

4.1.2. Arjuna

Desarrollado desde 1987 en la Universidad de *NewCastle upon Tyne*, *Arjuna* [Arjuna] se define como un sistema de programación orientado a objetos que busca la simplificación del proceso de desarrollo de aplicaciones distribuidas fiables. Opera sobre el modelo cliente/servidor y puede ejecutarse sobre entornos heterogéneos, usando C++ como lenguaje de programación. La fiabilidad la obtiene mediante transacciones atómicas anidadas distribuidas. Es, por lo tanto, más bien un sistema de transacciones que un sistema de comunicaciones de grupo. De hecho, se ha portado a CORBA, siendo compatible con el servicio de transacciones de esta arquitectura.

En este sistema [Parrington95], el estado de los objetos es administrado por un objeto de la clase *StateManager*. Los objetos se clasifican como:

- Recuperables: el *StateManager* trata de generar y mantener toda la información necesaria para recuperar el estado del objeto si éste se cae. Esta información se guarda en objetos de la clase *ObjectState*.
- Recuperables y persistentes: son objetos recuperables cuyo ciclo de vida puede exceder al de la aplicación que los creó. En estos casos, el *StateManager* intentará obtener el estado previo antes de activar el objeto.

- No recuperables ni persistentes: el *StateManager* no guarda ninguna información sobre estos objetos. Para simplificar el trabajo del *StateManager*, un objeto no puede cambiar su tipo en tiempo de ejecución.

Los objetos recuperables deben implementar tres métodos:

- *boolean save_state (ObjectState &state, ObjectType oType);* : el objeto sobre el que el *StateManager* invoca esta operación debe salvar su propio estado en el parámetro de la clase *ObjectState*. Mediante el parámetro del tipo *ObjectType*, es posible pasarle cierta información al objeto para que conozca el motivo de la operación.
- *boolean restore_state (ObjectState &state, ObjectType oType);* : es la operación simétrica a la anterior.
- *const TypeName type () const;* : devuelve el tipo como una cadena de caracteres, usado simplemente para los controles de concurrencia y persistencia.

El protocolo de replicación por defecto se basa en una única réplica primaria, que se comunica con réplicas en *backup*. Cuando una nueva réplica se incluye en el grupo, puede recibir entonces automáticamente el estado. Sin embargo, no se pueden incluir nuevos miembros en el grupo mientras haya usuarios operando sobre él. Es decir, la transferencia de estado se realiza con la seguridad de que ningún miembro procesará ninguna operación hasta que la transferencia haya finalizado, lo que simplifica enormemente el protocolo.

Consecuentemente, las aplicaciones desarrolladas en este sistema se bloquean durante la transferencia; a cambio, ésta se realiza de una forma muy simplificada, implementando en cada objeto dos operaciones básicas de almacenamiento y recuperación del estado.

4.1.3. Bast

Este sistema [Bast] es un *framework* de patrones para el desarrollo de aplicaciones distribuidas tolerantes a fallos. Está enfocado [Garbinato97], más que en los algoritmos mismos que soportan el desarrollo de aplicaciones distribuidas tolerantes, en el estudio de los patrones que facilitan el desarrollo de esas aplicaciones: qué abstracciones resultan adecuadas para modelar la fiabilidad, cómo representarlas para promover diseños flexibles y cómo implementarlas para promover la reusabilidad del código. Los patrones en los que se enfoca este sistema están relacionados con los protocolos de comunicaciones más que con la aplicación final: cómo implementar protocolos a partir de un *framework*, con unos patrones de diseño sobre esos protocolos bien especificados. Está desarrollado en Smalltalk, y está siendo portado a Java. Como el sistema *Phoenix*, ha sido desarrollado en el *Swiss Federal Institute of Technology de Lausanne*.

4.1.4. Universidad de Cornell: Isis / Horus / Ensemble / Spinglass

Estos cuatro sistemas son la evolución del sistema inicial propuesto y desarrollado por Ken Birman sobre su modelo de sincronía virtual.

El primer sistema, *Isis* [Isis], desarrollado desde 1983, implementaba una colección de técnicas para la construcción de software de sistemas distribuidos con un rendimiento eficiente, tolerante a fallos hardware y software, y explotando el paralelismo de los sistemas [Birman85]. El planteamiento básico fue de suministrar las herramientas necesarias para interconectar componentes no distribuidos, dando paso a un sistema distribuido fiable, con herramientas para manejar datos replicados, sincronizar operaciones distribuidas, recuperación automática tras errores, y reconfiguración automática del sistema.

A partir de un rediseño de este sistema, desde 1993 *Isis* evolucionó a *Horus* [Horus], una arquitectura de comunicaciones de propósito general con un soporte avanzado para el desarrollo de aplicaciones distribuidas robustas [Renesse96]. Se basa también en las comunicaciones de grupo, proporcionando a la aplicación final las primitivas de comunicaciones necesarias para desarrollar los requisitos de fiabilidad al mínimo coste. *Horus* existe como dos sistemas: una versión inicial en *C*, disponible comercialmente y gratuita para fines de investigación, y una nueva versión llamada *Ensemble*.

Ensemble [Ensemble] es una versión de *Horus* escrita a partir de 1996 en *Ocaml*, un dialecto de *ML*, y está disponible gratuitamente para todos los usuarios. Como *Horus*, se diseñó para resultar independiente de la plataforma. Provee a las aplicaciones de una librería de protocolos con la que construir rápidamente aplicaciones distribuidas fiables; éstas registran una serie de eventos con *Ensemble*, y los protocolos de *Ensemble* gestionan el envío y recepción fiable de mensajes, transferencia de estado, seguridad, detección de fallos y la reconfiguración del sistema [Hayden98]. Tiene una interfaz a *C* denominada *Hot*, y sobre esta interfaz se han desarrollado un conjunto de clases denominadas *Maestro* [Maestro], que implementan un modelo de transferencia de estado que detallamos en el siguiente capítulo. También han desarrollado una interfaz a Java, con vistas a tener *Ensemble* como un *plug-in* en *Netscape*.

Spinglass [Spinglass] es un proyecto iniciado recientemente (1999), que trata de obviar las limitaciones del modelo de sincronía virtual en grupos con gran cantidad de miembros. A diferencias de las tres generaciones previas comentadas, presenta un enfoque diferente basándose en protocolos que soportan mensajería multipunto con garantías de fiabilidad probabilísticas [Birman99].

4.1.5. Cactus

Cactus [Cactus] es un substrato de comunicaciones modular desarrollado en la Universidad de Arizona, soportando calidades de servicio que pueden cambiarse dinámicamente. Sobre un protocolo básico multipunto usando ordenación causal, se añaden modularmente los protocolos de pertenencia a grupo (*membership*) y ordenación total. Hay versiones disponibles en C++ y Java de este sistema, que es la continuación de otro proyecto de la misma universidad, llamado *Consul.*

Cactus y *Consul* se basan en la utilización de *máquina de estados replicados* [Schneider90]. Las aplicaciones se estructuran como máquinas de estado que mantienen colecciones de variables de estado. Cada máquina de estado recibe órdenes de otras máquinas de estado para modificar u obtener sus variables. Estas órdenes son deterministas y atómicas con respecto a otras órdenes, de tal forma que puede determinarse el estado de un objeto exclusivamente a partir de las órdenes de entrada.

Todo el sistema [Schlichting93] se desarrolla en torno a tres servicios: mensajería multipunto, pertenencia a grupo y recuperación (*recovery*). Este último servicio de recuperación es el encargado de sincronizar a una réplica que se hubiera caído con el resto de réplicas activas. Para ello usa una técnica de *checkpoints* y seguimiento de mensajes: cada réplica, sin necesidad de comunicarse con otras réplicas, va escribiendo *checkpoints* (instancias de su estado) y guarda los mensajes en un sistema de almacenamiento estable. Tras un fallo, la réplica recupera el último estado almacenado, y se comprueban los mensajes procesados para sincronizarla con otras réplicas.

Esta técnica es rápida cuando las réplicas se caen durante cortos periodos de tiempo, de otra manera resulta más efectiva la transferencia desde otra réplica activa.

4.1.6. Electra

Electra [Electra] fue probablemente el primer ORB que soportó de una forma no estándar la implementación de aplicaciones distribuidas fiables sobre CORBA. Su desarrollo se efectuó sobre *Isis* y no se ha completado su porte a los sistemas sucesores de aquél, *Horus* y *Ensemble*. La fiabilidad la obtiene a partir del sistema de comunicaciones de grupo sobre el que se implementa (teóricamente, cualquier sistema que implemente el concepto de grupos con comunicaciones fiables provee las primitivas necesarias para *Electra*). Implementa un protocolo propio de transferencia de estado, independiente de los protocolos de transferencia existentes en el substrato de comunicaciones.

Este ORB [Maffeis97] sobre *Isis* define un mecanismo básico de transferencia de estado, en la que el BOA (*Basic Object Adapter*) con el que trabaja (el POA, *Portable Object Adapter*, es una estandarización bastante posterior a *Electra*) define

dos métodos para controlar la transferencia de estado. Estos métodos son virtuales, y deben ser implementados por la clase final que es, por consiguiente, una instancia del *BOA*, en lo que resulta un enfoque poco tradicional:

- *getState*
- *setState*

Ambos métodos tratan el estado mediante el tipo genérico de datos CORBA *AnySeq*. La transferencia puede realizarse en varios pasos. Para ello, los dos métodos incluyen una indicación de finalización de la transferencia, que la aplicación debe rellenar convenientemente. El primer miembro que se incluye en un grupo no recibe el estado de otros miembros, es el único caso en que un miembro no recibe una invocación de su método *setState*.

También implementado sobre Isis, el mismo autor de *Electra* desarrolló *CyberBus*, un antecedente de *iBus*, donde la abstracción de grupo con la que se trabaja es mediante canales, a los que se pueden subscribir distintos miembros, tanto servidores de datos, como consumidores. De esta forma, permite implementar un sistema de comunicaciones en grupo bajo el modelo de mensajería en lugar del clásico cliente/servidor. En este caso, la clase de la aplicación que soportan tolerancia a fallos debe sobrecargar tres métodos, que son invocados automáticamente por el canal al que están suscritos:

- *obj_ask_state*: el objeto involucrado debe devolver su estado asociado, empleando el tipo *CORBA::AnySeq*.
- *obj_set_state*: permite fijar el estado de un determinado objeto.
- *obj_load_state*: se emplea para el primer objeto del grupo, de tal forma que no reciba el estado de otros miembros y pueda inicializarlo directamente.

Al contrario que en *Electra*, la transferencia se realiza ahora en un solo paso.

En ambos casos, el mecanismo de transferencia se basa en las listas de grupo que *Isis* suministra, donde cada miembro tiene un rango determinado. Los miembros más antiguos presentan un rango inferior, con lo que *Electra* puede emplear siempre el miembro más antiguo para realizar la transferencia a los nuevos miembros. Esta transferencia se realiza de forma automática, guiada por el software que gestiona el grupo. Durante la transferencia, el sistema queda bloqueado.

4.1.7. Ibus / MessageBus

Del mismo autor que *Electra*, existe desde 1996 una versión comercial [Ibus] de un servicio basado en el servicio de mensajería Java *JMS* [SUN99] que permite el desarrollo de aplicaciones distribuidas tolerantes a fallos, obteniéndose esa tolerancia con unas extensiones no estándar al servicio. Este servicio en Java no se basa en un modelo cliente/servidor, sino en un modelo de publicación/subscripción, donde las aplicaciones u objetos productores producen información en los canales

de comunicación, y las aplicaciones subscriptas a esos canales reciben esa información. Puesto que la información fluye por los canales de comunicación, no es necesaria la localización exacta de los subscriptores o emisores, facilitándose su migración entre distintas máquinas y procesos. El *JMS* llama *tópicos* a los enlaces de comunicación; los canales son estos tópicos, a los que se añade una calidad de servicio. Esta calidad de servicio, como ocurre en los grupos de *Ensemble*, se construye sobre la base de una pila de protocolos; incluyendo el protocolo adecuado, se obtiene tolerancia a fallos mediante un servicio de miembros GMS. Tanto los miembros que publican como los que subscriben pertenecen al mismo grupo y no hay ningún mecanismo específico para la transferencia de estado entre los emisores.

4.1.8. JavaGroups

Este sistema [JavaGroups], también proveniente de la Universidad de Cornell, es un conjunto de herramientas Java que facilita el desarrollo de sistemas distribuidos fiables. La abstracción sobre la que trabaja es la de grupos de objetos, que denomina *canales*; presenta tres implementaciones de canales; *JChannel*, no fiable, *EnsChannel*, fiable, trabajando sobre *Ensemble*, e *IbusChannel*, también fiable, trabajando sobre *Ibus*.

Esta implementación de un sistema de grupo de comunicaciones no fuerza una transferencia de estado cada vez que un nuevo miembro se incluye en el grupo, o empleando la notación del sistema, en el canal. Es función de este miembro la de dirigirse a otro miembro para solicitar el estado, pudiendo, además, hacerlo en cualquier momento de su ciclo de vida; dispone para ello de dos métodos:

- *GetState*: devuelve el estado de un miembro, normalmente del más antiguo, el coordinador.
- *GetStates*: devuelve el estado de todos los miembros.

Esta solicitud de estado es asíncrona; el objeto que la solicita recibirá eventualmente uno o más eventos *SetState*.

El protocolo de transferencia de estado tiene los siguientes pasos:

- Todos los miembros, incluyendo el que lo solicita, reciben un mensaje para hacer una copia de su estado. Antes de copiarlo, encolan los mensajes sucesivos que les lleguen, procesándolos únicamente cuando haya completado la copia de su estado.
- Un miembro, posiblemente el más antiguo, recibe una solicitud para devolver su estado. Alternativamente, esta solicitud puede dirigirse a todos los miembros.
- El nuevo miembro recibe el estado y procesa los mensajes encolados, tras los cuales puede empezar a responder a nuevos mensajes.

La versión actual de *JavaGroups* (0.6.6) sólo soporta la transferencia de estado cuando emplea su propia implementación de canal, *JChannel*. Las

implementaciones sobre *Ensemble* e *Ibus* no la soportan. Además, el protocolo expuesto precisa que el canal presente ordenación total de los mensajes multipunto.

Bajo *JavaGroups*, un objeto que vaya a intervenir en una transferencia de estado debe soportar la interfaz *StateExchange*, con tres métodos que permiten procesar cada uno de los pasos del protocolo explicado:

- *void SaveState();* : guarda el estado en una copia interna.
- *void Object GetState();* : devuelve la copia del estado almacenada previamente.
- *void SetState(Object new_state);* : transfiere el estado al miembro.

Este esquema permite una transferencia que se simplifica enormemente para la aplicación final; a cambio, le otorga muy poca flexibilidad, a la vez que impone requisitos adicionales, como es la ordenación total.

4.1.9. JGroup

El proyecto *JGroup* [JGroup] de la Universidad de Bolonia, se define como una integración de objetos distribuidos con la tecnología de grupos, dando paso al paradigma de *grupos de objetos*. Basado en Java, se incluye dentro de otro proyecto llamado *Relacs* [Relacs], para el desarrollo de herramientas con soporte sistemático de servicios y aplicaciones particionables en entornos distribuidos asíncronos.

Este sistema [Montresor98] se deriva del modelo de invocación remota de Java, empleándolo directamente para las comunicaciones internas y usando un modelo de arquitectura parecido, pero extendido a grupos. Contiene un compilador propio llamado *dmic* paralelo al empleado en RMI, *rmic*, y un registro de servidores específico para grupos llamado *dregistry*. Si en RMI los objetos remotos deben implementar una interfaz *Remote*, en JGroups la interfaz se llama *ExternalGMIListener* para su interacción con clientes e *InternalGMIListener* para la interacción con los demás miembros del grupo; las siglas *GMI* significan *Group Method Invocation*: invocación dinámica de grupos.

Una de las principales características de este sistema es la exposición de los problemas de red a la aplicación, bajo la suposición de que ésta tiene un mejor conocimiento del grupo para poder manejar esos problemas. Este planteamiento está motivado en que este sistema emplea el modelo de sincronía virtual extendido [Babaoglu96], soportando la existencia de varias particiones activas; en este caso, es necesario que cuando dos particiones se unifican de nuevo, puedan homogeneizar sus estados, y esa es una labor totalmente dependiente de la aplicación.

La re-unión de particiones la realiza el servicio llamado *SMS, state merging service*, y es necesario que en ese caso se implemente la interfaz *MergingListener*, que define dos operaciones:

- *Object getState (MemberId[] dests)*: obtiene el estado de una partición.

- *void putState(Object status, MemberId[] sources)*; informa del estado a la otra partición, que debe reconstruir un estado homogéneo

En estas operaciones, la listas de identidades son las de los miembros de la partición que se unifica. La operación *getState* se invoca exclusivamente sobre un miembro de una partición, al que se considera coordinador.

4.1.10. Nile

El proyecto *National Challenge Computing Project* [Nile] se desarrolló inicialmente para una aplicación específica (*CLEO High Energy Experiment*), donde debía crear un sistema autogestionado, tolerante a fallos y heterogéneo con cientos de estaciones de trabajo que accedieran a una base de datos superior a los cien terabytes, distribuida en distintas localizaciones de Canadá y Estados Unidos. Aunque el dominio de este sistema es más extenso, el de aplicaciones fácilmente paralelizables [Marzullo96] independientes de la localización de los recursos y los datos, no puede considerarse un sistema genérico de comunicaciones en grupo. Está desarrollado en Java por la Universidad de Cornell.

4.1.11. Phoenix

Phoenix [Phoenix] es un conjunto de herramientas para la construcción de aplicaciones distribuidas tolerantes a fallos a gran escala. Como *Bast*, ha sido desarrollado en el *Swiss Federal Institute of Technology de Lausanne*, pero no es un sistema de libre distribución.

Este sistema [Malloth96] trata la transferencia de estado como un problema de sincronización de un miembro que se une a un grupo y debe recibir el estado de otro miembro, no pudiendo procesar ninguna petición de servicio en tanto no se complete esa sincronización.

La abstracción de grupo que realiza *Phoenix* incluye tres tipos de objetos: receptores, clientes, y miembros. Hay una relación de herencia entre estos objetos, de tal forma que un cliente es un receptor y un miembro es un cliente. Un receptor puede unirse y excluirse de un grupo, y recibir información de éste. Un cliente puede además recibir los cambios de vistas así como enviar solicitudes de servicio. Finalmente, un miembro puede enviar también mensajes multipunto en el interior de un grupo. Cuando un cliente realiza una solicitud de servicio, la realiza a un único miembro que, a su vez, envía el mensaje multipunto.

La transferencia de estado la realiza automáticamente *Phoenix* en cuanto detecta un nuevo miembro, pero no ante clientes o receptores; en ese momento, invoca la operación *GetState* sobre un miembro antiguo del grupo, el cual responde con una cadena de *bytes*, que es transferida tal cual al nuevo miembro, que debe soportar la operación simétrica *PutState*.

Como conclusión, la transferencia es controlada automáticamente, seleccionándose un miembro antiguo como coordinador de la transferencia, la cual se realiza en un solo paso. Esta transferencia se considera atómica en el nuevo miembro, que no recibirá ningún mensaje en tanto no se sincronice con las demás réplicas activas.

4.1.12. RMP

El nombre de este sistema [RMP] es un acrónimo para *Reliable Multicast Protocol*. Su enfoque es el rendimiento del servicio de mensajería, y provee un servicio de mensajería multipunto fiable, atómico y con ordenación total sobre IP multipunto. Se ha desarrollado en la *Universidad de Illinois at Urbana-Champaign*.

4.1.13. Spread

Spread [Spread] es otro conjunto de herramientas, con soporte de comunicaciones de grupo para la creación de aplicaciones distribuidas en LANs y WANs. Su dominio básico de aplicaciones son las aplicaciones colaborativas, servidores fiables, transmisiones multimedia y sistemas de grupos en general (*groupware*). Para ello soporta todos los niveles de mensajería y ordenación, bajo el modelo de sincronía virtual extendida. Las aplicaciones la usan como una librería con la deben enlazarse; el lenguaje de programación empleado es C, aunque presenta también una interfaz Java. Está siendo desarrollando actualmente en la Universidad *Johns Hopkins*, *Baltimore*.

Spread [Amir98] no presenta un soporte específico para la transferencia de estado. La interfaz Java presenta dos métodos para la recepción de mensajes: *membershipMessageReceived* y *regularMessageReceived* pero ningún método específico para salvar o recuperar el estado. La aplicación debe construir su propio sistema de transferencia y usar el envío normal de mensajes. Cuando un miembro antiguo perciba la inclusión de un nuevo miembro (lo recibirá en un mensaje a través del método *membershipMessageReceived*) deberá construir un mensaje de estado que el nuevo miembro recibirá a través de *regularMessageReceived*.

La transferencia de estado queda entonces como una responsabilidad de la aplicación, que deberá implementar todos los detalles necesarios para que los nuevos miembros sincronicen sus estados con los miembros antiguos. Puesto que soporta sincronía virtual extendida, es también responsabilidad de la aplicación el desarrollar el protocolo de conciliación de los estados de los miembros cuando se reúnen tras una partición.

4.1.14. Totem

Desarrollado en la Universidad de Santa Barbara, California, *Totem* [Totem] es un conjunto de protocolos de comunicaciones que soporta la construcción de sistemas distribuidos tolerantes a fallos. Tiene dos características claves: el empleo de mensajes multipunto con ordenación total en lugar de ordenación causal, y el empleo del modelo de sincronía virtual extendida, permitiendo de esta manera que distintas particiones de una aplicación se mantengan operativas, en tanto la consistencia requerida sea correcta. El favorecer la ordenación total se basa en una fuerte optimización del sistema de mensajería que hace innecesario el empleo de orden causal.

En este sistema [Moser95], el modelo empleado para el envío de mensajes es la creación de un anillo lógico entre los componentes del grupo, anillo por el que circula un testigo con los mensajes. Para obtener una buena eficiencia, se emplean las características de mensajería multipunto disponibles en redes de área local.

Es uno de los pocos sistemas que enfoca la transferencia de estado como un problema prioritario. Al soportar distintas particiones, debe permitir la reconciliación de estados inconsistentes, lo que puede implicar el envío de mensajes con largas porciones del estado. Define cuatro opciones para resolver el problema de la transferencia:

- Parar el sistema, que se desbloquea tras el envío del estado.
- Todos los procesos mantienen *checkpoints* y almacenan los mensajes procesados desde el último *checkpoint*. La transferencia de estado se resuelve con el envío del último *checkpoint* y de todos los mensajes almacenados tras él. Aunque simple, puede suponer el empleo de grandes cantidades de espacio para almacenar la información temporal y cada proceso debe destinar recursos para almacenar los *checkpoints* e ir guardando los mensajes.
- Sólo un proceso, el que debe enviar la transferencia, queda inoperativo mientras realiza la transferencia. Una posibilidad en este esquema es que ese miembro realice primero una copia interna del estado, que emplea para la transferencia, quedando entonces operativo más rápidamente.
- El miembro que envía el estado permanece operativo, manteniendo un archivo de los cambios efectuados en el estado durante la transferencia. La transferencia incluye, consecuentemente, el estado y uno o varios mensajes posteriores incluyendo los cambios efectuados.

Con este esquema, la primera opción es válida cuando el estado a enviar es pequeño y el sistema no es crítico en el tiempo; en otro caso, es mejor tomar la tercera opción. La última opción es muy atractiva, pero supone una mayor complejidad en la aplicación final.

4.1.15. Transis

Transis [Transis], es una implementación de la Universidad Hebrea de Jerusalén de un grupo eficiente de comunicaciones para alta disponibilidad. Al igual que en *Totem*, *Transis* permite el particionado de grupos y da soporte a la transferencia de estado. Los miembros del grupo son procesos, aunque una interfaz Java permite la abstracción de clases.

En *Transis* [Dolev96], la transferencia de estado es todavía una responsabilidad de la aplicación, que debe realizar los pasos pertinentes para sincronizar correctamente los estados. De esta manera, para una transferencia simple (sin incluir particiones y reconciliación de estados), sus autores proponen el siguiente protocolo:

- Los miembros dejan de enviar mensajes (los mensajes quedan bloqueados).
- Cada miembro envía un mensaje de *parada* que empleará el miembro que debe enviar el estado para saber cuándo ha procesado todos los mensajes de un miembro.
- Cuando el miembro que debe enviar el estado recibe el mensaje de parada de todos los demás miembros, envía el estado en uno o varios mensajes, al nuevo miembro.
- Tras concluir la transferencia, se desbloquean los miembros y se reinicia el flujo de mensajes.

Por consiguiente, y tal como ocurre con *Totem*, se considera prioritario el problema de la transferencia, aunque sigue estando la responsabilidad del protocolo de transferencia en la aplicación, que debe escoger el método más apropiado para realizar la sincronización de estados.

4.1.16. xAmp

Este sistema [xAmp], desarrollado desde 1991 en la Universidad de Lisboa, es un acrónimo de *eXtended Atomic Multicast Protocol*. Básicamente, es un sistema de comunicaciones de grupo que proporciona primitivas que facilitan el desarrollo de aplicaciones distribuidas fiables. Este servicio [Rodrigues92] de grupo de comunicaciones fue desarrollado siguiendo el modelo de *Isis*, que encapsula en el subsistema de comunicaciones un juego de primitivas que soportan diferentes calidades de servicio, cuyo uso facilita el desarrollo de aplicaciones distribuidas fiables.

Está desarrollado en *C*, presentando al usuario un grupo de primitivas que soportan el concepto de grupos de miembros. De esta manera, hay primitivas con las que unirse o salirse de grupos (*xampGroupOpen, xampGroupClose*) y enviar mensajes al grupo con distintas condiciones de atomicidad, fiabilidad y ordenación (*reliableSend, atomicSend,* ...). La aplicación debe suministrar a su vez funciones con

las que manejar los eventos del grupo: *viewHandler*, para las notificaciones de cambios de vistas, *confHandler*, para recibir confirmaciones de solicitudes (como la de incluirse en un grupo) y *dataHandler*, donde se reciben mensajes de otros miembros del grupo.

No se contempla específicamente la transferencia de estado, dejando como responsabilidad de la aplicación el protocolo de la transferencia.

4.2. Transferencia de estado en *Maestro*

Maestro es un conjunto de clases C++ que permiten crear una abstracción de objetos sobre los sistemas de *Horus* y *Ensemble*. Existe una capa intermedia entre estos sistemas y *Maestro*, denominada *Hot*, que es una interfaz *C*; hay dos interfaces Hot, una para *Maestro* y otra para *Ensemble*, pero una única implementación de *Maestro*. Aunque estos sistemas definen un protocolo de transferencia de estado propio, *Maestro* implementa uno independiente, que es el que se detalla aquí. Esta explicación incluye la versión 0.51 sobre *Ensemble* y la evolución posterior realizada para solventar los errores de transferencia de estado de la primera.

Sensei se inició como una solución a los problemas de transferencia en *Maestro*, y desarrollamos un conjunto substituto de clases que, funcionando sobre *Ensemble-Hot*, implementa parte de la funcionalidad que requerimos para un protocolo de transferencia de estado completo. Ésta es la razón por la que detallamos este protocolo, al ser la base de los desarrollados en esta memoria.

4.2.1. Versión 0.51

La clase básica es *Maestro_GroupMember*, con la que es posible especificar la calidad de servicio que un miembro de un grupo usará (todos los miembros deben emplear la misma calidad de servicio). Esta calidad de servicio se especifica como una cadena de caracteres donde cada propiedad viene separada por ':', como por ejemplo "Gmp:Sync:Heal:Switch:Suspect:Flow:Primary". Dos operaciones, *join* y *leave* permiten la inclusión y exclusión del miembro en el grupo definido, y cada objeto sólo puede incluirse en un grupo. Si el objeto es el primer miembro de un grupo, éste se crea automáticamente. Mediante varias operaciones *send* y *cast*, es posible enviar mensajes punto a punto o multipunto, que se reciben con los respectivos métodos *grpMemb_ReceiveSend_Callback* y *grpMemb_ReceiveCast_Callback*.

Cuando se produce un cambio de vista, *Maestro* invoca [figura 4.1] tres operaciones:

- *grpMemb_Block_Callback*, para informar que la vista ha sido bloqueada.

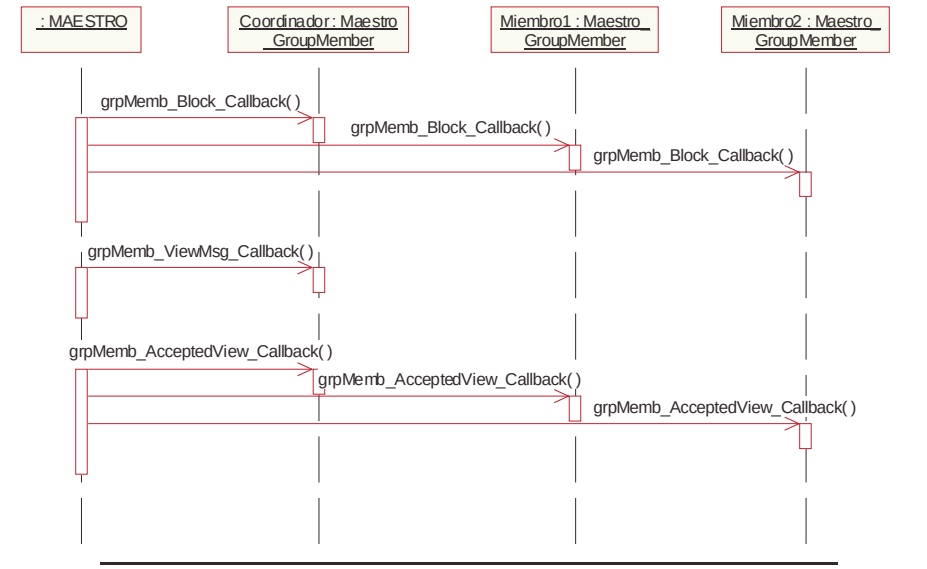

Figura 4.1. Mensajes en *Maestro_GroupMember* en un cambio de vista

- *grpMemb_ViewMsg_Callback*, con la nueva vista a instalar, que se invoca exclusivamente sobre el miembro con menor rango de la vista y que en la notación de *Maestro* se denomina coordinador. Este coordinador puede incluir alguna información tras una llamada a este método, que es entonces propagada a todos los miembros con el siguiente método.
- *grpMemb_AcceptedView_Callback*. Este método se invoca sobre todos los miembros con la nueva vista a instalar, así como con la información que transmita el coordinador de la vista. La vista contiene la lista ordenada de todos los miembros del grupo, junto a alguna información adicional.

Esta clase no proporciona ningún soporte de transferencia de estado. Para obtenerlo, se usa una clase especializada, *Maestro_ClSv*, que mantiene una relación de herencia con la anterior. Cuando se crea un objeto de este tipo, puede indicarse si pertenecerá al grupo como cliente o servidor y, en este último caso, cuál es el nivel de protección durante la transferencia:

- *MAESTRO_NO_XFER*: no hay transferencia.
- *MAESTRO_FREE_XFER*: hay transferencia; se puede enviar todo tipo de mensajes mientras dure la transferencia.
- *MAESTRO_PROTECTED_XFER*: hay transferencia, y los únicos mensajes permitidos son los de la transferencia y los considerados seguros para aquella. Para realizar esta distinción, cada mensaje incluye un atributo fijado por su emisor que indica cuándo se considera seguro procesar ese mensaje durante la transferencia.

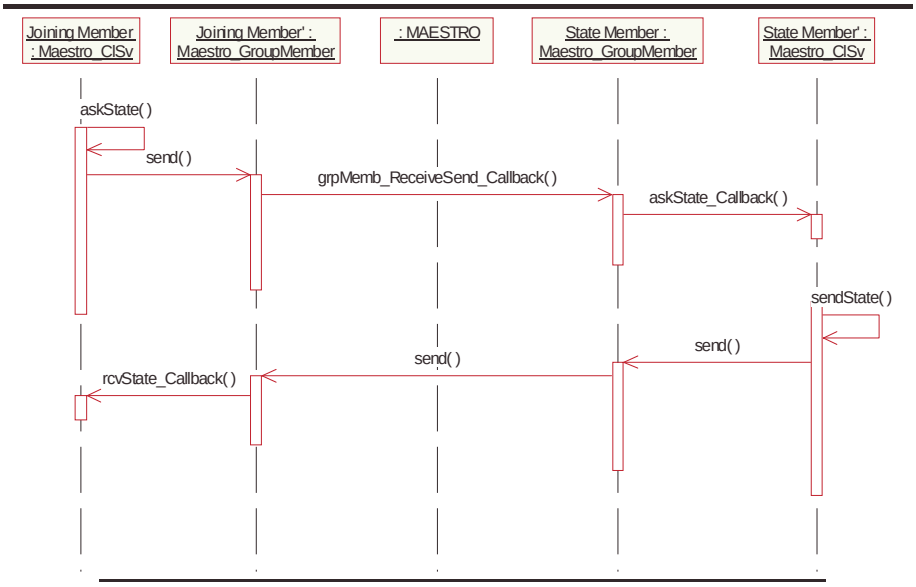

Figura 4.2. Mensajes en *Maestro_ClSv* en una transferencia de estado

- *MAESTRO_ATOMIC_XFER*: sólo los mensajes de transferencia de estado están permitidos durante la transferencia.

Puesto que esta clase realiza la distinción entre cliente y servidor, se incluyen nuevos métodos para enviar mensajes sólo a los servidores (*scast*), aunque se puede especificar una lista de clientes que recibirán también este mensaje.

La vista que se instala diferencia también una lista para los servidores y otra para los clientes; si debe producirse una transferencia de estado, la vista se considera de transferencia de estado y, como tal, se indica en la vista; cuando la transferencia finalice, se instalará una nueva vista. Cada miembro recibe también, con la vista, su papel durante la transferencia: si es emisor, receptor o no se involucra en ninguna transferencia.

La implementación de transferencia de estado sigue el paradigma *pull*: el nuevo miembro debe pedir a miembros antiguos el estado, siendo posible pedirlo por porciones. Es también responsabilidad del nuevo miembro decidir cuándo ha finalizado la transferencia. El nuevo miembro [figura 4.2] pedirá el estado a un determinado servidor invocando la operación *askState* definida en *Maestro_ClSv*, pudiendo incluir un mensaje donde especifica la porción de estado requerida. El servidor recibe la notificación de la petición de estado con *askState_Callback* y deberá responder, eventualmente, usando el método *sendState*. El miembro que espera el estado lo recibe con *rcvState_Callback*, pudiendo solicitar entonces nuevas porciones del estado o, si considera que la transferencia ha finalizando, notificando este evento con el método *xferDone*.

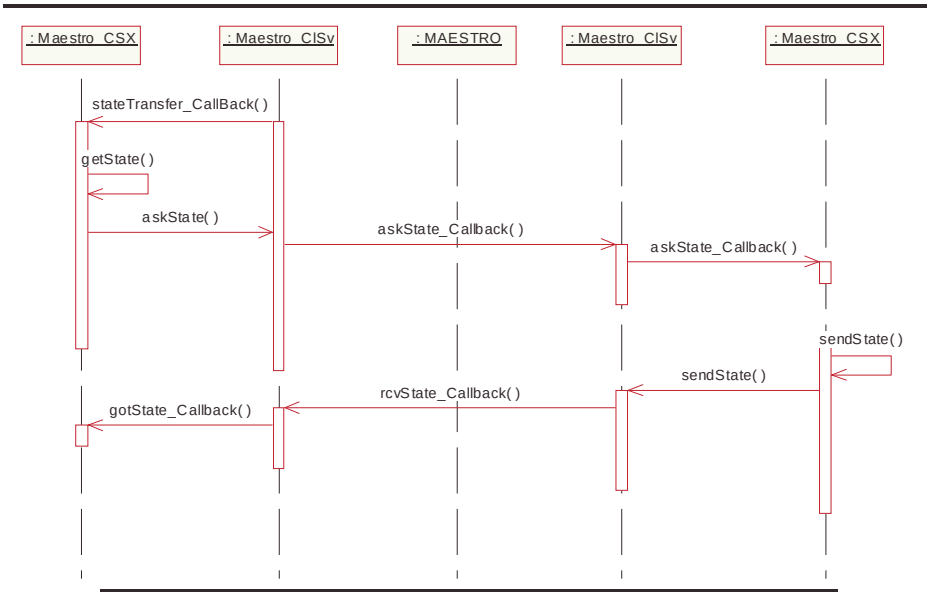

Figura 4.3. Mensajes en *Maestro_CSX* en una transferencia de estado

Durante la transferencia, los mensajes considerados no seguros son introducidos en una cola y serán enviados cuando finalice la transferencia. Este encolamiento lo realiza *Maestro* de forma automática.

Este diseño proporciona una gran flexibilidad a la aplicación. La transferencia se puede hacer en porciones, incluso podrían requerirse simultáneamente diferentes porciones a distintos miembros con estado. Si es preferible que la transferencia la realice un miembro con estado determinado, el coordinador puede enviar esta información cuando la vista se instale, ya que la vista se envía primero al coordinador con *clSv_ViewMsg_Callback*, que la devuelve a *Maestro* junto a un mensaje que será retransmitido a todos los miembros con el método *clSv_AcceptedView_Callback*. Y la aplicación puede decidir qué mensajes son seguros para el estado y pueden ser enviados durante la transferencia. Como no todos los mensajes son bloqueados bajo este esquema, el orden de procesamiento no será el especificado por los requisitos de la aplicación. Además, como los mensajes no son bloqueados en recepción, los mensajes provenientes de la vista anterior sí serán recibidos por los miembros.

Maestro_ClSv da posiblemente demasiada flexibilidad a la aplicación, que debe decidir qué miembro le transmite el estado o qué hacer si se cae ese miembro durante la transferencia. Por esta razón, existe una tercera clase denominada *Maestro_CSX*, subclase de la anterior, que simplifica la transferencia con la siguiente interfaz [figura 4.3]:

- *stateTransfer_Callback*: la invoca automáticamente *Maestro* sobre un miembro nuevo cuando debe iniciar una transferencia. Hasta el momento, sólo se ha

considerado la inserción de nuevos miembros en el grupo, pero estas operaciones son también invocadas si se produce una unión de dos particiones del mismo grupo. De hecho, tras la unión de particiones, antiguos miembros con estado pueden pasar a ser considerados como miembros sin estado y precisar una transferencia. *Maestro* asocia a cada transferencia una identidad, por lo que si se cae el miembro que transfiere el estado a un determinado nuevo miembro, este último recibirá de nuevo una notificación *stateTransfer_CallBack* con una nueva identidad de transferencia.

- *getState*: tras recibir una notificación para que se inicie una transferencia, el nuevo miembro solicita el estado con este método. Debe especificar la identidad de la transferencia, pero no debe preocuparse por encontrar un miembro con estado que se lo suministre, pues está implícito con la identidad de transferencia. Aunque se puede solicitar el estado en porciones, esta operación no es reentrante, es decir, deberá solicitarse secuencialmente.
- *askState_Callback*: el miembro con estado es notificado a través de este método para enviar la porción de estado requerida, respondiendo con mensajes *sendState*.
- *sendState*: envía el estado, o una porción de éste.
- *gotState_Callback*: el nuevo miembro recibe el estado con este método, pudiendo, a continuación, solicitar nuevas porciones de estado.
- *xferCanceled_Callback*: este método permite saber al nuevo miembro que la transferencia en curso ha sido cancelada. Posteriormente, recibirá una notificación para iniciar una nueva transferencia.
- *xferDone*: el nuevo miembro invoca este método cuando considera finalizada la transferencia.
- *resetState*: reinicializa el estado.

Esta clase simplifica, por lo tanto, las transferencias de estado en aquellas aplicaciones que no requieran especiales condiciones de transferencia.

Sin embargo, la razón por la que esta transferencia de estado no funciona en todos los casos es por un problema que proviene de la clase inicial, *Maestro_GroupMember,* que precisamente es la única que no soporta transferencia. En esta clase, se escoge al primer miembro de la vista y se le envía la nueva vista; éste la devuelve junto con un mensaje que es luego retransmitido a los demás miembros. Este mensaje es el utilizado por las subclases para enviar la información extendida de vista: la lista de servidores y la lista de clientes. Si el primer miembro de la lista es un miembro nuevo en el grupo, la vista que enviará es una donde ningún miembro tiene estado, pues no tiene constancia de que hubiera ningún miembro con estado: es una situación equivalente a un grupo que se crea inicialmente con *n* miembros. En estas condiciones, decidirá que no hace falta realizar ninguna transferencia y el protocolo general falla.

La base de este problema se encuentra en una suposición errónea: la lista de miembros viene ordenada, siendo el miembro más antiguo el de menor rango. Esta suposición era válida con *Horus*, pero no lo es con *Ensemble*, donde el primer miembro de la lista puede ser el miembro recién incorporado al grupo.

4.2.2. Versión 0.61

A partir de la versión 0.60, se resolvió el problema que planteaba las listas de miembros cuyo orden difería en cada vista. El método empleado es que todo miembro con estado envíe un mensaje multipunto con su estado. Para resolver cuál es el estado más actualizado, los estados se asocian a un número de versión dado por el par { número de mensajes recibidos, número de vistas aceptadas }. Cuando un servidor recibe un mensaje de estado actualiza el suyo propio si observa que es necesario.

De esta forma, el protocolo se simplifica enormemente, y se ejecuta en una sola fase, facilitando además el problema de conciliación de estados en caso de particiones. El rendimiento que se obtiene es aceptable si los cambios de vista son poco frecuentes, no hay muchos miembros y el estado es pequeño. Toda la flexibilidad que ofrecía el anterior protocolo se pierde, pero el equipo de *Ensemble* había comprobado que la mayoría de los usuarios de *Maestro* caían en la categoría de las aplicaciones cuyo rendimiento bajo el nuevo protocolo era aceptable y que se beneficiaban de la simplicidad del nuevo esquema.

Bajo estas condiciones, las clases *Maestro_ClSv* y *Maestro_CSX* no se emplean; en su lugar se utiliza *Maestro_GroupListener*, que implementa dos métodos básicos: *getState* y *setState*.

4.3. CORBA

La última especificación de CORBA soporta tolerancia a fallos mediante una especificación denominada *Fault Tolerant CORBA Specification v1.0* [OMG00], que no tiene aún el respaldo de una distribución comercial compatible. Por esta razón pueden esperarse cambios en la especificación, según se adquiere experiencia con la implementación. Además, la definición de la arquitectura incluye inicialmente la posibilidad de extensiones propietarias a la especificación, lo que limita la interoperabilidad entre distribuciones.

La tolerancia a fallos se consigue mediante la redundancia de objetos, introduciéndose una referencia especial a grupos de objetos, IOGR (*Interoperable Object Group Reference*), no definida en la especificación CORBA básica. Cada grupo se define con un conjunto de propiedades de tolerancia a fallos, como:

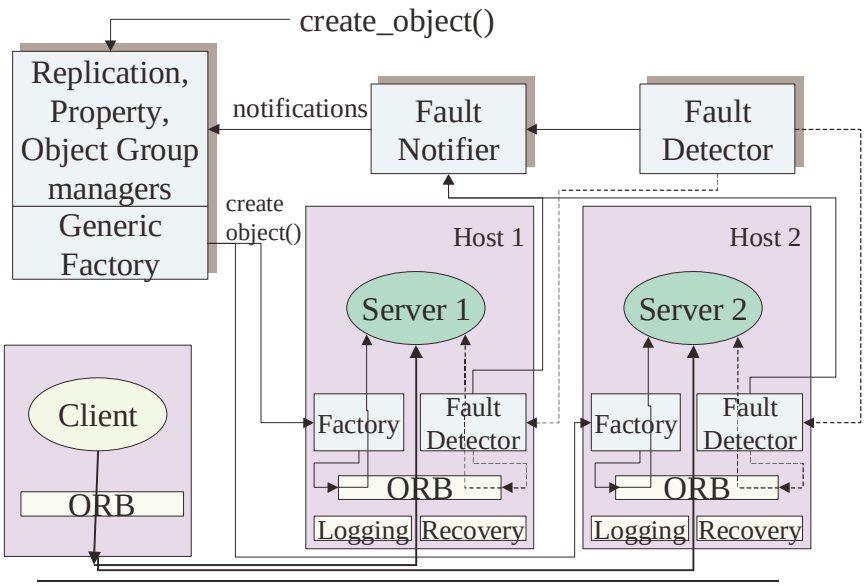

Figura 4.4. Arquitectura de tolerancia a fallos en CORBA

- Estilo de replicación: puede ser COLD_PASSIVE, WARM_PASSIVE, ACTIVE, STATELESS, ACTIVE_WITH_VOTING. La especificación actual no soporta este último estilo.
- Número inicial de réplicas.
- Mínimo número de réplicas.
- Tipo de pertenencia a grupo (*membership style)*: puede ser controlada por la infraestructura de CORBA o controlada por la misma aplicación, según se necesite un mayor o menor grado de transparencia.
- Estilo de consistencia: la consistencia puede estar también directamente controlada por la infraestructura o por la propia aplicación.

En los grupos directamente controlados por CORBA (*infrastructure-controlled membership*), la replicación de objetos resulta substancialmente transparente a la aplicación: la infraestructura crea nuevas réplicas o elimina miembros de forma automática, aunque la aplicación aún tiene determinado control sobre estas operaciones, violando en ese caso la transparencia. En los grupos directamente controlados por la aplicación (*application-controlled membership*), es la aplicación la que debe crear las nuevas réplicas e incluirlas en el grupo, así como eliminar miembros del grupo.

Esta arquitectura se muestra en la figura 4.4; dos servidores se encuentran replicados en dos máquinas diferentes. Los clientes acceden simultáneamente a ambos al emplear su IOGR. La aplicación solicita la creación de un objeto replicado

invocando la operación *create_object* sobre el *Replication Manager*, o controlador de replicación, que invoca a su vez al *GenericFactory*, o factoría genérica, que transforma esa solicitud en nuevas invocaciones a las factorías particulares que residen en cada máquina. El número de factorías invocadas, así como su localización, es dependiente de las propiedades del grupo. La aplicación puede, por otro lado, acceder directamente al controlador de grupos de objetos (*ObjectGroupManager*) invocando operaciones como *create_member*, *add_member* o *remove_member* y teniendo de esta manera más control sobre la adición y eliminación de miembros, así como sobre su localización. Estos controladores se encuentran replicados sobre diferentes máquinas, obteniéndose así uno de los requisitos de la especificación: que no haya un único punto de fallo en todo el sistema. De la misma forma se encuentra replicado el notificador de fallos (*Fault Notifier*) y el detector de fallos.

Sólo en los grupos activos todos los miembros responden a cada petición de servicio; si se definen como pasivos, un único miembro, definido como primario, recibe las peticiones y es, asimismo, el que las responde; si este miembro primario se cae, uno de los miembros pasivos se promociona a primario. Cuando el estilo de consistencia se ha definido como controlado por la aplicación, el miembro pasivo es automáticamente definido como primario y es responsabilidad de la aplicación el alcanzar la consistencia de estados. Sin embargo, si es controlada por la infraestructura, entran en funcionamiento los mecanismos de *Logging And Recovery*. Bajo estos mecanismos, todos los mensajes que son enviados al miembro primario son guardados para poder ser recuperados más tarde si hace falta reconstruir el estado. Periódicamente, estos mecanismos invocan el método *get_state* en la réplica primaria, que debe soportar la interfaz *Checkpointable*, de tal forma que la reconstrucción del estado no tenga que pasar por el procesado de todos los mensajes; cuando la réplica pasiva se promociona a primaria, recibe el último estado almacenado mediante la operación *set_state* y, a continuación, recibirá todos los mensajes que el anterior miembro primario procesó antes de caerse, garantizándose así la consistencia de sus estados. Es por tanto necesario que la aplicación sea determinista; obviamente, si el grupo es *STATELESS*, estos mecanismos no son necesarios. El intervalo con que se obtiene el estado de la aplicación es configurable.

La diferencia entre grupos pasivos *fríos* o *calientes* es que en los primeros, los miembros pasivos reciben sólo el estado cuando se promocionan a primarios. Si el grupo se ha definido como *WARM_PASSIVE*, esta transferencia se realiza con cada operación. En ambos casos se garantiza que al final de cada transferencia de estado, cada miembro del grupo involucrado en la transferencia tenga el mismo estado que presenta o presentaba el miembro primario. En el caso de grupos activos, la consistencia de estado requiere que este estado sea el mismo tras cada petición de servicio, por lo que precisan de un sistema fiable de comunicaciones multipunto bajo el modelo de sincronía virtual. El particionado de la red y, por lo tanto, del grupo no se soporta, es decir, no se incluye el modelo de sincronía virtual extendida en la especificación.

La replicación activa garantiza una rápida respuesta en caso de caída de miembros. La pasiva fría resulta más cómoda de emplear y, de hecho, puede resultar completamente transparente, pero requiere de mayores tiempos de espera si se cae el miembro primario, mientras se reconstruye el estado anterior. En el caso de grupos pasivos calientes, la promoción a primario es inmediata pero, puesto que el estado se transfiere con cada operación, resulta efectiva sólo cuando la transferencia del estado no implica más recursos que el procesado de cada mensaje, que es el caso de estados pequeños, principalmente.

La interfaz *Checkpointable* se define como:

```
interface Checkpointable {
       State get_state() raises (NoStateAvailable);
       void set_state(in State s) raises (InvalidState);
};
```

La especificación define asimismo una segunda interfaz para acelerar el proceso de almacenamiento y recuperación de estado en caso de estados grandes.

```
interface Updateable : Checkpointable {
       State get_update() raises (NoUpdateAvailable);
       void set_update(in State s) raises (InvalidUpdate);
};
```

Esta interfaz no implica que el estado se pueda transferir en varios pasos, pues la primera operación sigue siendo g*et_state*, que debe retornar el estado completo. La recuperación del estado involucra, en este caso, una operación *set_state*, seguida por un número indeterminado de operaciones *set_update* y una serie de mensajes ya procesados por el anterior miembro primario.

La figura 4.4 muestra dos de los componentes de esta arquitectura, *Logging* y *Recovering* (traceado y recuperación), que residen por debajo del ORB. Consecuentemente, esta especificación no define un servicio CORBA genérico que pueda instalarse sobre cualquier ORB y el servicio será, al menos inicialmente, dependiente de la distribución empleada, en lo referente al lado servidor. Además, todos los objetos replicados pertenecientes a un mismo dominio deben residir sobre ORBs del mismo distribuidor. En el lado cliente, este servicio afecta y modifica la especificación global CORBA, al introducir la referencia de grupos (IOGR) por lo que clientes que residan sobre previos ORBs no participarán en los beneficios de la tolerancia a fallos. Estos clientes pueden aún invocar los servicios sobre uno de los servidores, pero la solicitud no se dirigirá a los demás servidores replicados, por lo que si aquel servidor ha caído, el cliente verá directamente el fallo. Si estos ORBs no disponen de todas las características necesarias para soportar como clientes la tolerancia a fallos, es todavía posible que puedan obtener del IOGR las referencias a los servidores replicados y, en caso de fallo de uno, reintentar la solicitud a otros servidores, con lo que obtiene, al menos, soporte parcial de tolerancia de fallos.

4.4. Conclusiones

El modelo de sincronía virtual impone la transferencia de estado como una de las propiedades inherentes al sistema. No define, sin embargo, restricciones o requisitos sobre cómo realizar esa transferencia. Aun así, hay sistemas de comunicaciones fiables de grupo, como *xAmp*, *Transis* o *Spread*, que no la soportan internamente.

La transferencia de estado puede desarrollarse externamente al sistema de comunicaciones en grupo, utilizando las primitivas de comunicaciones que éste soporta. Sin embargo, como demostramos en un capítulo posterior, el modelo de sincronía virtual puede respetarse en estos casos sólo cuando el sistema de comunicaciones exterioriza parte de sus comunicaciones internas. Más precisamente, se requiere que este sistema informe a los miembros del grupo no sólo de los cambios de vistas, sino también del bloqueo de estas vistas cuando se inician las comunicaciones internas para acordar una nueva vista. Varios de los sistemas expuestos en esta sección soportan este evento incluso cuando ya soportan internamente la transferencia de estado y a pesar de no ser uno de los eventos definidos en el modelo de sincronía virtual.

En el caso de *JavaGroups*, el modelo soporta directamente la transferencia de estado, pero ésta se diseña sobre un sistema interno de comunicaciones y la transferencia en sí no la dirige *JavaGroups*, sino la aplicación misma. De esta forma, ocurre lo mismo que con *xAmp*, *Transis* o *Spread* y, bajo ciertas circunstancias, el modelo de sincronía virtual no se respeta.

La mayoría de los sistemas expuestos sí soporta con mayor o menor flexibilidad la transferencia de estado. En este caso, se emplean dos métodos alternativos y solamente *Totem* soporta ambos simultáneamente. El primer método es el almacenamiento de estados intermedios y mensajes de tal forma que sea posible reconstruir el estado final de un objeto sin necesidad de acceder a otras réplicas. Este planteamiento se sigue en CORBA y en *Cactus*, siendo atractivo al no bloquear en ningún momento ninguna de las réplicas activas. Sin embargo, requiere tiempo de procesado continuo por parte de la aplicación, donde cada réplica debe almacenar periódicamente su estado y el sistema debe mantener una lista de los mensajes procesados. Además, puede resultar más lenta la activación de la nueva réplica.

El método alternativo es la transferencia de estado entre réplicas. Este planteamiento es necesario en el caso del modelo de sincronía virtual extendida, donde distintas particiones deben acordar un estado común, que cada réplica asume entonces. Al emplear transferencia entre réplicas, es común bloquear todo el grupo (*Arjuna*, *Phoenix*, *Electra*, *CyberBus*) y realizar la transferencia en un solo paso.

Soportar transferencias en varios pasos, que puede resultar imprescindible en el caso de réplicas complejas, es funcional en varios sistemas, principalmente en la versión original de *Maestro* y en el complejo *Totem*. En el primer caso es sólo posible

emplear una transferencia *pull*, donde el nuevo miembro elige a su coordinador, aunque admite flexibilidad en la forma del bloqueo, pues permite bloquear mensajes específicos. En el caso de *Totem*, la aplicación puede seleccionar el método de bloqueo, bloqueando a todas las réplicas, a las réplicas que realizan la transferencia, o incluso a ninguna.

Sensei se enfoca en el problema de la transferencia de estado, estudiando las diferentes posibilidades y ofreciendo unos protocolos con la mayor flexibilidad posible. Esta flexibilidad incluye la elección de protocolos *push* o *pull* que afectan al rendimiento del sistema, la posibilidad de elegir al coordinador de cada transferencia y el poder realizar transferencias concurrentes simultáneamente, el soportar transferencias en varios pasos e incluso interrumpirlas y continuarlas en caso de cambios de vistas, el empleo de propiedades, o estado asociado al grupo como una entidad, o el poder bloquear el sistema y los mensajes entre réplicas, etc. Define los protocolos a bajo nivel que permiten esta flexibilidad y se estudia su rendimiento bajo diferentes condiciones, al ser muy distinto el entorno de una red local que un grupo comunicándose en Internet. Define también los protocolos a alto nivel, especificándose una interfaz de aplicación que soporta distintos grados de complejidad, de tal forma que las aplicaciones que precisan una transferencia de estado simple no deban soportar una interfaz compleja. Y estudia finalmente las condiciones de la transferencia de estado, esencial para entender las necesidades de bloqueo del sistema y cómo éstas afectan al modelo de sincronía virtual.

Capítulo 5 - Condiciones en la Transferencia de Estado

El principal propósito de este capítulo es estudiar la transferencia de una forma teórica, comprobando las condiciones que deben verificarse en cada momento en respuesta a los distintos eventos que se dan en el grupo durante esa transferencia.

El objetivo de la transferencia de estado es que todos los miembros del grupo alcancen un estado consistente, cuando uno o más miembros con estado no inicializado se unen a un grupo ya consistente. El grupo inicial podría tener miembros sin estado consistente sólo si ya había una transferencia en marcha cuando el nuevo miembro fue incluido en el grupo.

Esta definición del problema excluye ya una solución: que el futuro miembro contacte primero a un miembro del grupo para recibir el estado y, tras recibirlo, se una al grupo. Esta solución requeriría que el miembro contactado registrara los cambios producidos en su estado desde el momento en que lo transfiere hasta que el nuevo miembro se une efectivamente al grupo. Y esos cambios deberían entonces ser transferidos al nuevo miembro, lo que supone de nuevo un problema de transferencia de estado tal como se ha formulado.

Aunque la transferencia de estado puede contemplarse como una comunicación bidireccional entre dos conjuntos no vacíos de miembros en el mismo grupo, delimitaremos primero el problema a las comunicaciones unidireccionales necesarias para inicializar el estado del nuevo miembro con el estado compartido por

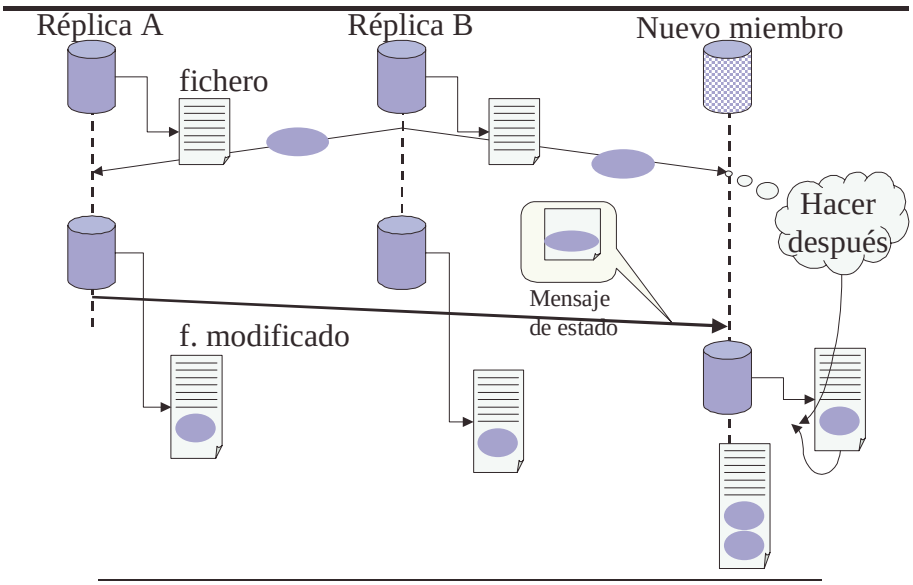

Figura 5.1. Mensaje con estado enviado tras recibir y procesar el mensaje con la modificación.

las réplicas ya inicializadas. El dominio del problema es un grupo bajo el modelo de sincronía virtual, donde algún nuevo miembro se une al grupo, lo que asume que hay ya al menos un miembro con estado inicializado. La solución debe emplear las primitivas de comunicaciones de grupo soportadas por el modelo de sincronía virtual.

Para entender este problema, consideraremos el ejemplo de un sistema distribuido de ficheros, donde cada réplica mantiene una copia de parte de los ficheros que el sistema sirve. Las réplicas siguen el modelo de sincronía virtual, empleando mensajes fiables con orden total: cada réplica ve los mismos mensajes en el mismo orden. Cuando una réplica se une al grupo, recibe los ficheros desde otra, en uno o más mensajes. Para los propósitos de este ejemplo, supondremos que una tercera réplica envía, durante esa transferencia, un mensaje para modificar un determinado fichero; este mensaje se interpreta como "*añadir en una determinada posición del fichero actual la porción de texto que se adjunta*". Es decir, no es un mensaje idempotente: si se procesa dos veces, el texto se añade dos veces.

La figura 5.1 muestra un hipotético escenario, donde una réplica envía el mensaje de estado tras recibir y procesar un mensaje que modifica ese estado. Este mensaje contiene, por consiguiente, el estado final en que debería quedar la nueva réplica. El nuevo miembro recibe dos mensajes, el de estado y el de modificación de estado, y no puede saber si el primero incluye ya en su estado las modificaciones incluidas en el segundo. En este escenario, recibe primero el mensaje de modificación del fichero y lo guarda para procesarlo tras recibir el estado: terminará consecuentemente en un estado inconsistente, pues incluirá las modificaciones sobre el estado ya modificado.

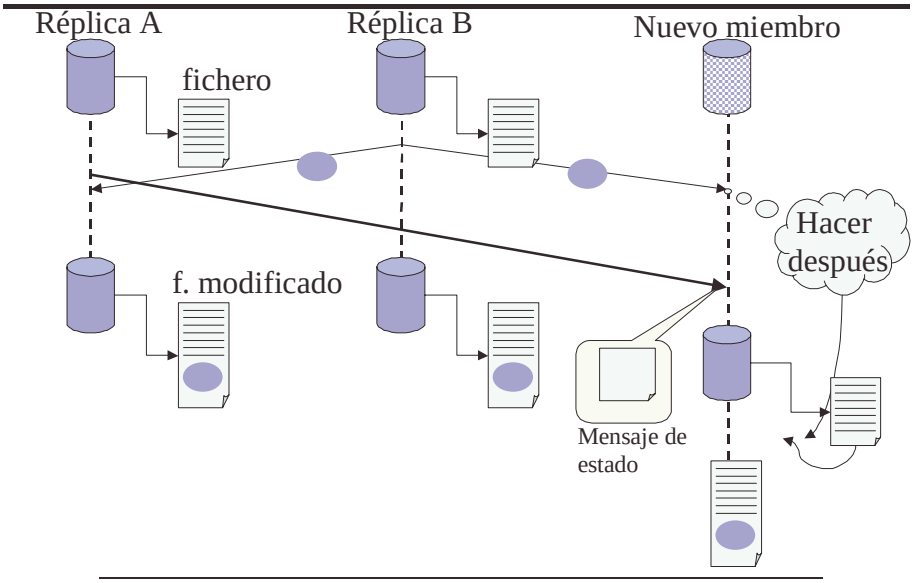

Figura 5.2. Mensaje con estado enviado antes de recibir el mensaje con la modificación

Este caso se solucionaría no procesando el segundo mensaje, pero el escenario dibujado en la figura 5.2 muestra el resultado de descartar ese mensaje bajo condiciones opuestas. En este caso, el mensaje de estado es enviado antes de procesarse el mensaje de modificación, por lo que la réplica que se une al grupo debe procesar ese mensaje para terminar en un estado consistente. Hay otros escenarios donde el mensaje no se guarda, pero en esos casos se pueden alcanzar igualmente estados inconsistentes.

La solución más sencilla sería bloquear el sistema completo durante la transferencia, de tal forma que no se pudieran enviar mensajes que modificaran el estado, pero el rendimiento del sistema disminuirá dramáticamente si el estado a enviar es grande o se producen abundantes cambios de vistas. En el anterior ejemplo, resulta evidente que sería suficiente con que la réplica que realiza la transferencia almacenara todos los mensajes recibidos hasta enviar el estado; el nuevo miembro también los almacenaría para procesarlos tras recibir el estado, con lo que ambos miembros terminarían en un estado consistente. El problema se complica si el estado debe ser enviado en varios mensajes y la réplica que lo envía se cae antes de completar la transferencia.

Como se ha descrito en el capítulo anterior, la transferencia desde otra réplica no es el único mecanismo posible para realizar la transferencia, pues el estado puede reconstruirse a partir de archivos de estado, como es el caso de *Cactus* [Schlichting93] o la especificación de tolerancia a fallos de CORBA. Esta aproximación implica que las réplicas deben emplear continuamente tiempo de proceso, en lugar de exclusivamente en los momentos en que la transferencia debe ser realmente realizada. Pero además, los archivos que guardan el estado deben

estar también replicados, lo que implica que a un nivel inferior existe todavía una transferencia de estado entre réplicas.

Las siguientes secciones estudian las condiciones que deben ser satisfechas por los grupos y sus mensajes en las transferencias de estado desde una o más réplicas activas. Es necesario generalmente bloquear las réplicas involucradas en la transferencia, excepto en los casos más simples, pero el bloqueo y filtrado de los mensajes dependen del tipo de orden empleado en los mensajes. También se consideran las transferencias en varios pasos, un requisito importante cuando el estado del grupo es suficientemente grande.

5.1. Modelo y definiciones

Este estudio de la transferencia de estado muestra que, incluso si no se bloquea ningún miembro durante la transferencia, es posible reconstruir el estado a partir de un conjunto de mensajes y un mensaje que contiene un estado alcanzado tras haber procesado un subconjunto del primer conjunto de mensajes. Debe notarse que el orden en que esos mensajes son recibidos depende de las restricciones de orden de la aplicación y, excepto cuando se emplea orden total, ese orden de los mensajes no puede predecirse en absoluto. Aunque la anterior conclusión puede resultar natural, formulamos el problema y demostramos una solución con una perspectiva formal. Ésto requiere introducir un modelo para el sistema distribuido con componentes replicados, así como la correspondiente notación básica.

Cada miembro tiene un estado, que contiene una parte común a todas las réplicas y una parte privada. La transferencia de estado sólo considera esa parte común. Dos miembros tienen estados consistentes si, en ausencia de nuevos mensajes en el grupo u otra interacción externa, el procesado de todos los mensajes pendientes sitúa a ambos miembros en el mismo estado final. El *mensaje de estado* es el mensaje enviado a un miembro que se incorpora al grupo, desde un miembro ya con estado al que llamamos *coordinador de la transferencia* o simplemente *coordinador*. Ese mensaje incluye el estado del coordinador en un momento dado, y puede haber varios mensajes de estado si éste es enviado en varias partes.

El modelo de sistema distribuido en que se realiza la transferencia de estado consiste en componentes replicados en grupos y un GMS que sigue las siguientes reglas:

- Los grupos siguen el modelo de sincronía virtual. No se considera sincronía virtual débil [Friedman95].
- Los cambios de estado en un miembro debido a cualquier interacción externa deben propagarse al resto de miembros en el grupo, usando las primitivas de comunicaciones de grupo. Son posibles dos comportamientos: dinámicamente no

uniforme, donde un miembro cambia su estado y lo comunica al grupo, y comportamiento dinámicamente uniforme, donde realiza primero la comunicación al grupo y entonces cambia su estado. En el segundo caso, consideramos que el miembro recibe su propio mensaje, que procesa al mismo tiempo que el resto de miembros en el grupo.

- Comportamiento determinista: dos miembros con el mismo estado que reciben los mismos mensajes, en el orden especificado por la aplicación, evolucionarán al mismo estado final.
- Los miembros que se incorporan al grupo no realizan ninguna acción antes de que la primera vista se instale y el miembro sea aceptado en el grupo.

La interacción entre el GMS y las réplicas se modela de la siguiente forma:

- Los miembros reciben secuencialmente los mensajes provenientes de otros miembros o el GMS: un mensaje cada vez, sin recibir el siguiente antes de haber procesado el primero. Si los miembros almacenan internamente los mensajes, el GMS los considerará procesados.
- El GMS instala una nueva vista sólo cuando todos los mensajes en la vista anterior han sido procesados. Es decir, el GMS está bloqueado en tanto que la vista queda instalada.
- El GMS informa a los miembros de la anterior condición de bloqueo: una vez que el GMS queda bloqueado, los mensajes enviados por un miembro son sólo recibidos por el grupo en la siguiente vista.
- Durante el periodo de bloqueo, un miembro todavía puede enviar mensajes, pero son almacenados por el GMS, que los enviará a los miembros en el grupo una vez que se instala la vista. Los mensajes enviados por un miembro pueden ordenarse *fifo*, pero los mensajes enviados por diferentes miembros durante este periodo de bloqueo se consideran concurrentes.

La siguiente notación expresa más formalmente la anterior discusión:

Estado: $S=s+s'$. El estado del miembro contiene una parte global (s) y una parte privada (s'). El problema de la transferencia de estado sólo se aplica a la parte global.

Miembros del grupo: $G=\{g_1, g_2 \ldots g_n\}\ (n \geq 1)$. El grupo incluye a todos los miembros, con o sin estado. $GS=\{g1, g2 \ldots g_m\}\ /\ (n \geq m)$ representa la lista de miembros inicializados, que comparten un estado común, y puede ser vacía. Los miembros no inicializados vienen dados por *GU*, que puede ser igualmente vacío. Esta notación asume que los miembros son incluidos directamente en el grupo y la transferencia de estado sólo es iniciada tras la inclusión del miembro en el grupo.

Sucesión de estados: $s_{iv}=\{s_{iv1}, s_{iv2} \ldots s_{ivl}\}$ Cada miembro *i* tiene en cada vista *v* una sucesión de estados. Cuando la información de la vista no es necesaria, la eliminamos de la notación: $s_i=\{s_{i1}, s_{i2} \ldots s_{il}\}$. En un sistema fuertemente síncrono, es

decir, con sincronía virtual fuerte y mensajes dinámicamente uniformes con orden total causal, todo miembro inicializado observa la misma secuencia de estados: $s_{in} = s_{jn}, \forall i,j \leq m$.

Mensajes:

- M_{ij} es el conjunto ordenado de mensajes ya procesados por *j* pero no por *i*, que ha sido enviado o debe ser enviado a *i*, si *j* no cae primero.
- $M_i = \sum_{j \neq i} M_{ij}$: son los mensajes que deben ser procesados por *i* porque los demás miembros en el grupo los han procesado ya.
- $M_{ij} = \phi, \ j \geq m$: los miembros que se suman al grupo no realizan ninguna acción antes de ser aceptados en el grupo.
- Si se diseña un grupo para tomar acciones exclusivamente tras recibir un mensaje, ningún miembro puede haber procesado algún mensaje que otros miembros no hayan procesado cuando una nueva vista es instalada. Estos miembros reaccionan a eventos externos enviando mensajes al grupo, y tomando la acción correspondiente sólo cuando el mensaje es recibido. Por lo tanto: $M_{ij} = \phi, \forall i,j$. Esos mensajes son mensajes *loopback*: los mensajes dinámicamente uniformes o procesados con orden total son mensajes *loopback*.
- $M_{ij} = \phi, \ \forall i,j \leq m \Rightarrow s_{iv1} = s_{jv1}, \forall i,j \leq m$.: Si M_{ij} es vacío para cualquier par de miembros, todo miembro con estado tiene el mismo estado tras instalarse una vista.
- M_{np} es el conjunto ordenado de mensajes que han sido ya enviados por algún miembro en el grupo, pero que ningún miembro ha procesado todavía.
- Orden de mensajes: $M \oplus M'$ significa procesar ambos grupos de mensajes en el orden especificado por la aplicación, en tanto $M + M'$ significa procesar primero M y luego M'.
- Conjuntos de mensajes. $M = M'$ significa que ambos conjuntos de mensajes son idénticos, incluyendo el orden de los mensajes. $M \approx M'$ significa que ambos conjuntos incluyen los mismos mensajes, pero el orden puede ser diferente, dependiendo de las restricciones de la aplicación.
- Subconjuntos de mensajes: $\overline{M}$ define un subconjunto de mensajes en M.

Mensaje de estado. $ms \ / \ s_O + ms = s$: un miembro en su estado inicial que recibe un mensaje de estado adopta el estado incluido en ese mensaje. Si el estado se envía en varias partes, la anterior regla se especifica como: $s_O + (ms_1 \oplus ms_2 \oplus \ldots \oplus ms_n) = s$. En general, ms_i^j se refiere a la porción *j*-ésima del estado s_i, y M^j es el grupo de mensajes en M que modifican la porción *j*-ésima; debe notarse que esos mismos mensajes podrían modificar igualmente otras porciones del estado, no únicamente la porción especificada, pero no puede haber otros mensajes que modifiquen esa porción dada.

5.2. Requisitos para la transferencia

Comenzamos imponiendo algunas restricciones que luego eliminamos: no se producen cambios de vistas durante la transferencia, el estado se envía en un único mensaje y no se envían mensajes en el grupo durante la transferencia. Finalmente, todos los mensajes en el grupo, incluyendo el mensaje de estado, siguen el mismo orden. Los siguientes requisitos obvian los mecanismos que deben implementarse para detectarse cuándo debe iniciarse una transferencia o cuándo debe darse una transferencia por concluida.

Suponemos tres miembros, uno de los cuales debe recibir el mensaje de estado. Los miembros con estado, *g1* y *g2* pueden presentar un estado diferente, pero deben ser consistentes y, consecuentemente, concluirán en el mismo estado final tras procesar todos sus mensajes pendientes. Los mensajes pendientes son aquellos enviados en la vista anterior durante el periodo de bloqueo, y son diferentes para cada miembro:

$$g_1 : Mnp \oplus M_1 \quad ,siendo \quad M_1 = M_{12} \qquad g_2 : Mnp \oplus M_2 \quad ,siendo \quad M_2 = M_{21}$$

Cualquiera de los miembros con estado puede ser el coordinador. Cada miembro procesará un subconjunto de los mensajes pendientes y alcanzará un estado intermedio, que puede ser diferente en ambos miembros. En ese momento el coordinador enviará el mensaje de estado, que contendrá ese estado intermedio alcanzado. A continuación, procesará el resto de los mensajes. Dividiendo el conjunto de mensajes a procesar en dos subconjuntos, es posible escribir:

$$g_{1:}\begin{bmatrix} mensajes: & Mnp \oplus M_{12} = M_A + M_B \\ estado: & \begin{bmatrix} s_{I1} + M_A = s_1 & (estado\ \ enviado) \\ s_1 + M_B = s_F \end{bmatrix} \end{bmatrix} \qquad g_{2:}\begin{bmatrix} mensajes: & Mnp \oplus M_{21} = M_C + M_D \\ estado: & \begin{bmatrix} s_{I2} + M_C = s_2 & (estado\ \ enviado) \\ s_1 + M_D = s_F \end{bmatrix} \end{bmatrix}$$

El miembro que recibe el estado debe procesar sus propios mensajes pendientes y el mensaje de estado para alcanzar el mismo estado final:

$$g_3 : \begin{bmatrix} mensajes: & Mnp \oplus M_3 = Mnp \oplus M_{31} \oplus M_{32} \quad (M_{31} \approx M_{21} \quad M_{32} \approx M_{12}) \\ estado: & \begin{bmatrix} recibido\ \ de\ \ g_1: & s_O + (Mnp \oplus M_3 \oplus ms_1) = s_F \\ recibido\ \ de\ \ g_2: & s_O + (Mnp \oplus M_3 \oplus ms_2) = s_F \end{bmatrix} \end{bmatrix}$$

Y, en general, para un número indefinido de miembros, los miembros con estado verificarán:

$$g_i / i \le m : \begin{vmatrix} Mnp \oplus M_i = M_A + M_B ; s_{Ii} + M_A = s_i ; s_i + M_B = s_F \\ s_i + M_B = s_O + ms_i + M_B = s_O + ms_i + \overline{(Mnp \oplus M_i)} = s_F \end{vmatrix} \quad [\delta 1]$$

Y los miembros que reciben el mensaje de estado deben cumplir, para cualquier coordinador:

$$g_j / j > m, \forall i \le m : \left| \begin{array}{c} M_j = \sum_k M_{jk} = \sum_{k \neq i} M_{jk} \oplus M_{ji} \approx \sum_{k \neq i} M_{ik} \oplus M_{ji} = M_i \oplus M_{ji} \\ s_O + (Mnp \oplus M_j \oplus ms_i) = s_O + (Mnp \oplus M_i \oplus M_{ji} \oplus ms_i) = s_F \end{array} \right. \quad [\delta 2]$$

Juntando las dos ecuaciones anteriores, obtenemos:

$$\forall i \le m : \quad s_O + ms_i + \overline{(Mnp \oplus M_i)} = s_O + (Mnp \oplus M_i \oplus M_{ji} \oplus ms_i) \quad [\delta 3]$$

Esta es la condición general que debe cumplirse en toda transferencia de estado: debe ser posible reconstruir el estado a partir de un conjunto de mensajes y un mensaje de estado cuyo estado asociado se ha alcanzado tras haber procesado un subconjunto de aquel conjunto de mensajes. Debe tenerse en cuenta que el orden en que esos mensajes y el mensaje de estado son recibidos depende de las restricciones de orden de la propia aplicación y, salvo que se emplee orden total, ese orden no puede predecirse en absoluto.

Un ejemplo es una aplicación replicada que mantiene una lista de clientes y cuya única operación de actualización se realiza mediante el mensaje '*añadir cliente si no está en la lista*'. El mensaje de estado incluye la lista completa de clientes, que constituye el estado de cada réplica. Este ejemplo verifica la anterior condición. Sin embargo, si el grupo permitiera operaciones como '*eliminar cliente*', la anterior condición no se cumpliría ya en todos los casos. Por ejemplo, como se muestra en la figura 5.3, pueden enviarse dos mensajes en el grupo, uno añadiendo un cliente y otro eliminando ese mismo cliente. Un miembro podría procesar el primer mensaje, enviando a continuación el mensaje de estado que incluiría, por lo tanto, ese cliente. A continuación procesaría el segundo mensaje y terminaría en un estado final que no incluye aquel cliente. El nuevo miembro podría procesar primero los dos mensajes, añadiendo y eliminado el cliente, y recibiría a continuación el mensaje de estado. Terminaría así en un estado inconsistente, pues su estado final contendría al cliente supuestamente eliminado.

Eliminando ahora la primera restricción y permitiendo el envío de nuevos mensajes durante la transferencia de estado, se obtiene una condición similar:

$$\forall i \le m : \quad s_O + ms_i + \overline{(Mnp \oplus M_i \oplus M)} = s_O + (Mnp \oplus M_i \oplus M_{ji} \oplus ms_i \oplus M) \quad [\delta 4]$$

Esta ecuación puede simplificarse en [δ5], que puede entenderse como [δ4] cuando el periodo de bloqueo previo al cambio de vista es tan corto que ningún miembro procesa o envía ningún mensaje:

$$M' \equiv Mnp \oplus M_i \oplus M_{ji} \oplus M \quad \Rightarrow \quad \forall i \le m : \quad s_O + ms_i + \overline{M'} = s_O + (ms_i \oplus M') \quad [\delta 5]$$

Debe destacarse que el subconjunto M' de mensajes en la parte izquierda de esta fórmula se compone de aquellos mensajes en M' que todavía no se han procesado en el momento en que el estado se envía, pero que el conjunto completo de mensajes, así como el mensaje de estado, deben ser procesados en la parte derecha de la fórmula. Además, el mensaje de estado debe ser procesado entre dos mensajes cualesquiera en ese conjunto, sin que el orden afecte al estado final

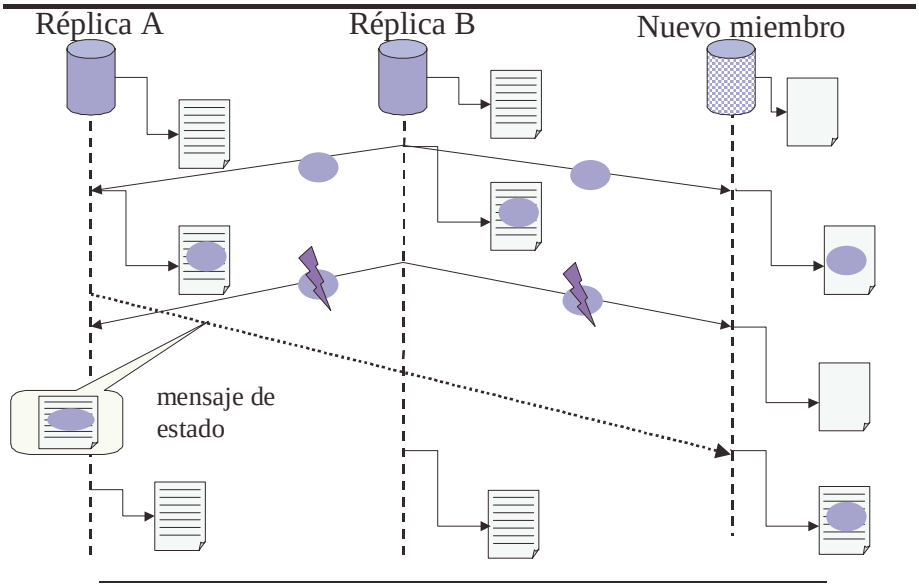

Figura 5.3. Ejemplo de escenario de transferencia con inconsistencia final de estados.

alcanzado. Una posibilidad de verificar esta condición es incluir en el mensaje de estado información sobre los mensajes que se han procesado para alcanzar el estado dado; por ejemplo, si los mensajes están numerados secuencialmente, bastaría con incluir el último número de secuencia.

Un caso interesante es aquél en que el estado se envía inmediatamente tras recibir la nueva vista, sin mensajes intermedios. El otro caso posible, enviarlo cuando se han procesado todos los mensajes pendientes, no resulta tan interesante por dos razones: primero, porque es difícil de implementar en un sistema asíncrono, al ser difícil detectar que no hay más mensajes pendientes y, segundo, porque debería inhibirse el envío de nuevos mensajes mientras se transfiere el estado. Cuando el estado se transfiere inmediatamente tras recibir la notificación de cambio de vista:

$$\left.\begin{array}{l} M'' \equiv \overline{M'} \equiv Mnp \oplus M_i \oplus M \\ M' \equiv M_{ji} \oplus \overline{M'} \equiv M_{ji} \oplus M'' \end{array}\right\} \quad \forall i \le m: \quad s_O + ms_i + M'' = s_O + (ms_i \oplus M_{ji} \oplus M'') \quad [\delta 6]$$

$$si \quad M_i = \phi \quad \forall i \quad \Rightarrow \quad \forall i \le m: \quad s_O + ms_i + M'' = s_O + (ms_i \oplus M'') \quad [\delta 7]$$

Esta última simplificación [δ7] es válida para grupos que envían exclusivamente mensajes *loopback*. Implica que no debe haber ninguna diferencia entre procesar primero el mensaje de estado y a continuación un conjunto de mensajes, o procesar el conjunto entero de mensajes, con el mensaje de estado incluido entre dos cualesquiera de esos mensajes. Un ejemplo de aplicación de grupo que cumple [δ7] es un contador donde todo mensaje tiene un significado relativo, como '*incrementar contador*' o '*decrementar contador*', y donde el mensaje de estado

se entiende también como un incremento relativo: '*incrementar contador en la cantidad especificada por el número dado en el mensaje de estado*'. Incluso este ejemplo sobresimplificado no verifica [δ5], porque el mensaje de estado podría ser enviado tras procesar alguno de los mensajes '*incrementar contador*' en la vista actual, mensajes que serán procesados igualmente por el nuevo miembro, terminando en estados (contadores) finales diferentes.

Las condiciones previas son difíciles de verificar en grupos genéricos. Incluso la condición más simple [δ7] carece de una solución general, porque ningún tipo de orden puede conseguir que el nuevo miembro reciba primero el mensaje de estado y luego los mensajes restantes, lo que resolvería directamente la ecuación.

Proponemos la siguiente solución para validar la transferencia de estado para cualquier tipo de mensajes de grupo y mensajes de estado: el mensaje de estado se envía inmediatamente tras el cambio de vista y el nuevo miembro encola todo mensaje hasta recibir y procesar el mensaje de estado. Una vez que lo procesa, los mensajes encolados se desbloquean y procesan en el orden de entrada. Si el grupo emplea mensajes no-*loopback*, es también necesario descartar M_{ji}, los mensajes que ya habían sido procesados por el coordinador en la vista previa. En ese caso:

$$s_O + (ms_i \oplus M_{ji} \oplus M'') = s_O + ms_i + (M_{ji} \oplus M'') = s_O + ms_i + M''$$

Con lo que se cumple la condición [δ6]. Como sólo deben descartarse los mensajes enviados por el coordinador *previos* al mensaje de estado (los enviados en la vista previa), esta solución implica que cuando el grupo contiene mensajes no-*loopback*, el orden de mensajes en el grupo debe ser, al menos, *fifo*. Y como un grupo puede usar tanto mensajes *loopback* como no-*loopback*, es la misma aplicación quien debe decidir los mensajes a descartar: aquellos que sean no-*loopback*.

Las ecuaciones previas, obtenidas para todo tipo de grupos, no cambian si hay más de un nuevo miembro, y no hay diferencia si hay un coordinador para cada nuevo miembro o un único coordinador para todos los miembros nuevos, y la transferencia se realiza entonces secuencial o concurrentemente. Si se realiza secuencialmente, la solución basada en el envío del estado inmediatamente tras el cambio de vista requiere ahora que el coordinador no procese ningún mensaje hasta que todas las transferencias se completen.

5.2.1. Cambios de vistas

Si se produce algún cambio de vista antes de que el coordinador envíe el mensaje de estado, la condición a verificarse en un grupo genérico sigue siendo la misma que cuando no hay cambios de vistas [δ5]. Para demostrarlo, supondremos que tras enviarse el mensaje de estado, ningún miembro envía mensajes adicionales y no se instalan nuevas vistas en el grupo. El nuevo miembro deberá procesar un conjunto de mensajes *M*. Primero procesará un subconjunto de esos mensajes, a continuación el mensaje de estado y, finalmente, los mensajes restantes.

$$\textit{mensajes}: \quad M = M_A + M_B \qquad \textit{estado}: \quad s_O + M_A + ms + M_B = s_F$$

Pero, si el grupo verifica [δ5], el orden en que el mensaje de estado es procesado no afecta al estado final:

$$s_O + M_A + ms + M_B = s_O + ms + M_A + M_B = s_O + ms + M = s_F$$

Y el mensaje de estado incluye un estado obtenido tras haber procesado un subconjunto de los mensajes en M, exactamente con la solución en [δ5]. Por consiguiente, no se precisan nuevas restricciones en el caso en que se produzcan cambios de vistas antes de que el mensaje de estado se envíe, incluso si el coordinador cae y otro miembro con estado se convierte en el nuevo coordinador.

Nuestra propuesta para resolver la transferencia de estado en grupos genéricos que no cumplen [δ5] implica que, debido al bloqueo de mensajes, los miembros involucrados en la transferencia pueden parecer miembros lentos respecto a los demás miembros en el grupo. Y lo que es más importante, cuando se produce un cambio de vista antes de que la transferencia finalice, una propiedad del modelo de sincronía virtual puede estar siendo violada. Este modelo establece que todo par de procesos que sean miembros de dos vistas consecutivas deben recibir el mismo conjunto de mensajes en el periodo comprendido entre ambas vistas. Cuando la vista cambia, tanto el coordinador de la transferencia como el nuevo miembro que la recibe tienen mensajes encolados que no pueden procesar porque la transferencia aún no ha finalizado. De hecho, los mensajes han sido recibidos, por lo que la propiedad se respeta, pero las aplicaciones en grupo que usen esta aproximación deben tener esta posibilidad en cuenta. El sistema puede modelarse de dos formas:

- El coordinador y el nuevo miembro son considerados como miembros excluidos temporalmente de la vista. Continúan bloqueando los mensajes, que sólo son procesados cuando la transferencia queda completada. Cuando los mensajes sean procesados, procederán probablemente de dos o más vistas pero, incluso así, se procesarán en el orden correcto.
- El nuevo miembro no se considera incluido en la vista en tanto no se le transfiera el estado. En este caso, el coordinador cancelaría la transferencia si se instala una nueva vista, procesando los mensajes encolados y reiniciando la transferencia en la nueva vista. El nuevo miembro debe descartar los mensajes encolados si recibe una nueva vista antes de recibir el mensaje de estado. Como debido a problemas de sincronización el coordinador podría haber enviado ese mensaje de estado antes de percibir que la vista estaba bloqueada, el nuevo miembro debe ser capaz de descartar este mensaje, que recibirá en la siguiente vista. Una posibilidad es incluir la identidad de la vista en el mensaje de estado.

Ambas soluciones excluyen, cuanto menos, al nuevo miembro como perteneciente al grupo. Puede verse entonces como una solución equivalente a aquella en la que el futuro miembro del grupo solicita el estado a un miembro perteneciente al grupo y, cuando lo recibe, se incorpora a ese grupo.

Además, debe considerarse que un miembro que ha encolado los mensajes puede caerse mientras los procesa. Volviendo de nuevo al modelo de sincronía virtual, si un miembro es incluido en una nueva vista debe haber procesado todos los mensajes de la anterior vista. Sin embargo, si un miembro recibe la notificación de nueva vista y decide entonces procesar los mensajes encolados, puede en ese momento caerse, sin haber procesado todos los mensajes que le correspondían, y violando así el modelo de sincronía virtual, afectando a las aplicaciones que confían en este modelo para su correcto funcionamiento. La única solución para este problema implica que los miembros deben recibir la notificación del bloqueo de vista; en este momento procesarán los mensajes pendientes y, si se cayeran durante esta fase, no aparecerían como miembros de la nueva vista. Debe notarse que el modelo de sincronía virtual no define el evento de vista bloqueada, y no es necesario si la transferencia de estado se implementa directamente en el mismo modelo; este evento se produce en el grupo cuando los miembros reciben la petición de inclusión de algún nuevo miembro o han detectado la caída de algún antiguo miembro y comienzan a pactar el contenido de la nueva vista.

Si se instala una nueva vista y el coordinador de una transferencia en curso cae, otro de los miembros con estado debe asumir su papel y completar su trasferencia. Sin embargo, nuestra propuesta, como ocurría con las simplificaciones [δ6,7], asume que el mensaje de estado se envía antes de que el coordinador procese ningún mensaje de la nueva vista, es decir, de la vista en que se inicia la transferencia. Como resultado, el nuevo miembro espera un mensaje con el estado del coordinador tras la instalación de la vista. Si otro miembro asume el papel del antiguo coordinador, aún debe enviar aquel estado, para lo que hay varias posibilidades:

- Los miembros con estado guardan su estado tras cada cambio de vista. Como dos miembros podrían incorporarse al grupo en dos vistas consecutivas, esta solución implica que los miembros deberían ser capaces de almacenar varios estados.
- O bien esos miembros congelan sus estados, simplemente no procesando ninguno de los nuevos mensajes hasta que toda transferencia finalice: todos los miembros del grupo quedan bloqueados.
- O bien cada miembro con estado se comporta como coordinador y envía su estado a los nuevos miembros cuando se instala una vista.
- Una última y favorable solución es que el nuevo miembro descarte todos los mensajes encolados durante la vista previa, tal como si el miembro se incorporara al grupo en la última vista instalada. Esta solución es la misma que ya usamos previamente al modelar el sistema y es la que escogemos en nuestra aproximación.

Finalmente, debe considerarse el caso en que se instala la vista antes de que el coordinador envíe su estado, y ese coordinador deba transferir su estado igualmente

a nuevos miembros incorporados en la última vista. Nuestra aproximación implica que todos los nuevos miembros pasan a considerarse como miembros incorporados en la última vista, por lo que este caso no añade ninguna dificultad.

5.2.2. Transferencias de estados en varios pasos

Estados *grandes* pueden requerir dividir el mensaje de estado en varios mensajes. El nuevo miembro debe entonces recibir cada uno de los mensajes, que pueden ser enviados por diferentes coordinadores. Bajo estas condiciones, el nuevo miembro debe satisfacer:

$$g_j / j \geq m : s_O + \left(Mnp \oplus M_j \oplus ms_A^1 \oplus ms_B^2 \oplus \mathsf{K} \oplus ms_U^p\right) = s_F \quad [\delta 8]$$

Es decir, el estado final se obtiene a partir de los mensajes de estado y un conjunto de mensajes pendientes; pero esos mensajes de estado pueden provenir de diferentes miembros con distintos estados y que hayan ya procesado algunos o todos los mensajes que el nuevo miembro debe procesar. Esta condición general es evidentemente difícil de satisfacer de forma genérica. Un ejemplo de aplicación que la satisface en un grupo que almacena todos los mensajes enviados en el grupo, e inicializa a los nuevos miembros enviándoles esos mensajes; pero, en este caso, cada mensaje de estado se refiere a la misma instancia de estado, esto es: $s_A = s_B = \ldots = s_U$.

Incluso cuando hay un único coordinador, la condición anterior no se simplifica, pues los diferentes mensajes de estado podrían referirse a distintas instancias del estado, ya que el coordinador puede evolucionar al procesar mensajes entrantes. Si todo mensaje de estado se refiriese al mismo estado, la solución sí sería más fácil. Como vimos en la transferencia normal, es difícil esperar a procesar todo mensaje pendiente, y nos enfocaremos en el caso en que el estado se envía sin haber procesado ningún mensaje de la nueva vista. En este caso, es importante notar que no hay ninguna diferencia entre tener un único coordinador o varios si los mensajes son del tipo *loopback*, pues entonces todos los coordinadores presentan el mismo estado cuando las vistas son instaladas. El coordinador comienza en un estado inicial y evolucionará al estado final al procesar los mensajes pendientes (*Mnp* y M_i); el miembro nuevo procesará igualmente sus mensajes pendientes (*Mnp* y M_j) así como los mensajes de estado para alcanzar el mismo estado final. Si hay un único coordinador:

$$\left.\begin{array}{l} \left.\begin{array}{r} s_{Ii} + \left(Mnp \oplus M_i\right) = s_F \\ \underline{coordinador} : g_i / i \leq m : \quad s_O + ms_{Ii} = s_{Ii} \\ ms_{Ii} = ms_{Ii}^1 \oplus \mathsf{K} \oplus ms_{Ii}^p \end{array}\right\} s_O + \left(ms_{Ii}^1 \oplus \mathsf{K} \oplus ms_{Ii}^p\right) + \left(Mnp \oplus M_i\right) = s_F \\ \underline{nuevo \quad miembro} : g_j / j > m : s_O + \left(Mnp \oplus M_j \oplus ms_{Ii}^1 \oplus \mathsf{K} \oplus ms_{Ii}^p\right) = s_F \\ M_j = \sum_k M_{jk} = \sum_{k \neq i} M_{jk} \oplus M_{ji} \approx \sum_{k \neq i} M_{ik} \oplus M_{ji} = M_i \oplus M_{ji} \end{array}\right\} \Rightarrow$$

$$\Rightarrow \quad s_O + \left(ms_{Ii}^1 \oplus \mathsf{K} \oplus ms_{Ii}^p\right) + \left(Mnp \oplus M_i\right) = s_O + \left(Mnp \oplus M_i \oplus M_{ji} \oplus ms_{Ii}^1 \oplus \mathsf{K} \oplus ms_{Ii}^p\right) \quad [\delta 9]$$

Si hay varios coordinadores transfiriendo el estado, M_i debe ser el conjunto vacío para todos los miembros:

$$\left.\begin{array}{l} \left.\begin{array}{rl} & s_I + Mnp = s_F \\ \underline{coordinado\ res}: g_i / i \le m: & s_O + ms_I = s_{Ii} \\ & ms_I = ms_I^1 \oplus \mathsf{K} \oplus ms_I^p \end{array}\right\} s_O + \left(ms_I^1 \oplus \mathsf{K} \oplus ms_I^p\right) + Mnp = s_F \\ \overline{nuevo\quad miembro}: g_j / j > m: s_O + \left(Mnp \oplus ms_I^1 \oplus \mathsf{K} \oplus ms_I^p\right) = s_F \end{array}\right\} \Rightarrow$$

$$\Rightarrow \quad s_O + \left(ms_I^1 \oplus \mathsf{K} \oplus ms_I^p\right) + Mnp = s_O + \left(Mnp \oplus ms_I^1 \oplus \mathsf{K} \oplus ms_I^p\right) \quad [\delta 10]$$

Estas dos ecuaciones son *simples* complicaciones de las fórmulas obtenidas para transferencias en un solo mensaje ([δ6] y [δ7]): el orden en que todo par de mensajes, incluyendo los mensajes de estado, es procesado no puede afectar al estado final.

Existe un caso especial cuando el estado y los mensajes se definen de tal forma que cada mensaje sólo modifica una de las porciones del estado. Si hay *p* porciones, y *l* es una porción especifica, podemos obtener a partir de [δ8] para cada porción:

$$\left.\begin{array}{c} \left[s_O + \left(Mnp \oplus M_j \oplus ms_A^1 \oplus ms_B^2 \oplus \mathsf{K} \oplus ms_U^p\right)\right]^l = s_F^l \\ \forall j \ne l \rightarrow ms_X^j = \phi \end{array}\right\} \quad \Rightarrow \quad s_O^l + \left(Mnp^l \oplus M_j^l \oplus ms_i^l\right) = s_F^l \quad [\delta 11]$$

[δ11] debe verificarse en cada porción *l*, cuando el estado se recibe de un coordinador cualquiera *i*, que podría ser diferente para cada porción. Este coordinador habrá procesado una parte de los mensajes entrantes antes de haber enviado su mensaje de estado:

$$g_i / i \le m: \quad \left.\begin{array}{c} mensajes: Mnp \oplus M_i = M_A + M_B \rightarrow M_B = \overline{(Mnp \oplus M_i)} \\ estado: s_{Ii} + \left(Mnp \oplus M_i\right) = s_F \rightarrow s_{Ii} + M_A = s_i; s_i + M_B = s_F \end{array}\right\} \quad s_i + \left(Mnp \oplus M_i\right) = s_F \quad \Rightarrow$$

$$\Rightarrow porcion: \left.\begin{array}{c} s_i^l + (Mnp^l \oplus M_i^l) = s_F^l \\ s_O^l + ms_i^l = s_i^l \end{array}\right\} \quad s_O^l + ms_i^l + (Mnp^l \oplus M_i^l) = s_F^l \quad [\delta 12]$$

Comparando [δ11] y [δ12], obtenemos una ecuación similar a [δ3], que debe ser satisfecha para cada porción del estado, porciones que pueden ser transferidas posiblemente por coordinadores diferentes:

$$s_O^l + ms_i^l + \overline{(M_{np}{}^l \oplus M_i^l)} = s_O^l + (M_{np}{}^l \oplus M_i^l \oplus M_{ji}^l \oplus ms_i^l) = s_F^l \quad [\delta 13]$$

Si no se bloquean los mensajes mientras se realiza la transferencia, deben ser incluidos en la anterior ecuación:

$$s_O^l + ms_i^l + \overline{(M_{np}{}^l \oplus M_i^l \oplus M^l)} = s_O^l + (M_{np}{}^l \oplus M_i^l \oplus M_{ji}^l \oplus M^l \oplus ms_i^l) = s_F^l \quad [\delta 14]$$

Para obtener una solución genérica para todo tipo de grupos, extendemos la solución propuesta para transferencias en un solo paso. Los mensajes deben quedar ahora bloqueados en el nuevo miembro hasta que el último mensaje de estado se reciba, y es la aplicación la que define cuál es el último mensaje de estado. Si no hay mensajes *loopback*, debe descartarse cada mensaje proveniente del coordinador que

sea *previo* al último mensaje de estado, por lo que se requiere cuanto menos ordenación *fifo*. Esta solución implica que sólo se empleará un miembro como coordinador para una determinada transferencia. Sin embargo, ya que el coordinador no puede procesar ningún mensaje en tanto la transferencia se complete, tiene sentido usar un solo coordinador y no bloquear varios miembros a la vez.

5.2.3. Cambios de vistas en transferencias en varios pasos

Cuando puede descomponerse el estado de un grupo en varias partes y los mensajes actualizan exclusivamente una de esas partes, el problema de la transferencia de estado puede descomponerse en *n* transferencias de estado en un solo paso, donde cada transferencia se corresponde a cada una de las partes del estado. En este caso, los cambios de vistas introducen la misma complejidad que ya hemos observado en transferencias en un solo paso. En las aplicaciones en grupos que satisfacen [δ8], esta condición no se torna más complicada, aunque los miembros con estado deben ahora considerar el caso en que el coordinador o coordinadores caen antes de completar la transferencia.

La solución que presentamos en el punto previo, basada en el envío del estado del coordinador antes de procesar ningún mensaje de la nueva vista, puede emplear igualmente las estrategias que señalamos durante la discusión de transferencias en un solo paso. Refiriéndonos a la solución que directamente escogimos, donde el nuevo miembro debe descartar los mensajes en cada cambio de vista, debe notarse que ahora debe descartarse igualmente los mensajes de estado previos, lo que la convierte en una solución *cara*: el estado se divide en varias partes para acelerar la transferencia, pero debe reiniciarse cada vez que el coordinador se caiga. Incluso suponiendo poco usuales las repetidas caídas de los sucesivos coordinadores, un protocolo bidireccional entre el nuevo miembro y su coordinador aceleraría la transferencia, pues el nuevo miembro podría enviar al nuevo coordinador información sobre el estado de su transferencia.

5.3. Algoritmo de transferencia de estado

Los anteriores puntos han mostrado que sólo los sistemas más sencillos pueden realizar transferencias de estado básicas sin realizar ningún procesamiento adicional, en especial sin bloquear mensajes. El algoritmo que proponemos para resolver la transferencia de estado de forma genérica se resume en los siguientes puntos:

- El mensaje de estado debe enviarse inmediatamente tras el cambio de vista, y el nuevo miembro debe bloquear todo mensaje entrante en tanto no reciba y procese el mensaje de estado.

- Tras procesar el mensaje de estado, los mensajes bloqueados se desencolan y procesan en el orden de entrada. Si el grupo emplea mensajes no-*loopback*, deben descartarse los mensajes que fueron procesados por el coordinador en la vista anterior. En este caso, el grupo debe definir al menos orden *fifo*.
- No se considera al nuevo miembro como perteneciente al grupo mientras no se le transfiera el estado. Cuando se instala una nueva vista, el coordinador concluye cualquier transferencia en marcha y procesa los mensajes encolados. El nuevo miembro descarta, en este caso, esos mensajes y esperará a que se inicie una nueva transferencia en la siguiente vista. El mensaje de estado debería incluir la identidad de vista para que pueda descartarse si es recibido en la siguiente vista.
- Si se envía el mensaje de estado en varias porciones, los mensajes deben quedar bloqueados en el nuevo miembro hasta que se reciba el último mensaje de estado, siendo la aplicación la que decide cuál es este último mensaje de estado. Si no hay mensajes no-*loopback*, debe descartarse todo mensaje del coordinador *previo* al último mensaje de estado, lo que implica la necesidad de orden *fifo*, cuanto menos.
- Si se envía el estado en varios pasos y el coordinador se cae, la solución más sencilla es que el nuevo miembro descarte los mensajes de estado recibidos. El problema de rendimiento asociado a esta solución puede evitarse empleándose protocolos bidireccionales, donde el nuevo estado envía al nuevo coordinador el estado de la transferencia. Esta solución no resulta genérica, cada grupo deberá definir esos protocolos, siempre y cuando puedan permitir la continuación de transferencias fallidas.

5.4. Conclusiones

Este capítulo ha enfocado el problema de la transferencia cuando ésta se produce bajo condiciones que modifican el estado que se transfiere. Se han descrito las condiciones que deben verificar, tanto el grupo y sus miembros, como los mensajes definidos en ese grupo para que la transferencia pueda realizarse sin bloqueo de los miembros o los mensajes, observándose que sólo los grupos más simples verifican esas propiedades.

Se ha descrito de esta manera un algoritmo genérico que puede implementarse en cualquier aplicación para realizar una transferencia correcta en todos los casos. Este algoritmo precisa de un bloqueo de mensajes, y el impacto de este bloqueo sobre el modelo de sincronía virtual se ha detallado igualmente.

Al hacer el análisis de este impacto, hemos demostrado que es preciso que el GMS haga público el evento de bloqueo de vista; aunque este evento está generalmente disponible en los sistemas de comunicaciones fiables de grupo, el modelo de sincronía virtual no lo considera una propiedad del GMS. Y, en efecto, si

el GMS implementa directamente los protocolos de transferencia de estado, no es preciso que tal evento sea público.

El siguiente capítulo estudia los protocolos de bajo nivel que deben utilizarse entre los miembros del grupo durante la transferencia, con el objetivo de comparar las distintas posibilidades de implementación, y comparando igualmente el efecto de implementar esos protocolos en el mismo GMS o como un nivel por encima de aquél.

Capítulo 6 - Protocolos de Transferencia de Bajo Nivel

El capítulo anterior estudió las condiciones que deben verificarse para realizar la transferencia de estado hacia miembros nuevos que se incorporan a un grupo, y desarrolló un algoritmo que validaba esa transferencia de forma genérica. Este algoritmo se ha especificado de forma abstracta, y esta sección define unos protocolos de bajo nivel sobre los que implementar el algoritmo. En particular, estos protocolos son bidireccionales, permiten el intercambio de información entre el coordinador de la transferencia y el nuevo miembro, lo que permite mejorar el rendimiento del algoritmo en el peor caso: transferencias en varios pasos con frecuentes caídas de los sucesivos coordinadores.

6.1. Requisitos de los protocolos

La transferencia de estado tiene varios problemas que resolver: consensuar cuál es el estado a transferir, quién debe por lo tanto recibirlo y quién inicia la transferencia. A continuación, habrá que determinar cómo efectuar la transferencia, involucrando a todo el grupo o no, cómo y quién inicia la transferencia o si ésta se realiza en un solo paso o en varios.

Los protocolos deben cumplir una serie de requisitos:

- *Minimizar el ancho de banda* requerido para efectuar la transferencia.

- Facilitar el envío del estado en varias fases, para facilitar a la aplicación el tratamiento de estados *grandes*.
- Cubrir, tanto la inicialización de miembros nuevos en el grupo, como la unión de particiones del mismo grupo.
- *Elección del coordinador*: la aplicación puede decidir cómo determinar el coordinador para la inicialización de un miembro nuevo; usar cualquier miembro con estado, favorecer el empleo de determinados miembros mediante el empleo de *pesos* o realizar balanza de carga entre los miembros con estado. También permite delegarse esta decisión a la aplicación, que podrá de esta manera seleccionar el coordinador apoyándose en información adicional, como pueda ser la proximidad física de miembros o la balanza de carga total.
- *Limitar la disponibilidad* del grupo durante la transferencia: puede ser necesario inhibir el envío de mensajes en el grupo, lo que supone que el grupo pierde temporalmente su disponibilidad, al no poder responder a ninguna petición de servicio. En el polo contrario, puede considerarse que la transferencia de estado es muy corta o que los mensajes pueden recibirse sin complicar la transferencia de estado y, en ese caso, no se inhibe ningún mensaje durante estas transferencias, evitando que el grupo pierda temporalmente su disponibilidad como servidor. Un grado intermedio se obtiene cuando la transferencia de estado se considera *parcialmente segura* y se permite el envío de los mensajes definidos como seguros; la aplicación debe entonces diferenciar entre mensajes seguros y no seguros. En este caso, se puede también diferenciar entre la transferencia de estado y la unión de particiones, definiéndose así distintos grados de seguridad para ambos escenarios.
- *Grupos de varios niveles*. Si un grupo distingue entre funcionalidad cliente y servidor, la transferencia sólo involucrará a los servidores. De la misma forma, en los grupos que presentan replicación pasiva, en uno o varios niveles, también debe realizarse una transferencia ante determinados eventos dependientes de la aplicación; un ejemplo es la activación de un miembro en *backup* como consecuencia de la caída de un miembro activo, que deberá primero alcanzar un estado consistente al resto de miembros activos mediante una transferencia de estado *voluntaria*. Sin embargo, un miembro no activo se considera no perteneciente al grupo, pues no interviene en los mensajes, su actualización es dependiente de la aplicación y no lo consideramos como un apartado especial en la transferencia de estado.
- *Propiedades* de los miembros. El problema de la transferencia de estado se centra en la replicación de un estado sobre un conjunto de réplicas de un determinado objeto. Si la distribución de réplicas no es uniforme y puede considerarse que cada miembro posee un conjunto de propiedades que deben ser conocidas por todos los miembros, estas propiedades no formarán parte del estado ni del problema de la transferencia de estado. La transferencia de estado es un problema lógicamente mayor que la transferencia de propiedades. Ejem-

plos de propiedades son el peso asociado a cada miembro o su localización física, que puede servir a la aplicación para seleccionar como coordinador al miembro más cercano.

- *Inicialización del grupo.* El primer miembro que crea un grupo, crea un estado inicial y empezará a dar un servicio, transfiriendo su estado si nuevos miembros se incluyen en el grupo. Sin embargo, por problemas de partición de la red, un miembro puede considerarse creador de un grupo al no poder comunicarse con el resto de los miembros, y desarrollar un estado incompatible. Para soslayar esta situación, la aplicación debe comunicar al servicio de transferencia de estado cuándo un miembro o conjunto de miembros sin estado pasan a considerarse como miembros con estado, con completa funcionalidad.

La transferencia de estado está en un nivel intermedio entre la aplicación y el GMS. De hecho, gran parte de la funcionalidad requerida en la transferencia de estado puede trasladarse al GMS para lograr un mejor rendimiento. Los protocolos de transferencia han sido desarrollados para los dos casos posibles, con o sin el soporte del GMS, lo que permitirá estudiar las ventajas o desventajas de extender el GMS con una funcionalidad ajena a su servicio: en su definición, las vistas que se envían al grupo incluyen únicamente una lista ordenada de los miembros del grupo y ninguna información adicional sobre el estado.

En los grupos de partición primaria existen dos formas posibles de realizar la transferencia de estado: transferencia *push*, donde los miembros con estado coordinan la transferencia hacia el miembro o miembros sin estado, y transferencia *pull*, donde los miembros sin estado solicitan la transferencia de estado.

6.2. Transferencia *push*

El algoritmo de transferencia *push* precisa que cada miembro conozca la lista de miembros con estado y los miembros que éstos están coordinando. La transferencia se realiza de la siguiente manera, como se visualiza en la figura 6.1:

- *Selección de un coordinador* para cada miembro sin estado, siguiendo las pautas definidas por la aplicación: usando pesos, balanza de carga, etc... Esta selección es determinista, cada miembro del grupo decide el coordinador para cada nuevo miembro sin realizar comunicaciones internas, pero todos seleccionan a los mismos coordinadores.
- El coordinador envía al nuevo miembro la lista de miembros con estado y los miembros que coordinan, en un mensaje al que denominamos mensaje *p*, mensaje de **p**rotocolo. Si se han definido propiedades, este mensaje incluye las de los miembros con estado. En cuanto lo recibe, el nuevo miembro pasa a conocer el estado de transferencias en el grupo: qué miembros tienen estado y a qué miembros están coordinando.

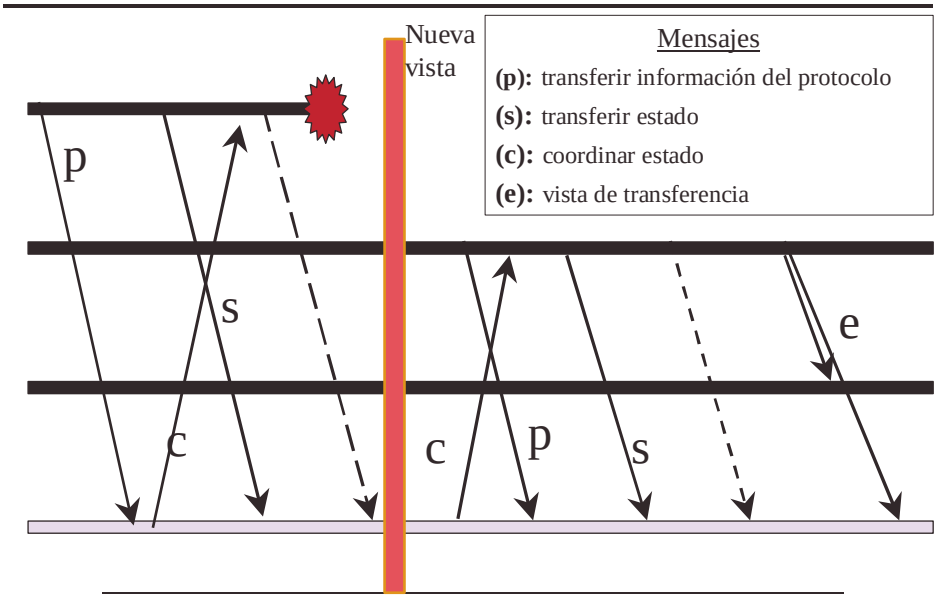

Figura 6.1. Protocolo de transferencia *push*.

- Si el nuevo miembro ha definido alguna propiedad, las envía al coordinador en un mensaje de **c**oordinación llamado mensaje *c*. Se denomina mensaje de coordinación porque permite al nuevo miembro enviar información a su coordinador; por ejemplo, sobre antiguas transferencias frustradas, como explicamos más adelante.
- El coordinador transfiere el estado en uno o varios mensajes de estado (***s**tate messages*), denominados mensajes *s*.
- Al finalizar la transferencia, el coordinador envía un mensaje multipunto final (***e**nd message*), denominado mensaje *e*, con el que los demás miembros pasan a considerar al nuevo miembro como miembro con estado. Este mensaje contiene las propiedades del nuevo miembro.

Si un coordinador se cae, los miembros sin estado a los que coordina se distribuyen entre los demás miembros con estado; por esta razón, cada miembro debe guardar la lista de miembros de cada coordinador, y la elección del coordinador debe ser determinista, pues no existe comunicación entre los miembros con estado sobre esa elección. En particular, si se cae durante una transferencia, el miembro al que coordinaba puede seleccionar directamente un nuevo coordinador, si ya recibió del anterior el mensaje *p*. En este caso, puede enviarle automáticamente un mensaje *c* de coordinación de la transferencia, acelerando el protocolo. En la figura se muestra que este mensaje *c* sería concurrente con el mensaje *p* recibido del coordinador, pues éste último no puede saber si el nuevo miembro recibió el mensaje *p* en su anterior transferencia, por lo que debe consecuentemente enviarlo.

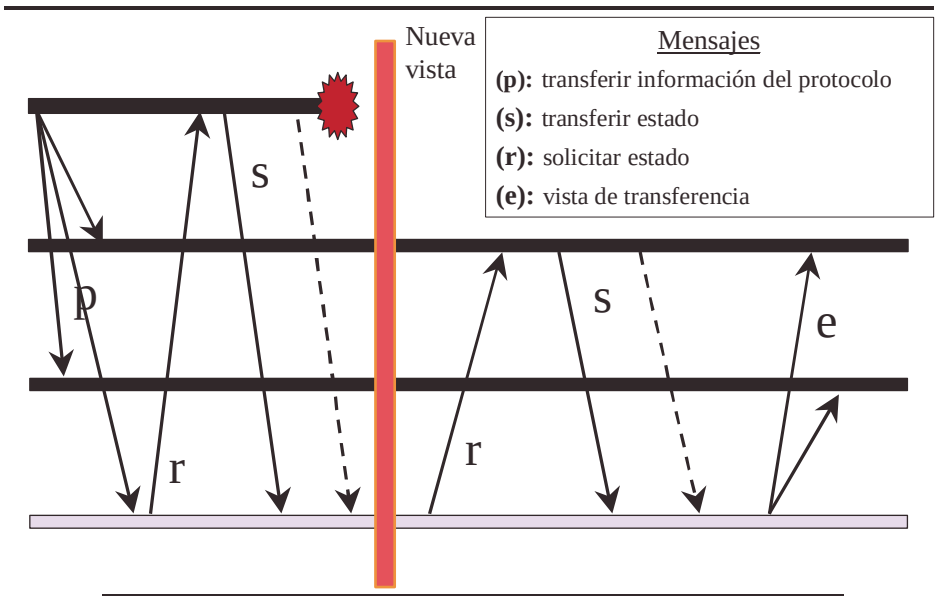

Figura 6.2. Protocolo de transferencia *pull*.

El contenido del mensaje *c* de coordinación, no se define a este nivel del protocolo, de la misma forma que no se definen los mensajes *s* de estado; ambas definiciones deben realizarse a un nivel superior de este protocolo, tal como se detalla en el siguiente capítulo, al definir la interfaz de la aplicación.

Debe notarse que las propiedades no son estáticas y, cuando un miembro las modifica, debe enviar un mensaje a los demás miembros. Durante la transferencia, el nuevo miembro no puede modificar sus propiedades, debe esperar a que el coordinador envíe el mensaje *e*. Sin embargo, si se produce un cambio de vista, el protocolo permite al nuevo miembro enviar sus propiedades actualizadas al coordinador en el mensaje *c*.

Este protocolo no incluye ninguna restricción de concurrencia, permitiendo la transferencia simultánea de estado a varios miembros. En particular, si dos o más miembros son asignados al mismo coordinador, este último puede inhibir la concurrencia al ser incapaz de manejar varias transferencias al mismo tiempo; esta limitación deberá incluirse en algún nivel por encima de este protocolo.

6.3. Transferencia *pull*

Con transferencia *pull*, visualizada en la figura 6.2, cada miembro sólo debe conocer la lista de los miembros con estado, lo que simplifica el algoritmo empleado:

- Un miembro con estado es seleccionado determinísticamente para enviar en un mensaje multipunto *p* la lista de los miembros con estado, así como sus propiedades. Este miembro no es el coordinador, pero puede escogerse igualmente a partir de los pesos asignados a cada miembro.
- Cada miembro sin estado selecciona un coordinador. No hay comunicaciones entre estos miembros, por lo que no se soporta balanza de carga. El coordinador recibe un mensaje *r* de solicitud de estado (***r****equest state message*).
- El coordinador envía entonces el estado en *mensajes s*.
- Al concluir la transferencia, el miembro sin estado envía un mensaje multipunto *e*, indicando así su promoción a miembro con estado. En este mensaje incluye sus propiedades.

Si el coordinador se cae, el miembro sin estado selecciona un nuevo coordinador mediante un *mensaje r*, donde incluye el estado actual de la transferencia. En este caso, no es necesario que ninguno de los miembros con estado envíe de nuevo el mensaje *p*, puesto que al ser multipunto, pueden saber perfectamente si el nuevo miembro lo recibió o no.

Varios miembros sin estado pueden seleccionar al mismo coordinador para realizar la transferencia de estado. Si esta concurrencia no es deseable, deberá inhibirse en algún nivel por encima de este protocolo. También se definirá de esta manera el contenido del mensaje *r*, tal como se vio en la transferencia *push*.

6.4. Transferencia *push-one step*

El protocolo *push* puede simplificarse cuando el estado se transfiere en un solo mensaje, mediante el protocolo *push-one step*. En este caso, se selecciona un único coordinador que transfiere el estado en un mensaje multipunto, que incluye asimismo las propiedades de los miembros con estado. Cada miembro sin estado debe realizar a su vez un envío de sus propiedades con un mensaje multipunto.

A pesar del nombre, el protocolo *push* aún puede usarse para transferencias en un único paso, y como se ve en la siguiente comparativa, ésta es la solución más favorable en la mayoría de los casos.

6.5. Comparativas de protocolos

6.5.1. Protocolos *push* y *push-one step*

Las figuras 6.3 y 6.4 comparan los protocolos *push* y *push-one step* para pequeños estados, asumiendo que el coordinador elegido no se cae durante la

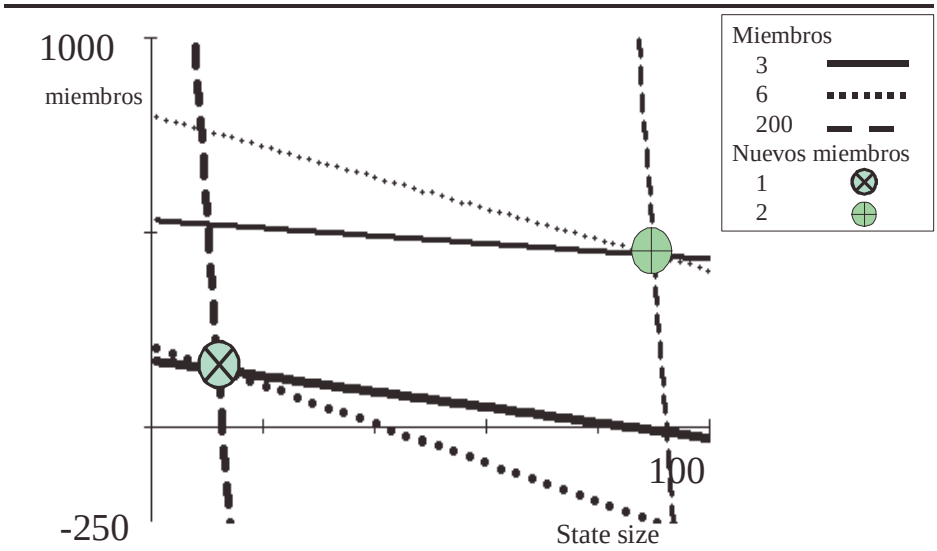

Figura 6.3. Comparativa entre *push* y *push-one step*, sin propiedades. Valores positivos implican peor rendimiento del protocolo *push*.

transferencia. Los gráficos comparan la cantidad de información enviada en ambos protocolos para distintos tamaños de estado, indicando los valores positivos un peor rendimiento para el protocolo *push*. La primera figura se refiere al caso donde no se usan propiedades, y la segunda figura incluye el soporte de propiedades.

Los cálculos se han realizado con unos tamaños de cabecera de los mensajes de 32 bytes y tamaños de identidades de los miembros del grupo de 6 bytes. Para las propiedades, se han empleado pequeños tamaños, con sólo 10 bytes por miembro. Finalmente, los mensajes *c* de coordinación en el protocolo *push* se han considerado con tamaños de 4 bytes, suponiendo simplemente dos contadores enteros, uno que indique la fase de la transferencia y otro para el número total de fases requeridas. Se observa que el protocolo en un sólo paso tiene mejor comportamiento para estados pequeños, y se degrada rápidamente según aumenta el número de miembros del grupo; este comportamiento se acentúa si se aumentan los tamaños de las cabeceras de los mensajes o de las propiedades:

- Como el protocolo *push-one step* realiza la transferencia en un solo mensaje multipunto, la inclusión simultánea de varios miembros en el grupo repercute en un mejor comportamiento de este protocolo.
- El empleo de propiedades supone en el protocolo *push-one step* el envío adicional de un mensaje multipunto por cada nuevo miembro, lo que se traduce en un peor rendimiento respecto al protocolo normal, tanto peor cuanto mayores sean esas propiedades.

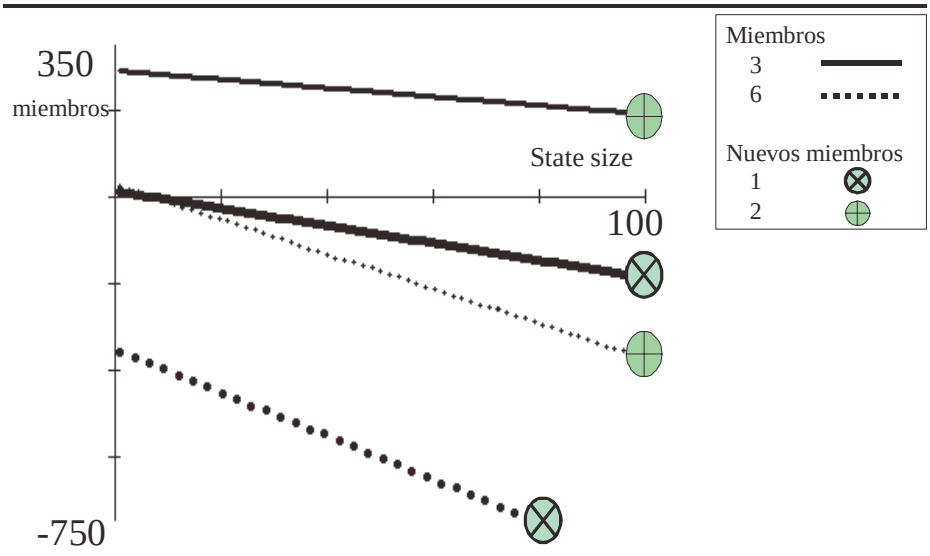

Figura 6.4. Comparativa entre *push* y *push-one step*,con propiedades. Valores positivos implican peor rendimiento del protocolo *push*.

- Los gráficos se han realizado suponiendo que no hay soporte multipunto; con éste, el protocolo en un solo paso resultará siempre preferible al protocolo *push*, al hacer uso extensivo de los mensajes multipunto.

Por lo tanto, sin soporte multipunto, el empleo de transferencia con protocolo *push-one step* se limita a estados muy pequeños, tanto más según aumenta el tamaño del grupo, y dando muy mal rendimiento si es necesario el soporte de propiedades. Debe notarse que, lógicamente, el protocolo *push* también permite transferencias en un único paso, donde se envía un único mensaje *s*.

6.5.2. Protocolos *push* y *pull*

La figura 6.5 muestra la diferencia en la cantidad de información enviada en transferencias *push* y *pull*, según el número de miembros del grupo. Valores positivos indican mejor rendimiento del protocolo *pull;* los cálculos se han realizado con los mismos parámetros (tamaños de mensajes, etc.) que se han detallado en la comparativa entre protocolos *push* y *push-one step*:

- Cuanto mayor es el tamaño del grupo, mejor es comparativamente el rendimiento del protocolo *push*. Esto se debe al mayor número de mensajes necesarios para realizar un envío multipunto; por esta misma razón, un aumento en el tamaño de los mensajes como, por ejemplo, emplear cabeceras mayores, implica un peor rendimiento del protocolo *pull*.

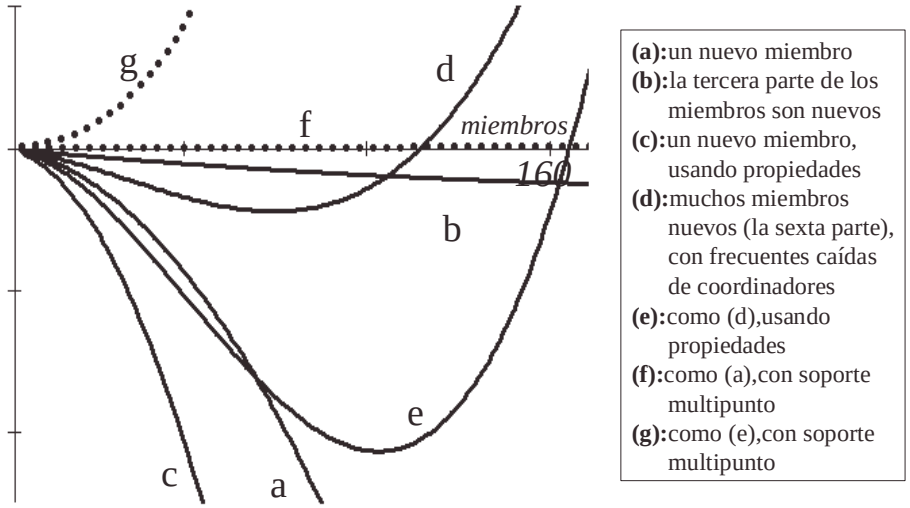

Figura 6.5. Comparativa entre protocolos *push* y *pull*.

- La inclusión simultánea de varios miembros en el grupo repercute en un mejor comportamiento del protocolo *pull*.
- El uso de propiedades implica un mayor tamaño de los mensajes multipunto necesarios bajo protocolo *pull*, lo que se traduce en un peor rendimiento de este protocolo según el tamaño de las propiedades aumenta.
- Con soporte multipunto, es favorable el empleo del protocolo *pull*.

Resumiendo: sin soporte multipunto, el protocolo *push* tiene un mejor comportamiento que el protocolo *pull*, beneficiándose este último de situaciones con muy frecuentes caídas de miembros y de inclusiones simultáneas de nuevos miembros en el grupo. Con soporte multipunto, el protocolo *pull* tiene mejor rendimiento.

6.6. GMS con soporte de transferencia de estado

Aunque el GMS no implemente los protocolos de transferencia de estado, puede incluir un soporte básico si añade en cada vista información sobre los miembros con estado. De esta forma, los nuevos miembros conocen desde el principio los miembros con estado, si los hay. Este soporte podría además extenderse para incluir asimismo las propiedades de los miembros.

El impacto en el GMS es muy pequeño, y es únicamente preciso que los miembros que adquieren el estado del grupo lo comuniquen al GMS para que éste

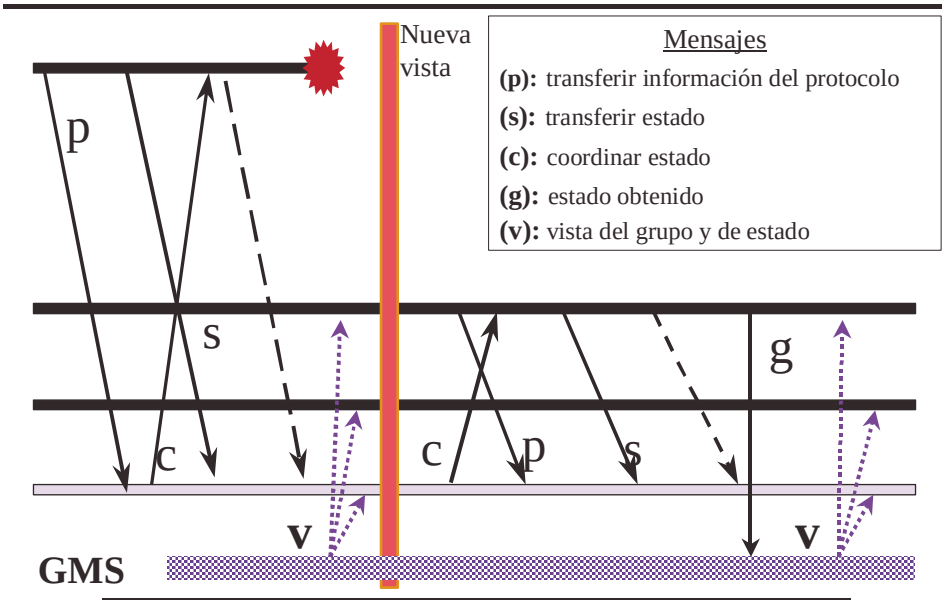

Figura 6.6. Protocolo *push* con soporte de transferencia en el GMS

pueda instalar una nueva vista. Obviamente, este soporte mínimo está orientado al protocolo *pull*, donde hace falta que los nuevos miembros reciban primero información sobre el estado del grupo para poder dirigirse entonces a un miembro válido y solicitarle el estado.

La figura 6.6 muestra los cambios en el protocolo *push*. El coordinador, una vez transferido el estado, comunica al GMS la actualización del nuevo miembro, en lugar de enviar un mensaje multipunto a todo el grupo. El GMS instalará eventualmente, posiblemente tras esperar a que todos los nuevos miembros hayan recibido el estado, una nueva vista que incluye la nueva información de estado.

La figura 6.7 muestra los cambios en el protocolo *pull* que, al no precisar del primer mensaje multipunto *p*, resulta claramente superior al protocolo inicial. Una comparación entre el protocolo *push* sin soporte del GMS y el protocolo *pull* con soporte, clarificaría cuándo la inclusión de tal funcionalidad en el GMS resulta útil. No obstante, tal comparativa dependerá de la implementación particular de las vistas en el GMS. Así, si cada vista implica un mensaje multipunto entre los miembros del GMS incluyendo sólo los cambios y no la vista completa, la información a enviar bajo protocolo *pull* es menor y el soporte de la transferencia de estado en el GMS queda justificado.

Tanto *Maestro* [Vaysburd98] como *JavaGroups* [Ban98] (ambos sobre *Ensemble*) utilizan transferencia *pull*, pero en ambos casos se elimina el mensaje inicial que envía un miembro con estado para informar de cuáles son los miembros correctos que pueden realizar la transferencia; el primero realiza la suposición errónea de que la vista incluye los miembros más antiguos con los rangos más bajos,

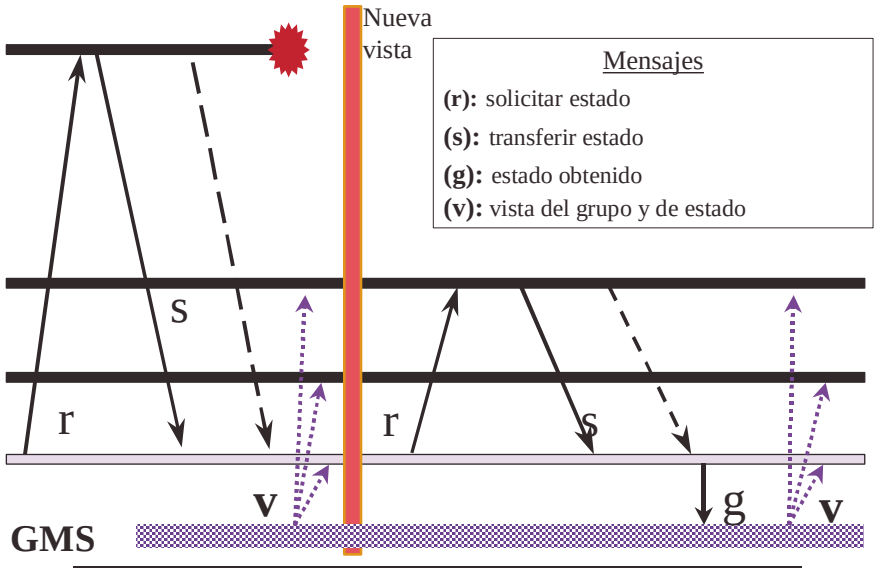

Figura 6.7. Protocolo *pull* con soporte de transferencia en el GMS

por lo que el miembro sin estado siempre se dirige al miembro con menor rango. En *JavaGroups*, el miembro sin estado se dirige a cualquiera de los demás miembros, sin saber si tiene estado o no; si no le responde en un tiempo determinado, la solicita a otro miembro.

6.7. Grupos particionables

Si el GMS presta soporte de estado, en el caso de grupos de particiones primarias basta con que contenga constancia de los miembros con estado. Si el grupo es particionable, debe agrupar los miembros según el estado que posean, pues distintos miembros pueden presentar estados inconsistentes, y el GMS admitirá dos niveles de soporte:

- Asociar a cada miembro un identificador único de estado, que es enviado en las vistas.
- Implementar la interfaz definida en [Babaoglu95] en *sincronía virtual extendida*, una forma de agrupar los miembros según su estado y permitir a la aplicación la integración de *subvistas* y *conjuntos de subvistas* hasta alcanzar un estado único (una vista única).

Sensei no incluye el estudio de grupos particionables, por lo que limitamos los protocolos de transferencia a los grupos primarios.

6.8. Conclusiones

Este capítulo se ha centrado en los protocolos de bajo nivel necesarios para realizar la transferencia de estado en grupos primarios. Hay dos enfoques posibles: un miembro con estado se dirige al nuevo miembro y le transfiere el estado, o el nuevo miembro obtiene información sobre el grupo y se dirige entonces a uno de los miembros con estado para solicitarle la transferencia.

Para cada posibilidad, hemos desarrollado un protocolo de bajo nivel que permite implementar protocolos de aplicación flexibles, como demostramos en el siguiente capítulo. Finalmente, hemos verificado las condiciones que favorecen el empleo de uno u otro protocolo. Estas condiciones han incluido el uso de propiedades o no, y el posible soporte por parte del GMS para la transferencia de estado, estudiando el impacto del número de miembros en el grupo y del número de nuevos miembros en cada vista. Aunque para la comparativa hemos partido de valores precisos para los tamaños de los mensajes, hemos observado también el impacto de variaciones en esos tamaños.

Como regla general, el protocolo *push* precisa una menor transferencia de información, salvo que se disponga de soporte multipunto. Como contrapartida, es más complicado de implementar, pues cada miembro debe mantener el estado de transferencia de todos los demás miembros.

Capítulo 7 - Interfaz de Aplicación de Transferencia de Estado

Los dos capítulos anteriores han especificado un algoritmo de transferencia de estado y unos protocolos de bajo nivel sobre los que implementar esa transferencia de estado. Este capítulo se centra en la interfaz de alto nivel que la aplicación empleará, interfaz que permite la implementación del algoritmo de transferencia de estado planteado.

Esta interfaz la especificaremos como un servicio CORBA, en OMG/IDL, así como en Java, y mostraremos su implementación sobre los protocolos de bajo nivel descritos en el capítulo anterior.

El *Object Management Group* ha aprobado recientemente el servicio estándar de tolerancia a fallos en CORBA mediante replicación de entidades [OMG99]. Esta especificación se refiere a la replicación de objetos en un entorno CORBA, e intenta soportar un amplio rango de requisitos. Incluye soporte de replicaciones activa y pasiva. Permite el control de la creación y consistencia de las réplicas por la aplicación o bien por la infraestructura de tolerancia a fallos. Realiza el mantenimiento y recuperación del estado, ya sea automáticamente o personalizado según la aplicación. Y este soporte se proporciona intentando minimizar en todo caso el impacto sobre el ORB.

Sin embargo, la especificación deja al menos un punto abierto: la transferencia de estado en grupos de réplicas activas con un estilo de consistencia controlado por

la aplicación, transferencia que debe ser suficientemente flexible para soportar los requisitos de muy variadas aplicaciones.

7.1. Tolerancia a fallos en CORBA

El principal componente en la especificación de tolerancia a fallos de CORBA para el manejo de grupos de objetos es el *ReplicationManager*, o supervisor de replicación, que debería replicarse a su vez para así ser tolerante a fallos. La interfaz del *ReplicationManager* soporta por herencia las tres interfaces siguientes: *PropertyManager* (supervisor de propiedades), *GenericFactory* (factoría de grupos) y *ObjectGroupManager* (supervisor de grupos de objetos).

La interfaz *PropertyManager* define propiedades de tolerancia a fallos en grupos de objetos, que determinan las responsabilidades de la infraestructura de tolerancia a fallos. Las propiedades más relevantes son *ReplicationStyle* (estilo de replicación), *MembershipStyle* (estilo de pertenencia a grupo) y *ConsistencyStyle* (estilo de consistencia).

La propiedad *ReplicationStyle* determina cómo se mantiene el estado en las réplicas; puede ser *stateless* (sin estado), *cold passive* (estado pasivo frío), *warm passive* (estado pasivo caliente), *active* (estado activo) y *active with voting* (estado activo con votación), aunque el último no está soportado en la actual especificación. Las replicaciones pasivas se basan en que un único miembro, llamado primario, procesa todas las solicitudes de servicio dirigidas al grupo, grupo que contiene otros miembros denominados pasivos o *backup*. Si la replicación es caliente, los miembros pasivos reciben periódicamente el estado de la réplica primaria; si es replicación fría, la actualización se realiza únicamente cuando es estrictamente necesario porque la réplica primaria se haya caído. Bajo replicación activa, todas las réplicas procesan todas las solicitudes de servicio al grupo, y se suprimen tanto las solicitudes repetidas como las respuestas duplicadas. Este estilo requiere que el grupo se diseñe bajo el modelo de sincronía virtual [Birman87].

La propiedad *MembershipStyle* determina como se maneja la pertenencia al grupo, si es directamente controlada por la aplicación (*application-controlled*) o por la infraestructura de tolerancia a fallos (*infrastructure-controlled*). En el último caso, el *ReplicationManager* invocará a las factorías requeridas para crear los miembros necesarios para satisfacer las propiedades del grupo, como el número inicial o mínimo de réplicas. Si la aplicación controla la pertenencia al grupo, el *ReplicationManager* dispone de operaciones para insertar o extraer miembros de los grupos, definir el miembro primario y las localizaciones de cada miembro del grupo.

La consistencia del grupo, determinada por la propiedad *ConsistencyStyle*, puede ser también controlada por la aplicación (*application-controlled*) o por la infraestructura (*infrastructure-controlled*). En el primer caso, la aplicación debe asegurar su propia consistencia en cada momento: tras invocaciones al grupo o

cuando éste cambia porque una nueva réplica es añadida al grupo, la cual debe por lo tanto recibir el estado actual del grupo. Alternativamente, si es controlada por la infraestructura de tolerancia a fallos, los objetos que soportan la aplicación deben implementar una interfaz básica, *Checkpointable*, que define operaciones simples para obtener o establecer el estado:

```
interface Checkpointable {
    State get_state ();
    void set_state (in State s);
};
interface Updateable : Checkpointable {
    State get_update ();
    void set_update (in State s);
};
```

Opcionalmente, los objetos pueden implementar una interfaz especializada sobre la anterior, *Updateable*, que define adicionalmente operaciones para establecer y obtener el estado incrementalmente. La infraestructura maneja los mecanismos requeridos para mantener una consistencia fuerte entre las réplicas de grupo y para instalar un estado consistente en cada nueva réplica. Esta consistencia significa, bajo replicación pasiva, que tras la transferencia de estado a un miembro pasivo, éste es consistente con la réplica primaria o, si la replicación pasiva es *cold*, con la anterior réplica primera. Con replicación activa, consistencia fuerte requiere que tras cada invocación de servicio al grupo, toda réplica tenga el mismo estado. Si la replicación mantiene algún estado, es decir, no es *stateless*, el comportamiento de cada miembro debe ser determinista y cada miembro debe comenzar en el mismo estado.

Esta especificación implica que, cuando la infraestructura de tolerancia a fallos debe mantener la consistencia, las réplicas deben devolver periódicamente su estado. Esta infraestructura almacena los mensajes procesados por cada réplica y obtiene su estado en intervalos periódicos acordes a la propiedad *CheckpointInterval*, definida en unidades de décimas de microsegundos. Al incorporarse una nueva réplica al grupo, el estado de éste puede obtenerse a partir del último estado almacenado y de los mensajes procesados tras haber almacenado ese estado, de ahí la necesidad del comportamiento determinista de la aplicación.

La transferencia de estado estudiada en *Sensei* involucra precisamente a los grupos de réplicas activas donde el estado se obtiene a partir de las otras réplicas de forma directa. Bajo la notación CORBA, el dominio de *Sensei* son los grupos activos con un valor de la propiedad *ConsistencyStyle* tal que la consistencia es manejada por la propia aplicación. Finalmente, la especificación CORBA no soporta el particionado de grupos, por lo que no es necesario incluir la consistencia entre grupos particionados.

7.2. Diseño de la interfaz

La interfaz que especificamos busca la flexibilidad necesaria para adaptarse a complejos requisitos de transferencia. Sin embargo, debe mantenerse simple en los casos que precisen transferencias sencillas. Por esta razón, un miembro que realice transferencias en un único paso debe implementar simplemente la interfaz *Checkpointable*, tal como lo define la especificación CORBA. Bajo JavaRMI, esta interfaz es:

```
public interface Checkpointable extends java.rmi.Remote{
    State get_state () throws java.rmi.RemoteException;
    void set_state (State s) throws java.rmi.RemoteException;
}
```

Donde el estado viene definido como:

```
public interface State extends java.io.Serializable {}
```

Los siguientes apartados implican una mayor flexibilidad en la transferencia:

- Transferencias en varios pasos.
- Comportamiento ante cambios de vistas.
- Propiedades de miembros.
- Elección del coordinador.
- Transferencias concurrentes.

Cada apartado es explicado a continuación, antes de mostrar la interfaz especificada. Al describir esta interfaz, los objetos o clases involucrados han sido nombrados en inglés, para mantener la coherencia con otras especificaciones OMG/IDL.

7.2.1. Transferencias en varios pasos

Cuando se transfiere el estado en varias partes, se precisa alguna sincronización entre el coordinador y el nuevo miembro para identificar la parte que se transfiere. Esta sincronización puede definirse simplemente por el orden de las partes que se transfiere pero, especialmente para soportar un protocolo bidireccional entre ambas partes que permita reanudar transferencias interrumpidas, se requiere algún mecanismo más elaborado de sincronización. Aunque la misma porción de estado puede incluir esa sincronización como parte del estado, definir un objeto separado para ese propósito permitirá un diseño mejor y más claro. Este objeto lo llamamos *coordinación de fase* o *PhaseCoordination* en la interfaz, y es la propia aplicación quien lo define.

Desde el punto de vista del sistema de transferencia de estado, esta fase simplemente debe indicar cuándo una transferencia ha concluido. Un ejemplo es una fase que contiene el número de porciones en que se ha dividido el estado y la identidad de la siguiente porción a transferir. El final de la transferencia se determina en este caso cuando la identidad y el número de porciones del estado coinciden. Esta fase se define en OMG/IDL como:

```
valuetype PhaseCoordination {
    public boolean is_transfer_finished ();
}
```

Una fase que siempre devuelve *true* define una transferencia en un único paso, obviamente.

La definición de este objeto se realiza en CORBA mediante la funcionalidad de *objetos por valor*. Si en su lugar se hubiera empleado un tipo *interface*, chequear la finalización de la transferencia requeriría una llamada remota. Y si en su lugar se empleara un tipo *struct*, no habría posibilidad de extenderlo para incluir comportamiento específico para la aplicación, pues no admiten herencia en CORBA. De esta forma, se puede definir la fase del anterior ejemplo de la siguiente manera:

```
valuetype ExamplePhase : PhaseCoordination {
    public long chunks;
    public long nextChunk;
};
```

Cuando se accede al coordinador para obtener su estado, debe devolver un objeto *PhaseCoordination* asociado a ese estado, que es transferido al nuevo miembro. Casi todas las operaciones en la interfaz de transferencia incluyen fases de coordinación.

Bajo JavaRMI, la definición de la fase es:

```
public interface PhaseCoordination extends java.io.Serializable{
    public boolean is_transfer_finished ();
}
```

Al ser serializable, el objeto que implemente la interfaz es visto localmente en la máquina virtual que lo recibe y el acceso a sus métodos no es remoto.

7.2.2. Cambios de vistas

Si una transferencia se interrumpe porque el coordinador haya caído, es posible que el nuevo miembro envíe con una fase la información sobre esa

transferencia interrumpida a su nuevo coordinador, de tal manera que no tenga que reiniciarse desde el principio.

Sin embargo, como ya hemos demostrado al estudiar las condiciones en la transferencia, la sincronización entre el nuevo miembro y el coordinador, cuando una transferencia anterior es interrumpida, no es en absoluto un proceso obvio. Así, la solución genérica que propusimos fue la de cancelar toda transferencia que no hubiese finalizado cuando una nueva vista se instalase. En principio, hay dos soluciones alternativas, aunque debe notarse que no son completamente compatibles con el modelo de sincronía virtual, al romper uno de los principios: dos miembros cualesquiera en dos vistas consecutivas deben procesar los mismos mensajes.

La primera solución implica que, cuando se realiza la transferencia, tanto el coordinador como el miembro coordinado se encuentran lógicamente fuera del grupo: ninguno procesa ningún mensaje hasta que se complete la transferencia. Si el coordinador se cae, el miembro coordinado debe ser capaz de enviar a su nuevo coordinador información sobre el estado recibido y los mensajes procesados, de tal manera que éste pueda conocer la porción de estado que le falta. Los mensajes son por lo tanto encolados, no siendo procesados hasta que la transferencia finalice naturalmente o porque uno de los participantes en la transferencia se caiga. Si es el coordinador el que se cae, los mensajes encolados en el nuevo miembro son eliminados de la cola.

Esta solución es la que permite una transferencia de estado más sencilla para la aplicación, que se despreocupa de los cambios de vista. El objetivo que se persigue con el modelo de sincronía virtual es obtener estados consistentes mediante el procesado de los mismos mensajes en un orden adecuado. Si el procesado de un mensaje depende de información de la vista como, por ejemplo, del número de miembros en el grupo, no podrá emplearse esta solución.

La otra solución implica que sólo el miembro coordinado se encuentra fuera del grupo durante la transferencia. Si llega una nueva vista, los mensajes son enviados al coordinador para que los procese, mientras que el miembro coordinado puede, bien descartar, bien recibir, todos los mensajes de esa vista. La motivación de esta solución es permitir al coordinador evaluar los cambios en el estado, cambios que puede enviar en una vista posterior.

Esta segunda opción es similar a la solución que propusimos como genérica: ambos miembros reciben todos los mensajes, pero si hay un cambio de vista la transferencia se reinicia automáticamente. En este caso, se deja a la aplicación la opción de que reinicie esa transferencia o pueda evaluar todos los cambios que, en algunos casos, puede suponer una optimización al protocolo, a costa de complicar la lógica de la aplicación.

Para permitir las distintas opciones, un parámetro permite definir cómo se realiza la transferencia. Esta propiedad en el sistema de transferencia de estado la denominamos *BehaviourOnViewChanges* y puede tomar los siguientes valores:

- *MembersOnTransferExcludedFromGroup*: los miembros en la transferencia se consideran excluidos del grupo. Los mensajes quedan encolados y sólo se envían a la aplicación cuando la transferencia ha concluido.
- *StatelessMembersDoNotBelongToGroup*: los miembros sin estado no pertenecen al grupo. Estos miembros no reciben mensajes en tanto no reciban el estado; ante un cambio de vista, la transferencia se considera que debe reiniciarse, aunque se mantiene el mismo coordinador para permitir optimizaciones. El servidor no procesa mensajes en tanto realiza la transferencia pero, ante un cambio de vista, interrumpe ésta y recibe todos los mensajes no procesados en esa vista, que debe procesar antes de asumir la nueva vista.
- *StatelessMembersBelongToGroup*: todos los miembros se consideran incluidos en el grupo. Ésta es la única opción que permite una solución que se adhiere al modelo de sincronía virtual. Se comporta como en el anterior caso, pero el nuevo miembro recibe los mensajes antes de procesar la nueva vista y es responsabilidad de la aplicación el realizar toda la sincronización necesaria, o bien, descartar toda resincronización y comenzar de nuevo la transferencia. Los mensajes sólo se envían al final de la vista, no durante ésta, y la aplicación debe decidir si los puede procesar o no.

7.2.3. Propiedades de miembros

El estado del grupo puede entenderse como el estado de un determinado miembro que fuera el único miembro de ese grupo. Sin embargo, las propiedades están asociadas a cada miembro en particular y no tienen visibilidad fuera del grupo, es decir, ni son parte del estado, ni deberían afectar al resultado de ninguna operación sobre el grupo. Las operaciones relativas al mismo grupo sí pueden verse afectadas y un ejemplo es la elección del coordinador. Ejemplos básicos de estas propiedades son las localizaciones de cada miembro o pesos específicos para realizar determinadas tareas.

Otra diferencia entre el estado y las propiedades de los miembros reside en su transferencia. Cuando un nuevo miembro se incluye en el grupo, el grupo transfiere su estado a este miembro. Sin embargo, las propiedades deben transmitirse en ambas direcciones: se considera que un nuevo miembro no tiene estado pero sí propiedades.

La especificación de tolerancia a fallos en CORBA define una estructura de propiedades llamada *Property*, que podemos emplear para este propósito:

```
typedef CosNaming::Name Name;
typedef any Value;
```

```
struct Property {
        Name nam;
        Value val;
};
typedef sequence <Property> Properties;
```

La definición de propiedades bajo JavaRMI es:

```
class Property implements java.io.Serializable {
        public interface Value extends java.io.Serializable {}
        public String nam;
        public Value val;
}
```

La interfaz *PropertyManager* en la especificación CORBA define métodos para asociar estáticamente propiedades por defecto para cualquier grupo creado por ese *manager*, o dinámicamente para un grupo específico. Por su complejidad, los detalles del manejo de propiedades lo posponemos a una sección posterior y sólo definimos aquí la interfaz OMG/IDL que deben soportar los miembros del grupo para recibir las notificaciones de cambios en las propiedades de otros miembros:

```
interface PropertyListener {
        void properties_updated (in Location loc);
};
```

7.2.4. Elección del coordinador

Hay varias estrategias para seleccionar el miembro responsable de coordinar una determinada transferencia de estado: transferencia *pull*, donde el nuevo miembro selecciona a ese coordinador, o transferencia *push*, donde son los miembros con estado los que lo seleccionan. El capítulo anterior mostró que el protocolo *push* presenta un mejor rendimiento en aquellos casos en que no se dispone de soporte multipunto por hardware: por esta razón, el servicio de transferencia debe predefinirse como *push* o *pull*, ya que no es posible que el servicio seleccione automáticamente el mejor modo de funcionamiento.

Bajo las dos estrategias, la elección del coordinador puede realizarse automáticamente por la infraestructura de transferencia de estado, o bien directamente por la aplicación. En el primer caso, la aproximación más sencilla es permitir que cualquier miembro sea el coordinador. Una posibilidad más elaborada es asociar pesos a cada miembro, permitiendo así balancear la carga de transferencia entre los miembros con estado, o bien asignar las transferencias a uno o varios miembros predeterminados, aquellos que tengan mayores pesos.

Cuando la aplicación selecciona al coordinador, los miembros deben implementar la interfaz *CoordinatorElector*, que define únicamente un método denominado *get_coordinator*, método que, al ser invocado, debe devolver determinísticamente uno de los miembros con estado presentados:

```
interface CoordinatorElector {
  Location get_coordinator (in Locations locs);
};
```

CORBA define *Location* como *CosNaming::Name*, que es a su vez un *string*. Por lo tanto, los miembros vienen identificados por una cadena de caracteres y la aplicación selecciona al coordinador a partir de estas identificaciones. En el método anterior, sólo los miembros con estado, susceptibles de ser coordinadores, son enviados en el único parámetro.

La operación debe ser determinista, todos los miembros consultados deben devolver el mismo miembro; sin embargo, si el protocolo empleado es *pull*, es responsabilidad exclusiva del nuevo miembro el escoger ese coordinador, por lo que en este caso ese determinismo no es necesario.

Cuando la infraestructura selecciona al coordinador, el algoritmo es diferente para protocolo *push* y *pull*. En el primer caso, se balancea la carga sobre los distintos miembros con estado y, en el segundo caso, se selecciona el coordinador de acuerdo con su rango en la vista: el miembro sin estado con menor rango selecciona al miembro con estado con menor rango, etc. Debe notarse que este segundo algoritmo no balancea realmente la carga. Por ejemplo, si una vista instala un nuevo miembro y, antes de que concluya su transferencia, se instala una segunda vista con otro nuevo miembro, ambos seleccionan el mismo coordinador.

Para dar una mayor flexibilidad a este esquema, este algoritmo permite definir una determinada propiedad que denominamos *peso del coordinador*. Si esta propiedad está definida y tiene asociado un valor numérico, se entiende como el peso que cada miembro tiene para ser seleccionado como coordinador.

7.2.5. Concurrencia en la transferencia

Si el grupo contiene más de un miembro sin estado consistente, es necesario realizar más de una transferencia. Estas pueden realizarse simultáneamente si hay varios coordinadores o bien cuando un único coordinador las realiza concurrentemente. La primera aproximación depende de la elección del coordinador, cuando dos miembros sin estado seleccionan diferentes coordinadores. La segunda opción tiene dos posibilidades, que pueden determinarse de nuevo mediante propiedades en el sistema de transferencia de estado. Estas dos posibilidades no son mutuamente excluyentes.

En primer lugar, definimos la propiedad booleana *ConcurrentTransfersAllowed*, describiendo cuándo un servidor puede realizar simultáneamente transferencias independientes a distintos clientes. Esta opción no es compatible lógicamente con que el servidor se encuentre fuera del grupo en tanto se realiza la transferencia: si inicia una transferencia en una vista y antes de concluirla se instala una nueva vista con un nuevo miembro y debe transferirle igualmente el estado, habrá un desfase lógico debido a los mensajes de la vista anterior que no ha podido procesar. Por esta razón, las transferencias concurrentes sólo pueden realizarse, en este caso, sobre miembros que la inician en la misma vista.

La segunda posibilidad viene definida con la propiedad booleana *MultipleTransfersAllowed*: el servidor realiza la transferencia a varios clientes simultáneamente y todos reciben el mismo estado. Esto implica que todos deben devolver la misma coordinación de fase en cada paso. Si alguno no la devuelve, debe considerarse que funciona erróneamente y ser expulsado del grupo.

7.3. Interfaz de transferencia

La sección anterior mostró varios aspectos que influyen directamente en la interfaz final: las transferencias pueden realizarse en varios pasos, pueden ser interrumpidas al cambiar las vistas y pueden incluir concurrentemente varios miembros coordinados.

La interfaz se define, entonces, incluyendo en cada método las fases de coordinación que hemos definido, que permiten realizar la sincronización en las transferencias en varios pasos y en las transferencias interrumpidas. La siguiente lista muestra los métodos que deben soportarse, sin detallar su funcionalidad:

- *start_transfer*: se invoca sobre el coordinador, que devuelve una fase a emplear en los siguientes pasos de la transferencia. El coordinador recibe aquí la lista de miembros a coordinar, en caso de una transferencia concurrente.
- *stop_transfer*: se invoca, tanto sobre el coordinador, como sobre el nuevo miembro, para indicar el final de una transferencia de estado, bien porque ésta haya finalizado, bien porque uno de los miembros en la transferencia haya caído.
- *get_state*: en comparación con el método con el mismo nombre en la interfaz *Checkpointable*, incluye ahora un parámetro con la fase que se espera y la siguiente fase a enviar. A partir de la fase que se devuelve, el sistema de transferencia de estado obtiene la información necesaria para saber cuándo una transferencia ha finalizado.
- *set_state* incluye también un parámetro con la fase asociada al estado que se establece.

- *sync_transfer*: invocado sobre el nuevo miembro para obtener cuál es la fase que espera. Se invoca al principio de la transferencia y tras instalarse cada vista. En el caso de emplear la aproximación donde se consideran que los miembros en transferencia están excluidos del grupo (*MembersOnTransferExcludedFromGroup*), se invoca, lógicamente, una única vez.
- *interrupt_transfer*: se invoca sobre el coordinador y sobre el nuevo miembro en cada momento en que la transferencia quede interrumpida. En el caso de *MembersOnTransferExcludedFromGroup* no se invoca, entonces, en ningún momento.
- *continue_transfer*: invocación al reanudarse una transferencia. Tampoco se invoca nunca en el caso de *MembersOnTransferExcludedFromGroup*.

Estos métodos se especifican en una nueva interfaz, *StateHandler*, que no tienen ninguna relación de dependencia con la interfaz *Checkpointable*. Sin embargo, en lugar de especificar una única interfaz, la funcionalidad queda desglosada en dos interfaces con relación de herencia, la primera de los cuales soporta la funcionalidad del caso más simple, donde toda transferencia queda cancelada tras un cambio de vista. La interfaz más simple es *BasicStateHandler*, que bajo OMG/IDL es:

```
interface BasicStateHandler {
  void  start_transfer     (in Locations joining_members,
                            inout PhaseCoordination phase);
  State get_state          (inout PhaseCoordination phase);
  void  set_state          (in State s,
                            in PhaseCoordination phase);
  void  stop_transfer      (in Locations falling_members,
                            in boolean transfer_finished);
};
```

Esta interfaz queda especializada, en el caso de poder continuar transferencias interrumpidas, mediante la interfaz *StateHandler*:

```
interface StateHandler : BasicStateHandler {
  void  sync_transfer      (in Location coordinator,
                            inout PhaseCoordination phase);
  void  interrupt_transfer (inout PhaseCoordination phase);
  void  continue_transfer  (in Locations joining_members,
                            inout PhaseCoordination coordinator_phase,
                            in PhaseCoordination joining_phase);
};
```

La interfaz resulta compleja para poder soportar todas las posibilidades de transferencia. Aun así, resulta mucho menos compleja que la lógica que la propia aplicación debe incluir para soportar transferencias interrumpidas. La sección

posterior de *ejemplos de uso* muestra esta complejidad y cómo la interfaz definida permite soportar las distintas posibilidades.

7.3.1. sync_transfer

Esta operación, sólo soportada en la interfaz más compleja, *StateHandler*, se invoca sobre el nuevo miembro cuando se va a iniciar la transferencia y, a continuación, tras cada interrupción de transferencia debida a cambios de vistas, de tal forma que el miembro pueda devolver una fase de coordinación que exprese su actual estado de transferencia.

```
void  sync_transfer (in Location coordinator, inout PhaseCoordination phase);
```

El primer parámetro informa sobre el coordinador asignado y el segundo parámetro devuelve la fase esperada. Si había ya una transferencia en curso, cancelada por la caída del coordinador previo, este método se invoca con la última fase devuelta al recibir una notificación *interrupt_transfer*, que será una referencia nula en otro caso.

Si dos miembros susceptibles de pertenecer a la misma transferencia simultánea devuelven dos fases distintas, no podrán recibir el mismo estado simultáneamente.

7.3.2. start_transfer

Esta operación se invoca sobre el coordinador cuando se inicia una transferencia, para permitir cualquier preprocesado que la aplicación precise, y poder sincronizar la transferencia con el nuevo miembro.

```
void  start_transfer (in Locations joining_members, inout PhaseCoordination phase);
```

El primer parámetro incluye las localizaciones de los miembros nuevos, que tendrán la misma transferencia de estado. Si la aplicación soporta transferencias concurrentes de un coordinador a varios miembros, cada uno de ellos bajo una transferencia diferente, este método es invocado repetidamente para cada uno de esos miembros, en lugar de incluirlos a todos en el primer parámetro.

Si se soporta sólo la interfaz *BasicStateHandler*, el segundo parámetro no contiene ninguna información válida. En caso contrario, antes de invocarse esta operación, el sistema habrá solicitado al nuevo miembro su fase de coordinación empleando el método *sync_transfer*, y esta fase se transfiere ahora al coordinador.

Devuelve, en ese mismo segundo parámetro, una fase que se empleará en la siguiente llamada a *get_state*.

7.3.3. get_state

Como en el caso de su operación homónima en la interfaz *Checkpointable*, *get_state* devuelve el estado de un miembro del grupo y se invoca, exclusivamente, sobre miembros que presenten un estado consistente.

```
State get_state (inout PhaseCoordination phase);
```

El estado o porción de estado, se devuelve con el primer parámetro; el segundo parámetro contiene la fase que determina el estado o porción de estado a devolver, proveniente de una llamada anterior a *start_transfer* o *get_state*. En este mismo parámetro debe devolver la fase que indica el siguiente paso de la transferencia.

Esta operación se invoca repetidamente sobre el coordinador mientras que la fase devuelta no indique el final de la transferencia.

Esta operación no incluye las identidades de los miembros coordinados cuando, bajo una transferencia concurrente, el mismo coordinador pueda recibir invocaciones *get_state* concurrentes. Si la aplicación necesita transferir diferentes estados según sea el miembro destino, puede emplear la fase de coordinación con ese fin.

7.3.4. set_state

```
void  set_state (in State s, in PhaseCoordination phase);
```

Con este método se establece el estado en el nuevo miembro. El primer parámetro define ese estado y el segundo, la fase de coordinación asociada.

7.3.5. interrupt_transfer

Se invoca sobre el coordinador y nuevo miembro o miembros cuando la transferencia se interrumpe por un cambio de vista. En el caso de *MembersOnTransferExcludedFromGroup*, no se invoca en ningún momento.

```
void interrupt_transfer (inout PhaseCoordination phase);
```

Debe devolverse en su único parámetro una fase que indique el estado del miembro y la transferencia. Este mismo parámetro se usa para indicar la última fase dada por este miembro en una operación previa de transferencia.

La fase devuelta será empleada más adelante en la siguiente llamada a *continue_transfer*, en el caso del coordinador, o a *sync_transfer*, en el caso del nuevo miembro. Entre ambas llamadas, el miembro recibirá probablemente (y dependiendo del comportamiento ante vistas seleccionado) mensajes del grupo que alterarán su estado, por lo que esta fase permite al miembro sintetizar los cambios producidos por esos mensajes.

7.3.6. continue_transfer

Esta operación se invoca exclusivamente sobre el coordinador, una vez que una transferencia ha sido interrumpida, enviándosele las fases dadas por el coordinador y el nuevo miembro en la anterior llamada a *interrupt_transfer/sync_transfer*. Como ocurre con *interrupt_transfer*, en el caso de *MembersOnTransferExcludedFromGroup*, esta operación no se invoca en ningún momento.

```
void continue_transfer (in Locations joining_members,
                        inout PhaseCoordination coordinator_phase,
                        in PhaseCoordination joining_phase);
```

El primer parámetro incluye las identidades de los miembros a coordinar, pues puede haber variado con el cambio de vista que produjo la interrupción de la transferencia. El segundo parámetro incluye la fase que este miembro devolvió en la llamada a *interrupt_transfer* y, en este parámetro, debe devolver la fase que será empleada en las sucesivas llamadas a *get_state*. Finalmente, el tercer parámetro incluye la fase devuelta por el miembro coordinado tras invocar *sync_transfer*.

7.3.7. stop_transfer

```
void stop_transfer (in Locations falling_members, in boolean transfer_finished);
```

Cuando se invoca sobre el coordinador, esta operación da información sobre los miembros excluidos de la transferencia porque hayan caído. Al invocarse, tanto sobre el coordinador como sobre el nuevo miembro, el segundo parámetro indica cuándo la transferencia ha concluido. En el coordinador, si la transferencia se considera cancelada en cambios de vistas, este parámetro se recibirá siempre con valor *true*.

Al estudiar las condiciones que deben cumplir las aplicaciones y sus mensajes para realizar la transferencia de estado, comprobamos que es preciso que el GMS informe a la aplicación del bloqueo de vistas; es en ese momento cuando el sistema de transferencia de estado realiza las llamadas a *interrump_transfer* y desencola los mensajes que han sido enviados durante la vista. Por esa razón, un sistema puede recibir el evento de interrupción de vista cuando realmente ésta ya haya finalizado porque el otro miembro en la transferencia haya caído. En este caso, el miembro recibe el evento de interrupción y, a continuación, el de finalización de la transferencia.

7.4. Propiedades de miembros

Al describir el diseño de la interfaz, introdujimos el uso de propiedades sin entrar en detalle sobre su manejo. Las propiedades no están ligadas a ningún tipo en particular y el único requisito es que se especifican mediante una cadena de caracteres. En la especificación CORBA, la interfaz *PropertyManager* define la forma de asociar propiedades a grupos.

La granularidad en esta interfaz permite especificar propiedades exclusivamente a nivel de grupo, no sobre los propios miembros. Para las propiedades que definen el comportamiento del grupo, como sus posibilidades de concurrencia en las transferencias de estado, esta granularidad es suficiente. Al discutir la elección del coordinador, introdujimos también una nueva propiedad que define el peso de cada miembro en una transferencia. Sin embargo, esta propiedad no es asignable a un grupo, sino a miembros particulares, donde la granularidad dada por el *PropertyManager* es insuficiente.

Para estos casos de granularidad insuficiente, predefinimos una nueva propiedad que asocia a cada miembro un conjunto de propiedades; su tipo asociado se define como una secuencia de propiedades para cada miembro en el grupo:

```
typedef sequence <Property> Properties;

struct MemberProperties {
        Location member;
        Properties props;
};
typedef sequence <MemberProperties> MemberPropertiesList;
```

Para la actualización dinámica de propiedades, la interfaz *PropertyManager* incluye un método para modificar las propiedades de un grupo, *set_properties_dynamically*:

```
interface PropertyManager {
     /* . . . */
     void set_properties_dynamically (in ObjectGroup object_group,
                                      in Properties overrides)
        raises (ObjectGroupNotFound, InvalidProperty, UnsupportedProperty);
     Properties get_properties       (in ObjectGroup object_group)
        raises (ObjectGroupNotFound);
};
```

Sin embargo, no resultaría práctico que un determinado miembro cambiara una o varias de sus propiedades asociadas mediante este método, pues entonces deberían especificarse todas las propiedades del grupo. Además, habría problemas de concurrencia si varios miembros cambiaran simultáneamente sus propias propiedades.

Esta interfaz se extiende entonces con dos nuevas operaciones:

```
interface PropertyManager {
     /* . . . */
     void set_member_properties       (in ObjectGroup object_group,
                                       in Location member,
                                       in Properties overrides)
        raises (ObjectGroupNotFound, InvalidProperty, UnsupportedProperty);
     Properties get_member_properties (in ObjectGroup object_group,
                                       in Location)
        raises (ObjectGroupNotFound);
};
```

Al emplear pertenencia a grupo controlada por la infraestructura, las factorías (*GenericFactory*) reciben las propiedades de cada miembro cuando éste debe ser creado:

```
interface GenericFactory {
  Object create_object (in TypeId type_id,
                        in Criteria the_criteria,
                        out FactoryCreationId id);
  void delete_object   (in FactoryCreationId id);
};
```

En nuestra propuesta, las propiedades quedan incluidas en el parámetro del tipo *Criteria*: un miembro recibe sus propias propiedades, las de los demás miembros del grupo se le transfieren durante la transferencia de estado. Cuando las propiedades cambian dinámicamente, un miembro que implementa la interfaz *PropertiesListener* recibe la notificación del cambio.

Definimos a continuación las propiedades que hemos mencionado en ésta y las anteriores secciones para implementar la transferencia de estado:

```
const string COORDINATOR_WEIGHT = “CoordinatorWeight”;
const string MEMBER_PROPERTIES = “MemberProperties”;
const string BEHAVIOUR_ON_VIEW_CHANGES = “BehaviourOnViewChanges”;
       const long MEMBERS_ON_TRANSFER_EXCLUDED_FROM_GROUP = 0;
       const long STATELESS_MEMBERS_DO_NOT_BELONG_TO_GROUP = 1;
       const long STATELESS_MEMBERS_BELONG_TO_GROUP = 2;
const string CONCURRENT_TRANSFERS_ALLOWED = “ConcurrentTransfersAllowed”;
const string MULTIPLE_TRANSFERS_ALLOWED = “MultipleTransfersAllowed”;
const string USING_MEMBER_PROPERTIES = “UseMemberProperties”;
const string COORD_ELECTION_APP_CTRL = “CoordinatorElectedByApplication”;
```

Las cuatro últimas propiedades tienen valores booleanos; basta con que se definan para que se asuma la propiedad asociada. Para el comportamiento en caso de cambios de vistas, el valor asociado es un entero, con valores ya predefinidos.

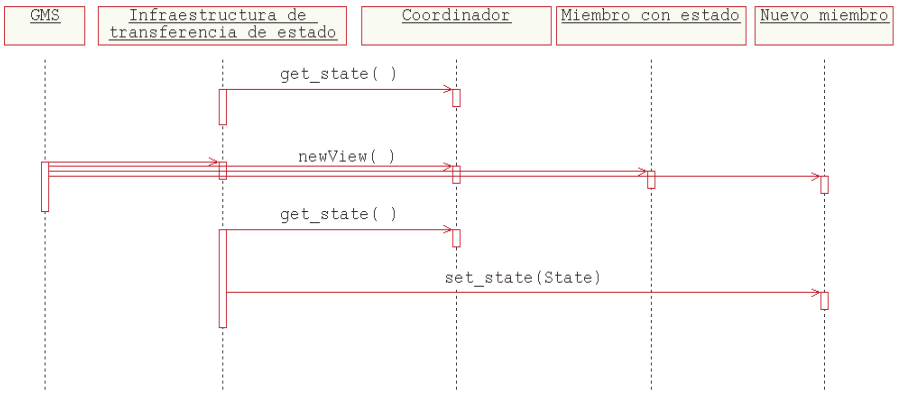

Figura 7.1. Interfaz *Checkpointable*

7.5. Ejemplos de uso (*use cases*)

7.5.1. Interfaz *Checkpointable*

La figura 7.1 muestra el siguiente escenario, donde un cambio de vista provoca que el protocolo se reinicie.

- Una aplicación crea un nuevo miembro en un grupo que emplea la interfaz de transferencia de estado *Checkpointable*.
- Se selecciona como coordinador a uno de los miembros con estado.
- Se solicita el estado al coordinador.
- Se transfiere este estado al nuevo miembro; si se produce un cambio de vista antes de enviar este estado, debe reiniciarse el proceso: seleccionar coordinador, solicitarle estado y transferirlo al nuevo miembro.

7.5.2. Interfaz *BasicStateHandler*

Mostrar esta interfaz requiere observar su funcionamiento ante diversos eventos externos. La figura 7.2 muestra un escenario básico donde no se producen

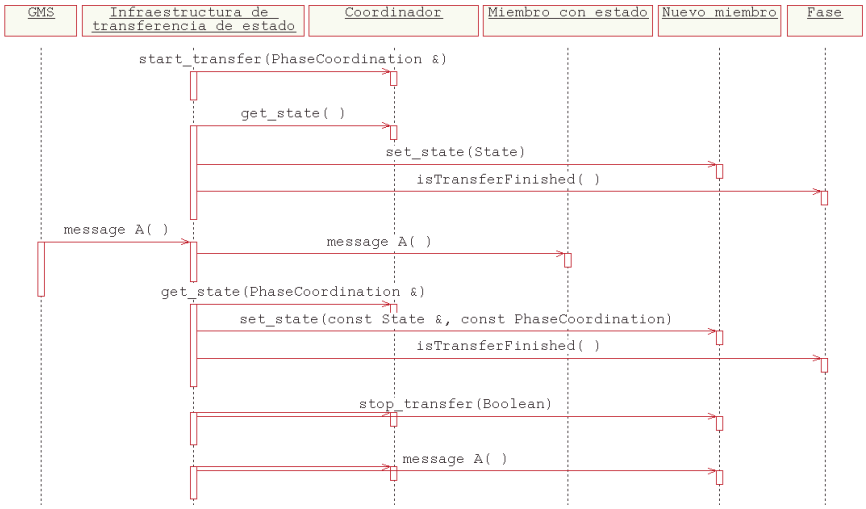

Figura 7.2. Interfaz *BasicStateHandler* sin cambios de vistas

cambios de vistas. En esta figura, se envía un único mensaje al grupo durante la transferencia.

- Una aplicación crea un nuevo miembro en un grupo que emplea la interfaz de transferencia de estado *BasicStateHandler*.
- Se selecciona como coordinador a uno de los miembros con estado.
- A partir de este momento, se bloquean todos los mensajes al coordinador y al nuevo miembro, para ser procesados tras concluir la transferencia.
- El coordinador recibe la notificación de inicio de transferencia, con la identificación de los miembros a los que va a realizar esa transferencia. Recibe también una fase sin valor inicial y devuelve una coordinación de fase que será usada en las siguientes etapas.
- El sistema de transferencia de estado entra en un bucle:
- 1- Verifica con esa coordinación de fase si la transferencia ha finalizado.
- 2- Solicita al coordinador el estado, pasándole la coordinación de fase que ha devuelto en el paso anterior y esperando una nueva coordinación de fase para la siguiente etapa.
- 3- Se transfiere el estado y la coordinación devueltas al nuevo miembro o nuevos miembros.
- Al concluir la transferencia, el sistema comunica al coordinador y al nuevo o nuevos miembros que la transferencia ha concluido.

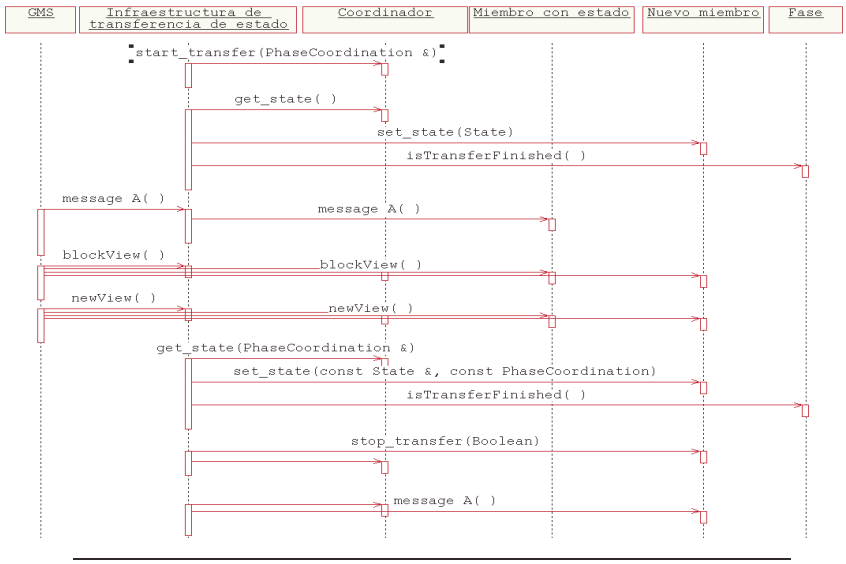

Figura 7.3. Interfaz *BasicStateHandler*, *MembersOnTransferExcludedFromGroup*

- Los mensajes encolados durante la vista se envían para su procesamiento por el coordinador y nuevo o nuevos miembros. Si el coordinador tuviera más miembros que coordinar, no podrá todavía procesar esos mensajes, que seguirían encolados.

El escenario se complica si se producen cambios de vistas. La figura 7.3 muestra un cambio de vista en un grupo en modo *MembersOnTransferExcludedFromGroup*, donde los miembros involucrados en la transferencia no se ven afectados por cambios de vistas; la figura 7.4 incluye el mismo escenario, en el caso en que el coordinador se cae.

- El inicio sigue la misma pauta que en el previo escenario pero, antes de concluir la transferencia, se produce un cambio de vista. Si el coordinador se cae, los nuevos miembros reciben la notificación de parada de transferencia y, eventualmente, ésta se reiniciará con otro coordinador. Si uno o más de los miembros coordinados se cae, el coordinador recibe una notificación de parada de transferencia sobre esos miembros y, si todos los miembros hubieran caído, el parámetro *finished_transfer* tendría un valor *true*.
- El coordinador mantiene los mensajes encolados hasta el momento en una segunda cola, ya que debe conocer en todo momento los mensajes encolados provenientes de la vista actual o de vistas anteriores.
- La transferencia continúa con los pasos anteriores.
- Al concluir la transferencia, el sistema comunica al coordinador y al nuevo o nuevos miembros que la transferencia ha concluido.

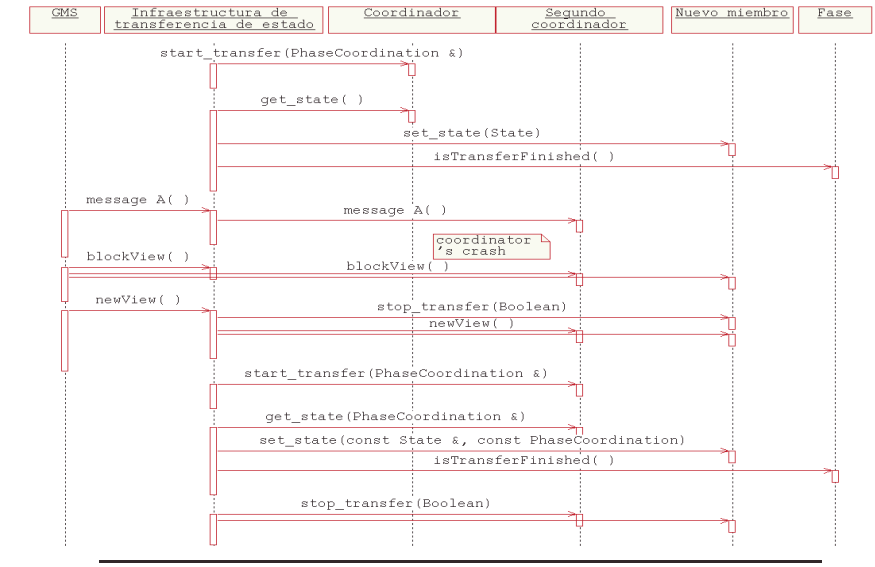

Figura 7.4. Interfaz *BasicStateHandler*, *MembersOnTransferExcludedFromGroup* con caída del coordinador.

- Los mensajes encolados por los nuevos miembros durante las vistas que ha durado la transferencia son desbloqueados y procesados. El coordinador solamente puede procesar los mensajes encolados en vistas anteriores. Los de la vista actual sólo los podrá procesar si no tiene pendiente ninguna transferencia de estado a otro miembro.

Falta en estos escenarios mostrar un caso en que el coordinador debe iniciar una segunda transferencia en la misma vista en que concluye una transferencia previa. Por su semejanza con otros ejemplos, ese caso se muestra más adelante, en la figura 7.6. Primero observamos un ejemplo en el que los cambios de vista provocan una cancelación de la transferencia:

- Una aplicación crea un nuevo miembro en un grupo que emplea la interfaz de transferencia de estado *BasicStateHandler* y especifica como propiedad del grupo que el comportamiento ante cambios de vistas es cancelar la transferencia: *StatelessMembersDoNotBelongToGroup*. Es posible emplear también *StatelessMembersBelongToGroup*, pero la utilidad de esta aproximación es poco clara si los miembros no pueden continuar una transferencia cancelada.
- Los pasos de selección del coordinador y el bucle de obtención del estado y su transferencia al nuevo miembro son idénticos a los ejemplos anteriores, pues el comportamiento sólo difiere ante un cambio de vista.
- Se produce un cambio de vista. El sistema comunica al coordinador y nuevo o nuevos miembros que la transferencia se para, sin haber concluido. Los mensajes encolados en el coordinador son desbloqueados, este miembro sólo

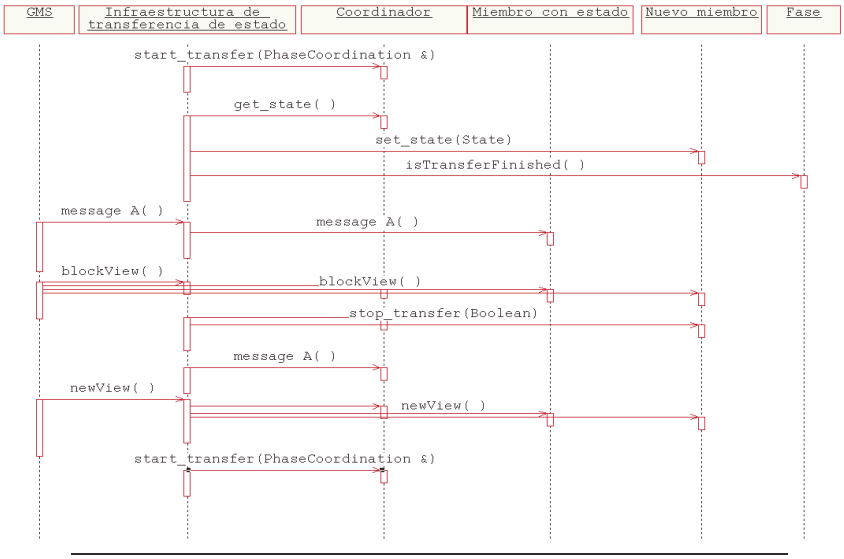

Figura 7.5. Interfaz *BasicStateHandler*, *StateMembersBelongToGroup*

recibirá el evento de nueva vista tras haber procesado esos mensajes. El nuevo miembro, sin embargo, descarta esos mensajes encolados.

- Si no se ha caído el nuevo miembro, la transferencia debe continuar, por lo que el sistema la reiniciará, usando el mismo coordinador. Si se hubiera caído, se selecciona directamente un nuevo coordinador, sin informar al nuevo miembro. El coordinador recibe la notificación de inicio de transferencia y los pasos anteriores se repiten en el mismo orden.
- Al concluir la transferencia, el sistema comunica al coordinador y al nuevo o nuevos miembros que la transferencia ha finalizado.
- Los mensajes encolados durante la última vista se envían para su procesamiento por el coordinador y nuevo o nuevos miembros. Si el coordinador tuviera más transferencias por realizar, no podrá todavía procesar esos mensajes.

Este escenario se muestra en la figura 7.5, aunque los pasos en la segunda vista se han obviado. Para ver el último paso de este escenario, en el que el coordinador no puede procesar los mensajes bloqueados hasta finalizar la nueva transferencia en marcha, empleamos el siguiente escenario, que refleja la figura 7.6:

- El inicio del escenario se realiza como en el caso anterior y, antes de finalizar la transferencia, una nueva vista se instala para insertar un nuevo miembro, al que se le asigna el mismo coordinador. La transferencia en marcha se para.
- Tras instalarse la vista, se reinicia la transferencia entre el coordinador y el anterior miembro. Los mensajes en el grupo están bloqueados para los dos

Figura 7.6. Interfaz *BasicStateHandler*, *StateMembersBelongToGroup* con sucesivas coordinaciones.

miembros de la transferencia, así como para el nuevo miembro, que aún no ha iniciado ninguna transferencia.

- La primera transferencia de estado termina; el miembro que ha adquirido el estado puede procesar los mensajes enviados en esta vista, mientras que el coordinador debe, todavía, finalizar la segunda transferencia antes de poderlos procesar.

7.5.3. Interfaz *StateHandler*

Para mostrar esta interfaz, empleamos dos casos, con y sin caída del coordinador. Usamos como modos del grupo *StatelessMembersDoNotBelongToGroup* y *StatelessMembersBelongToGroup*, ya que no es necesario un ejemplo con *MembersOnTransferExcludedFromGroup*, pues en este modo la interfaz se usa sólo mínimamente, al no invocarse las operaciones *continue_transfer* o *interrupt_transfer*. En ambos ejemplos, se omiten los mensajes *get_state* y *set_state* que se realizan durante la transferencia.

La figura 7.7 muestra un escenario donde los nuevos miembros no se consideran miembros del grupo:

- Una aplicación crea un nuevo miembro en un grupo que emplea la interfaz de transferencia de estado *StateHandler* y especifica como propiedad del grupo *StatelessMembersDoNotBelongToGroup*.

Figura 7.7. Interfaz *StateHandler*, *StateMembersDoNotBelongToGroup* sin caída del coordinador

- El escenario contiene tres miembros con estado y un nuevo miembro, aún sin estado consistente. Se selecciona como coordinador uno de los miembros con estado.
- El primer paso de la transferencia es consultar al nuevo miembro mediante *sync_transfer* sobre la fase que espera inicialmente.
- Se transfiere esta fase al coordinador, que inicia entonces la transferencia, mediante los mensajes *get_state*, *set_state* que no se muestran en la figura. Los miembros involucrados en la transferencia no reciben los mensajes del grupo; el escenario muestra un mensaje A que es recibido exclusivamente por los dos miembros con estado no involucrados en la transferencia.
- Se produce un cambio de vista por la caída de un miembro con estado. El sistema transfiere el bloqueo de vista a los miembros y comunica al coordinador y nuevo o nuevos miembros que la transferencia se interrumpe. Los mensajes encolados en el coordinador son desbloqueados, este miembro sólo recibirá el evento de nueva vista tras haber procesado esos mensajes. El nuevo miembro, sin embargo, descarta esos mensajes encolados.
- Tras instalarse la nueva vista, la transferencia se reinicia. El nuevo miembro recibe, de nuevo, una solicitud de sincronización de la transferencia, que incluye, ahora, la fase que devolvió durante *interrupt_transfer*.
- El coordinador recibe una petición *continue_transfer*, con la fase que devolvió en *interrupt_transfer* y la dada por el nuevo miembro en *sync_transfer*. La opción

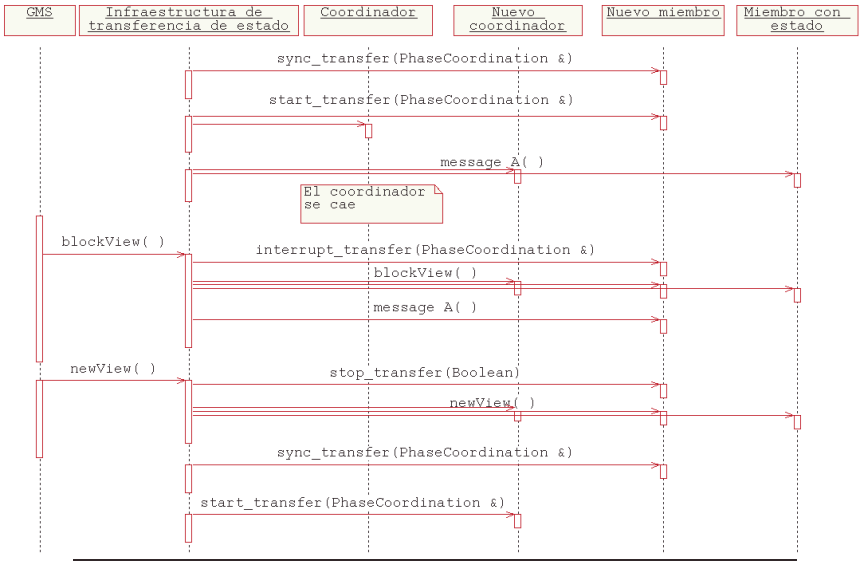

Figura 7.8. Interfaz *StateHandler*, *StateMembersNotBelongToGroup* con caída del coordinador

segura por defecto es reiniciar la transferencia, pero el intercambio de información puede permitir a una aplicación alterar este comportamiento.

- Cuando la transferencia concluye, se envía a los miembros que la realizaron cualquier mensaje encolado. Estos mensajes encolados sólo pueden provenir de la vista actual. Como en los ejemplos anteriores, si el coordinador tuviera más transferencias por realizar, no podría todavía procesar esos mensajes.

La figura 7.8 muestra un escenario parecido donde los nuevos miembros se consideran parte del grupo aún sin estado:

- Los pasos iniciales se realizan como en el caso anterior, pero el grupo se define con propiedad *StatelessMembersBelongToGroup*.
- El coordinador se cae antes de concluir la transferencia; el GMS envía un evento de bloqueo de vista.
- Al bloquearse la vista, la transferencia se para, por lo que el único miembro sobreviviente de la transferencia recibe un evento *interrupt_transfer*.
- Los mensajes bloqueados se envían al nuevo miembro, que se considera parte del grupo.
- Al instalarse la vista, el sistema de transferencia de estado detecta la caída del coordinador, por lo que invoca *stop_transfer* sobre el nuevo miembro, antes de comunicarle la nueva vista.
- Se selecciona un nuevo coordinador, y se reinicia la transferencia invocando primero *sync_transfer* sobre el nuevo miembro.

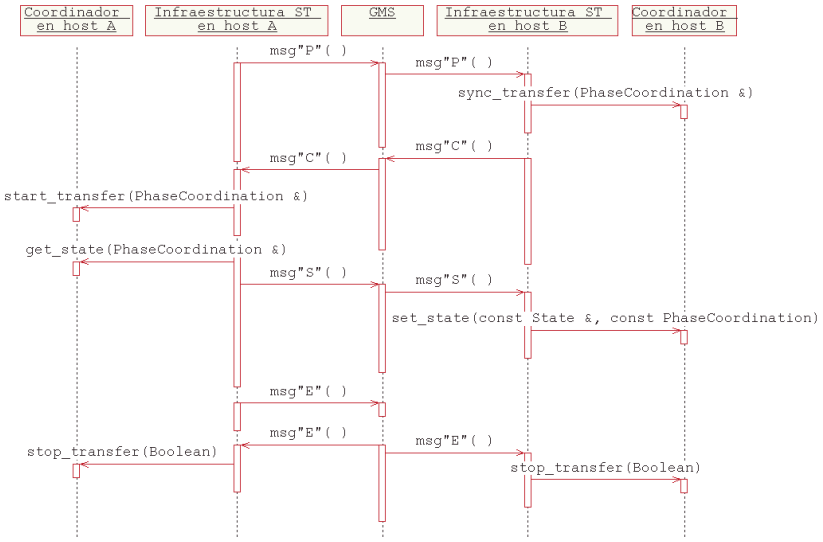

Figura 7.9. Implementación sobre protocolo *push* sin cambios de vistas

- El nuevo coordinador recibe el evento *start_transfer* y se reinician todos los pasos previos.

7.6. Implementación sobre protocolos de bajo nivel

El anterior capítulo mostró los protocolos de bajo nivel y sus beneficios relativos, y este capítulo, al desarrollar el protocolo de alto nivel, no ha mostrado ningún énfasis sobre el protocolo subyacente. El motivo es que los protocolos *pull* y *push* se diferencian únicamente en la elección inicial del coordinador, por lo que en líneas generales, el comportamiento es idéntico y la única excepción es, precisamente, la interfaz de alto nivel de elección del coordinador, que puede ser no determinista en el caso *pull*.

Mostrar la implementación detallada de los protocolos implicaría una larga lista de condicionales, teniendo en cuenta los tres modos de comportamiento del grupo y las tres posibles interfaces a implementar. Como, además, la conversión entre los mensajes del protocolo de bajo nivel y las operaciones en la interfaz de aplicación es prácticamente directa, nos limitamos a mostrar dos escenarios que muestran esta conversión, usando protocolo *push*.

La figura 7.9 muestra un escenario común, válido para las interfaces *BasicStateHandler* y *StateHandler*, sin cambios de vistas durante la transferencia. En este caso, los tres modos de comportamiento generan las mismas operaciones. Cuando se producen cambios de vistas, el comportamiento es ya totalmente

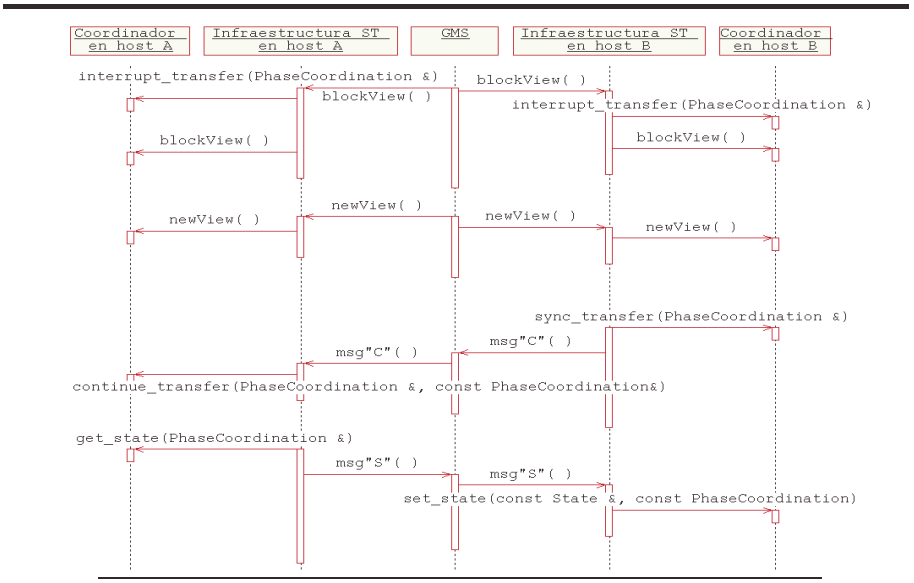

Figura 7.10. Implementación sobre protocolo *push* con cambios de vistas

diferente y las operaciones difieren completamente si se emplea la interfaz *BasicStateHandler*, donde la transferencia queda cancelada, o la interfaz *StateHandler*, donde se considera interrumpida; sin embargo, en ambos casos, los mensajes subyacentes son los mismos. La figura 7.10 muestra el cambio de mensajes y las operaciones durante un cambio de vista para la interfaz *StateHandler* cuando no se emplea modo *MembersOnTransferExcludedFromGroup*.

El anterior capítulo desarrolló un protocolo más, denominado *push-one step*, donde el estado se transfiere en un único mensaje multipunto. Empleando este protocolo de bajo nivel, no es posible, evidentemente, implementar ninguna de las interfaces más avanzadas, y la aplicación está limitada a la interfaz *Checkpointable*.

7.7. Conclusiones

Con la interfaz de aplicación desarrollada, es posible implementar una transferencia de estado considerablemente compleja, al habilitarse un canal de comunicación bidireccional entre las partes que participan de esa transferencia.

Aunque teóricamente las transferencias de estado no básicas deberían cancelarse ante cada cambio de vista, las necesidades de una aplicación específica pueden hacer obviar este requisito, a costa de no respetar escrupulosamente el modelo de sincronía virtual: si la aplicación se programa con esa premisa, esta violación del modelo puede realizarse sin ninguna repercusión negativa.

Además de este soporte de transferencias interrumpidas y transferencias en varios pasos, la interfaz soporta concurrencia en la transferencia, empleo de propiedades de miembros y diferentes posibilidades en la elección del coordinador. Y, pese a la complejidad de la interfaz, ésta se ha definido de tal forma que aplicaciones que precisan transferencias simples no deban implementar operaciones complejas.

Por último, la interfaz ha quedado definido en JavaRMI y en OMG/IDL, y su especificación se ha realizado maximizando su compatibilidad con la actual especificación de tolerancia a fallos de CORBA. Basándonos en esta interfaz, hemos realizado, asimismo, su implementación sobre *Sensei*, el sistema de comunicaciones en grupo que hemos desarrollado específicamente en este proyecto. Esta implementación se detalla en el capítulo 10.

Capítulo 8 - SENSEIGMS

SenseiGMS es un sistema de grupo de comunicaciones fiables desarrollado específicamente para este proyecto. Está basado en el modelo de sincronía virtual y sobre este sistema se han implementado los protocolos de transferencia de estado presentados en los anteriores capítulos, así como varias técnicas desarrolladas para la simplificación de sistemas fiables, descritas en los siguientes capítulos.

Su diseño se ha realizado bajo tres requisitos:

- Soportar una funcionalidad simple y básica para facilitar su implementación, pues el objetivo de este proyecto no es el desarrollo de un sistema de comunicaciones fiables de grupo, sino la implementación de los protocolos de transferencia de estado estudiados y de herramientas que simplifiquen el desarrollo de aplicaciones replicadas.
- Limitar su funcionalidad a la descrita en el modelo de sincronía virtual, permitiendo únicamente las extensiones disponibles en otros sistemas de comunicaciones ampliamente conocidos, como es el envío de mensajes punto a punto. El objetivo de este requisito es poder sustituir este sistema por esos sistemas de comunicaciones existentes, generalmente más complicados funcionalmente y con mejor rendimiento.
- Presentar una interfaz orientada a objetos y con soporte de mensajes tipados, con el objetivo de facilitar el desarrollo de las capas situadas sobre este sustrato de comunicaciones.

Este capítulo presenta el diseño de SenseiGMS y los algoritmos que lo sustentan, así como el modelo de programación que soporta. Las razones que han impuesto la creación de un sistema propio de comunicaciones en grupo se reducen básicamente a la necesidad de disponer de un sistema sobre CORBA y JavaRMI; en el momento de desarrollar la mayor parte de la tesis, no había disponibilidad de un sistema de comunicaciones en grupo sobre CORBA, a pesar de que la especificación de tolerancia a fallos de esta arquitectura estuviera ya aceptada. De la misma forma, la necesidad de desarrollar nuestro propio algoritmo para las comunicaciones multipunto fiables se justifica al observar que los algoritmos existentes se basan normalmente en el empleo de primitivas de comunicaciones multipunto no fiables (UDP), cuando la especificación de nuestro sistema incluye sólo soporte de comunicaciones fiables punto a punto.

8.1. Diseño

SenseiGMS soporta aplicaciones CORBA y JavaRMI. Su interfaz está orientada a objetos, especificada directamente en OMG/IDL y en Java, y como las aplicaciones deben definir sus mensajes de la misma forma, el soporte de mensajes tipados es directo.

La interfaz de aplicación de SenseiGMS está basada en la definida en *Isis* que, a su vez, ha sido la base para un gran número de sistemas de comunicaciones en grupo: *Horus*, *Ensemble*, *Totem*, *Transis* o *JavaGroups*.

Las comunicaciones se realizan directamente empleando CORBA o JavaRMI. La aproximación empleada en otros sistemas de comunicaciones es crear las pilas de protocolos sobre UDP o, incluso, directamente sobre IP, pilas que además resultan dinámicas en la selección de los algoritmos de ordenación de mensajes. Esta elección implica un peor rendimiento en SenseiGMS, pero simplifica su diseño. Adicionalmente, la elección de un *middleware* de alto nivel como substrato de comunicaciones permite emplear las facilidades y servicios asociados al mismo. Por ejemplo, empleando *applets* autentificados, se podrían desarrollar *applets* tolerantes a fallos más fácilmente que si el soporte se realiza directamente sobre *sockets*. De la misma manera, se pueden aprovechar otras facilidades como *http tunneling*, ya disponibles al trabajar con JavaRMI o CORBA, o la descarga dinámica de código.

Otra de las decisiones de simplificación del diseño es el soporte exclusivo de orden total causal en la secuenciación de mensajes. El algoritmo empleado para obtener el orden total causal, es mediante el paso de un testigo sobre un anillo lógico compuesto por los componentes del grupo, que puede cambiar su estructura según se incluyan nuevos miembros en el grupo o antiguos sean excluidos. La decisión de soportar exclusivamente orden total está influenciada, además, por tres motivos:

- La programación de aplicaciones replicadas se simplifica enormemente cuando los mensajes se procesan en orden total.

- El protocolo que sustenta el orden total puede soportar este orden a un coste no mayor que con empleo de orden causal o simplemente con mensajes *fifo* punto a punto fiables [Amir95].
- La especificación del servicio de tolerancia a fallos de CORBA incluye únicamente el soporte de este orden de secuenciación.

Para soportar simultáneamente ambas plataformas, SenseiGMS define su interfaz en OMG/IDL o en JavaRMI, e implementa separadamente los algoritmos, comunes en los dos casos, empleando Java. Desde el punto de vista de implementación, tanto la interfaz OMG/IDL, como la interfaz JavaRMI, comparten un mismo *package*, llamado *sensei.middleware.GMS*, aunque se localizan en distintos directorios. Los algoritmos comunes, definidos en el paquete *sensei.GMS*, se compilan sobre una de las plataformas previas, simplemente seleccionando un *classpath* diferente. El único requisito, en este caso, es que ambas interfaces sean completamente compatibles.

La interfaz que SenseiGMS presenta a la aplicación, que incluye, por ejemplo, operaciones para el envío de mensajes fiables, constituye su interfaz pública. Sin embargo, para realizar las comunicaciones internas también en CORBA o JavaRMI, como, por ejemplo, el paso del testigo entre miembros de un grupo, es necesario definir una segunda interfaz en los correspondientes lenguajes. Esta última interfaz es privada; en caso de querer sustituir SenseiGMS por otro grupo de comunicaciones, es necesario soportar exclusivamente la interfaz pública.

Finalmente, SenseiGMS no presenta ningún soporte de transferencia de estado. A pesar del énfasis de esta tesis en los algoritmos de transferencia de estado, esta decisión permite implementarlos totalmente en una capa superior, soportando así más fácilmente su migración a otros sistemas de comunicaciones en grupo.

8.1.1. Interfaz pública

Esta interfaz se basa en tres tipos básicos:

- Identidades de miembros, para lo que se emplea un simple entero, que debe ser único para cada miembro del grupo.
- Definición de mensajes. Cada aplicación puede definir sus propios mensajes, que deben pertenecer a una jerarquía de objetos definida. Estos mensajes pueden definirse en detalle obteniéndose, así, mensajes tipados.
- Definición de vistas, acorde al modelo de sincronía virtual. Cuando un grupo cambia su composición, cada miembro del grupo recibe una lista con todos los miembros. Esta lista constituye la vista que incluye, asimismo, una identidad que se define de forma numérica.

En OMG/IDL, su especificación es:

```
typedef long GroupMemberId;
typedef sequence <GroupMemberId> GroupMemberIdList;

struct View
{
   long              viewId;
   GroupMemberIdList members;
   GroupMemberIdList newMembers;
   GroupMemberIdList expulsedMembers;
};

valuetype Message {};
```

En SenseiGMS, por conveniencia, las vistas incluyen también una lista con los miembros excluidos y los nuevos miembros en el grupo. Es importante destacar que esta información no supone una funcionalidad añadida: un miembro nuevo recibe una vista en la que todos los demás miembros son listados como nuevos. Este detalle es importante para los algoritmos de transferencia de estado, que pueden simplificarse si las vistas contienen información con la que discernir cuándo un miembro es más antiguo que otro. Siguiendo el objetivo de no ofrecer mayor funcionalidad que otros sistemas de comunicaciones, esta información no se incluye.

La definición de mensaje como *valuetype* es esencial. Si se define como *interface*, el mensaje no es realmente transferido a cada miembro del grupo, que sólo recibe una referencia remota; si se define como *struct*, se transfiere a cada miembro del grupo, pero no se pueden definir nuevos mensajes por herencia.

Bajo JavaRMI, la especificación resulta similar; sin incluir los constructores para la clase, es:

```
final public class View implements java.io.Serializable
{
   public int viewId;
   public int[] members;
   public int[] newMembers;
   public int[] expulsedMembers;
}

public abstract class Message
  implements java.io.Serializable
{
}
```

Un objeto debe implementar la interfaz *GroupMember* para poder pertenecer a un grupo. Esta interfaz se especifica en JavaRMI como:

```
public interface GroupMember extends java.rmi.Remote
{
   public void processPTPMessage(int sender, Message message) throws RemoteException[2];
   public void processCastMessage(int sender, Message message) throws RemoteException;
   public void memberAccepted(int identity, GroupHandler handler, View view)
                              throws RemoteException;
   public void changingView() throws RemoteException;
   public void installView(View view) throws RemoteException;
   public void excludedFromGroup() throws RemoteException;
}
```

A través de esta interfaz, el miembro recibe todas las comunicaciones del grupo que, siguiendo el orden mostrado, es:

- Recepción de mensajes multipunto: todos los miembros ven la misma secuencia de mensajes. El miembro que envía un mensaje también lo recibe.
- Recepción de mensajes punto a punto. SenseiGMS respeta el orden total tanto entre mensajes multipunto como entre mensajes punto a punto; sin embargo, ésta no es una característica común a otros sistemas de comunicaciones fiables, donde estos mensajes pueden recibirse con antelación a mensajes multipunto que les preceden en el orden establecido. Por esta razón, ninguno de los algoritmos en Sensei es dependiente de este orden en mensajes punto a punto.
- Aceptación del miembro en el grupo, que es el primer evento que se recibe en todos los casos.
- Evento de cambios de vista. Se recibe antes de que se instale una nueva vista, e implica que el servicio de pertenencia de miembros ha detectado que es necesario un cambio de vista y está negociando la nueva composición del grupo. Este evento no se incluye en el modelo de sincronía virtual, pero es común en los sistemas de comunicaciones en grupo. Los protocolos de transferencia de estado precisan este evento, como se demostró en los capítulos anteriores.
- Instalación de una nueva vista.
- Evento de exclusión del grupo, ya sea por petición del miembro, o porque el GMS considere, erróneamente, que este miembro ha caído.

Un miembro interactúa con los demás miembros del grupo a través de la interfaz *GroupHandler*, que define la interfaz pública del servicio de pertenencia a grupos o GMS. Esta interfaz no define cómo crear el grupo o cómo incluir un miembro en un grupo ya existente, pues esta funcionalidad es específica a la implementación de SenseiGMS y se define como parte de su interfaz privada. Hay una relación biunívoca entre *GroupMember* y *GroupHandler*, es decir, cada miembro

[2] Todas las excepciones son *java.rmi.RemoteException.*

del grupo mantiene una instancia del servicio de pertenencia a grupos. Otros sistemas de comunicaciones en grupo permiten, sin embargo, que una instancia de este servicio controle varios miembros o, incluso, varios grupos, pero esta aproximación no implica una mayor o menor funcionalidad en ningún caso.

Esta interfaz se especifica en JavaRMI como:

```
public interface GroupHandler extends java.rmi.Remote
{
   public boolean castMessage(Message message) throws RemoteException;
   public boolean sendMessage(int target, Message message) throws RemoteException;
   public boolean leaveGroup() throws RemoteException;
   public int getGroupMemberId() throws RemoteException;
   public boolean isValidGroup() throws RemoteException;
}
```

Las operaciones definidas son:

- Enviar un mensaje al grupo, que es recibido por todos los miembros, incluido el que lo envía. Devuelve un valor que indica si el mensaje será procesado en la vista actual. A partir del momento en que el servicio de pertenencia al grupo bloquea una vista para negociar la composición de la siguiente, todos los nuevos mensajes que se envíen en el grupo son bloqueados, siendo sólo enviados en la siguiente vista.
- Enviar un mensaje a un miembro determinado del grupo. Para mantener una interfaz coherente, esta operación devuelve un valor *true* si el mensaje es recibido por el miembro destinatario en la misma vista.
- Abandonar el grupo. Esta operación no es inmediata, el miembro no es efectivamente expulsado hasta recibir el evento de expulsión.
- Detectar si un grupo es válido.
- Obtener la identidad de miembro asociada.

La especificación en OMG/IDL de estos tipos es paralela:

```
interface GroupMember
{
   void processPTPMessage(in GroupMemberId sender, in Message msg);
   void processCastMessage(in GroupMemberId sender, in Message msg);
   void memberAccepted(in GroupMemberId identity, in GroupHandler handler,
                       in View theView);
   void changingView();
   void installView(in View theView);
   void excludedFromGroup();
};
```

```
interface GroupHandler
{
   boolean castMessage(in Message msg);
   boolean sendMessage(in GroupMemberId target, in Message msg);
   boolean leaveGroup();
   GroupMemberId getGroupMemberId();
   boolean isValidGroup();
};
```

8.1.2. Interfaz privada

La parte privada de la interfaz define las operaciones que los miembros del GMS deben soportar para implementar su propio protocolo de comunicaciones, en este caso, basado en el paso de un testigo. La definición principal es la del miembro del GMS, denominada *SenseiGMSMember*, que es una extensión de la definición de *GroupHandler* presentada en la sección anterior. En esta sección, los tipos están especificados empleando únicamente OMG/IDL pues, como se ve en los casos anteriores, la especificación JavaRMI es inmediata.

Las operaciones definidas en esta interfaz están fuertemente ligadas a los algoritmos empleados para mantener el modelo de sincronía virtual. Dos características importantes, desarrolladas en mayor profundidad al explicar el protocolo, son:

- Cada miembro puede comunicarse con los otros cuando posee el testigo. Como el principal problema en este tipo de comunicación es que el testigo aparezca duplicado en el grupo, se define de tal forma que sea identificable sin posibilidad de duplicados. Empleamos su anglicismo, *token*, para nombrarlo:

```
struct Token
{
   GroupMemberId creator;
   long id;
};
```

- En la interfaz pública, los miembros reciben vistas con las identidades de todos los miembros del grupo. Sin embargo, para realizar las comunicaciones, es necesario que se conozcan las referencias a estos miembros, no sólo sus identidades, lo que supone el empleo de vistas internas. Adicionalmente, existe el concepto de vista temporal, como cada una de las vistas que el grupo negocia mientras se discute la composición final del grupo; por esta razón, la identidad de vista interna incluye un campo identificando la vista como temporal:

```
struct InternalViewId
{
   long id;
```

```
      long installing;
   };
   typedef sequence <SenseiGMSMember> SenseiGMSMemberList;
   struct InternalView
   {
      InternalViewId viewId;
      SenseiGMSMemberList members;
      GroupMemberIdList memberIds;
      GroupMemberIdList newMembers;
   };
```

La interfaz *SenseiGMSMember* se define como:

```
interface SenseiGMSMember : GroupHandler
{
   boolean receiveToken(in GroupMemberId sender, in Token theToken);
   boolean receiveView(in GroupMemberId sender, in InternalView view,
                       in Token viewToken);
   boolean receiveMessages(in GroupMemberId sender, in MessageList messages,
                           in MessageId msgId);
   boolean confirmMessages(in GroupMemberId sender, in MessageId msgId);
   boolean receivePTPMessages(in GroupMemberId sender, in MessageList messages,
                              in MessageId msgId);
   AllowTokenRecoveryAnswer allowTokenRecovery(in GroupMemberId sender,
                                               in InternalViewId myViewId);
   boolean recoverToken(in GroupMemberId sender);
   boolean addGroupMember(in SenseiGMSMember other);
   boolean createGroup();
   boolean joinGroup(in SenseiGMSMember group);
};
```

Estas operaciones se describen junto a su significado en la siguiente sección, al explicar el algoritmo empleado. Las tres últimas operaciones son las que permiten crear o extender grupos, tomando como argumento una referencia a un objeto del mismo tipo. Es decir, sólo un miembro que implemente esta interfaz privada puede incluirse en el grupo y, por esa razón, es preciso definir estas operaciones como privadas.

Para que una aplicación pueda crear réplicas sin necesidad de acceder a esta interfaz privada, Sensei incluye una interfaz externa denominada SenseiGMNS que contempla la funcionalidad generalmente disponible en otros sistemas de pertenencia a grupo. GMNS significa *GroupMembershipNamingService* pues, además de incluir la creación y extensión manual de grupos, permite manejarlos de forma simple, asociando un nombre a los grupos. Este servicio, totalmente ligado a la implementación de SenseiGMS, se describe en el capítulo 11.

Todas las operaciones devuelven un valor booleano; si éste es *false*, implica que el miembro destino no acepta la operación efectuada, generalmente porque considere al miembro fuente no perteneciente al grupo. La excepción es la operación *allowTokenRecovery*, que precisa más información y se define como:

```
struct AllowTokenRecoveryAnswer
{
   boolean validCommunication;
   boolean recoveryAllowed;
   MessageId lastMessage;
};
```

8.2. Algoritmo de paso de testigo

SenseiGMS se basa en el paso de un testigo que circula entre los miembros del grupo; la posesión del testigo permite realizar el ordenado de los mensajes de una forma sencilla, y el algoritmo queda sólo complicado por la necesidad de regenerar el testigo si el miembro que lo controlaba se cae, e impedir a la vez que dos testigos circulen simultáneamente en el grupo.

Casi todas las operaciones de grupo que un miembro puede realizar, tales como enviar mensajes, expulsar a miembros sospechosos de haber caído o incluir nuevos miembros, se realizan, exclusivamente, una vez que el miembro entra en posesión del testigo. La única excepción se refiere a las operaciones de recuperación del testigo.

El paso de un testigo no es la única forma de obtener orden total. En Ameba [Kaashoek91], por ejemplo, el orden total se alcanza mediante un proceso especial, denominado secuenciador, al que los demás procesos envían sus mensajes punto a punto; el secuenciador los redistribuye luego a los demás miembros con el orden adecuado. El protocolo *Psync* [Peterson89] crea un orden parcial en los mensajes que puede ser convertido en orden total.

Sin embargo, son más numerosos los sistemas basados en paso de testigo. *Totem* define un algoritmo muy eficiente de paso de testigo que admite, además, el particionado de la red. Al igual que en SenseiGMS, el miembro que posee el testigo es el único que envía mensajes aunque, en otros casos [Chang84], cualquier miembro puede enviar mensajes en cualquier momento, produciendo, sin embargo, altas latencias cuando hay numerosos mensajes en el grupo o ante caídas de miembros. Similar al protocolo de *Totem* es el protocolo *TPM* [Rajagopalan89], aunque no admite el particionado de los grupos. Estos protocolos se basan en el soporte de mensajes multicast *best-effort*, principalmente UDP, mientras que SenseiGMS se construye sobre las comunicaciones punto a punto de CORBA o RMI, lo que ha provocado la necesidad de desarrollar un algoritmo propio.

8.2.1. Paso de testigo y estructura en anillo

Los miembros pueden considerarse lógicamente dispuestos en una estructura de anillo, donde la vista define los componentes y su respectivo orden. Este anillo es dinámico, expandiéndose según nuevos miembros se insertan en el grupo o los antiguos lo abandonan o son excluidos. La implementación actual del algoritmo inserta los miembros lógicamente tras el miembro que los introduce en el grupo.

Un testigo no es, en realidad, mas que un tipo especial de mensaje transferido punto a punto. Cuando una nueva vista temporal se crea, se define un nuevo testigo con una identidad única, que el miembro que envía la vista propaga junto con esa vista. Este testigo circula, a continuación, en el grupo y los miembros deben verificar su identidad. Dos problemas pueden aparecer con este paso de testigo. El primer problema es que el miembro que posee el testigo se caiga, con lo que el testigo se pierde. Todos los miembros disponen de un temporizador que se reinicia cada vez que el miembro recibe una comunicación. Si el temporizador vence, el miembro trata de regenerar el testigo, según se detalla en el apartado posterior de recuperación del testigo. El segundo problema viene asociado a éste y consiste en que el miembro considerado caído fuera simplemente muy lento y otro genere un nuevo testigo, terminando el grupo con varios testigos activos. Este punto se soluciona con que cada miembro admita sólo un testigo como válido, con lo que no es posible que un mismo miembro transmita dos o más testigos diferentes. Esta solución conlleva que el grupo puede particionarse, pero no se compromete la integridad de sus comunicaciones.

Este paso de testigo, realizado a través de la operación *receiveToken*, implica que el grupo está realizando comunicaciones continuamente, incluso cuando la aplicación está inactiva sin enviar mensajes. El sistema puede detectar esta inactividad y retrasar el envío del testigo entre miembros en estos casos. En cualquier caso, otros sistemas no basados en paso de testigo deben realizar también comunicaciones periódicas para detectar caídas de miembros.

8.2.2. Detección de errores

La detección de errores se realiza a partir de las comunicaciones normales: envío de mensajes y vistas, paso de testigo, etc. Si un miembro no puede acceder a otro, lo considera caído. Adicionalmente, el miembro contactado debe responder en un tiempo límite, configurable según sea la calidad de la red.

Puesto que el paso del testigo se realiza continuamente, la caída del miembro se detectará de forma razonablemente rápida, salvo que el miembro que posee el testigo sea el que se haya caído. Por esta razón, todo miembro espera recibir alguna comunicación del grupo en un tiempo dado y si no se produce, inicia el protocolo de regeneración del testigo.

Problemas en la red o miembros lentos pueden suponer detecciones de errores erróneas. Sin embargo, una vez que un miembro considera a otro erróneo tal decisión es permanente, y lo comunica a los demás miembros del grupo que pasan a considerarlo erróneo igualmente. En estas condiciones, rechazan cualquier comunicación proveniente de este miembro falsamente erróneo.

A la inversa, si un miembro accede a otro y este último rechaza la comunicación, el primero le considera erróneo igualmente y se lo comunica a los demás. Esta situación puede suponer la división del grupo, pero sólo uno, o incluso ninguno, de los nuevos grupos tendrá consenso y sobrevivirá, puesto que SenseiGMS no soporta particionados de la red. Los sistemas de sincronía virtual extendida permiten que varios subgrupos permanezcan activos tras su división por problemas de comunicaciones en la red; si esta funcionalidad no se soporta, debe asegurarse que sólo uno de los grupos, como máximo, permanezca activo. Este objetivo se logra imponiendo un requisito sobre las vistas: una vista temporal sólo se acepta si contiene, al menos, la mayoría de los miembros que se incluían en la anterior vista *permanente*. Esta implementación implica que un grupo de dos miembros donde uno realmente cae, perderá el consenso y se volverá inoperativo; por esta razón, SenseiGMS permite definir grupos como pertenecientes a una red fiable, relajando, en este caso, el anterior requisito y permitiendo que el consenso se mantenga con vistas conteniendo sólo la mitad de los miembros de la anterior vista permanente. En estos casos, decimos que SenseiGMS trabaja con *consenso débil*.

8.2.3. Envío de mensajes

El modelo de sincronía virtual no prescribe el soporte de mensajes punto a punto entre los miembros de un grupo; su soporte bajo un algoritmo de paso de testigo es sin embargo sencillo, incluso respetando el orden entre mensajes punto a punto y mensajes al grupo. Es además importante: aunque bajo CORBA o JavaRMI cada componente puede realizar comunicaciones punto a punto con otros componentes, la información incluida en la vista no contiene las referencias a otros miembros, sino sus identidades, una abstracción que sólo tiene significado bajo SenseiGMS.

Un miembro encola todos los mensajes a enviar, que procesa cuando recibe el testigo. Bajo condiciones normales, cuando recibe ese testigo, itera sobre la cola de mensajes, enviándolos al miembro especificado, o a todos los miembros si es un mensaje multipunto. Si tiene varios mensajes multipunto, los puede agrupar en un sólo envío, minimizando el número de comunicaciones y delegando en el *middleware* la división del mensaje en submensajes si aquél resulta demasiado grande. Para el envío de mensajes, la interfaz *GroupHandler* incluye tres operaciones:

- *receiveMessages*: recibe conjuntos de mensajes multipunto.
- *receivePTPMessages*: la operación equivalente para recibir mensajes punto a punto.

- *confirmMessages*: completa el envío de los mensajes, que se realiza en dos pasos: el miembro envía el mensaje y, a continuación, confirma que se puede procesar. Como un miembro puede enviar varios mensajes, el envío de un segundo mensaje directamente confirma el primero y sólo es necesario realizar una confirmación de mensajes en el último paso, antes de liberar el testigo. Los miembros del grupo procesan los mensajes multipunto sólo cuando reciben esta confirmación, mientras que los mensajes punto a punto no necesitan, evidentemente, una confirmación.

Si el miembro se cae antes de enviar todos los mensajes, el algoritmo de recuperación del testigo debe verificar los mensajes procesados por los miembros activos, de tal forma que esos mensajes sean procesados por los demás miembros. Cada mensaje incluye, por lo tanto, una identidad de mensaje, con la vista en que se envía, y un número secuencial que se reinicia en cada nueva vista. La recepción de un mensaje con mayor número secuencial confirma automáticamente al anterior. Los mensajes deben, entonces, encolarse cuando se reciben y cada miembro almacena la identidad del último mensaje procesado, que es utilizado en el algoritmo de recuperación del testigo. Esta identidad de mensaje, que no se mostró con la interfaz privada, es en OMG/IDL:

```
struct MessageId
{
   long view;
   long id;
};
```

Un miembro que recibe un mensaje punto a punto de otro miembro da automáticamente por confirmados sus anteriores mensajes multipunto; sin embargo, el mensaje punto a punto no puede incrementar su identidad de mensaje (*id*, el número secuencial), ya que no lo reciben todos los miembros. Por ello, en la identidad de los mensajes punto a punto, sólo tiene significado la identidad de la vista y no el número secuencial asociado.

8.2.4. Manejo de vistas

Todos los miembros del grupo deben recibir las mismas vistas. Éstas pueden ser temporales, cuando los miembros discuten su contenido, o permanentes. La aplicación sólo recibe las permanentes y se garantiza que todos los miembros de la aplicación reciben las mismas vistas. Sin embargo, no es necesario que todos los miembros reciban las mismas vistas temporales.

Las vistas se envían a través de la operación *receiveView* en la interfaz *SenseiGMSMember*. Una vista es enviada cuando un miembro desea cambiar la composición del grupo, bien sea por la inserción de nuevos miembros, o tras sospechar que un miembro ha caído y debe ser expulsado. Cada vista viene

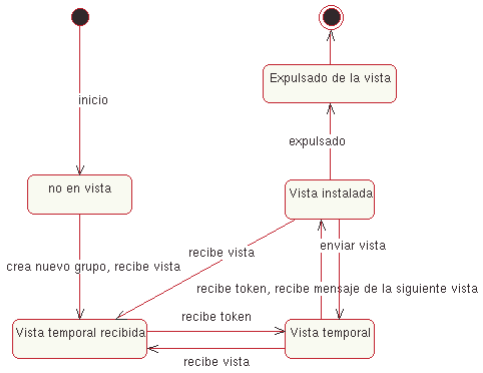

Figura 8.1. Estados de *vistas* en SenseiGMS

identificada por una identidad de vista, e incluye la lista ordenada de los demás miembros del grupo y sus identidades, así como las de los miembros que son nuevos en el grupo (identidades que no son luego propagadas con las vistas públicas). Esta lista es ordenada, pero no se puede asegurar que el primer miembro sea el más antiguo del grupo.

Siguiendo el modelo de sincronía virtual, los mensajes deben ser procesados por todos los miembros en la misma vista. Los cambios de vistas bloquean, por lo tanto, este envío. Cada miembro válido puede estar en tres estados: *vista instalada*, *vista temporal*, y *vista temporal recibida*. Cuando un miembro en estado de *vista instalada* envía una nueva vista, debe primero enviar todos los mensajes y bloquear el envío de nuevos mensajes, antes de pasar a estado de *vista temporal*. En este estado, cuando la aplicación solicita el envío de un mensaje, se le notifica que no será enviado hasta la siguiente vista.

Las vistas se instalan en dos pasos; un miembro envía su vista, que siempre presenta una identidad temporal y, a continuación, pasa el testigo al siguiente miembro de la vista. Todos los miembros recibirán este testigo y podrán cambiar el contenido de la vista; en este caso, serán nuevas vistas temporales. Si el miembro que envía la vista recibe el testigo sin que la vista haya cambiado, considerará que la vista es válida y la propagará como permanente a la aplicación.

Cuando un miembro recibe una vista, pasa a estado de *vista temporal recibida*. Cuando reciben el testigo en la siguiente vuelta, tienen una oportunidad para enviar sus mensajes y, a continuación, pasan a estado de *vista temporal*, bloqueando igualmente los mensajes. Un miembro en *vista temporal* pasa a *vista instalada*

cuando recibe el testigo en la siguiente vuelta; si, estando en *vista temporal*, recibe una vista de otro miembro, pasa a *vista temporal instalada* sin desbloquear los mensajes. El diagrama de estados en la figura 8.1 visualiza este algoritmo.

Hay un caso especial: un miembro en estado de *vista temporal* que recibe un mensaje perteneciente a una vista posterior. Este caso implica que otro miembro en el anillo ha pasado ya a estado de *vista instalada* y envía nuevos mensajes; como consecuencia, pasa, automáticamente, a estado de *vista instalada* y lo comunica a la aplicación.

8.2.5. Gestión de grupos

Un grupo se crea a partir de un único miembro, que crea un testigo inicial que se envía a sí mismo. Esta aproximación se puede optimizar para evitar el paso del testigo cuando hay un solo miembro en el grupo. Para extender el grupo, el potencial miembro debe contactar a alguno de los miembros válidos, y solicitar su inserción. Este miembro encolará esa solicitud y, cuando reciba el testigo, considerará si el estado del grupo permite la inserción de nuevos elementos. Esta consideración se tiene en cuenta a continuación, al explicar los cambios de vistas.

Los siguientes métodos de la interfaz *SenseiGMSMember* son necesarios para cubrir esta funcionalidad:

- *createGroup*.
- *addGroupMember*. Incluye eventualmente al miembro especificado como parámetro de la operación en el grupo propio.
- *joinGroup*. Es un método redundante, definido en función de *addGroupMember*, que bloquea la llamada hasta que se completa la inserción en el grupo.

Tras recibir una solicitud de inclusión en el grupo, el miembro contactado añadirá, eventualmente, a ese miembro o nuevos miembros, creando una nueva vista que envía al grupo que incluye a esos nuevos miembros; éstos, al recibir esta vista, pueden ya considerarse aceptados en el grupo. Este proceso puede fallar si el miembro contactado se cae o pierde el consenso, en cuyo caso, los nuevos miembros deberán repetir la solicitud a otro miembro. La obtención de referencias a miembros existentes no queda cubierta en la definición del algoritmo.

Para abandonar voluntariamente un grupo, un miembro debe enviar, cuando recibe el testigo, una vista que no le incluya y, a continuación, liberar el testigo. Un miembro que abandona de esta forma el grupo afecta a la forma en que se resuelve el consenso, excluyéndolo de la anterior vista permanente al contabilizar la mayoría. Por ejemplo, en condiciones normales, un grupo de dos miembros donde uno cae pierde el consenso, pero esto no ocurre si el miembro solicita su exclusión.

Este protocolo no soporta el concepto de grupos *exclusivos*: cualquier instancia del tipo *SenseiGMSMember* puede solicitar su inclusión en otro grupo si obtiene una

referencia válida y ninguno de los miembros existentes puede vetar tal inclusión. Desde el punto de vista de seguridad, implica, por ejemplo, que un miembro podría ser incluido y enviar entonces vistas erróneas diferentes a distintos miembros del grupo, provocando que éste se particione en grupos sin consenso. Esta seguridad puede, sin embargo, incluirse en una capa superior. Por ejemplo, empleando JavaRMI, es posible especificar una factoría propia de *sockets* que emplee sockets seguros o valide todo inicio de comunicaciones: sin afectar a SenseiGMS, la aplicación ha superpuesto un nivel de seguridad en las comunicaciones.

8.2.6. Protocolo de recuperación del testigo

Un miembro que no recibe ninguna comunicación tras un determinado periodo de tiempo considera que el testigo se ha perdido e inicia el protocolo de recuperación de aquél. Este protocolo se realiza en tres fases. En la primera fase, el miembro contacta a los demás miembros a través de la operación *allowTokenRecovery*, solicitándoles su permiso para recuperar el testigo. Si alguno de los miembros responde negativamente, este miembro aborta la recuperación del testigo.

Un miembro responde negativamente por dos posibles razones: porque posea el testigo, o porque tenga prioridad para recuperarlo. La segunda razón busca evitar que dos miembros recuperen simultáneamente el testigo. Un miembro considera que tiene prioridad sobre otro si su identidad de vista (temporal/permanente) es superior a la del otro miembro y, cuando es igual, si su identidad de miembro de grupo es superior a la del otro miembro; de esta forma, si un miembro ha enviado su propuesta de vista a un subconjunto del grupo y se cae, ese subconjunto tiene prioridad para instalar una nueva vista. La solicitud de recuperación del testigo incluye, por lo tanto, la identidad de vista del miembro que envía la solicitud. Al responder, los miembros incluyen, igualmente, la identidad del último mensaje procesado, de tal manera que sea posible conocer la mayor identidad de mensaje procesado por algún miembro en el grupo.

En la segunda fase, el miembro considera que puede recuperar el testigo, y envía su decisión a los demás miembros, empleando la operación *recoverToken*. Cuando estos miembros reciben esta señal, consideran que el testigo se ha perdido y que está siendo regenerado por el miembro dado. El grupo no aceptará entonces ninguna nueva comunicación del grupo (paso de testigo, o mensajes, etc.), aceptando únicamente otra comunicación *recoverToken* o un cambio de vista proveniente del miembro que está recuperando el testigo. De esta manera, se impide que el anterior testigo cause problemas si el miembro que lo poseía era muy lento o simplemente había un problema en las comunicaciones de red. Se impide también que dos miembros puedan recuperar el testigo simultáneamente, también por problemas de comunicaciones. Si fuera éste el caso, el grupo se dividiría entre los miembros que han recibido las solicitudes de recuperación de cada miembro; eventualmente se puede provocar que el grupo se vuelva inoperativo, pero impide la

inconsistencia de que dos miembros consideren que poseen el testigo y envíen simultáneamente comunicaciones contradictorias al grupo.

En la tercera fase, el miembro envía la vista a los miembros. Esta vista excluye a los miembros que no pudo contactar en la primera o segunda fase y a los miembros que respondieron negativamente al evento de recuperación del testigo. Si tras estas exclusiones el miembro ha perdido el consenso, se considera automáticamente excluido.

Un miembro que es expulsado del grupo cuando no tiene el testigo ejecutará eventualmente este protocolo, descubriendo que no tiene consenso para continuar operativo, enviando entonces el evento de exclusión de grupo a la aplicación. Si el miembro es expulsado cuando tiene el testigo, fracasará en su intento de transferirlo y, de nuevo, al intentar instalar una vista, detectando entonces su pérdida de consenso igualmente.

8.3. Uso de SenseiGMS

El siguiente capítulo, dedicado a la metodología de diseño de grupos de objetos replicados, se centra en los problemas generales al diseñar aplicaciones tolerantes a fallos y muestra métodos genéricos y las herramientas necesarias para soportar esa metodología. Esta sección se limita a mostrar un ejemplo particular sobre SenseiGMS, obviando, al menos, un problema de uso que con las herramientas mostradas en el siguiente capítulo tendría una fácil solución.

8.3.1. Diseño de un servicio replicado

La sección dedicada al diseño de SenseiGMS mostró su interfaz pública y las directrices generales para emplear replicación de objetos: definir los mensajes que se emplearán en las comunicaciones del grupo, implementar la interfaz *GroupMember* y obtener una referencia a un objeto *GroupHandler*, ya sea creando un nuevo grupo o incluyéndose en uno existente; mediante esta referencia se accede al grupo, y es preciso procesar los mensajes provenientes de éste.

El ejemplo a implementar es un caso real: un servicio de directorio de miembros de grupos, de tal forma que nuevos miembros puedan obtener fácilmente referencias a otros miembros ya existentes en un determinado grupo, que se nombra mediante una cadena de caracteres. Este es uno de los servicios existentes en Sensei, denominado SenseiGMNS, con una interfaz más complicada para ser funcional a los clientes del grupo y no sólo a sus miembros.

Para implementar este servicio, empleamos una estructura de datos denominada *GMNSdata*, que contiene una lista de todos los grupos existentes y, para cada grupo, la lista de miembros asociados. Si la máquina que contiene esta

estructura se cae, el servicio se vuelve inaccesible, por lo que la implementación debe ser tolerante a fallos: es un componente replicado.

La definición de este componente es, en OMG/IDL:

```
interface GMNSdata : GroupMember
{
   void insert (in string groupName, in GroupHandler member, in string memberName);
   void remove (in string groupName, in GroupHandler member);
   GroupHandler get (in string groupName);
};
```

Las operaciones se definen como:

- *insert*: incluye un miembro en el grupo dado, creando el grupo si es necesario. Cada miembro tiene un nombre, solamente útil para su visualización.
- *remove*: elimina un miembro del grupo y el grupo mismo si se queda vacío.
- *get*: devuelve un elemento cualquiera de un grupo.

Un miembro que desea incluirse en un determinado grupo, accede a este componente[3] mediante la operación *get* para obtener referencias a otros miembros del grupo. Si no hay referencias, crea el grupo e introduce en este componente su referencia con la operación *insert*. Si hay otros miembros, *get* devuelve uno de esos miembros, al que solicita su inclusión en el grupo, insertando finalmente en este componente su referencia. Sin embargo, hay problemas de concurrencia: puede darse el caso de dos futuros miembros de un mismo grupo, todavía no existente, que accedan a este componente, observando ambos que el grupo no existe y creando entonces dos grupos independientes, para insertar, a continuación, sus referencias bajo el mismo nombre de grupo. Para este ejemplo, obviamos este problema, que se trata en los siguientes capítulos.

Si un componente *GMNSdata* recibe una solicitud *insert* o *remove*, debe comunicárselo a las demás réplicas *GMNSdata* de tal forma que todo el grupo mantenga el mismo estado. Si la solicitud es *get*, el componente puede dar esa información sin realizar ninguna comunicación al grupo. Definimos dos mensajes:

```
valuetype GMNSdataInsertMessage :
          Message
{
   public string groupName;
   public string memberName;
   public GroupHandler member;
};
```

```
valuetype GMNSdataRemoveMessage :
    Message
{
   public string groupName;
   public GroupHandler member;
};
```

[3] Tal acceso no es directo, un componente intermedio, el GMNS, es el encargado de acceder ordenadamente al *GMNSdata*, evitando colisiones bajo múltiples solicitudes.

Cada mensaje incluye la información necesaria para poder ejecutar en cada réplica el mismo código, lo que, en general, supone incluir como atributos cada uno de los parámetros definidos en la operación asociada.

Desde un punto de vista formal, no es preciso que *GMNSdata* herede de *GroupMember*: es suficiente que se defina un componente que implemente ambas interfaces.

8.3.2. Implementación

La implementación es diferente si se emplea JavaRMI o CORBA y, en este último caso, varía según el lenguaje de implementación escogido y el adaptador de objetos empleado. En el caso de emplear Java bajo CORBA, usando el *Portable Object Adapter*, una implementación posible es crear una clase que herede de *GMNSdataPOA*, que es generada automáticamente por el compilador de OMG/IDL. Con RMI, una posibilidad es crear una clase que herede de *java.rmi.server.UnicastRemoteObject* e implemente la interfaz *GMNSdata*. En ambos casos, el algoritmo es el mismo y, puesto que en los dos casos se emplea Java, el código que implementa la funcionalidad *GMNSdata* es común para CORBA y RMI. Para no entrar en detalles innecesarios sobre este ejemplo concreto, suponemos que existe una estructura de datos interna capaz de manejar la lógica del problema (sin replicación) y con las mismas operaciones definidas en la interfaz *GMNSdata* (sin incluir las operaciones de *GroupMember*). La instancia de esta estructura la denominamos *internalGMNSData*.

La figura 8.2 muestra la idea básica para el procesado de mensajes en este ejemplo: al recibir una solicitud de servicio, la réplica no la procesa inmediatamente, sino que envía un mensaje al grupo con la información sobre la operación a realizar. El miembro que envía el mensaje, también lo recibe, en el orden adecuado, y es entonces cuando procesa la operación. La ventaja de esta aproximación es que hay un único punto de procesamiento, tanto para las operaciones directamente invocadas en esta réplica, como para los mensajes provenientes de esas mismas operaciones al ser invocadas sobre otras réplicas del grupo.

Este ejemplo es un caso simplificado pues, normalmente, el servidor debe devolver algún valor al cliente y es entonces necesario sincronizar de alguna manera la invocación de la operación con su procesamiento, como detallamos en el siguiente capítulo. Incluso en este simple ejemplo, no puede garantizarse que la siguiente porción de código se ejecute satisfactoriamente:

```
map.insert(“Grupo A”, specificGroupHandler, “Miembro 1”);
if (map.get(“Grupo A”)==null)
{
   //Error, el grupo no tiene miembros tras haber insertado uno?
}
```

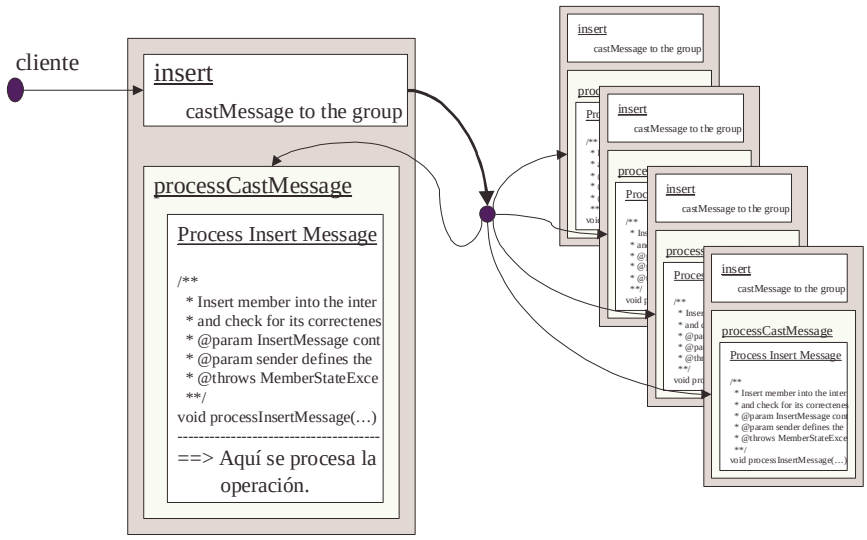

Figura 8.2. Tratamiento de mensajes empleando orden total.

El motivo de este error es que cuando la operación *insert* vuelve, no ha sido aún procesada, por lo que el valor devuelto por la operación *get* puede indicar que el grupo dado no tiene ningún miembro. Y si la operación se sincroniza, la principal desventaja es el mayor tiempo de proceso que el cliente experimenta. En la siguiente lista aparecen en cursiva los pasos que deben ser ejecutados en adición al proceso normal, si el servidor no estuviera replicado:

- Recibir la petición del cliente.
- *Crear un mensaje con la información de la operación a procesar.*
- *Enviar este mensaje al grupo.* Lo cual incluye enviar el mensaje a cada una de las réplicas garantizando que todas obtienen una copia. En el peor caso, implica enviar a cada réplica tres mensajes.
- *Recibir el mensaje.*
- Procesar la operación solicitada.
- *Sincronizar este procesamiento con la operación invocada para poder devolver al cliente los resultados de esa operación.*
- Devolver al cliente los resultados de la operación, si fuera necesario.

Estos pasos adicionales deben ejecutarse en cualquier caso en un componente replicado activamente, pero el cliente no experimenta estos mayores tiempos de procesamiento si el sistema se programa de tal forma que los mensajes sean dinámicamente no uniformes. En ese caso, cuando un cliente invoca la operación *insert*, el servidor la procesa inmediatamente y envía a continuación (o incluso

concurrentemente) el mensaje al grupo, pero el cliente observa el mismo tiempo de procesamiento que si el servidor no estuviera replicado. Obviamente, el servidor tiene ahora una mayor carga de trabajo, por lo que no será capaz de soportar el mismo número de operaciones por unidad de tiempo.

Emplear mensajes dinámicamente no uniformes implica normalmente cambiar los algoritmos de los grupos, pues no todas las réplicas procesan los mismos mensajes en el mismo orden: la réplica que envía el mensaje no lo procesa. Por ejemplo, al programar una estructura de datos que implemente un mapa replicado, estos mensajes producirán, normalmente, inconsistencias en el grupo: si dos miembros insertan concurrentemente dos valores distintos para una misma clave, las réplicas terminarán con estados diferentes. Esta restricción no se da, sin embargo, en el caso de *GMNSdata*, donde el valor de cada entrada del mapa es, a su vez, un conjunto de valores sin requerimientos de orden.

El siguiente código implementa un servidor replicado *GMNSdata* usando SenseiGMS bajo JavaRMI y empleando mensajes dinámicamente uniformes:

```
public class GMNSdataImpl extends UnicastRemoteObject implements GMNSdata
{
   public GMNSdataImpl throws RemoteException {}

   public void insert(String groupName, GroupHandler member, String memberName)
   {
      handler.castMessage(new GMNSdataInsertMessage(groupName, member, memberName));
   }

   public void remove(String groupName, GroupHandler member)
   {
      handler.castMessage(new GMNSremoveMessage(groupName, member));
   }

   public GroupHandler get(String groupName)
   {
      return internalGMNSdata.get(groupName);
   }

   public void processCastMessage(int snd, Message msg)
   {
      if (msg instanceof GMNSdataInsertMessage)
      {
         GMNSdataInsertMessage iMsg = (GMNSdataInsertMessage) msg;
        internalGMNSdata.insert(iMsg.groupName, iMsg.member, iMsg.memberName);
      }
      else if (msg instanceof GMNSdataRemoveMessage)
      {
```

```
            GMNSdataRemoveMessage rMsg = (GMNSdataRemoveMessage) msg;
            internalGMNSdata.remove(rMsg.groupName, rMsg.member);
         }
      }

      public void changingView() {}

      public void void installView(View view) {}

      public void processPTPMessage(int snd, Message msg) {}

      public void memberAccepted(int id, GroupHandler handler, View view)
      {
         this.handler = handler;
      }

       public void excludedFromGroup()
      {
         System.exit(0);
      }

      GroupHandler handler;
   }
```

- Este servidor no necesita información sobre las demás réplicas, luego no procesa las vistas que recibe, y no necesita código en las operaciones *installView* o *changingView*. Cuando el miembro es aceptado en el grupo, recibe mediante *memberAccepted* una instancia del tipo *GroupHandler*, que guarda para utilizar posteriormente en sus interacciones con el grupo.
- El algoritmo no precisa de mensajes punto a punto, luego tampoco es necesario ningún código en *processPTPMessage*. Al recibir mensajes del grupo en *processCastMessage* debe primero discernir el tipo de mensaje recibido, procesándolos acordemente. Si el mensaje es *GMNSdataInsertMessage*, lee la información del mensaje e invoca esa operación sobre la estructura de datos interna, tratando de forma similar el mensaje *GMNSdataRemoveMessage*.
- Las operaciones *insert* y *remove* se tratan creando los mensajes correspondientes y enviándolos al grupo a través de la instancia *GroupHandler* recibida cuando el miembro fue aceptado.
- La operación *get* es procesada inmediatamente, empleando la estructura de datos interna.

El código mostrado no trata errores (la instancia de *GroupHandler* es remota, luego su acceso puede siempre provocar excepciones) ni realiza transferencia de estado, pero muestra las características fundamentales de SenseiGMS: orientación a

objetos, empleo de mensajes tipados, comunicaciones en grupo fiables con orden total. Es posible llevar la orientación a objetos más lejos empleando herramientas genéricas que definan los mensajes automáticamente a partir de una interfaz dada y transformen la recepción de mensajes en llamadas a procedimientos específicos según el mensaje recibido; este tema se trata también en el siguiente capítulo.

Tampoco muestra este código cómo se crea o se extiende el grupo: si la implementación se realiza totalmente en Java, la clase *ActiveGroupMember* en el paquete *sensei.GMS* implementa la interfaz *SenseiGMSMember*, necesaria para la creación o extensión de grupos. El primer miembro crea una instancia de esta clase e invoca la operación *createGroup*. El segundo y posteriores miembros deben obtener de alguna manera una referencia a este miembro, por ejemplo, accediendo a un fichero donde esa referencia se haya guardado. Estos miembros crean de la misma forma una instancia de la clase *ActiveGroupMember* e invocan la operación *joinGroup*, pasando la referencia del primer miembro del grupo. Este proceso se realiza automáticamente y sin necesidad de discernir entre 'primero' o 'posterior' miembro del grupo empleando SenseiGMNS, detallado en el capítulo 11.

8.3.3. Configuración

El algoritmo emplea varias variables que pueden cambiar el comportamiento del sistema, permitiendo así su optimización en función del entorno de trabajo. Estas propiedades son:

- GMS.*channelLiveness*: periodo en milisegundos en que un canal de comunicación espera una respuesta. Transcurrido este tiempo, el miembro correspondiente se considera caído. Por defecto son 10 segundos.
- GMS.*maxProcessMaxDelay*: periodo en milisegundos que define el retraso máximo en que la aplicación puede procesar un mensaje dado. Transcurrido este tiempo, que por defecto es de 5 segundos, Sensei considerará que la aplicación es muy lenta y excluirá al correspondiente miembro del grupo.
- GMS.*tokenStoppedPeriod*: retención del testigo en caso de inactividad en el sistema. Cuando el sistema no realiza ninguna comunicación, cada miembro retiene el testigo el tiempo dado por esta variable, 50 milisegundos por defecto.
- GMS.*tokenLostTimeout*: periodo en milisegundos que tarda un miembro que no recibe ninguna comunicación en iniciar el algoritmo de recuperación del testigo. Por defecto, es de 6 segundos.
- GMS.*weakConsensus*: si esta variable booleana se define *a true*, se emplea un modelo de consenso débil, donde es suficiente que la mitad de los miembros de una vista sobrevivan para que el grupo conserve el consenso.

Estas propiedades se definen en un fichero específico, aunque cada grupo puede definir configuraciones distintas o, simplemente, redefinir alguno de los

anteriores valores. No es necesario tampoco que todos los miembros de un mismo grupo definan la misma configuración.

8.4. VirtualNet

SenseiGMS se ha probado en profundidad, comprobando situaciones límite mediante el empleo de instancias de *SenseiGMSMember* con una implementación deliberadamente errónea del algoritmo expuesto. Estas alteraciones del algoritmo incluyen:

- Los mensajes no son enviados a todos los miembros en el grupo.
- La vista no se envía a todos los miembros.
- La confirmación de los mensajes no se envía a todos los miembros.
- Los mensajes no se envían hasta que se recibe un cambio de vista.
- Los mensajes no se envían hasta la siguiente vista.
- Se retrasa la inserción de los miembros que lo solicitan hasta que se reciba un cambio de vista.
- El paso del testigo se realiza lentamente.
- Miembros se caen en situaciones comprometidas.

Comprobar el comportamiento de Sensei en condiciones reales es una tarea más complicada, al precisar de entornos muy variados. Por ejemplo, comprobar un enlace más lento, o la caída de uno o más enlaces, es difícil de realizar, incluso cuando sea posible realizar la desconexión manual de esos enlaces físicos. Un entorno real puede involucrar ordenadores físicamente muy distantes, conectados por Internet a través de un número desconocido de nodos, donde estos nodos pueden caerse o donde algunas conexiones pueden tener grandes fluctuaciones de rendimiento. Para solucionar este problema, hemos desarrollado una aplicación independiente de Sensei denominada *VirtualNet*.

La filosofía de VirtualNet es simple: simular una red en un solo ordenador, donde se pueden definir nodos y enlaces que pueden alterarse en cualquier momento, ya sea definiendo diferentes rendimientos o desactivando temporal o permanente esos nodos o enlaces. Cada miembro debe asociarse a un nodo, y sus comunicaciones con otros miembros suponen que debe existir una ruta virtual entre sus nodos.

La versión inicial de VirtualNet sólo funciona con JavaRMI, empleando un compilador propio para generar los *stubs* y *skeletons*, que verifican esa ruta virtual antes y después de acceder al objeto remoto. La forma de asociarse a un nodo es empleando el servicio de nombrado de RMI (*java.rmi.Naming*), que se redefine para VirtualNet. Así, en lugar de emplear *java.rmi.Naming*, se emplea *vnet.remote.Host*,

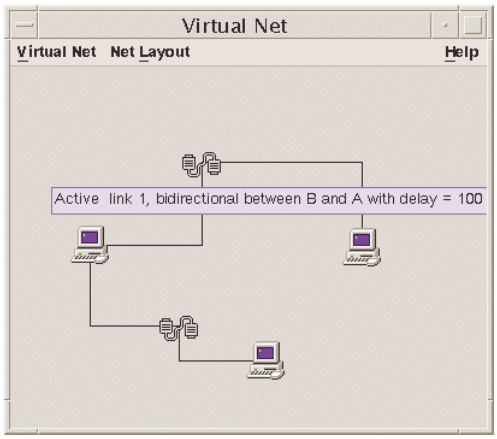

Figura 8.3. VirtualNet

que define exactamente la mismo interfaz, facilitando la prueba de las aplicaciones. Sólo debe cambiarse el nombre con que se publica la aplicación, que debe ser del tipo: `/virtualNetName/Host/Server` (nombre de la red virtual creada, máquina a emplear, nombre del servidor). Por ejemplo, `/redMadrid/MaquinaUCM/GMNS`.

Una aplicación no necesita, sin embargo, emplear el registro de RMI para operar. Este registro es sólo una forma cómoda de obtener referencias a otros objetos remotos existentes. De hecho, Sensei no emplea el registro en ningún momento. Por esta razón, se ha desarrollado una nueva versión que, por el contrario, funciona de momento sólo con CORBA. VirtualNet contiene ahora la lógica para comprobar las rutas virtuales entre dos nodos cualesquiera, nodos a los que se les puede asignar cualquier nombre. Puede soportar múltiples redes virtuales, cada una de las cuales se instala en un determinado puerto. La aplicación final debe conectarse a este puerto, indicando el nombre de la máquina en que reside y, a continuación, puede consultar indefinidamente si hay rutas virtuales a otras máquinas. VirtualNet dispone de una interfaz gráfica que permite visualizar dinámicamente el estado de la red virtual, tal como muestra la figura 8.3.

Los enlaces pueden ser monodireccionales o bidireccionales y, tanto los nodos, como los enlaces, pueden tener un retraso asociado que se procesa en cada chequeo de ruta. Es decir, al contactarse VirtualNet para chequear si una ruta es válida o no, una respuesta positiva se efectuará en el tiempo necesario para recorrer virtualmente cada uno de los nodos y enlaces en la ruta calculada. Pueden realizarse varios chequeos simultáneamente.

El protocolo de acceso a VirtualNet es muy simple y una aplicación en cualquier lenguaje puede emplearlo para comprobar las rutas. Sin embargo, es necesario que estos accesos sean lo más transparentes posible para que el empleo de esta herramienta sea práctico. Bajo CORBA es posible implementar *interceptors*, cuyo objetivo es interceptar cualquier acceso remoto. VirtualNet incluye interceptores que acceden a la red virtual en cada acceso y emiten excepciones *org.omg.CORBA.COMM_FAILURE* cuando la ruta no es válida. Para ello, añaden en cada comunicación el nombre de la máquina que realiza la comunicación y, cuando el servidor recibe esa comunicación, puede chequear la ruta. De la misma forma, cuando el servidor devuelve la respuesta al cliente, éste puede volver a comprobar la ruta.

Para que una aplicación emplee VirtualNet debe incluir la siguiente línea de código al principio del programa, antes de iniciar el ORB[4]:

```
args=vnet2.OOCInterceptor.setup(null, args, props);
```

El primer parámetro indica el receptor de los mensajes de error que, por defecto, es el dispositivo normal de salida. El segundo parámetro contiene los argumentos de la línea de comando y, el tercero, las propiedades con que se inicializará posteriormente el ORB. Los argumentos asociados a VirtualNet son eliminados para no interferir con la lógica de la aplicación. Estos argumentos requeridos son:

- *-virtualNetHost host*: indica la máquina donde se ejecuta el programa de VirtualNet que, por defecto, es "*localhost*".
- *-virtualNetPort port*: indica el puerto donde VirtualNet espera conexiones.
- *-virtualHostName name*: indica el nombre de la máquina virtual, que deberá estar definido en la red virtual asociada.

Si los dos últimos argumentos están presentes, el interceptor se instala como cliente y como servidor, y se emplea entonces activamente la red virtual.

8.5. Conclusiones

SenseiGMS es un sistema de comunicaciones fiables en grupo que implementa sobre CORBA y JavaRMI una interfaz de programación de aplicaciones similar al soportado en otros sistemas de comunicaciones en grupo. La funcionalidad implementada se reduce a la mínima común con esos sistemas de comunicaciones, de tal forma que la implementación posterior, tanto de los algoritmos de

[4] En este ejemplo se está empleando el interceptor OOC, específico para *Orbacus*.

transferencia de estado expuestos, como de las herramientas de apoyo al desarrollo de aplicaciones replicadas mostradas en los siguientes capítulos, pueda reimplementarse sin cambios en cualquier sistema de comunicaciones en grupo.

La necesidad de SenseiGMS se ha basado en la no disponibilidad de sistemas similares sobre CORBA o JavaRMI. La especificación del servicio de tolerancia a fallos de CORBA define, para la replicación activa, un grupo de comunicaciones en sincronía virtual situado lógicamente por debajo del ORB, con el objetivo de lograr un mejor rendimiento. SenseiGMS implementa el sistema de comunicaciones *sobre* el ORB, degradando su rendimiento en beneficio de una simplificación en su implementación. Cuando esté disponible una implementación compatible con la actual especificación, confiamos en la fácil migración de la interfaz de SenseiGMS sobre esa implementación, teniendo en cuenta la mínima funcionalidad soportada.

SenseiGMS define, sin embargo, dos facilidades adicionales sobre otros sistemas de comunicaciones en grupo tradicionales: orientación a objetos y soporte de mensajes tipados. Estas dos facilidades no resultan, por otro lado, difíciles de implementar sobre esos sistemas.

El objetivo de simplificar la implementación de SenseiGMS afecta, también, a su soporte de orden en los mensajes fiables, admitiendo sólo orden total. Este orden se obtiene mediante un algoritmo de paso de testigo, diseñado sobre mensajes fiables punto a punto. Las aplicaciones replicadas se programan más fácilmente empleando orden total que causal o *fifo*, y el diseño de herramientas genéricas está también basado en ese orden, lo que garantiza que esta simplificación, específica en la funcionalidad de SenseiGMS, no afecta al dominio de aplicaciones que soporta.

Probar la corrección de SenseiGMS, así como de las aplicaciones que lo usan, implica la misma problemática asociada a sistemas distribuidos, donde deben probarse procesos ejecutándose sobre múltiples máquinas. Aquí se añade el problema de probar los enlaces mismos, pues los algoritmos empleados son muy dependientes de cualquier retraso o alteración en las comunicaciones. Con el objetivo de cubrir esta necesidad, VirtualNet permite simular una red de cualquier complejidad sobre una única máquina, ofreciendo facilidades para alterar su topología y la calidad de las comunicaciones en cualquier momento. Además, su empleo no implica importantes cambios en la aplicación a probar, limitándose su impacto a la adición de una simple línea de código al principio del programa.

Capítulo 9 - METODOLOGÍA DE DESARROLLO

La replicación de una aplicación es esencial para hacerla tolerante a fallos, pero esa replicación resulta cara de realizar. Además de la duplicación de recursos hardware que precisa, hay un impacto negativo en el rendimiento, ya sea en la carga de trabajo soportada, en el tiempo de respuesta de servicio o incluso en ambos parámetros.

Empleando replicación activa, las réplicas deben mantener su consistencia mutua continuamente, por lo que toda actualización en una réplica debe propagarse a las demás. El impacto de esta propagación depende del sistema de comunicaciones en grupo y su algoritmo empleado. El protocolo optimizado de *Totem* [Totem] requiere, por ejemplo, dos vueltas y media del testigo para entregar fiablemente un mensaje en orden total. SenseiGMS requiere para el mismo cometido media vuelta del testigo más el envío de tres mensajes al miembro más lento del grupo, aunque la eficiencia se incrementa cuando un miembro envía varios mensajes en la misma vuelta.

Y esta propagación sobre el grupo es requerida para cada actualización en la aplicación en condiciones normales. Si el grupo formado por las réplicas cambia su composición, pierde su disponibilidad mientras se negocian los cambios, y es difícil calcular el impacto o duración de este bloqueo. Por ejemplo, en SenseiGMS, un cambio debido a la inclusión o exclusión voluntaria de un miembro se realiza en dos vueltas de testigo, más el envío de un mensaje al miembro más lento del grupo. Pero

si hay una partición del grupo o se cae el miembro que posee el testigo, debe verificarse la regeneración del testigo, comprobarse el consenso, enviarse nuevas vistas, etc.

El coste de la replicación activa puede reducirse empleando replicaciones pasivas, donde la consistencia sólo debe mantenerse a intervalos dados de tiempo, o incluso nunca, en el caso más simple. Cuanto menor sea la consistencia, mayor es el tiempo requerido para recuperar el servicio en caso de que la réplica primaria falle. Y además, sigue existiendo un impacto continuo de rendimiento, pues esa réplica primaria debe aún guardar su estado en dispositivos persistentes. Sin embargo, la especificación del servicio de tolerancia a fallos de CORBA da un fuerte soporte al empleo de replicaciones pasivas y, aunque cubre el uso de grupos activos, su soporte puede resultar insuficiente para aplicaciones complejas, como se detalla en este capítulo.

Si el empleo de replicaciones pasivas supone mejores tiempos de respuesta en la mayoría de los casos, ofrece por otro lado una menor disponibilidad que puede ser prioritaria para determinadas aplicaciones como, por ejemplo, controladores de dispositivos esenciales en un avión. Y si las aplicaciones pueden asegurar una buena calidad de las comunicaciones entre sus réplicas, el coste de la replicación activa puede reducirse a márgenes perfectamente aceptables para tales aplicaciones.

Por ejemplo, un servidor conectado a Internet, dando un determinado servicio mediante *Web*, puede disponer de réplicas situadas geográficamente de tal forma que ningún cliente deba acceder a servidores físicamente muy distantes, lo que supondría peores tiempos de respuesta. Y si la conexión entre las réplicas no se hace mediante Internet, sino en una red con una calidad de servicio fija y elevada, los tiempos requeridos para actualizar todas las réplicas pueden ser adecuados, además de minimizar las probabilidades de indisponibilidad del grupo por cambios en su composición.

Para estos casos donde la red soporta unos parámetros de calidad de servicio adecuados, es también posible emplear protocolos de comunicaciones optimizados, como el desarrollado en *Spinglass* [Spinglass], redundando en un mejor servicio del grupo.

Entendidas las limitaciones asociadas al modelo de replicación activa, este capítulo se centra en la programación de aplicaciones tolerantes a fallos bajo ese modelo. Empleando replicación pasiva, es preciso modificar la aplicación para que almacene su estado de forma periódica, pero este cambio resulta mínimo al compararlo con los cambios necesarios para emplear replicación activa. Es posible emplear diversas técnicas que simplifiquen el desarrollo de aplicaciones replicadas activamente, por lo que nos centramos en el soporte necesario para implementar esas técnicas; en particular, estudiamos el soporte ofrecido en CORBA, detallando sus insuficiencias para aplicaciones no básicas.

Los problemas de rendimiento en grupos activos pueden precisar de algoritmos fuertemente optimizados y específicos para una aplicación concreta. Sin embargo, el enfoque seguido en este capítulo es generalista, por lo que nos basamos en el empleo de orden total causal en las comunicaciones. Es decir, todas las réplicas reciben los mismos eventos en el mismo orden, lo que permite diseñar algoritmos más generales y simples que empleando órdenes menos estrictos.

9.1. Sincronización de la respuesta

En una aplicación cliente/servidor, el servidor define una interfaz que el cliente emplea para solicitar sus servicios. Generalmente el cliente queda bloqueado mientras el servidor procesa su solicitud, aunque esto no es siempre cierto, como puede[5] ser el caso de las operaciones *oneway* en CORBA. En una aplicación fiable, este servidor debe contactar a las demás réplicas y obtener así una respuesta consensuada, que es entonces entregada al cliente; este proceso implica en general la necesidad de sincronizar la solicitud del servicio con la elaboración de la respuesta.

Este problema fue brevemente esbozado en el capítulo anterior, al desarrollar un ejemplo de uso con SenseiGMS, y mostrado gráficamente en la figura 8.2. Ahora emplearemos una interfaz muy sencilla, con una única operación que muestra el patrón general de diseño a emplear; esta interfaz es la de un servidor que genera números únicos de forma secuencial, especificado en OMG/IDL como:

```
interface NumberGenerator
{
   long getNumber();
};
```

Una posible implementación de este servidor replicado se basa en que cada réplica mantenga una variable con el último número generado por el grupo. A esta variable la denominamos *lastGeneratedNumber*. Cuando una de estas réplicas recibe una petición *getNumber*, realiza los siguientes pasos:

- Envía un mensaje al grupo, de tal forma que todas las réplicas tengan conocimiento de que está en curso una petición.
- Las réplicas, al recibir el mensaje previo, incrementan automáticamente la variable que almacena el último número generado. Este paso también lo realiza la réplica que envió el mensaje.

---

[5] El estándar CORBA [OMG98] no especifica que la invocación de una operación *oneway* deba impedir el bloqueo del cliente en tanto el servidor la procesa. Sin embargo, un comportamiento no bloqueante resulta una implementación lógica y generalizada.

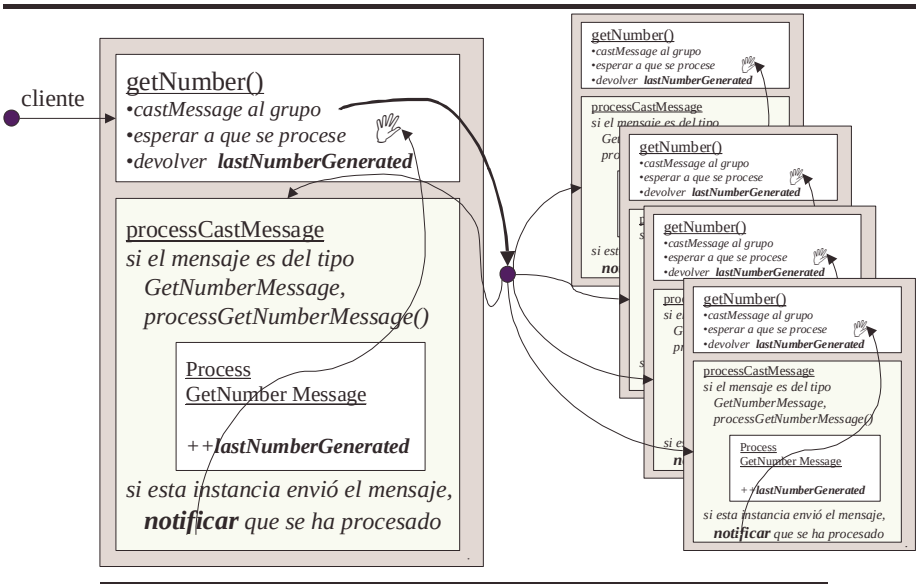

Figura 9.1. Sincronización de la respuesta

- Cuando la réplica que envió el mensaje lo ha procesado, puede ya devolver una respuesta al cliente, el valor contenido en *lastGeneratedNumber*.

Luego la réplica que envía el mensaje debe esperar la notificación de que ha sido procesado; la figura 9.1, similar a la figura 8.2, la extiende mostrando la sincronización requerida e incluye además en pseudo-código el escenario descrito.

Es importante destacar que la secuencia previa de pasos no es correcta. Bajo una implementación general de *threads* o hilos de control, no es posible asumir que un determinado *thread* en estado de espera que pasa a estado activo reciba inmediatamente tiempo de proceso. Por ello, es perfectamente posible que desde el momento en que se realiza la notificación de que el mensaje ha sido procesado hasta que se lee el valor de la variable *lastGeneratedNumber*, la réplica procese otro mensaje que modificaría de nuevo esta variable, devolviendo consecuentemente un valor erróneo.

La figura 9.2 muestra una alternativa a la anterior sincronización, más complicada pero necesaria en todas las operaciones que devuelven algún valor, sea como parámetro de salida o como valor de retorno:

- La réplica recibe en un *thread* la petición *getNumber*. Envía un mensaje al grupo, de tal forma que todas las réplicas tengan conocimiento de que una petición está en curso.
- Espera a que el mensaje se reciba, con lo que ese *thread* (*thread* de envío) queda suspendido.

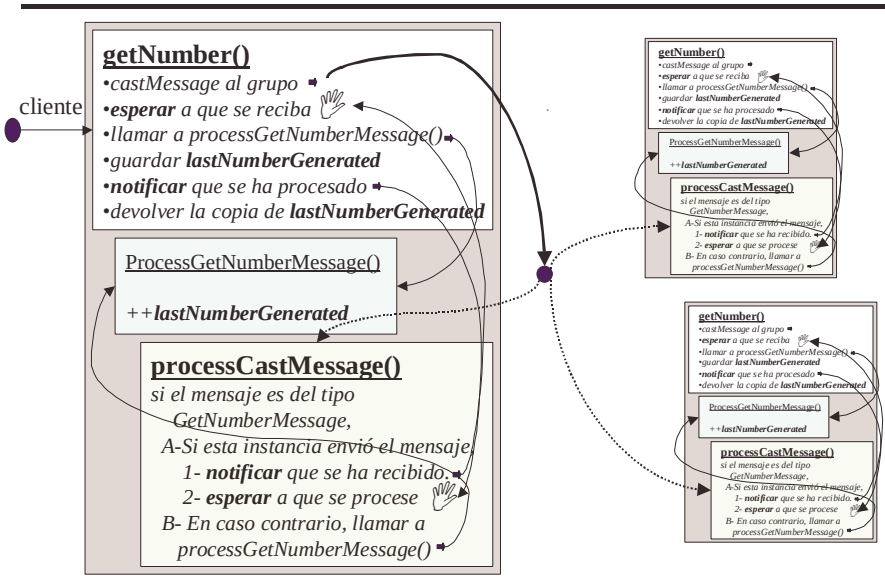

Figura 9.2. Alternativa para la sincronización de la respuesta

- El mensaje se recibe en un *thread* independiente (*thread* de recibo); en la réplica que envió el mensaje, este mensaje debe procesarse en el *thread* de envío. Por ello, se despierta a ese *thread*, y este *thread* de recibo pasa a estado suspendido. Las demás réplicas procesan el mensaje directamente en el *thread* de recibo.
- Cuando el *thread* de envío despierta, procesa el código asociado al mensaje, incrementando el valor de la variable *lastGeneratedNumber*, que será devuelta al cliente. A continuación, despierta al *thread* de recibo, que no debe hacer ya nada más. El *thread* de envío termina también al completarse la petición *getNumber*.

Es necesario en este escenario que el *thread* de recibo no finalice hasta que se haya procesado el mensaje en el *thread* de envío; en caso contrario, la réplica podría recibir nuevos mensajes, con lo que estaría procesando operaciones en paralelo, con resultados imprevistos salvo que incluyera sus propias opciones de sincronización, lo que complica de nuevo el escenario.

En este segundo caso, cuando una réplica recibe un mensaje que no envió, lo procesa como en el primer caso, sin requerir ninguna sincronización. En los dos escenarios previos, hay que considerar la situación en que una réplica es expulsada del grupo mientras está procesando una petición de servicio. En ese caso, los *threads* que estén suspendidos deben desbloquearse, devolviendo alguna excepción al cliente.

9.2. Transformación de operaciones en mensajes

En el ejemplo del anterior apartado se envía un mensaje en respuesta a la petición de operación del cliente. Como ya se vio en el anterior capítulo, ésa es la estrategia general: cada mensaje incluye la información necesaria para poder ejecutar en cada réplica el mismo código. Esta aproximación implica definir un mensaje para cada operación, que define como atributos cada uno de los parámetros de entrada definidos en la operación asociada.

Es necesaria una extensión a lo descrito en el anterior capítulo para permitir que una misma réplica pueda procesar varias operaciones concurrentemente. Para cada una de esas operaciones se envía un mensaje y su *thread* asociado se bloquea hasta que el mensaje se reciba o se procese. Por lo tanto, es necesario asociar cada mensaje al *thread* que lo envía, y por eso cada mensaje incluye una identidad fácilmente asociable a ese *thread*. Usando JavaRMI, esta identidad podría ser directamente la asignada por Java al *thread*, pero en CORBA debe ser independiente del lenguaje de programación final. Escogemos un valor entero para identificar al *thread*, y el algoritmo de envío de mensajes y bloqueo de *threads* debe ser capaz de asignar a cada *thread* una identidad única. Es importante notar que esta identidad sólo tiene sentido para la réplica que envía el mensaje, por lo que se puede asignar una misma identidad a mensajes provenientes de distintas réplicas.

Con CORBA, todos los parámetros *in* y *inout* deben incluirse en el mensaje. La operación *double obtainPercentage (in string key, inout long values, out boolean error)* se transforma en el mensaje:

```
valuetype ObtainPercentageMessage : Message
{
       public long id; //identidad del mensaje
       public string key; //parámetro de entrada
       public long values; //parámetro de entrada
};
```

No es necesario incluir información de los parámetros de salida, pues no es preciso que las demás réplicas devuelvan esa información. En el caso de JavaRMI, todos los parámetros son de entrada, luego el mensaje incluye todos los parámetros[6] de la operación.

Esta transformación de operaciones en mensajes es fácilmente automatizable, y una herramienta puede generar los mensajes necesarios a partir de la definición de las interfaces. Este proceso de automatización es extensible al propio servidor

[6] Que los parámetros sean de entrada, no implica que no puedan tratarse como parámetros de salida. Por ejemplo, es posible enviar a un método Java una instancia *StringBuffer* que únicamente se emplee para devolver una cadena de caracteres. En este caso, no es necesario incluir el atributo asociado en la definición del mensaje.

replicado: es posible crear todo el código y clases necesarias para definir un objeto replicado a partir de la definición de su interfaz y una implementación no replicada de ese servidor.

Para entender esta automatización, partimos de una clase *NumberGeneratorImpl* que implementa sin ningún soporte de replicación la interfaz *NumberGenerator* definida en el apartado anterior. El primer paso es la generación de un mensaje para la operación *getNumber*. Este mensaje no contiene ningún atributo asociado a la operación, sólo la identidad de mensaje:

```
valuetype GetNumberMessage : Message
{
       public long id;
};
```

Se genera también una clase *GroupNumberGeneratorImpl* que implementa igualmente la interfaz *NumberGenerator* e incluye toda la lógica de grupo necesaria. La lógica de aplicación se delega a la instancia de la clase *NumberGeneratorImpl*, con lo que la operación *getNumber* resulta ahora en Java:

```
int getNumber() throws InvalidGroupException
{
       int id = getRequestId();
       Message message = new GetNumberMessage(id);
       castMessageAndBlockUntilReceived(message);
       int ret = theNumberGeneratorImpl.getNumber();
       messageProcessed(id);
       return ret;
}
```

Las operaciones en cursiva se explican posteriormente; este código realiza los siguientes pasos:

- Obtiene una identidad única para la operación en curso.
- Crea el mensaje asociado a la operación, incluyendo la identidad obtenida.
- Envía el mensaje al grupo y se bloquea hasta que se reciba el mensaje.
- Se invoca la operación sobre la instancia no replicada del servidor.
- Se comunica que la operación ha sido procesada, tal como se describió en el anterior apartado.
- El valor devuelto por la instancia no replicada es el valor que se devuelve al cliente del grupo.

El código de recepción del mensaje sigue las pautas dadas anteriormente:

```
void processCastMessage(int sender, Message message)
{
        if (message instanceof GetNumberMessage) {
                GetNumberMessage msg = (GetNumberMessage) message;
                if (sender==myself) {
                        unblockThreadAndWait(msg.id);
                }
                else {
                        theNumberGeneratorImpl.getNumber();
                }
        }
        else
                . . . .
}
```

- Se verifica cada uno de los tipos de mensaje esperado.
- Si la réplica que recibe el mensaje es la que lo envió, despierta el *thread* asociado y se bloquea hasta que ese *thread* concluye.
- En caso contrario, se invoca directamente la operación sobre la instancia no replicada. Si esta operación precisa de parámetros de entrada, sus valores se leen del mensaje recibido[7].

Esta automatización emplea el segundo escenario descrito para la sincronización de la respuesta en el anterior apartado, incluso para operaciones que no devuelven valores, pues es válido para todo tipo de operaciones.

El código descrito ha empleado cuatro primitivas, comunes para cualquier servidor:

- *getRequestId:* devuelve una identidad única. Es fácilmente implementable a partir de una variable con tipo entero que se incrementa con cada llamada.
- *castMessageAndBlockUntilReceived:* envía el mensaje al grupo y bloquea al *thread* que lo llama hasta que se reciba el mensaje.
- *unblockThreadAndWait:* desbloquea al *thread* que tenga la identidad dada como parámetro y bloquea automáticamente al *thread* que lo llama.
- *messageProcessed:* desbloquea al *thread* asociado al mensaje especificado.

Esta automatización permite, por lo tanto, generar rápidamente un servidor replicado a partir de una instancia no replicada de ese servidor. El proceso descrito no es completo, ya que es necesario escribir manualmente el código asociado a las

[7] En C++, para los parámetros de salida deberían definirse variables temporales que se descartarían tras invocarse la operación. Lo mismo ocurriría en Java para los parámetros manualmente excluidos del mensaje, como explicó la anterior anotación.

operaciones de transferencia de estado. Empleando JavaRMI, es posible emplear sus mecanismos de serialización para implementar esa transferencia, pero con CORBA la implementación debe realizarse independientemente.

9.3. Comportamiento no determinista

A pesar de los beneficios aparentes de la automatización descrita en la sección previa, sólo las aplicaciones más sencillas podrán replicarse de esa manera. El anterior enfoque implica replicar el comportamiento de la aplicación a partir de su interfaz. Pero una interfaz define las operaciones a soportar, no la implementación o estructuras de datos necesarias. Si en el ejemplo anterior el generador de números únicos debiera devolver números aleatorios, su interfaz sería aún la misma:

```
interface NumberGenerator
{
   long getNumber();
};
```

Sin embargo, la implementación pasa a considerarse no determinista: la misma secuencia de eventos no produce la misma secuencia de resultados.

Internamente, la aplicación puede definir ahora una estructura de datos que incluya todos los números ya generados. Cuando se solicita un nuevo número, se obtiene uno al azar y se comprueba en la anterior estructura de datos si está libre o ha sido ya asignado; si no lo está, se puede generar otro o simplemente iterar en la tabla a partir del número anterior buscando el primer número libre. Esta tabla es evidentemente finita, pero obviamos este hecho para los propósitos de este ejemplo.

Al no haber cambiado la interfaz *idl*, el proceso de automatización produciría el mismo código detallado en la sección previa, lo que provocaría resultados incorrectos:

- Cuando una réplica recibe la solicitud *getNumber*, envía el mensaje *GetNumberMessage*.
- Cada réplica, al recibir este mensaje, invoca la instancia no replicada del servidor.
- Las distintas réplicas, si el algoritmo de aleatoriedad es bueno, producirán distintos números aleatorios, con lo que la consistencia de sus estados se pierde. Distintas réplicas consideran que los números asignados son distintos, con lo que producirán eventualmente números no únicos.

La solución a este problema es sencilla: la réplica que recibe la solicitud *getNumber* genera directamente un número aleatorio, que es incluido en el mensaje *GetNumberMessage*. Los demás miembros del grupo comprueban si ese número está libre y, si no es así, iteran sobre su estructura de datos interna hasta obtener el

siguiente número libre, pues generar otro número aleatorio produciría el mismo resultado erróneo. Pero a pesar de la sencillez de la solución, el problema es ya incompatible con la automatización descrita.

Este problema no se asocia sólo a comportamientos no deterministas. Si por ejemplo el servidor debe enviar el resultado de la operación a una página *Web* o, en general, actuar como cliente de un servidor diferente, sólo una de las réplicas debería efectuar la operación correspondiente. Este problema es inherente al servicio de tolerancia a fallos de CORBA, como detallamos al final de este capítulo (de hecho, ese servicio especifica claramente que las aplicaciones soportadas deben ser deterministas).

9.4. Replicación de componentes

Un comportamiento no determinista es uno de los casos que impiden la automatización propuesta en la replicación de un servidor. Otro caso ya señalado se da cuando ese servidor debe acceder a ciertos recursos externos; si por ejemplo un servicio debe acceder a un servidor GPS que devuelve la posición geográfica de un determinado localizador, y ese servidor cobra por cada acceso, el servidor a replicar debería evitar que todas las réplicas consultaran al servicio GPS.

En general, todo servidor cuya implementación dependa de otros servidores no podrá automatizarse con el método descrito. Y este problema persiste incluso si la dependencia se produce a nivel de interfaz. Por ejemplo, la clase *java.util.Vector* que implementa la interfaz *java.util.List* podría utilizarse para crear un contenedor replicado; sin embargo, esta clase devuelve en algunas operaciones *iteradores* que permiten observar los elementos contenidos en el *Vector*. Y esos iteradores deben modificarse para soportar las características replicadas del contenedor asociado, lo que implica modificar el código que produce esos iteradores para crear los nuevos con soporte de replicación.

Por lo tanto, la automatización del proceso de replicación es únicamente aplicable a servidores sencillos. Por otra parte, es posible identificar estructuras en el servidor que puedan replicarse automáticamente. Volviendo al ejemplo del generador de números únicos y aleatorios, se identifica fácilmente como replicable la estructura de datos interna que contiene los números ya generados. Esta estructura se programa separadamente con la siguiente interfaz:

```
interface NumberSet
{
       boolean set (in long Number);
};
```

La única operación definida, fija como asignado el número dado en el parámetro y devuelve *true* si el número estaba todavía libre (no asignado).

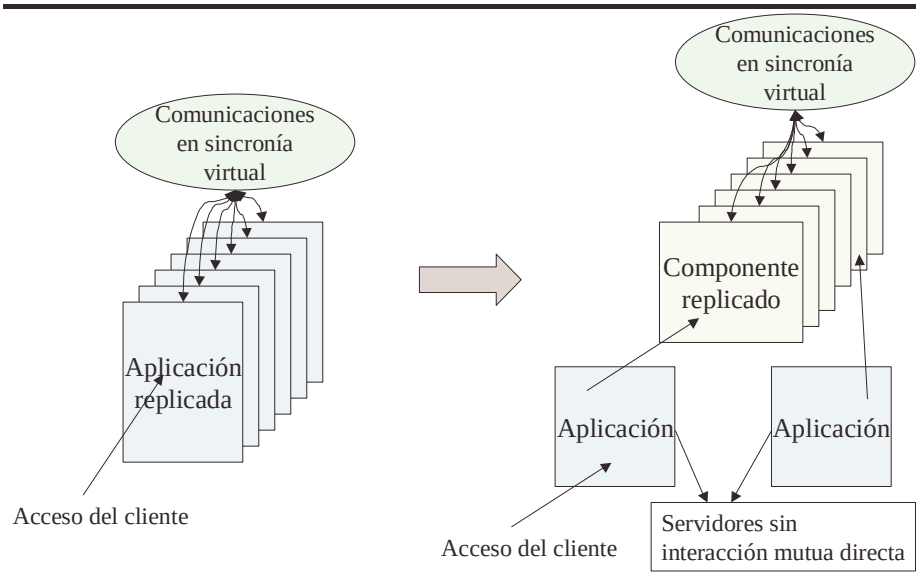

Figura 9.3. Enfoque en la replicación de componentes

El servidor no replicado se programa empleando una instancia de la interfaz *NumberSet*, denominada *theNumberSet* en el siguiente código, que debería ser mejorado para una situación real donde el contenedor de números no es infinito. No obstante, muestra la idea del algoritmo: iterar sobre el contenedor hasta encontrar una posición libre, comenzando desde una posición aleatoria:

```
int getNumber()
{
        int initial = getRandomNumber();
        while (!theNumberSet.set(initial)) {
                ++initial;
        }
        return initial;
}
```

La interfaz *NumberSet* sí es automáticamente replicable, con lo que sólo es necesario cambiar el código del servidor para instanciar y emplear la clase que implementa el comportamiento replicado. Este servidor, al recibir una petición *getNumber* no se comunica con sus réplicas sino que ejecuta el mismo código empleado en el servidor no replicado. Sin embargo, cada vez que ahora accede a *theNumberSet* emplea una estructura replicada que sí se comunica con sus demás réplicas.

La figura 9.3 muestra el cambio de enfoque de esta aproximación: no se replica el servidor completo, sino los componentes que lo precisan. Los servidores forman un grupo tolerante a fallos pero no interaccionan directamente entre ellos, sino a

través de esos componentes. La abstracción que esos componentes replicados crean es la de un único componente compartido (memoria compartida).

9.5. Librerías de componentes replicados

El lenguaje C++ [ISO98, Stroustroup97] se creó y extendió inicialmente sin una librería de contenedores, lo que provocó que cada programador diseñara e implementara sus propias soluciones para pilas, listas, árboles binarios, etc, o que se adhiriera a determinadas soluciones comerciales, pocas veces compatibles entre sí. La estandarización posterior del lenguaje incluyó una librería estándar de *templates* (STL) [ISO98, Plauger95], simplificando considerablemente este escenario. Un programador puede emplear distintas implementaciones sabiendo que son compatibles, y usar la que produzca mejores rendimientos. El empleo de servicios replicados está poco extendido a pesar de sus indudables beneficios. Y la causa no es únicamente el menor rendimiento que se consigue, sino la dificultad de su diseño. Si se dispone de librerías de componentes replicados, ese diseño se facilitará considerablemente.

El principal problema es entonces considerar qué componentes deben ser replicados. En general, el comportamiento de una aplicación depende del estado de sus datos. Esto no implica un determinismo, que empleando los mismos datos esa aplicación produzca siempre los mismos resultados, pero sí una consistencia de éstos. Consecuentemente, es comprensible enfocarse en los componentes que almacenan esos datos.

Cada aplicación define sus propias estructuras de datos, que pueden ser desde simple tipos existentes en el lenguaje, como enteros, a complejas estructuras, como un árbol binario que contenga a su vez listas de valores enteros. Sin embargo, es posible factorizar esas estructuras complejas en términos de los contenedores estándar.

El ejemplo mostrado en la sección anterior podría simplificarse si dispusiéramos en Java de un *java.util.Set* replicado o, hablando en términos C++, de un *std::set* replicado. Además de la simplificación que se obtiene en el diseño de servidores replicados, la principal ventaja es la optimización que puede y debe lograrse sobre esos componentes. SenseiUMA es la parte de este proyecto que trata el diseño de estos contenedores replicados. Este diseño está directamente basado en la colección de componentes disponibles en Java 1.2.

9.6. Soporte de concurrencia

Al enfocar la replicación sobre los componentes de una aplicación, es necesario estudiar los problemas de concurrencia que pueden aparecer. En una aplicación no

replicada, el empleo de múltiples *threads* complica la implementación de sus componentes para soportar el acceso concurrente desde esos *threads*. En una aplicación basada en componentes replicados, esos componentes pueden entenderse como objetos compartidos al que acceden las distintos aplicaciones del grupo, por lo que deben soportar directamente un acceso concurrente.

Por ejemplo, la interfaz *NumberSet* utilizada en la sección de replicación de componentes se diseñó de tal manera que permitiera acceso concurrente desde varios servidores. Si se hubieran seguido las reglas generales para escribir buen código, que dictan la definición de múltiples operaciones simples en lugar de complicadas operaciones multifunción [Maguire93], se habrían definido dos métodos claramente diferenciados:

```
interface NumberSet
{
       void set (in long Number);
       boolean test(in long Number);
};
```

Un método comprueba si un número ha sido asignado, y el otro lo asigna. Sin embargo, esta interfaz no podría haberse usado directamente para producir componentes replicados: dos servidores podrían comprobar simultáneamente si un mismo número estaba libre, obteniendo ambos una respuesta afirmativa y asignándolo erróneamente [figura 9.4]. Al juntar los dos métodos, se obtiene una operación atómica que soporta accesos concurrentes.

Esta definición de operaciones atómicas específicas no es posible al emplear componentes genéricos. En este ejemplo, el componente empleado sería un *Set*, definido como un contenedor de elementos que no admite duplicados, o un *BitSet*, definido como un contenedor numerado de valores booleanos. Basándonos en la clase *java.util.BitSet*, serían necesarias las dos operaciones siguientes, expresadas en Java como:

- *boolean get (int bitIndex);* Devuelve el valor del bit en el índice dado.
- *void set (int bitIndex);* Asigna al bit dado el valor sí/cierto/*true*.

Y tal como sucedía con las operaciones de la interfaz previa *NumberSet*, se observan los mismos problemas de concurrencia descritos. La solución en el caso de acceso concurrente bajo múltiples *threads* es el empleo de monitores: un *thread* debe adquirir un monitor antes de efectuar una operación crítica, liberándolo al concluirla. Distintos *threads* acceden al mismo monitor, lo que impide que ejecuten concurrentemente la operación crítica.

Por extensión, la solución para el problema de concurrencia en componentes replicados es el empleo de monitores replicados. Estos monitores deben soportar dos operaciones, con las siguientes semánticas:

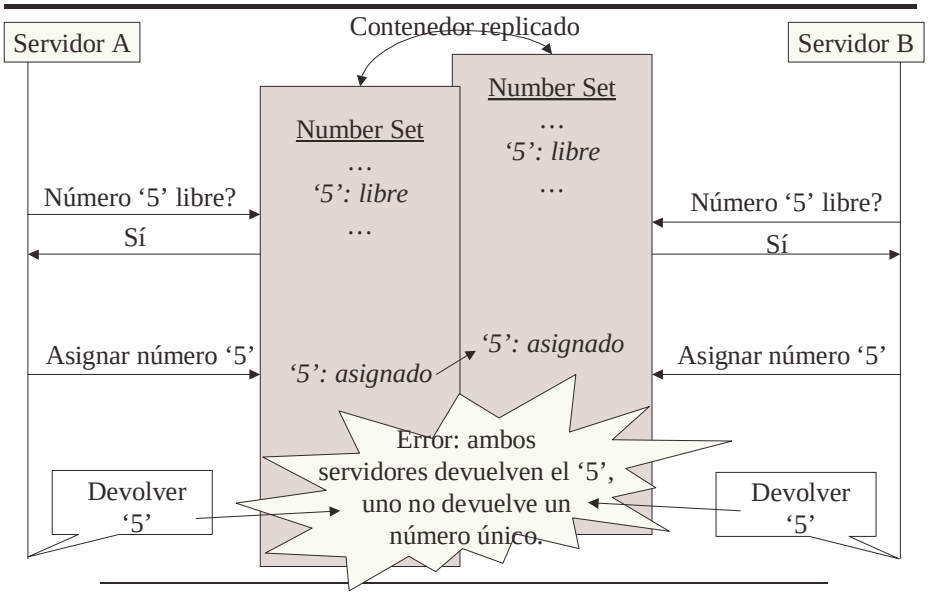

Figura 9.4. Problema de concurrencia en *NumberSet*

- *lock*: adquisición del monitor. Ante una adquisición concurrente desde varias réplicas, sólo una lo obtiene. Las demás réplicas que pretendan adquirir ese monitor quedan bloqueadas hasta que se libere el monitor o hasta que el componente que lo ha adquirido se caiga o sea excluido del grupo. Una vez que el monitor se libere, cualquiera de las réplicas bloqueadas puede obtenerlo, pasando a estado no bloqueado. Una misma réplica puede adquirir un monitor repetidas veces, es decir, el monitor es reentrante.
- *unlock*: libera un monitor previamente adquirido, produciéndose un error si la réplica no había adquirido el monitor. Si esa réplica hubiera adquirido el monitor múltiples veces, debe invocarse esta operación el mismo número de veces para liberar efectivamente el monitor,

Estas dos operaciones son suficientes si no se plantean condiciones de guarda (*wait/notify*), necesarias normalmente en aplicaciones *multithread* cuando diversos *threads* deben comunicarse entre sí.

Con esta definición, una réplica que adquiere un monitor puede obtenerlo repetidamente sin precisar liberarlo entre adquisiciones. Este monitor controla, de esta manera, regiones críticas entre diferentes réplicas, pero no entre diferentes *threads* de una misma réplica.

Empleando un monitor y un *bitset* replicados, el servidor *NumberSet* podría codificarse como:

```
number = getRandomNumber();
replicatedMonitor.lock();
```

```
while (replicatedBitSet.get(number)) {
     number++;
}
replicatedBitSet.set(number);
replicatedMonitor.unlock();
return number;
```

Este código es completamente análogo al empleado en una aplicación *multithread*; la diferencia existe sólo en los objetos empleados que, al estar replicados, pueden provocar errores en sus comunicaciones con las demás réplicas. Esta situación es paralela a la existente en cualquier aplicación CORBA: el acceso a objetos remotos se realiza bajo la abstracción de que son locales y la transparencia sólo se pierde en el menor rendimiento obtenido y en la necesidad de verificar los problemas de comunicaciones.

9.7. Transferencia de estado

Al promover la factorización de aplicaciones replicadas en componentes replicados, el empleo de una interfaz de transferencia de estado flexible resulta obviamente útil. Sin soporte de transferencia de estado, cada componente debería implementar sus propios protocolos y, si la interfaz de transferencia existente es poco flexible, provocaría una mala adaptación de esos componentes que podría redundar en un pobre rendimiento. En cualquier caso, la transferencia de estado ha sido el tema central en varios capítulos anteriores, con lo que nos limitamos aquí a exponer de nuevo la necesidad de un soporte flexible de transferencia de estado.

Un aspecto de los protocolos de transferencia que se expuso en los capítulos previos fue la necesidad de transferencias en varios pasos, especialmente útil en el caso de estados grandes. Al factorizarse una aplicación en componentes menores, ese requisito es aún necesario por varias razones:

- La aplicación puede enviar el estado de todos sus componentes en un solo paso o aprovechar el protocolo para enviar el estado en sucesivos pasos, cada componente por separado.
- Un componente separado puede ser aún suficientemente grande por sí solo. Tomando un contenedor como ejemplo, puede emplear sucesivos pasos para transferir su estado cuando contiene muchos elementos, o transferirlo en un único paso en caso contrario.

Como se vio anteriormente, un servidor no puede, en principio, procesar mensajes de otras réplicas o de clientes mientras efectúa una transferencia. En el caso de transferencias grandes, esa pérdida de servicio puede ser no permisible, con lo que sería preciso que la aplicación no se bloqueara y transmitiera su estado en diferentes etapas, implementando algoritmos que le permitieran transferir los cambios de estados producidos durante la transferencia en curso. Estos algoritmos

no son triviales, lo que supone otra ventaja para el argumento de emplear librerías de componentes replicados, que implementan ya estos algoritmos de forma optimizada.

9.8. Gestión de componentes replicados

El empleo de componentes replicados ha mostrado tres ventajas respecto a la replicación monolítica de la aplicación:

- Mayor facilidad para automatizar su implementación a partir de un componente no replicado.
- Minimizar el tiempo de desarrollo mediante reusabilidad de código, al emplear librerías con componentes replicados ya desarrollados, probados y optimizados.
- Simplificación del código, al acceder a esos componentes replicados como componentes estándar, no replicados.

Sin embargo, estos componentes no son objetos *normales*, tienen un ciclo de vida especial. Cada componente, una vez creado, debe acceder a un servicio GMS para crear un grupo o incluirse en uno ya existente. Y cada componente pertenece a un grupo distinto, con lo que la aplicación debe esperar a que todos los componentes pertenezcan a sus respectivos grupos para poder emplearlos.

Además, como se vio en los anteriores capítulos, un miembro puede ser excluido de un grupo accidentalmente, en caso de que resulte lento o sus comunicaciones con otras réplicas sean lentas. La exclusión de un componente dado de entre los muchos que una aplicación puede contener supondrá una merma total o parcial de la disponibilidad de esa aplicación. En este caso, es posible corregir esa expulsión incluyendo al componente de nuevo en un grupo, pero las causas que han provocado su exclusión accidental implican que muy probablemente otros componentes habrán sido expulsados igualmente de sus grupos. En cualquier caso, la aplicación debe manejar estos eventos y parar eventualmente su actividad.

Hay dos aproximaciones en el empleo de múltiples componentes. La primera es crear múltiples grupos, lo que implica la necesidad de coordinación desde la aplicación. La segunda es crear un único grupo donde cada componente aparece como *subgrupo*. Para minimizar la confusión con grupos y subgrupos, llamaremos *dominio* al grupo principal, que es el único que debe emplear el GMS para crear o incluirse en su grupo de réplicas, y componentes a esos subgrupos. Cuando un componente se comunica con sus réplicas, el dominio incluye una identidad asociada a ese subgrupo en los mensajes, de tal forma que el dominio destino pueda demultiplexar éstos al componente adecuado, como muestra la figura 9.5.

El modelo de dominios supone además un empleo menor de recursos. Empleando SenseiGMS, cada grupo crea su propio anillo, con un testigo independiente; cada miembro comprueba sus comunicaciones con el grupo, vigila la

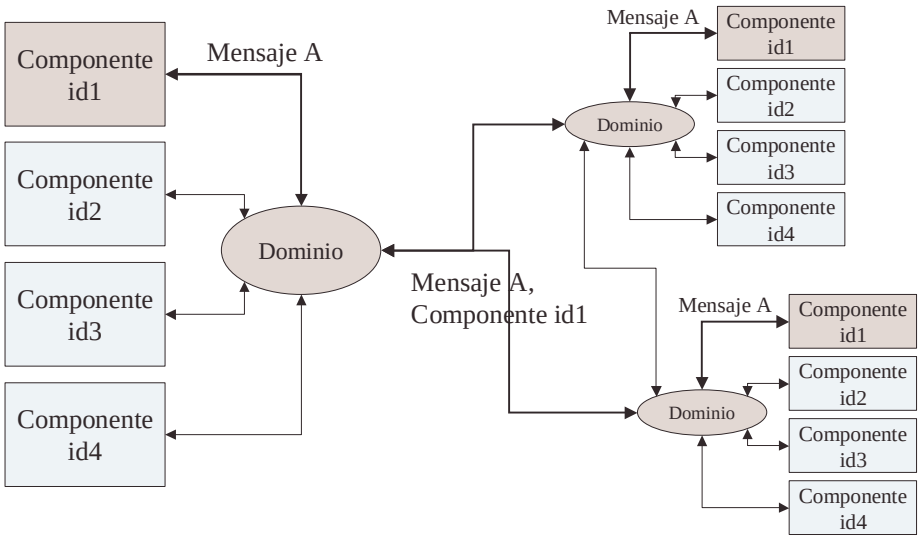

Figura 9.5. Empleo de *dominios*

pérdida o duplicación del testigo, etc. Si existe un único grupo, se limita ese empleo de recursos. Además se simplifica la interacción entre réplicas, al no tener que coordinarse los grupos de los distintos componentes; en su lugar, los componentes son los subgrupos que deben subscribirse al dominio mediante una sencilla operación. El dominio sólo debe asegurar que cada componente tenga la misma identidad en cada miembro del dominio. El estado del dominio lo forman el estado de esos componentes más las propiedades asociadas al dominio, como son las identidades de los componentes. Como el primer paso tras la inclusión del dominio en su grupo es la transferencia de estado, la coherencia en la asignación de identidades a componentes en los miembros del dominio puede comprobarse desde un principio. Este razonamiento implica que los componentes se registran en el dominio antes de que éste se una a su grupo.

Para entender mejor el concepto, emplearemos un ejemplo de un servicio de ficheros replicado. En este servicio [figura 9.6], los componentes son los directorios y ficheros, donde los directorios agrupan ficheros lógicamente. Todos los miembros del dominio contienen los mismos directorios y ficheros, de tal forma que el servicio es tolerante a fallos. Cada uno de los componentes en este ejemplo se registra en el dominio y la aplicación incluye a continuación el dominio en su grupo de réplicas. Cuando el dominio recibe los mensajes de transferencia de estado, obtiene el estado de los directorios y de los ficheros, así como información de los otros miembros en el dominio. En particular, recibe la confirmación de que todos los miembros han definido los mismos componentes con las mismas identidades.

Este ejemplo demuestra también la necesidad de dinamismo en el empleo de componentes. El servidor mostrado no puede extender el número de ficheros que

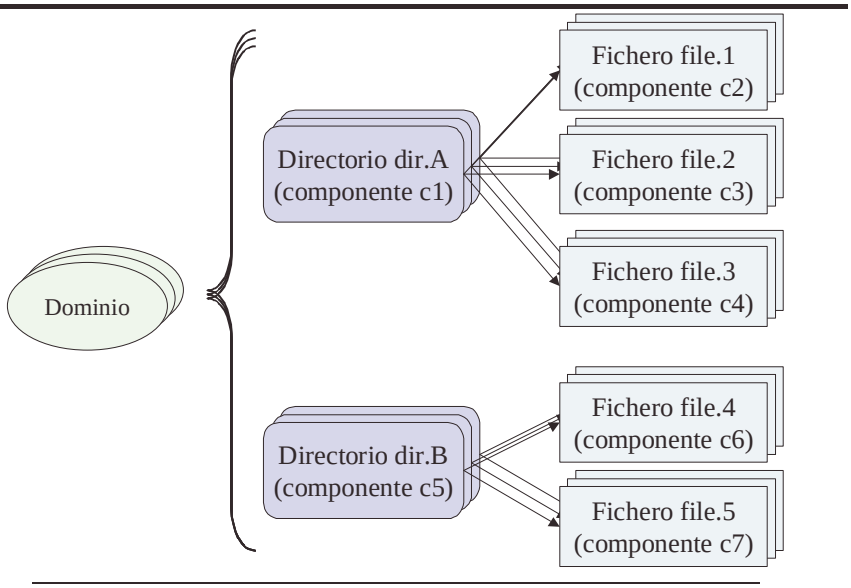

Figura 9.6. Servicio replicado de ficheros

sirve, puesto que los componentes deben ser conocidos antes de la inclusión en el grupo, momento en el que se podría obtener fácilmente esa información. Además, si se pretende poder emplear componentes replicados sin diferenciarlos de otros componentes *normales* en la aplicación, es preciso que se puedan crear esos componentes dinámicamente. Siguiendo con el escenario descrito, un cliente de la aplicación replicada podría solicitar la creación de un nuevo fichero en un directorio dado: el servidor crea en este caso el componente que representa el fichero y le asigna una identidad, comunicándoselo entonces a las demás réplicas.

Hay, por lo tanto, componentes estáticos, que se registran antes de incluirse en el grupo de réplicas, y componentes dinámicos, registrados tras la inclusión en el grupo. Todos los miembros del dominio deben registrar los mismos componentes estáticos, mientras que los componentes dinámicos son creados por un miembro del dominio, propagándose su creación a los otros miembros.

Un dominio no es más que un grupo que multiplexa/demultiplexa los mensajes de sus componentes, pero define a su vez sus propios mensajes. La creación de un componente dinámico puede implementarse entonces a partir de un mensaje específico del dominio, incluyendo información sobre el componente que se crea. Este componente debe existir en el espacio de memoria de cada una de las réplicas, por lo que cuando el dominio recibe el mensaje de creación del componente, solicita a la aplicación que cree dinámicamente ese componente con la información obtenida. De este requisito se deduce que la aplicación debe soportar una operación *callback* para crear componentes dinámicamente. Por ejemplo, en Sensei este método es:

```
GroupMember subgroupCreated(  in GroupMemberId creator, in SubgroupId id,
                              in DynamicSubgroupInfo info);
```

Se recibe la identidad dada al componente e información sobre el tipo de componente, tal como la haya dado la réplica que lo creó.

Durante el proceso de transferencia de estado, un dominio que se incluye en un grupo recibe información sobre los grupos dinámicos ya creados, de tal forma que pueda crearlos en su espacio de memoria antes de recibir el estado asociado a esos componentes. Es también posible eliminar componentes dinámicos, lo que se traduce en otro mensaje del dominio y en la necesidad de otro método *callback* en la aplicación. Sin embargo, no es posible eliminar los componentes estáticos, pues su existencia se verifica con cada transferencia de estado. Introducir estas diferencias entre ambos tipos de componentes puede parecer una complicación, teniendo en cuenta, además, que los dinámicos son más flexibles que los estáticos, pero las aplicaciones replicadas que precisen únicamente de componentes estáticos se benefician de no tener que soportar la funcionalidad adicional necesaria por los dinámicos.

Que las identidades de los componentes sean únicas implicará generalmente una necesidad de comunicaciones en el dominio para consensuar la unicidad de cada identidad. Por esta razón, es el mismo dominio el que genera las identidades en caso de componentes dinámicos. Esta es la otra diferencia con los componentes estáticos; cuando éstos se registran, el dominio aún no pertenece a un grupo, con lo que no puede comunicarse con otros miembros, lo cual deja tres opciones para la obtención de identidades:

- Las identidades son asignadas secuencialmente según se registran los componentes. Todos los dominios deben registrar entonces en ese mismo orden todos sus componentes estáticos, lo que puede originar problemas en caso de aplicaciones *multithread*.
- Las identidades son dadas por la aplicación al registrar el componente estático.
- La asignación de identidades se pospone hasta la inserción del dominio en un grupo: complica la aplicación y los componentes, que deben implementar la funcionalidad necesaria para recibir esa información. Además, si esa asignación se produce siguiendo el orden en que se registraron los componentes, se da el mismo problema que con la primera opción.

Por estos motivos, es la aplicación la que define las identidades de los componentes estáticos y el dominio el que las asigna para componentes dinámicos. Debe notarse que, sin el concepto dado de dominios, el empleo de componentes dinámicos es también posible cuando cada componente emplea su propio grupo de réplicas. En este caso la aplicación debe definir otro grupo especial para manejar los mensajes para la creación y destrucción de esos componentes.

Volviendo al ejemplo dado de servidores replicados de ficheros, su implementación puede realizarse ahora empleando únicamente componentes

dinámicos. Sin embargo, si la estructura de ficheros es jerárquica, el directorio raíz siempre estará presente, lo que da sentido a tener un único componente estático asociado a ese directorio. Cuando este servidor recibe una petición de creación de un fichero, crea el componente asociado:

```
public FileDescriptor createFile(String name, DirectoryDescriptor descriptor)
  throws DirectoryNotExisting, FileAlreadyExists
{
        Directory dir = getDirectory(descriptor);
        if (dir.contains(name)) {
                throw new FileAlreadyExists();
        }
        File file = new File(domain, name, descriptor);
        dir.add(file);
        return file.getDescriptor();
}
```

Aunque este código resulta muy similar al de un servidor no replicado, la primera diferencia se da ya en el constructor para el componente *File*, que necesita dos parámetros adicionales: el dominio, donde debe registrarse, y el descriptor del directorio, pues esta información debe propagarse a las demás réplicas:

```
File(Domain domain, String name, DirectoryDescriptor descriptor)
{
        DynamicSubgroupInfo info = new FileInfo(name, descriptor);
        subgroupId  = domain.createSubgroup(info, this);
}
```

Cuando el dominio crea el subgrupo, envía un mensaje a las demás réplicas del dominio, y sólo lo considera creado cuando ese mensaje es procesado, asignándole entonces una identidad única. Cada réplica debe suministrar su propio *callback* para crear el componente:

```
GroupMember subgroupCreated(int creator, int id, DynamicSubgroupInfo info)
{
        if (info instanceof FileInfo) {
                FileInfo fileInfo = (FileInfo) info;
                File file = new File(id, info.name);
                Directory dir = getDirectory(descriptor);
                dir.add(file);
                return file;
        }
        . . .
}
```

En este código, que no incluye condiciones de error, la aplicación decide el tipo de componente a crear en función de la información recibida. El constructor

empleado para el componente que representa el fichero es diferente al que se empleó anteriormente: ahora no precisa registrarse en el dominio, pues ya lo está, recibe su identidad asociada, que ya es conocida, y no precisa recibir el directorio en que se encuentra, pues esa información es redundante para el componente *fichero*.

El empleo de componentes dinámicos es entonces más complicado, con la necesidad de funcionalidad en la aplicación para crear componentes en demanda, y en los componentes para soportar diferentes constructores con cada escenario.

9.9. Transacciones

Los monitores replicados se han definido de forma similar a los monitores empleados en aplicaciones *multithread* pues solucionan un problema también similar: secuencializar el acceso concurrente a un componente o región crítica de código para evitar actualizaciones paralelas incompatibles.

Sin embargo, la concurrencia en componentes replicados implica un problema adicional que no se asocia a aplicaciones *multithread*, sino a bases de datos distribuidas: es preciso sincronizar el acceso a la base de datos para impedir que dos aplicaciones la actualicen de forma incompatible. Además, es necesario proteger a la base de datos en caso de que una aplicación se caiga sin haber completado sus modificaciones. La solución a este problema es el empleo de *transacciones* [Gray93].

En el caso de una aplicación que deba actualizar varios componentes replicados de forma coordinada, su caída, o expulsión accidental del grupo, antes de completar todos los cambios conlleva una inconsistencia en el estado de esos componentes. Es necesario, entonces, un mecanismo que proteja a los componentes replicados de caídas de las aplicaciones cuando éstas realizan actualizaciones sobre varios componentes simultáneamente. Este mecanismo lo llamamos, por paralelismo, transacción.

Empleando el ejemplo del servicio replicado de ficheros, la operación de mover un fichero a un directorio distinto podría escribirse de la siguiente manera, sin detallar cómo comenzar o finalizar la transacción:

```
void moveFile( String name, DirectoryDescriptor origin, DirectoryDescriptor target)
  throws DirectoryNotExisting, FileNotExisting FileAlreadyExists
{
       Directory dirOrigin = getDirectory(origin);
       Directory dirTarget = getDirectory(target);
       File file = dirOrigin.get(name);
       Start transaction
       dirOrigin.remove(file);
       dirTarget.add(file);
       End transaction
}
```

La transacción es necesaria para evitar que, si la aplicación cae tras borrar el fichero del primer directorio pero antes de incluirlo en el segundo, el resultado sea la pérdida de ese fichero.

El mecanismo de transacciones puede basarse en componentes que incluyan su soporte directamente: deben ser capaces de deshacer, no sólo el último cambio efectuado, sino todos los cambios realizados desde el momento en que se inició la transacción. Evidentemente, este soporte puede resultar muy complicado de implementar. Además, se complica el empleo de esos componentes, que deben recibir información sobre la transacción en curso; así, el bloque anterior de código relativo a la transacción podría escribirse como:

```
. . .
       Transaction transaction = startTransaction();
       dirOrigin.remove(file, transaction);
       dirTarget.add(file, transaction);
       endTransaction(transaction);
. . .
```

Es decir, las operaciones sobre el componente incluyen la información de la transacción. Lo que significa que la aplicación debe emplear transacciones incluso para actualizaciones que no las precisan, como es el caso en que afectan a un único componente. O bien, es necesario duplicar el número de operaciones en el componente, con y sin soporte de transacciones, complicando aún más ese componente.

Una posibilidad alternativa se basa en limitar los cambios en los componentes a la réplica que realiza la transacción, haciéndolos visibles a las demás réplicas cuando la transacción se completa. Como las operaciones sobre los componentes se traducen en mensajes al grupo, esta solución implica encolar esos mensajes durante la transacción. Esos mensajes deben, sin embargo, ser recibidos por la réplica que realiza la transacción, que sí debe ser capaz de observar los cambios efectuados. Este bloqueo de mensajes es fácil de realizar al emplear el concepto de dominios explicado en la sección anterior: puesto que todos los mensajes son multiplexados a través del dominio, es simple encolarlos a este nivel, mientras que si cada componente definiera su propio grupo, cada componente sería responsable de realizar ese bloqueo. El código relativo a la transacción se transforma ahora en:

```
. . .
       domain.startTransaction();
       dirOrigin.remove(file);
       dirTarget.add(file);
       domain.endTransaction();
. . .
```

Y el uso de la transacción resulta transparente para los componentes.

Hay dos problemas con esta solución por sus implicaciones con el modelo de sincronía virtual. El primero es que no todos los miembros del grupo observan los mismos mensajes en el mismo orden. Si otro miembro envía un mensaje durante la transacción, la réplica que realiza esa transacción observa ese mensaje entre los demás mensajes que la réplica está enviando, pues no hay un bloqueo de los mensajes que se reciben. Sin embargo, las demás réplicas ven primero el mensaje dado y luego todos los mensajes de la transacción, enviados cuando ha finalizado.

Por otro lado, el grupo podría instalar una nueva vista mientras un miembro efectúa una transacción, lo que presenta tres opciones:

- Posponer la instalación de la nueva vista hasta que la transacción se complete. Pero la transacción es un mecanismo lógico implementado sobre el GMS, que entonces debería ser extendido para conocer el estado de las transacciones. Esto implicaría un cambio muy importante, principalmente por la necesidad de redefinir el comportamiento del GMS (especialmente la detección de miembros caídos).
- Cancelar la transacción, que la aplicación debe reiniciar una vez que se instala la vista. Implica volver a escribir la lógica de tratamiento de vistas en la aplicación, rompiendo la transparencia que se logra con el empleo de componentes replicados. Además, los componentes en la réplica que realiza la transacción deberán deshacer los cambios efectuados hasta el momento, solución que hemos descartado por compleja.
- Instalar la vista y completar la transacción en la siguiente vista. Esta solución es la más viable, pero presenta una nueva incompatibilidad con el modelo de sincronía virtual: todos los miembros del grupo no observarán los mismos mensajes entre dos vistas consecutivas. El miembro que realiza la transacción ve los mensajes en la vista que se envían, mientras que sus réplicas los reciben en la vista en que concluye la transacción.

Por consiguiente, esta segunda implementación de las transacciones implica una menor complejidad de los componentes y de su empleo, pero viola el modelo de sincronía virtual. Sin embargo, como el correcto uso de las transacciones, tal como ocurre con en las bases de datos o con los monitores en las aplicaciones *multithread*, depende de la aplicación, ésta puede escoger cuándo esa violación es permisible.

Este argumento se basa en el cambio de enfoque realizado: la aplicación se basa ahora en componentes replicados, sin precisar un conocimiento de detalles de bajo nivel como es el cambio de vistas. La actualización del componente *directorio* que contiene las referencias a unos componentes *fichero* no se ve influida por el cambio en la composición del grupo de réplicas de ese componente *directorio* o componentes *fichero*, en tanto la actualización se realice apropiadamente en todos los componentes y las nuevas réplicas que se incluyan reciban un estado consistente.

Así, si un miembro inicia una transacción y recibe durante ésta un mensaje de otro miembro, la aplicación debe haber sido programada de tal manera que ese mensaje no afecte a la transacción. La transacción encierra, por lo tanto, una región crítica de código, pero ésta es una responsabilidad de los monitores y resulta lógico, consecuentemente, implementar las transacciones con monitores:

```
. . .
       domain.startTransaction(monitor);
       dirOrigin.remove(file);
       dirTarget.add(file);
       domain.endTransaction();
. . .
```

El inicio de la transacción implica la adquisición del monitor especificado, que es liberado cuando la transacción finaliza. Si el monitor no está libre, la transacción no puede comenzar.

Siguiendo con el mismo ejemplo, si el mensaje que la aplicación recibe durante esta transacción es el correspondiente a una operación de renombrado de un fichero tal como *`dirTarget.rename(file, newName)`*, implica que la aplicación no ha protegido correctamente la actualización del recurso *dirTarget* con un monitor, pues permite que dos réplicas modifiquen concurrentemente ese mismo recurso. En condiciones normales, las dos réplicas deberían haber escrito un código similar a:

Réplica A

```
. . .
       monitorTarget.lock();
       domain.startTransaction(monitorOrigin);
       dirOrigin.remove(file);
       dirTarget.add(file);
       domain.endTransaction();
       monitorTarget.unlock();
. . .
```

Réplica B

```
. . .
monitorTarget.lock();


dirTarget.rename(file,newName);

monitorTarget.unlock();
. . .
```

En estas condiciones, la réplica B no hubiera podido enviar el mensaje dado, pues estaría bloqueada esperando a adquirir el monitor asociado al directorio destino (o sería la réplica A la bloqueada). Los únicos mensajes que la réplica A puede recibir, siempre y cuando la aplicación se programe adecuadamente, son aquellos que no afecten al resultado de la transacción en curso: mensajes relativos a componentes no actualizados durante la transacción o mensajes que no actualicen componentes.

Es el mismo razonamiento empleado en el modelo de sincronía virtual. Este modelo permite la inclusión o expulsión dinámica de miembros, pero los miembros que pertenecen al grupo deben recibir los mismos mensajes en el mismo orden. Posteriormente, ese requisito de orden total puede reducirse y emplear órdenes menos restrictivos, siempre que una réplica sea incapaz de detectar sus

inconsistencias de estado con otras réplicas: si esta condición se cumple, los clientes de esas réplicas tampoco podrán observar las inconsistencias. Al emplear transacciones, el orden se altera y la consistencia entre miembros está sólo asegurada en los momentos donde no haya transacciones, siendo responsabilidad de la aplicación el ocultar esas inconsistencias temporales.

Sin embargo, la replicación de una aplicación específica puede aún requerir el empleo de orden total causal y comportamiento dinámico uniforme: si una réplica ha efectuado una determinada operación, las demás réplicas la deben efectuar igualmente, incluso si la primera ha caído. Si esa operación se realiza bajo una transacción, los demás miembros no tendrán ninguna información de las operaciones efectuadas si el miembro se cae antes de finalizarla, pues ésa es la finalidad buscada. Como consecuencia, este tipo de aplicaciones no admite el empleo de transacciones.

Las bases de datos permiten tres operaciones básicas con transacciones: iniciarlas, completarlas y abortarlas. Además, soportan transacciones anidadas [Moss85]. La segunda solución para las transacciones tiene un inconveniente con respecto a la primera planteada, basada en componentes capaces de deshacer sus cambios: una réplica no puede abortar una transacción.

La posibilidad de anidar transacciones es, por otra parte, muy conveniente; como cada transacción se realiza en torno a un monitor, implica, primero, la adquisición de nuevos monitores y, segundo, el bloqueo de los mensajes en tanto la última transacción no haya finalizado. Puesto que los monitores replicados se han definido como reentrantes, no es un problema emplear para una transacción anidada un monitor ya adquirido en una transacción externa.

Por otro lado, que una transacción bloquee los mensajes en la réplica que los envía, implica un problema con el empleo de monitores: los demás miembros deben saber que ese monitor ha sido adquirido, o podrían adquirirlo simultáneamente. Pero si esa adquisición del monitor precisa de comunicaciones con el grupo, no es posible su empleo durante las transacciones, lo que impide, a su vez, el empleo de transacciones anidadas. Ésta es la razón por la que el último listado de código no se escribió como:

Réplica A

```
. . .
       domain.startTransaction(monitorOrigin);
       monitorTarget.lock();
       dirOrigin.remove(file);
       dirTarget.add(file);
       monitorTarget.unlock();
       domain.endTransaction();
. . .
```

Réplica B

```
 . . .

monitorTarget.lock();

dirTarget.rename(file,newName);
monitorTarget.unlock();
```

El monitor *monitorTarget* es obtenido durante la transacción, pero la réplica B sólo recibe información de esa adquisición tras concluir la transferencia, cuando recibe a su vez información de su liberación. Al desconocer esa adquisición, ha podido realizar una actualización inconsistente sobre un recurso con acceso supuestamente exclusivo.

En la transferencia de estado se bloquean todos los mensajes entrantes en tanto la transferencia se realiza, pero la aplicación puede mejorar el rendimiento de la aplicación definiendo mensajes que pueden pasar ese bloqueo. De la misma forma, la aplicación puede definir mensajes que no son bloqueados durante la transacción, de forma que puedan emplearse los escenarios previamente descritos.

Las dos operaciones soportadas por un monitor replicado, *lock* y *unlock*, implican la definición de dos mensajes. El mensaje asociado a *lock* no debe ser bloqueado durante transacciones y es recibido por todos los miembros. Un monitor replicado está programado para reaccionar correctamente a la caída del miembro que lo posee, por lo que su exclusión de la transacción no implica posibles inconsistencias. El mensaje asociado a *unlock* sí debe ser bloqueado durante la transacción; en caso contrario, el miembro podría realizar, protegido por un monitor, una serie de cambios sobre uno o más componentes, y otra réplica, al adquirir el monitor liberado, realizar cambios incompatibles con los del primer miembro, que aún no ha finalizado su transacción y, por consiguiente, no ha hecho públicos sus cambios efectuados.

Esta solución al problema implica que los mensajes enviados en el dominio deben tener una propiedad que precise su comportamiento ante transacciones, de la misma forma que es necesaria otra propiedad para su comportamiento ante una transferencia de estado.

Hay, precisamente, una interacción importante entre transacciones y transferencia de estado: una réplica no debe coordinar una transferencia de estado en tanto no complete todas sus transacciones en curso. Este razonamiento se explica con la figura 9.7, donde una réplica inicia una transacción y, antes de concluirla, recibe una vista con un nuevo miembro al que debe coordinar. Antes de recibir la vista, la réplica envía un mensaje *m1* al grupo, que la transacción bloquea; sin embargo, la réplica procesa ese mensaje, actualizando el estado del componente *C*, que pasa a tener un estado {*m1*}. Si la transacción no impide la transferencia, envía inmediatamente su estado al nuevo miembro (en un mensaje que no se bloquea por la transacción), que recibe por tanto ese componente *C* con su estado actual, {*m1*}. A continuación, la réplica continúa su transacción, y envía un segundo mensaje *m2*, también bloqueado, que actualiza el estado de C a {*m1*,*m2*}. Al finalizar la transacción, el nuevo miembro recibe los mensajes que han sido encolados: *m1* y *m2*. Si procesa ambos mensajes, su estado pasa a ser {*m1*,*m1*,*m2*}, inconsistente con el del grupo salvo que *m1* sea idempotente. En caso contrario, debe obtener información adicional para ser capaz de descartar el mensaje *m1*. El escenario se

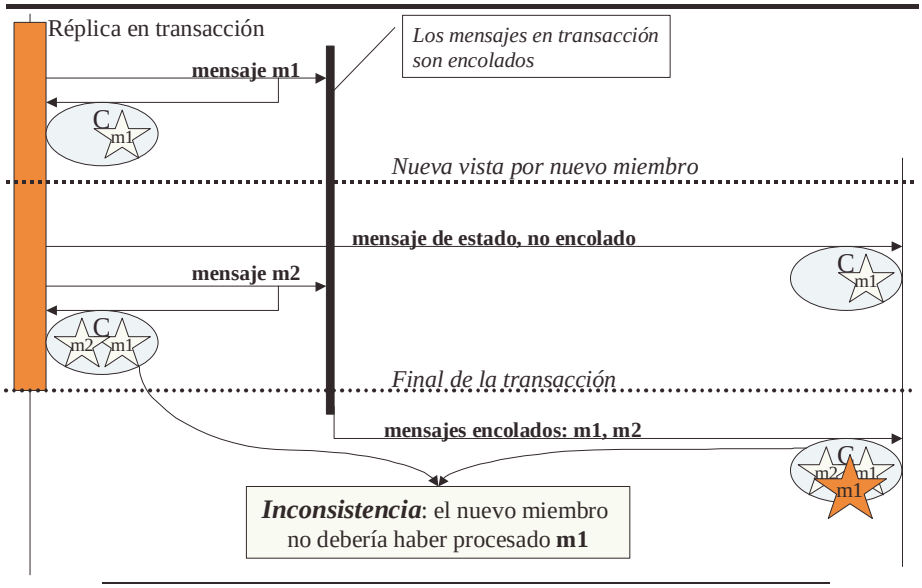

Figura 9.7. Interacción entre transacciones y transferencia de estado

complica aún más al considerar que la réplica en transacción puede también enviar mensajes que la transacción no bloquea.

Es posible complicar los mecanismos de transferencia de estado para solucionar este escenario, incluyendo cierta información adicional en los mensajes o, por ejemplo, bloqueando el mensaje de transferencia de estado durante la transacción. Pero impedir que un miembro que procesa una transacción sea el coordinador de una transferencia resulta además conveniente. Una transacción implica la adquisición de un monitor, y una réplica que adquiere un monitor bloquea a otras réplicas, por lo que resulta apropiado que un miembro en transacción (o simplemente poseyendo un monitor) no sea escogido como coordinador de una transferencia, o retrase ésta en tanto la finaliza.

Una consecuencia de definir un mensaje como no transaccionable es que tampoco puede ser bloqueado durante una transferencia de estado. Para comprobar esta afirmación, suponemos un caso donde un miembro del grupo está realizando una transferencia y un nuevo miembro se incluye en este mismo grupo. Aunque la transferencia no se inicia hasta finalizar la transacción, ambos miembros deben ya bloquear los mensajes del grupo. Un componente dado de este miembro ha definido un mensaje específico como no transaccionable y lo envía al grupo, sin que este envío sea bloqueado; sin embargo, el mensaje no está definido como no bloqueable durante las transferencias, con lo que ni alcanza al nuevo miembro, que lo encola, ni puede ser procesado por este miembro, que no puede entonces evolucionar en su transacción.

Consecuentemente, la definición de un mensaje como no transaccionable implica que el componente que lo emplea debe ser capaz de construir su estado aunque reciba mensajes en desorden (los bloqueados y no bloqueados en transferencia se reciben en distinto orden), lo que complica sus algoritmos de transferencia. De los componentes definidos en SenseiUMA, ninguno hace uso de este tipo de mensajes, limitándose, de momento, su empleo a los monitores.

Finalmente, existe un efecto lateral asociado al bloqueo de mensajes durante las transacciones: puesto que los mensajes no son enviados hasta la finalización de la transacción, se mejora el rendimiento de la aplicación. Cada operación sobre un componente se traduce, siguiendo las primeras secciones de este capítulo, en un mensaje al grupo y el bloqueo de la réplica en tanto el mensaje se recibe y se procesa. Bajo una transacción, esos bloqueos son resueltos inmediatamente, resultando en un código más rápido. Además, puesto que todos los mensajes son enviados al finalizar la transacción, se obtiene un beneficio adicional, pues los sistemas de comunicaciones fiables en grupo, incluido SenseiGMS, tienen un rendimiento mejor cuando una réplica envía sucesivos mensajes: la fiabilidad de las comunicaciones precisa que los mensajes sean enviados y confirmados (y descartados, según el sistema), pero el envío de un segundo mensaje implica la confirmación del anterior, ahorrándose el sistema un paso del protocolo. Dependiendo del tamaño de cada mensaje, es también posible agrupar los correspondientes a la transacción en un único mensaje, que es luego dividido en el dominio distinto.

9.9.1. Transacciones y el modelo de sincronía virtual

El modelo de componentes desarrollado en Sensei sobre el modelo de sincronía virtual ha impuesto la necesidad de transacciones para mantener la coherencia de los datos en caso de caídas. Sin embargo, las relaciones entre el modelo de transacciones y el modelo de sincronía virtual han sido ya ampliamente estudiadas con anterioridad, pues ambos modelos sirven, independientemente, para el mismo fin: la construcción de aplicaciones fiables.

Tanto los algoritmos de transacciones, como los empleados para las comunicaciones multipunto en el modelo sincronía virtual, deben resolver problemas de *consenso*. Se ha demostrado que existe un mecanismo básico, *DT-multicast*, con el que se pueden construir ambos modelos [Guerraoui95], partiendo de una implementación del protocolo de compromiso (*commit protocol*).

Sin embargo, la integración de ambos modelos se ha realizado normalmente mediante la implementación de uno sobre el otro. Por ejemplo, puesto que un mensaje fiable multipunto tiene por sí mismo semánticas transaccionales [Schiper96], es posible obtener la atomicidad requerida para una transacción encapsulándola en un único mensaje fiable multipunto. La propuesta de implementación de transacciones en SenseiDomains emplea este mismo enfoque.

Es más habitual emplear el modelo de sincronía virtual para incrementar la disponibilidad de sistemas basados en transacciones. En este caso, el sistema está basado en transacciones y el modelo de sincronía virtual queda generalmente oculto. Una opción [Little00] es que las transacciones se efectúen sobre sistemas replicados, pero el principal inconveniente es que cada una de las réplicas debe procesar cada transacción, incluso aquellas que no se completan y deben abortarse. Se obtiene un mejor rendimiento cuando los cambios se realizan sobre una única réplica que, a continuación, los propaga a las otras réplicas [Patiño00]; la transacción todavía puede abortarse, en caso de colisión con otras transacciones.

SenseiDomains también procesa cada transacción en una única réplica, propagando los cambios al final, pero el enfoque es radicalmente diferente. SenseiDomains no es un sistema basado en transacciones, sino un sistema basado en el modelo de sincronía virtual que soporta un mecanismo similar a las transacciones para preservar la consistencia de varios componentes en caso de caída de una réplica.

Una transacción debe verificar cuatro propiedades, denominadas *ACID* por su denominación en idioma inglés: Atomicidad (*Atomicity*), consistencia (*Consistency*), aislamiento (*Isolation*) y persistencia (*Durability*). Una transacción, o se completa y los cambios efectuados no se pierden incluso ante fallos o caídas, o se aborta, en cuyo caso el estado del sistema es el que tuviera antes de que se efectuara la transacción. La propiedad de aislamiento implica que las transacciones concurrentes no observan los cambios que se producen en otras transferencias hasta que se completan, y esta propiedad no se verifica en SenseiDomains para las transacciones anidadas ni las concurrentes iniciadas por una misma réplica.

En SenseiDomains, una transacción protege exclusivamente sobre caídas en un miembro y no es el mecanismo básico para obtener fiabilidad. Por esta razón, aislar los cambios producidos por transacciones iniciadas por una misma réplica no conlleva ningún beneficio al modelo, donde el empleo de las transacciones es totalmente transparente para la réplica que las efectúa (si una transacción no observa los cambios que se producen en otra, esta transparencia se pierde).

Además, la concurrencia de las transacciones no es tampoco una prioridad; un sistema basado en transacciones busca optimizar esta concurrencia, de tal forma que puedan procesarse tantas transacciones concurrentes como sea posible. Los mecanismos de control de concurrencia se clasifican como pesimistas u optimistas, y el empleo de monitores, aun siendo el más habitualmente empleado, entra en el grupo de mecanismos pesimistas. Sin embargo, hay varios tipos de monitores y los empleados en SenseiDomains son de lectura/escritura, considerados los menos concurrentes. Otros sistemas basados en transacciones, como *TransLib* [Jiménez99], emplean monitores basados en conmutación, donde es posible definir las condiciones que permiten conmutar dos operaciones y procesarlas concurrentemente. Por ejemplo, con un monitor de lectura/escritura, una operación de creación de un fichero en un directorio no es concurrente con ninguna otra

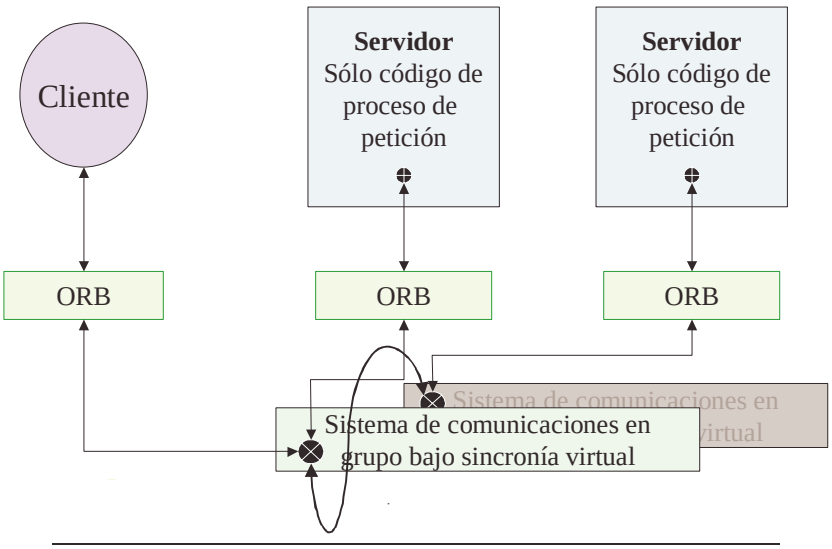

Figura 9.8. Modelo del servicio de tolerancia a fallos CORBA

operación de modificación de ese directorio; con un monitor basado en conmutación de operaciones, es posible crear un fichero y borrar simultáneamente otro, siempre que no tengan el mismo nombre.

9.10. Comparación con el modelo de CORBA

Como se expuso en capítulos anteriores, el servicio de tolerancia a fallos de CORBA soporta replicación activa y pasiva y, dentro de la pasiva, diferentes opciones de *backup*. En todos los casos la aplicación se programa bajo un mismo modelo, lo que permite cambiar el tipo de replicación sin modificar la aplicación, al menos teóricamente.

Esta filosofía implica que las réplicas se programan sin ningún conocimiento de las otras réplicas (aunque pueden obtener esa información) y es el servicio el encargado de emplear el modelo de sincronía virtual e implementarlo transparentemente para las réplicas.

La figura 9.8 muestra el modelo empleado en este servicio CORBA. Su especificación incluye la posibilidad de que el modelo de sincronía virtual que sustenta las operaciones en grupo se implemente por debajo del ORB, para así obtener un mejor rendimiento. En este caso, la petición del cliente es recibida directamente por este sistema de comunicaciones, que la envía a los demás miembros del grupo. Este sistema ordena todas las comunicaciones al grupo,

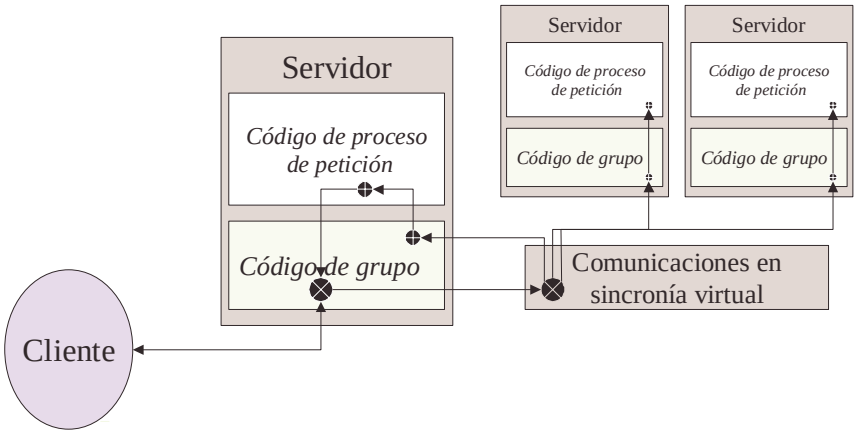

Figura 9.9. Esquema simple de la sincronización de la respuesta

enviándolas secuencialmente a las réplicas. Las réplicas no precisan incluir ningún código de grupo, sólo el código de procesado de las peticiones de los clientes.

Por otra parte, la figura 9.9 muestra de forma esquemática el modelo de sincronización expuesto en este capítulo: el cliente accede a un servidor, que incluye internamente el código para comunicar a sus réplicas la operación en curso, que se procesa en orden total con respecto a las demás operaciones que el grupo está realizando. Es posible diferenciar el código que procesa realmente la petición, similar o igual al que precisaría el servidor si no estuviera replicado, del código empleado para mantener la consistencia en el grupo.

Ambos modelos son muy similares, pero el servicio CORBA oculta todas las comunicaciones al grupo, lo que permite un diseño de las réplicas muy simplificado. En principio, la única diferencia entre un servidor no replicado y otro que lo está, es el soporte de las primitivas de transferencia de estado. Sin embargo, esta simplificación supone importantes inconvenientes.

El primer problema que presenta es la imposibilidad de optimizar las comunicaciones. Un importante número de operaciones en todo servidor lo constituyen las operaciones de consulta, que no actualizan su estado. Estas operaciones no precisan comunicarse al grupo y pueden procesarse inmediatamente. Bajo CORBA, esta optimización no es inicialmente posible; una posibilidad sería que esas operaciones se definieran en la interfaz IDL a partir de atributos de sólo lectura, que se traducen en operaciones de lectura de ese atributo. Sin embargo, la especificación CORBA no incluye la limitación de que estas operaciones no puedan

actualizar el estado del servidor, por lo que sería preciso una extensión propietaria en la interfaz IDL para realizar esta optimización.

Si el servidor es capaz de procesar una petición de consulta directamente, es posible suponer que el siguiente código podría producir un error:

```
server.assign(”Nombre”, ”Luis”);
if (server.read(”Nombre”) =! ”Luis”) {
       ERROR(”Incongruencia”);
}
```

Si la operación *read* se procesa inmediatamente, puede, de hecho, procesarse antes que la operación *assign*, con lo que el código anterior produciría un error. Sin embargo, el cliente que ejecuta el anterior código, lo hace secuencialmente, es decir, no ejecuta la operación *read* hasta que *assign* ha finalizado, lo que evita la anterior incongruencia.

El segundo problema está asociado al modelo de transferencia de estado, basado en mecanismos de archivo: obtener periódicamente el estado de la réplica que se almacena junto con todos los mensajes recibidos desde que se obtuvo el estado. El estado se puede obtener cada vez de forma completa o devolviendo simplemente los cambios efectuados.

La especificación del servicio no detalla estos mecanismos, en particular no precisa si todas las réplicas deben devolver periódicamente su estado o sólo se accede a una de las réplicas. Tampoco precisa cómo almacenar el estado. Una solución sería emplear un servicio de archivo tolerante a fallos, con lo que bastaría con acceder a una de las réplicas. Sin embargo, esta solución es complicada por la dificultad asociada a ese servicio de archivo y la solución básica es acceder a cada réplica y almacenar su estado directamente en la máquina donde se ejecuta.

Este tipo de transferencia es perfectamente lógico en caso de replicación pasiva, donde sólo una réplica conoce el estado de la aplicación. Pero la obtención del estado implica el bloqueo de la réplica que lo transfiere, con lo que bloquear todos los miembros del grupo de forma periódica sólo puede redundar en un mal rendimiento.

El tercer y más importante problema está ligado con la imposibilidad del servicio de soportar aplicaciones no deterministas. Pero, tal como ocurría con los mecanismos de automatización del proceso de replicación desarrollados al principio de este capítulo, el problema real se traduce en la imposibilidad de acceder a componentes externos. Volviendo al ejemplo del generador de números únicos aleatorios, todas las réplicas deben ser capaces de generar el mismo número aleatorio para mantener la consistencia del grupo. La solución planteada fue el empleo de un componente externo que generara ese número aleatorio, pero esta solución es, si no imposible, al menos muy difícil de implementar bajo CORBA.

Si todas las réplicas acceden al componente externo, ese componente recibe múltiples accesos: en el caso del generador de números, no generará uno solo, como se desea, sino tantos como réplicas haya. Consecuentemente, el servicio de tolerancia a fallos debe interceptar no sólo las llamadas a la réplica, sino las llamadas efectuadas por ésta, y coordinarlas con las llamadas de las otras réplicas del grupo para que se realice efectivamente un único acceso al componente dado.

Es posible realizar esta interceptación y coordinación de llamadas, aun cuando no está incluida en la especificación CORBA e implica un peor rendimiento del sistema. Además el problema persiste porque no todos los accesos externos se realizan a componentes CORBA y el servicio será incapaz de interceptar y coordinar llamadas a un servidor *ftp* o *http*, por ejemplo. E incluso con respecto a componentes CORBA, no todas las llamadas deben ser tampoco interceptadas.

Por ejemplo, una aplicación puede emplear una segunda aplicación que ofrece un servicio de traceado y que se ejecuta en la misma máquina que la primera. Asumiendo que esta segunda aplicación es CORBA, no tendría sentido que el servicio de tolerancia a fallos interceptara las llamadas de todas las réplicas y enviara sólo información de trazas de una de las réplicas: al realizar este traceado se pretende saber qué ocurre con cada réplica. Pero si ése fuera precisamente el objetivo, la aplicación no podría emplear ese servicio de traceado en el caso que su interfaz no fuera CORBA. Por ejemplo, con una interfaz basada en un protocolo propietario empleando directamente *sockets*.

Por lo tanto, el modelo CORBA soporta las mismas limitaciones que la automatización del proceso de replicación que se propuso en este capítulo, donde se producía un servidor replicado a partir de su interfaz y una instancia del servidor no replicado. Esa automatización dejaba sin cubrir un aspecto: las operaciones de transferencia de estado, que son precisamente las mismas operaciones que deben añadirse sobre un servidor no replicado al emplear el servicio de tolerancia a fallos de CORBA.

En el caso de la automatización, las limitaciones se resolvieron factorizando la aplicación en componentes, pero este paso no es posible bajo CORBA, al menos bajo la especificación actual. Como consecuencia, y salvo que se use replicación pasiva, el dominio de aplicación del servicio de tolerancia a fallos de CORBA se reduce a servidores sencillos.

9.11. Conclusiones

La metodología de construcción de aplicaciones tolerantes a fallos expuesta a lo largo de este capítulo se basa en el empleo de componentes replicados, intentando reducir al máximo la lógica asociada a las comunicaciones en grupo entre réplicas en una aplicación tolerante a fallos, que se programa mediante el empleo de esos componentes. Esta propuesta se fundamenta en dos ideas: el empleo de librerías

existentes de componentes ya probados y optimizados, y la posibilidad de automatizar el proceso de replicación de componentes sencillos.

Un servicio tolerante a fallos contiene, en este caso, servidores sin un conocimiento mutuo, donde toda interacción se realiza a través de los componentes replicados, que ofrecen la abstracción de componentes *compartidos*. Esta compartición de recursos implica que los servidores los acceden simultáneamente, por lo que precisan de soluciones que impidan resultados inconsistentes por actualizaciones concurrentes. Estas soluciones son los monitores replicados y las transacciones.

Para facilitar la utilización de esos componentes, su empleo se simplifica mediante el concepto de *dominios*, que permiten gestionarlos en su conjunto, y se soporta su dinamismo: los componentes pueden crearse y eliminarse dinámicamente, aumentando su semejanza a componentes *normales* y, por consiguiente, su facilidad de uso.

En comparación, el servicio de tolerancia a fallos de CORBA define un modelo muy simplificado, sin estos problemas de concurrencia. La diferencia estriba en la granularidad en la replicación: CORBA replica un servidor de forma monolítica mientras que la metodología expuesta en este capítulo permite definir esa replicación a nivel de componentes. Como consecuencia, el servicio de tolerancia a fallos CORBA tiene muy poca aplicabilidad en replicación activa, limitándose a servidores sencillos.

El empleo de componentes replicados implica la posibilidad de diseñar aplicaciones tolerantes a fallos a alto nivel, sin necesidad de emplear las primitivas directamente soportadas en el modelo de sincronía virtual, basadas en vistas y mensajes. Esta metodología se basa totalmente en este modelo, que queda sin embargo oculto para la aplicación.

Este capítulo ha definido las facilidades necesarias para soportar esta replicación basada en componentes y ha especificado su implementación en términos del soporte dado por un sistema de comunicaciones en grupo basado en sincronía virtual. De hecho, hace uso exclusivo de las primitivas de comunicaciones empleando orden total causal, sin precisar el soporte de otros órdenes menos restrictivos, lo que, sin embargo, implica la necesidad de que el grupo de comunicaciones sobre el que se implemente tenga un buen rendimiento con las comunicaciones con orden total.

Capítulo 10 - SENSEIDOMAINS

SenseiDomains es la implementación del soporte de la metodología desarrollada en el anterior capítulo, para el desarrollo de aplicaciones tolerantes a fallos mediante el empleo de componentes replicados.

La idea básica es que todas las comunicaciones deben realizarse a través de SenseiDomains, el cual las delega a su vez al sistema de comunicaciones en grupo subyacente. Los principales requisitos sobre dicho sistema de comunicaciones en grupo son el empleo de un modelo de sincronía virtual y el soporte de comunicaciones con orden total causal. Aunque el sistema de comunicaciones subyacente pueda incluir otros órdenes menos restrictivos en los mensajes, en la versión actual de SenseiDomains no se han considerado. La implementación actual se realiza sobre SenseiGMS, pero su dependencia con este sistema de comunicaciones se limita a su interfaz pública. No hay ningún detalle de implementación de SenseiGMS sobre el que se base SenseiDomains, lo que asociado a la generalidad de la interfaz de SenseiGMS, implica una fácil migración a otros sistemas de comunicaciones de grupo.

En lo referente a la transferencia de estado, SenseiDomains implementa la interfaz de aplicación mostrada en el capítulo 7, con pequeñas variaciones al no tratarse de una implementación del servicio de tolerancia a fallos de CORBA. Las variaciones introducidas no restan, sin embargo, flexibilidad a la transferencia, que presenta todas las características que se detallaron en aquel capítulo. Esta transferencia de estado se implementa totalmente en SenseiDomains, sin precisar que el sistema de comunicaciones preste un soporte específico de transferencia. El

capítulo 4 se centró en los sistemas de comunicaciones en grupo existentes, mostrando el soporte que presentan a la transferencia de estado, soporte que es, cuando existente, inferior al que precisa SenseiDomains.

Además de la limitación al empleo de comunicaciones con orden total, SenseiDomains no soporta el particionado del grupo. Esa falta de soporte no viene dada por SenseiGMS, que tampoco lo soporta, sino porque, tanto la metodología expuesta en el capítulo anterior, como el modelo de transferencia de estado presentado en los capítulos 5 al 7, son insuficientes para aplicaciones que permiten particiones del grupo.

La metodología expuesta en el capítulo anterior se centró en su aplicabilidad bajo CORBA, pero es igualmente válida para JavaRMI. Al igual que SenseiGMS, SenseiDomains define una interfaz común para JavaRMI y CORBA, compartiendo los algoritmos de implementación. En este capítulo explicamos exclusivamente la interfaz OMG/IDL; esta interfaz es considerablemente más extensa que la necesaria para SenseiGMS, al igual que la implementación resulta también mucho más compleja, por lo que incluir también la descripción de las clases e interfaces Java precisas para trabajar con JavaRMI supondría una extensión innecesaria, ya que sería repetitivo.

La interfaz CORBA se reparte físicamente en ocho partes que agrupan de forma lógica las distintas definiciones; a cada uno de los ficheros le corresponde una sección en este capítulo, para mantener esa agrupación lógica. Todas las definiciones se incluyen en el paquete Java *sensei.middleware.domains*, lo que implica que están definidas en el módulo *domains*, incluido en el módulo *middleware*, definido a su vez en el módulo *sensei*.

El nombre SenseiDomains viene dado por el elemento principal de la metodología, el dominio, que permite agrupar y gestionar los componentes replicados: todas las características de la metodología se definen en torno a este concepto de dominio. La interfaz definida para el manejo de dominios es *DomainGroupHandler*; sin embargo, esta interfaz depende [figura 10.1] en parte de las demás entidades de SenseiDomains, por lo que no se define hasta las últimas secciones.

10.1. Excepciones

En `DomainExceptions.idl` se define una única excepción, correspondiente a las operaciones efectuadas por el dominio bajo un estado incorrecto. Son posibles otras excepciones, pero asociadas a características específicas del dominio, definidas junto a esas características en las secciones siguientes.

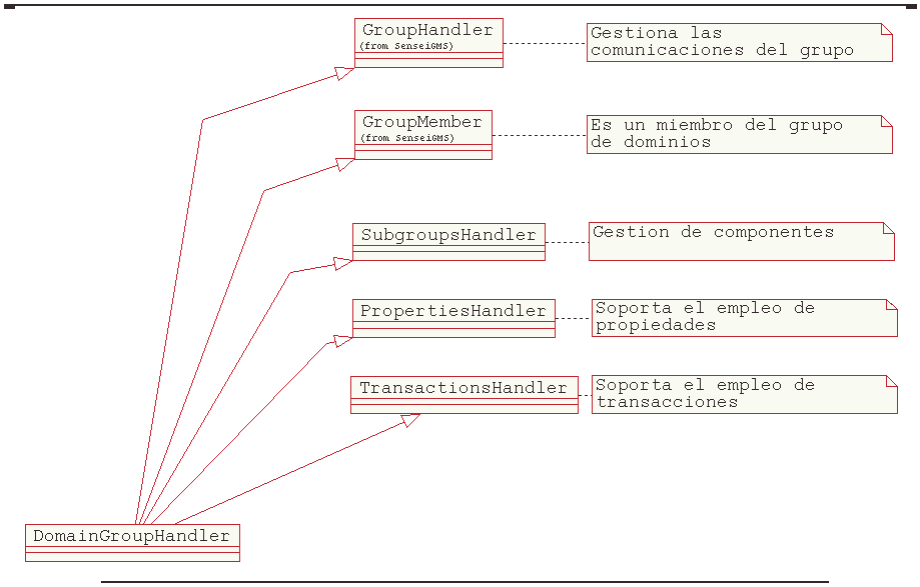

Figura 10.1. Jerarquía de herencia en *DomainGroupHandler*

La mayoría de las operaciones en el dominio son susceptibles de recibir una excepción *MemberStateException*, definida como:

```
enum MemberStateExceptionReason
{
        MemberNotJoined,
        MemberJoined,
        MemberWithoutState,
        MemberExcluded
};

exception MemberStateException
{
        MemberStateExceptionReason reason;
};
```

10.2. Transferencia de estado

La definición de la funcionalidad de transferencia de estado se incluye en `StateTransfer.idl`. El capítulo 7 desarrolló la interfaz de aplicación de un servicio tolerante a fallos CORBA con una transferencia de estado flexible; SenseiDomains no es una implementación de este servicio CORBA, por lo que la interfaz soportada es ligeramente diferente, con el objetivo de integrarse con el modelo genérico de grupos

que emplea SenseiGMS. En cualquier caso, la flexibilidad es la misma, así que la explicación de las siguientes definiciones no es exhaustiva, para no duplicar la información ya dada en el capítulo 7.

Esta definición incluye tipos directamente definidos en la especificación del servicio de tolerancia a fallos de CORBA, pero redefine algunos, bien para adaptarse a la interfaz de SenseiGMS, bien para añadir alguna funcionalidad. Estos tipos son:

```
abstract valuetype State {};

typedef GroupMemberId Location;
typedef sequence <Location> Locations;

interface Checkpointable : GroupMember
{
        State getState();
        void setState (in State s);
};

typedef string Name;
typedef string Value;
struct Property
{
        Name nam;
        Value val;
};
typedef sequence <Property> Properties;
```

- *State*: es el estado a transferir entre réplicas, definido en CORBA como un tipo *any*. SenseiDomains emplea un *valuetype* de tal forma que las aplicaciones puedan definir su propio tipo y no prescindan del soporte de tipado del lenguaje.
- *Location*: define la identidad de cada miembro del grupo, que viene dada en CORBA por una cadena de caracteres (`CosNaming::Name`). SenseiGMS define estas identidades con *GroupMemberId*, y empleamos este mismo tipo para *SenseiDomains*. De la misma forma, *Locations* define un conjunto de miembros.
- *Checkpointable*: define la interfaz que todos los miembros de un grupo con soporte de transferencia de estado deben implementar, correspondiente al soporte más simple en SenseiDomains. En CORBA, las operaciones emplean nombres separados, como *get_state* en lugar de *getState*, y la interfaz no extiende, obviamente, la definición SenseiGMS de *GroupMember*.
- *Property*: las propiedades asociadas al grupo, o a un miembro dado, se definen como pares {nombre, valor}. Bajo SenseiDomains, tanto el nombre como el valor de la propiedad se asocian a una cadena de caracteres para simplificar su tratamiento, mientras que en CORBA los valores son tipos *any*.

La transferencia de estado planteada en los capítulos anteriores soporta las transferencias en varias etapas. Como se explicó en la sección 7.2.1, la aplicación debe definir una entidad que contenga la información necesaria para sincronizar las diferentes etapas. SenseiDomains sólo emplea ese tipo para decidir cuándo una transferencia ha finalizado, predefiniendo, por lo tanto, una parte de esa entidad:

```
valuetype PhaseCoordination
{
       public boolean transferFinished;
};
```

La aplicación define su propia coordinación de fase, extendiendo generalmente este tipo previo. En cada etapa, debe asignar el valor correspondiente al campo *transferFinished* de la fase, donde un valor *true* marca el final de la transferencia.

Un componente incluido en el dominio define el soporte de transferencia de estado que precisa a partir de la interfaz que ese componente implementa. Las interfaces de transferencia de estado que puede implementar son:

- *StateHandler*: completo soporte de transferencias en varias etapas, admitiendo la interrupción de las transferencias por cambios de vistas.
- *BasicStateHandler*: soporte de transferencias en varias etapas, pero los cambios de vistas suponen la reiniciación completa de la transferencia.
- *ExtendedCheckpointable*: soporte básico, similar al definido en CORBA.
- *Checkpointable*: soporte básico, tal como lo define CORBA.
- Sin implementar ninguna de las anteriores interfaces: sin soporte de transferencia.

Estas interfaces se especifican a continuación, sin incluir *Checkpointable*, que ya se ha definido anteriormente. La única adición con respecto a la información del capítulo 7 es la interfaz *ExtendedCheckpointable*, que sólo se diferencia de *Checkpointable* en la operación *assumeState*. Esta operación permite que un componente pueda decidir cuál es su estado si no hay ninguna otra réplica que se lo pueda transferir.

```
interface ExtendedCheckpointable : Checkpointable
{
    void assumeState ();
};

interface BasicStateHandler : GroupMember
{
    void assumeState ();
    void startTransfer (in Locations joiningMembers, inout PhaseCoordination phase);
    State getState (inout PhaseCoordination phase);
```

```
    void setState (in State s, in PhaseCoordination phase);
    void stopTransfer (in Locations joiningMembers, in boolean transferFinished);
};

interface StateHandler : BasicStateHandler
{
    void syncTransfer (in Location coordinator, inout PhaseCoordination phase);
    void interruptTransfer (inout PhaseCoordination phase);
    void continueTransfer (in Locations joiningMembers,
                           inout PhaseCoordination coordinator,
                           in PhaseCoordination joining);
};
```

Cada componente de un mismo dominio puede implementar una interfaz diferente, no es preciso que todos los componentes requieran el mismo soporte de transferencia de estado. Además, esta interfaz soporta las transferencias concurrentes: por ejemplo, cuando una transferencia comienza, la operación *startTransfer* incluye el conjunto de miembros que recibirán el estado. Sin embargo, la implementación actual se limita a transferencias en serie: no es posible que un miembro transfiera el estado a varios miembros concurrentemente.

SenseiDomains permite la elección del coordinador de cada transferencia mediante las siguientes definiciones:

```
struct CoordinatorInformation
{
    Location statusMember;
    Locations currentCoordinations;
};
typedef sequence <CoordinatorInformation> CoordinatorInformationList;

interface CoordinatorElector
{
    Location getCoordinator (in Location loc, in CoordinatorInformationList info);
};
```

Cada componente del dominio puede requerir un soporte de transferencia de estado diferente pues, en caso contrario, se limitaría la reusabilidad de los componentes al poder emplear conjuntamente sólo aquellos que definieran el mismo soporte de transferencia. Sin embargo, no es posible que cada componente decida el coordinador para su transferencia: la aplicación debe registrar un único objeto que implemente la interfaz *CoordinatorElector*, con la que elige un coordinador cada vez que se precisa una transferencia. Para realizar esta elección, la aplicación recibe una lista con todos los miembros con estado y, para cada uno de esos miembros, las transferencias que están realizando en ese momento dado, de tal forma que pueda

realizar balanza de carga (esta posibilidad no se incluye en la interfaz dada en el capítulo 7).

No es necesario que la aplicación implemente esta interfaz: en caso contrario, la elección la realiza el dominio mismo, como se explica más adelante.

El último tipo definido especifica el comportamiento de la aplicación ante cambios de estados:

```
enum BehaviourOnViewChanges
{
       MembersOnTransferExcludedFromGroup,
       StatelessMembersDoNotBelongToGroup,
       StatelessMembersBelongToGroup
};
```

Durante la transferencia, SenseiDomains bloquea los mensajes a los miembros que la realizan, salvo que esos mensajes se definan como de no bloqueo. Ante un cambio de vista, *BehaviourOnViewChanges* define las acciones a tomar con esos mensajes:

- *MembersOnTransferExcludedFromGroup*: los miembros en la transferencia se consideran excluidos del grupo, por lo que no se ven afectados por los cambios de vista. Los mensajes quedan encolados y sólo se envían a la aplicación cuando la transferencia ha concluido. Permite implementar las transferencias más sencillas, pero no respeta completamente el modelo de sincronía virtual. Puesto que la transferencia no se interrumpe, no resulta lógico implementar la complicada interfaz *StateHandler* cuando se define este comportamiento.
- *StatelessMembersDoNotBelongToGroup*: considera que los miembros sin estado no pertenecen al grupo. Ante un cambio de vista, la transferencia se interrumpe y el miembro que la coordina recibe los mensajes que se hubieran bloqueado, que debe procesar antes de instalar la nueva vista. El miembro sin estado, sin embargo, no recibe esos mensajes, que son descartados.
- *StatelessMembersBelongToGroup*: todos los miembros se consideran incluidos en el grupo, siendo ésta la única opción que permite una solución que se adhiere completamente al modelo de sincronía virtual. Se comporta como en el anterior caso, pero el nuevo miembro recibe también los mensajes bloqueados antes de procesar la nueva vista.

Puesto que SenseiDomains soporta transacciones que, como mostró el capítulo anterior no se adhieren estrictamente al modelo de sincronía virtual, el comportamiento por defecto ante vistas es el que soporta una transferencia más simple: *MembersOnTransferExcludedFromGroup*.

Los protocolos que soportan la interfaz de transferencia de estado fueron definidos como *push* o *pull*, según el miembro sin estado recibiera automáticamente desde otro miembro el estado o tuviera que solicitarlo explícitamente. La interfaz

pública en ambos casos es la misma, luego la elección del protocolo afecta únicamente al rendimiento. Se demostró que, excepto en casos concretos y con soporte multipunto del medio empleado, el protocolo *push* ofrece mejores rendimientos, por lo que la versión actual de SenseiDomains implementa únicamente este protocolo.

10.3. Propiedades

Al estudiar la transferencia de estado se realizó una distinción entre el estado de un miembro y sus propiedades, y se definió la interfaz necesaria para soportar éstas. Esa interfaz estaba muy ligada al servicio de tolerancia a fallos CORBA, empleando entidades básicas de ese servicio, como el *PropertyManager* o las *GenericFactory* (sección 7.4). La solución aportada pretendía integrarse en la especificación hecha del servicio CORBA, introduciendo los mínimos cambios posibles para obtener la flexibilidad deseada. Esta integración no es precisa en SenseiDomains, e introducir las entidades necesarias supondría una complejidad adicional sin aportar ninguna ventaja.

Por esta razón, `Properties.idl` define las interfaces necesarias para soportar la funcionalidad de propiedades descrita en el capítulo 7, pero sin emplear las entidades CORBA. Este cambio permite, además de incrementar la funcionalidad, un manejo de las propiedades más flexible que en la descripción inicial. Posteriormente a esta primera descripción hemos introducido el concepto de dominio, agrupando componentes que podrían definir sus propias propiedades. Sin embargo, las propiedades deben emplearse a nivel de dominio, no a nivel de componente, y asociándose, por tanto, al servidor replicado y no a los componentes que lo constituyen.

La estructura *MemberProperties* asocia a un miembro dado un conjunto de propiedades:

```
struct MemberProperties
{
       GroupMemberId member;
       Properties props;
};
typedef sequence <MemberProperties> MemberPropertiesList;
```

De forma genérica, cuando una entidad debe usarse en listas se define un tipo específico, identificado para *MemberProperties* como *MemberPropertiesList*.

El capítulo 6 desarrolló los protocolos de bajo nivel que soportan la transferencia de estado y de propiedades entre miembros. El empleo de propiedades afecta a esos protocolos, que deben emplear más mensajes y con más información, incluso si un grupo específico no hace uso de esas propiedades. Para optimizar este

caso e impedir que el rendimiento de un grupo se resienta por una facilidad que no emplea, es posible especificar que el dominio no utiliza propiedades, en cuyo caso no se permite emplear las operaciones de manejo de esas propiedades. Si se emplean, se obtiene un error, asociado a la siguiente excepción:

```
exception PropertiesDisabledException
{
};
```

Todos los miembros del dominio deben utilizar la misma política de empleo de propiedades: o todos los miembros soportan propiedades o ninguno lo hace. Si se emplean propiedades, un miembro del dominio puede recibir notificaciones de cambios en las propiedades de otros miembros. Esta notificación no incluye los cambios, sólo su ámbito:

```
interface PropertiesListener
{
       void propertiesUpdated (in Location loc);
};
```

Las propiedades se modifican y se consultan empleando la interfaz *PropertiesHandler*:

```
interface PropertiesHandler
{
       void enableProperties() raises (MemberStateException);
       boolean arePropertiesEnabled();
       void setPropertiesListener(in PropertiesListener listener)
              raises (PropertiesDisabledException);
       MemberPropertiesList getAllProperties ()
              raises (MemberStateException, PropertiesDisabledException);
       MemberPropertiesList getPropertyForAllMembers (in Name nam)
              raises (MemberStateException, PropertiesDisabledException);
       Properties getMemberProperties (in Location loc)
              raises (MemberStateException, PropertiesDisabledException);
       Value getMemberProperty (in Location loc, in Name n)
              raises (MemberStateException, PropertiesDisabledException);
       Properties getProperties() raises (PropertiesDisabledException);
       Value getProperty(in Name n) raises (PropertiesDisabledException);
       void setProperties (in Properties props)
              raises (MemberStateException, PropertiesDisabledException);
       void addProperties (in Properties props)
              raises (MemberStateException, PropertiesDisabledException);
       void removeProperties (in Properties props)
              raises (MemberStateException, PropertiesDisabledException);
};
```

Estas operaciones se dividen en cuatro tipos. El primer tipo agrupa las operaciones que afectan al manejo global de propiedades:

- *enableProperties*: por defecto, el dominio no soporta propiedades, que deben habilitarse explícitamente, siendo imposible deshabilitarlas posteriormente. Además, esta operación debe efectuarse antes de incluir al miembro en un grupo, momento en que se inician los protocolos entre miembros, o se recibe una excepción *MemberStateException*.
- *arePropertiesEnabled*: método de consulta, permite conocer si las propiedades han sido habilitadas.
- *setPropertiesListener*: registra la instancia *PropertiesListener* que recibe las notificaciones de cambios de propiedades. Esta instancia puede modificarse en cualquier momento, salvo que no se hayan habilitado las propiedades. El argumento puede ser nulo para dejar de recibir notificaciones.

El resto de operaciones sólo pueden ser invocadas si las propiedades han sido habilitadas. El segundo grupo de operaciones permite modificar las propiedades de un miembro del grupo. Debe notarse que las propiedades de un miembro específico sólo pueden ser modificadas por ese miembro y siempre que no haya sido expulsado del grupo, en cuyo caso se lanzaría una excepción *MemberStateException*:

- *setProperties*: fija el conjunto de propiedades, eliminando toda propiedad previamente definida.
- *addProperties*: añade las propiedades dadas, sobrescribiendo las ya existentes con el mismo nombre.
- *removeProperties*: elimina las propiedades especificadas. No se considera un error eliminar una propiedad no definida.

Las operaciones que permiten observar las propiedades del miembro propio conforman el tercer grupo:

- *getProperties*: devuelve el conjunto de propiedades definidas para el propio miembro.
- *getProperty*: devuelve una propiedad específica del propio miembro; si no se ha definido tal propiedad, se devuelve una referencia nula.

El último grupo incluye a las operaciones que devuelven las propiedades de otros miembros del grupo. Como sólo es posible obtener información de otros miembros si se ha completado la transferencia de estado, es un error invocar a las siguientes operaciones sobre miembros sin estado:

- *getAllProperties*: devuelve las propiedades de todos los miembros.
- *getPropertyForAllMembers*: devuelve el valor de una propiedad específica en cada uno de los miembros del grupo.
- *getMemberProperties*: devuelve las propiedades de un miembro específico.

- *getMemberProperty*: devuelve una propiedad específica de un miembro determinado.

Gran parte de la funcionalidad proporcionada puede parecer superflua, pues los tres últimos grupos de operaciones podrían sintetizarse en dos operaciones: *setProperties*, *getProperties*. Como contrapartida, se obtiene una mayor facilidad de uso y un mejor rendimiento en casos específicos, como por ejemplo, al emplear *getPropertyForAllMembers*.

La interfaz *DomainGroupHandler* implementa la interfaz *PropertiesHandler*, es decir, todo el manejo de propiedades se realiza a través de la definición de dominio.

10.4. Componentes

En `SubgroupsHandler.idl` se define la gestión de componentes propuesta en el capítulo anterior, que emplea indistintamente los términos subgrupo y componente. Esta propuesta distinguía entre dos tipos de componentes, estáticos y dinámicos, según fuera su ciclo de vida. Ambos tipos son identificados en el dominio empleando una identidad, pero los primeros deben suministrar esa identidad mientras que es el dominio el que asigna la identidad de los segundos.

Las identidades de los componentes se definen como enteros en SenseiDomains, que incluye, además, una limitación adicional, al diferenciar los rangos de identidades que pueden emplearse para componentes estáticos y dinámicos. Se define, también, una identidad de componente universal, aplicable a todos los subgrupos del dominio:

```
typedef long SubgroupId;
typedef sequence<SubgroupId> SubgroupIdsList;
const long EVERY_SUBGROUP = 0;
const long MAX_STATIC_SUBGROUP_ID = 65535;
```

Al limitar los rangos de identidades, intentar registrar un componente con una identidad inválida o ya usada, provoca un error, para lo que define una excepción específica *SubgroupsHandlerException*, usada también al emplear componentes dinámicos erróneamente:

```
enum SubgroupsHandlerExceptionReason
{
       SubgroupIdAlreadyInUse,
       InvalidStaticSubgroupId,
       InvalidDynamicSubgroupId,
       DynamicBehaviourNotRegistered
};
exception SubgroupsHandlerException
{
```

```
        SubgroupsHandlerExceptionReason reason;
};
```

Si el dominio emplea componentes dinámicos, un miembro lo crea tras haberse incorporado al grupo y los demás miembros deben crear ese mismo componente en su espacio de memoria. Para que estos miembros conozcan el componente que deben crear, el primero debe enviar cierta información que le permita a la aplicación seleccionar el tipo de componente adecuado. Esta información se define a partir del tipo *DynamicSubgroupInfo*, definido simplemente como:

```
valuetype DynamicSubgroupInfo
{
};
```

La aplicación debe definir un tipo concreto, a partir del anterior, que incluya esa información, que es totalmente opaca para SenseiDomains. Como facilidad adicional, se incluye un tipo concreto para el caso en que la información pueda darse mediante una cadena de caracteres:

```
valuetype DynamicSubgroupInfoAsString : DynamicSubgroupInfo
{
        public string info;
};
```

El empleo de componentes dinámicos implica que la aplicación debe suministrar funcionalidad adicional que permita la creación de componentes en demanda, así como su eliminación. Esta funcionalidad se proporciona al crear y registrar un objeto que implemente la interfaz *DynamicSubgroupsUser*:

```
interface DynamicSubgroupsUser
{
    GroupMember acceptSubgroup (in SubgroupId id,
                                in DynamicSubgroupInfo info);
    GroupMember subgroupCreated(in GroupMemberId creator,
                                in SubgroupId id,
                                in DynamicSubgroupInfo info);
    void subgroupRemoved       (in GroupMemberId remover,
                                in SubgroupId id,
                                in DynamicSubgroupInfo info);
};
```

- *subgroupRemoved*: comunica a la aplicación que un miembro del grupo ha eliminado un componente dado. La razón de esa eliminación, si es necesaria, se incluye mediante un parámetro de tipo *DynamicSubgroupInfo*.
- *acceptSubgroup* y *subgroupCreated*: ambas operaciones se emplean para solicitar a la aplicación que cree un componente dado. La primera operación se invoca

cuando el miembro se une al grupo y recibe información de los componentes dinámicos ya existentes, mientras que la segunda se invoca cuando el miembro ya pertenece al grupo. La diferencia es que sólo en el primer caso el componente recibe una transferencia de estado.

La distinción entre *acceptSubgroup* y *subgroupCreated* se comprende mejor con un ejemplo. Un componente que implementa un árbol binario replicado se crea vacío, sin elementos, por lo que no es necesario que se transfiera información a las demás réplicas cuando se crea dinámicamente. En este caso, esas réplicas reciben una llamada *subgroupCreated* con la que crean un componente *vacío*. Si en algún momento se incluye un nuevo miembro en el grupo, recibe la solicitud *acceptSubgroup* para crear ese componente pero, en este caso, su estado vacío no es consistente con el grupo y debe recibir el estado desde otra réplica.

Esta aproximación es obviamente una generalización, pues hay componentes que pueden necesitar para su inicialización de una información dada. En ese caso, esta información puede incluirse en el tipo *DynamicSubgroupInfo*, siempre y cuando su tamaño no sea considerado *grande* pues, en caso contrario, el componente debería definir sus propios métodos de inicialización de estado. Por ejemplo, en un sistema replicado de ficheros, el componente *fichero* debería necesitar para su construcción el nombre asignado. Si se pretende crearlo directamente con contenido, el contenido de un fichero podría fácilmente exceder los megabytes, en cuyo caso es difícilmente defendible la inclusión de esa información en un objeto de tipo *DynamicSubgroupInfo* y su replicación en un único paso: el componente debe definir sus propias operaciones para soportar esta inicialización.

La gestión de componentes se realiza a través de la interfaz *SubgroupsHandler*:

```
interface SubgroupsHandler
{
        void setDynamicSubgroupsUser(in DynamicSubgroupsUser user)
                raises (MemberStateException);
        void registerSubgroup(in SubgroupId uniqueSubgroupId, in GroupMember subgroup)
                raises (MemberStateException, SubgroupsHandlerException);
        void castSubgroupCreation(in DynamicSubgroupInfo info)
                raises (MemberStateException, SubgroupsHandlerException);
        SubgroupId createSubgroup(in DynamicSubgroupInfo info, in GroupMember subgroup)
                raises (MemberStateException, SubgroupsHandlerException);
        boolean removeSubgroup(in SubgroupId id, in DynamicSubgroupInfo info)
                raises (MemberStateException, SubgroupsHandlerException);
        SubgroupIdsList getSubgroups();
        GroupMember getSubgroup(in SubgroupId id);
        SubgroupId getSubgroupId(in GroupMember subgroup);
};
```

- *setDynamicSubgroupsUser*: registra la instancia que recibe las notificaciones de creación y eliminación de componentes dinámicos. Si el miembro no define

ninguna instancia, no soportará componentes dinámicos. Tal como ocurre con las propiedades, todos o ninguno de los miembros del grupo deben soportar componentes dinámicos.

- *registerSubgroup*: registra un componente estático en el dominio, especificando su identidad. Los componentes estáticos deben registrarse antes de incorporar el dominio a su grupo.
- *castSubgroupCreation*: comunica al grupo que se va a crear un nuevo componente dinámico. En esta operación no se incluye el componente que se crea, por lo que deberá crearlo posteriormente en demanda, como las demás réplicas, al recibir la notificación *subgroupCreated*.
- *createSubgroup*: creación de un componente dinámico, como en el anterior caso. Sin embargo, el dominio obtiene ahora ese componente y, por tanto, la aplicación no recibirá posteriormente la notificación *subgroupCreated*. Esta operación se bloquea hasta que cada réplica del grupo recibe la notificación de creación del nuevo componente.
- *removeSubgroup*: elimina el componente dado. Sólo es posible eliminar los componentes dinámicos.
- *getSubgroups*: permite obtener la lista de componentes definidos, tanto estáticos como dinámicos.
- *getSubgroup*: devuelve el componente asignado a la identidad especificada, o una referencia nula, si no existe tal componente.
- *getSubgroupId*: devuelve la identidad del componente dado. Si el componente no pertenece al dominio, el valor devuelto es *EVERY_SUBGROUP*.

Los componentes no deben emplear ninguna nueva interfaz, implementan la interfaz *GroupMember* definida en SenseiGMS, pues la abstracción que da el dominio es la de agrupar componentes que de otra forma deberían definir su propio grupo independiente de réplicas, implementando igualmente la interfaz *GroupMember*. La interfaz *DomainGroupHandler* implementa la interfaz *SubgroupsHandler* y gestiona, por lo tanto, los componentes del dominio.

10.5. Mensajes

SenseiGMS es el sistema de comunicaciones empleado para enviar fiablemente mensajes entre réplicas. Sin embargo, los mensajes se definen a nivel de aplicación y su contenido es totalmente opaco para SenseiGMS.

SenseiDomains sí necesita enviar información adicional en cada mensaje como, por ejemplo, la identidad del componente al que afecta. Por esta razón, se define en `DomainMessage.idl` un mensaje con toda la información necesaria; este

mensaje debe ser extendido por la aplicación para que incluya su información específica, tal como ocurría bajo SenseiGMS. La definición base es:

```
typedef long MessageId;

valuetype DomainMessage : Message
{
        public SubgroupId subgroup;

        public boolean unqueuedOnST;
        public boolean unTransactionable;

        public boolean waitReception;
        public MessageId  id;
};
```

DomainMessage extiende la definición de *Message*, necesario para su envío por SenseiGMS. Define cinco atributos, de los que sólo tres son asignados por la aplicación.

- *subgroup*: identidad del componente al que afecta. Un valor *EVERY_SUBGROUP* se emplea para enviar el mensaje a todos los componentes.
- *unqueuedOnST*: si se define este atributo como *true*, el mensaje no es bloqueado durante las transferencias de estado. Un ejemplo de este tipo de mensajes son los definidos por el propio dominio para implementar la transferencia de estado. Por defecto se inicializa a *false*, todos los mensajes son bloqueados.
- *unTransactionable*: al definir este atributo como *true*, el mensaje no es bloqueado durante transacciones. El atributo previo bloquea los mensajes en recepción, en tanto este atributo los bloquea en envío. Por defecto se inicializa también a *false*.
- *waitReception* y *id*: usados internamente por el dominio.

El capítulo anterior mostró la restricción de que todo mensaje no transaccionable tampoco puede ser bloqueado durante transferencias. Por lo tanto, el valor del atributo *unTransactionable* es irrelevante si *unqueuedOnST* contiene el valor por defecto.

10.6. Control de concurrencia

La concurrencia sobre componentes replicados se controla mediante monitores y transacciones, ambos definidos en `Monitor.idl`.

La interfaz que debe soportar un monitor replicado se detalló en la sección 9.6. Su explicación incluyó la posibilidad de un empleo incorrecto de estos monitores

como, por ejemplo, liberarlos cuando no han sido previamente adquiridos, lo que supone la necesidad de una excepción específica, *MonitorException*:

```
exception MonitorException
{
};
```

El empleo de un monitor replicado puede provocar errores adicionales relacionados con el estado del miembro en el grupo, que se identifican con la excepción *MemberStateException*, ya definida. La interfaz *Monitor* se define como:

```
interface Monitor
{
       void lock() raises (MemberStateException);
       void unlock() raises (MonitorException, MemberStateException);
};
```

Donde la operación *lock* adquiere el monitor y *unlock* lo libera. La semántica concreta se explica detalladamente en el capítulo anterior; específicamente, los monitores deben ser reentrantes, es decir, un monitor puede ser readquirido repetidas veces.

La interfaz *Monitor* no define, sin embargo, un componente específico "monitor replicado", sino las operaciones que todo monitor debe soportar. La distinción es importante; en Java, por ejemplo, todos los objetos tienen asociado un monitor, por lo que puede considerarse que extienden una interfaz *Monitor* (con la semántica propia de monitores del lenguaje). De la misma forma, es posible definir un componente replicado cualquiera que incluya su propio acceso a recursos e implemente su interfaz en términos de esta interfaz *Monitor*. En el caso de una *lista*, por ejemplo, sería posible definirla como:

```
interface List : Monitor
{
 . . .
};
```

Este componente permitiría un acceso sincronizado sin necesidad de otro componente adicional que implementara el monitor replicado.

SenseiDomains define e implementa, además de esta interfaz, un componente replicado específico con esta funcionalidad, denominado *GroupMonitor*:

```
interface GroupMonitor : Monitor, ExtendedCheckpointable
{
};
```

Este componente define simplemente la misma interfaz *Monitor* y el soporte de transferencia de estado que precisa: una aplicación puede crear un componente

GroupMonitor para realizar el sincronizado sobre accesos concurrentes. Es el único componente replicado definido en implementado en SenseiDomains, por su necesidad para el empleo de transacciones.

Una aplicación puede realizar sus propias implementaciones de monitores, siempre y cuando extiendan la interfaz *Monitor*. Debe tenerse en cuenta que el mensaje asociado a la adquisición del monitor es no transaccionable, luego no se bloquea durante las transacciones. Por esta razón, es necesario considerar la posibilidad de reconstruir el estado del monitor a partir de mensajes de adquisición y liberación que pudieran llegar en orden incorrecto. La implementación en SenseiDomains soporta directamente este escenario.

Las transacciones son mecanismos necesarios para el empleo de componentes, pues permiten conservar la consistencia en el grupo aun cuando la réplica caiga sin haber completado una actualización que afecte a múltiples componentes. Sin embargo, puesto que su uso viola el modelo de sincronía virtual, una aplicación estrictamente adherida al modelo debe poder evitarlas. En este caso, el empleo de transacciones provoca una excepción:

```
exception TransactionException
{
};
```

Como detalle de implementación, en SenseiDomains no es posible especificar explícitamente cuándo las transacciones son permisibles o no. Este permiso se determina indirectamente a partir del valor dado a *BehaviourOnViewChanges*, el comportamiento ante cambios de vista. Este valor se emplea para definir el tratamiento de mensajes durante transferencias ante cambios de vistas, definiendo tres modos de los que sólo uno es totalmente compatible con el modelo de sincronía virtual. Esta compatibilidad implica una mayor complejidad en la implementación de los algoritmos de transferencia, por lo que su empleo implica una adherencia al modelo y la imposibilidad, por tanto, de usar transacciones. Si un dominio no se define con valor *MembersOnTransferExcludedFromGroup*, que es el valor por defecto, se considera que no soporta transacciones.

Las transacciones se manejan mediante la interfaz *TransactionHandler*:

```
interface TransactionsHandler
{
       void startTransaction(in Monitor locker)
              raises (MonitorException, TransactionException, MemberStateException);
       void endTransaction()
              raises (MonitorException, TransactionException, MemberStateException);
};
```

La transacción se realiza en torno al monitor especificado al iniciarla, no siendo necesario especificarlo de nuevo para completarla, aunque se aniden las

transacciones. Los cambios realizados en los componentes del dominio durante la transacción no se reflejan en las demás réplicas si la réplica que los realiza se cae o es expulsada del grupo sin haber completado todas sus transacciones en curso. La implementación de SenseiDomains implica que realizar la modificación de dos o más componentes en una transacción ofrece mejores resultados que si no se emplea esa transacción.

La interfaz *TransactionHandler* es extendida por *DomainGroupHandler*, por lo que las transacciones se manejan a través del concepto de dominio, como se propuso en el capítulo anterior.

10.7. Dominios

Como se adelantó en la introducción, el concepto de dominio lo define la interfaz *DomainGroupHandler*, especificada en `DomainGroupHandler.idl`.

Un dominio es un grupo que aglutina lógicamente una serie de componentes, que de otra forma hubieran constituido otros tantos grupos independientes, permitiendo así su gestión conjunta. Al implementar una funcionalidad adicional, la interacción con el dominio es más complicada que la que había con el grupo en SenseiGMS. La expulsión de un miembro de su grupo de comunicaciones se debe, en todos los casos, a problemas, reales o virtuales, en sus enlaces de comunicaciones. La expulsión de un miembro de un grupo gestionado mediante un dominio puede, sin embargo, deberse a otras razones, generalmente violaciones del modelo de dominios. Por este motivo se ha definido un tipo que enumera estas razones:

```
enum DomainExpulsionReason
{
       GMSLeaveEvent,
       WrongPropertyAllowance
       WrongStaticSubgroupsComposition,
       WrongCoordinatorElectionPolicy,
       WrongBehaviourMode,
       WrongDynamicPolicy,
       WrongSubgroupsTypes,
       SubgroupError
};
```

- *GMSLeaveEvent*: la expulsión no la ha originado el dominio, sino el mismo substrato de comunicaciones, sea por petición de la aplicación o por un error de las comunicaciones en el grupo.
- *WrongPropertyAllowance*: el miembro que se añade al grupo no presenta los mismos permisos de propiedades que los miembros ya existentes. Todos los miembros del grupo deben definir la misma política de empleo de propiedades.

- *WrongStaticSubgroupsComposition*: el miembro que se añade al grupo no ha definido los mismos componentes e identidades que los miembros ya existentes.
- *WrongCoordinatorElectionPolicy*: es también obligatorio que o todos o ninguno de los miembros del grupo sean capaces de elegir al coordinador de una transferencia.
- *WrongBehaviourMode*: el miembro que se incluye en el grupo ha definido un valor diferente para *BehaviourOnViewChanges* que los miembros ya existentes.
- *WrongDynamicPolicy*: no se ha cumplido el requisito asociado a componentes dinámicos, por el que todos los miembros del grupo deben emplear la misma aproximación en cuanto a permitirlos o no.
- *WrongSubgroupsTypes*: cada componente se instancia en cada miembro del dominio y, para que la transferencia de estado sea posible, es necesario que esos componentes implementen las mismas interfaces de soporte de transferencia en cada una de las réplicas. En caso contrario, se obtiene este motivo en la expulsión.
- *SubgroupError*: el dominio ha recibido una excepción al acceder a uno de los componentes, lo que supone su autoexclusión del grupo.

El dominio genera también eventos asociados al grupo de componentes en su conjunto, que la aplicación puede recibir si crea y registra una instancia de la interfaz *DomainGroupUser*:

```
interface DomainGroupUser
{
       void domainAccepted(in GroupMemberId id);
       void domainExpulsed(in DomainExpulsionReason reason);
       void stateObtained(in boolean assumed);
       void offendingSubgroup(in SubgroupId subgroup, in string reason);
};
```

- *domainAccepted*: evento de aceptación en el grupo, incluye la identidad de grupo asignada. Por otra parte, todos los componentes del dominio reciben el evento *memberAccepted* definido en *GroupMember*, que incluye igualmente la identidad de grupo, siendo obviamente la misma para todos los componentes del dominio.
- *domainExcluded*: expulsión del grupo, incluye la razón aducida por el dominio para tal expulsión. Los componentes reciben el evento *expulsedFromGroup*, aunque no el motivo de la expulsión.
- *stateObtained*: obtención de estado, evento obtenido una vez que el dominio se considera con estado, bien porque lo haya recibido o bien porque se le considere el miembro primario de un grupo sin miembros con estado. En este segundo caso, se considera que el dominio *asume* su estado.

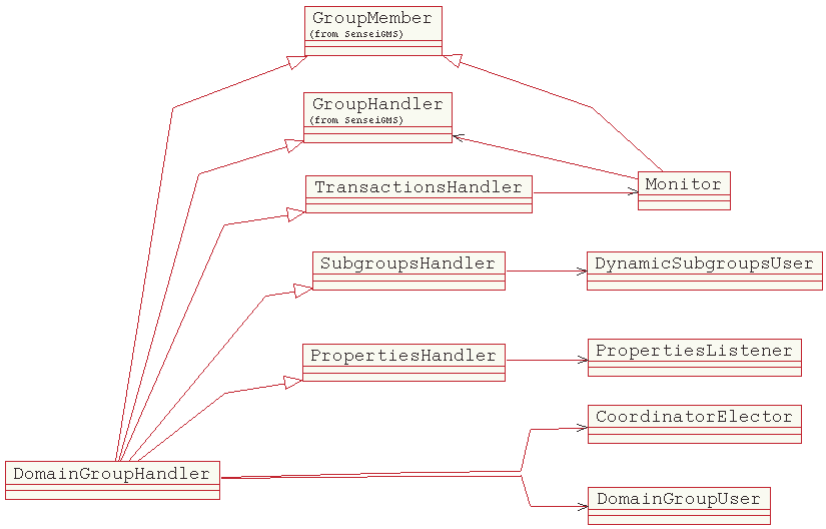

Figura 10.2. Principales dependencias de *DomainGroupHandler*

- *offendingSubgroup*: evento producido por un problema en un componente del dominio. A este evento le sigue eventualmente el de expulsión del grupo.

Finalmente, la interfaz *DomainGroupHandler* de define como:

```
interface DomainGroupHandler : GroupHandler, GroupMember,
        PropertiesHandler, SubgroupsHandler, TransactionsHandler
{
        void setBehaviourMode(in BehaviourOnViewChanges mode)
                raises (MemberStateException);
        void setCoordinatorElector(in CoordinatorElector elector)
                raises (MemberStateException);
        void setDomainGroupUser(in DomainGroupUser user)
                raises (MemberStateException);
        GroupMemberIdList getStatefulMembers() raises (MemberStateException);
        boolean syncCastDomainMessage(in DomainMessage message,
                                      in boolean normalProcessing)
                raises (MemberStateException);
        void syncCastDomainMessageProcessed()
                raises(MemberStateException, SyncCastDomainMessageProcessedException);
};
```

La figura 10.2 muestra las principales dependencias de esta definición:

- *DomainGroupHandler* se comporta como *GroupHandler*, que es la interfaz empleada en SenseiGMS para interaccionar con el grupo. Al emplear dominios,

toda la interacción con el grupo se realiza a partir de *DomainGroupHandler*, que implementa, por lo tanto, la funcionalidad de *GroupHandler* como, por ejemplo, la exclusión del grupo o el envío de mensajes. La interfaz *GroupMember*, que implementan los componentes, define el evento *memberAccepted*, por el que los componentes reciben la instancia *GroupHandler* cuando son aceptados en el grupo; al emplearlos en un dominio, la instancia que reciben en ese evento es la del dominio mismo.

- A su vez, *DomainGroupHandler* es el miembro de un grupo de réplicas, donde los otros miembros son instancias equivalentes del mismo dominio. Por esta razón debe implementar la interfaz *GroupMember*, según la define SenseiGMS.
- *DomainGroupHandler* es la entidad que gestiona las propiedades del dominio, implementando la interfaz *PropertiesHandler*. Admite el registro de una instancia *PropertiesListener* que recibe las notificaciones de cambios de propiedades en el grupo.
- *DomainGroupHandler* es el gestor de componentes, que implementa la interfaz *SubgroupsHandler*. Es capaz de soportar componentes dinámicos si se registra una instancia *DynamicSubgroupsUser* en el dominio.
- Por último, *DomainGroupHandler* extiende la interfaz *TransactionsHandler*, por lo que soporta el uso de transacciones sobre los componentes que gestiona.

Además del comportamiento definido por herencia, soporta la siguiente funcionalidad:

- *setBehaviourMode*; permite fijar el comportamiento de la transferencia de estado ante cambios de vistas. El comportamiento por defecto es el correspondiente a *MembersOnTransferExcludedFromGroup*; en caso de cambiarlo, el nuevo valor debe fijarse antes de incluir al dominio en el grupo.
- *setDomainGroupUser*: registra la instancia *DomainGroupUser* que recibe los eventos del dominio.
- *getStatefulMembers*: devuelve la lista de miembros del grupo con estado. La lista de todos los miembros, con o sin estado, es una información que ya se recibe a través de las vistas.
- *setCoordinatorElector*: registra la instancia *CoordinatorElector* empleada para la elección del coordinador. Si no se define ninguna instancia, el algoritmo por defecto en SenseiDomains distribuye las transferencias entre los miembros con estado, realizando una balanza de carga efectiva.

Una característica especial de SenseiDomains es que soporta una balanza de carga con pesos. Es posible asignar a cada miembro del dominio la propiedad "*weight*", cuyo valor determina su peso en esta balanza de carga. Por ejemplo, un miembro en el que esta propiedad tiene el valor "*2.0*" soporta el doble número de transferencias que otro con valor "*1.0*", mientras que uno con valor "*0.0*" sólo intervendrá en la transferencia si no hay otros miembros con estado con un valor

superior. De esta forma, es posible fijar el papel de cada miembro en la transferencia sin necesidad de crear un componente *CoordinatorElector*. Por ejemplo, un grupo que destina un miembro específico para las transferencias, de tal forma que los demás miembros no pierdan disponibilidad, puede asignar a ese miembro un peso grande, por ejemplo "*1000*", dejando el valor por defecto en los otros miembros.

Quedan dos operaciones de la interfaz *DomainGroupHandler* por explicar, relacionadas con el patrón de sincronización de mensajes que se desarrolló en la primera sección del capítulo anterior. Este patrón permite sincronizar la solicitud de un servicio en una réplica con el proceso asíncrono de obtener una respuesta tras comunicarse con las otras réplicas. Además, soporta dos variantes: la elaboración de la respuesta se hace en el mismo *thread* que realiza la comunicación con el grupo, o en el *thread* que recibe las comunicaciones del grupo. Si la solicitud de servicio requiere devolver algún valor al cliente, debe emplearse, preferiblemente, la primera variante; si no devuelve ningún valor, ambas variantes son adecuadas. Las operaciones son:

- *syncCastDomainMessage*: envía el mensaje al dominio y se bloquea este *thread* hasta que ese mismo mensaje se reciba. El parámetro *normalProcessing* indica dónde se procesa ese mensaje; si su valor es *true*, se procesa en la recepción del mensaje, y el bloqueo sólo termina cuando se ha completado su proceso. Si es *false*, el bloqueo finaliza al recibir el mensaje, lo que permite al proceso que inició la llamada *syncCastDomainMessage* despertarse y procesar el mensaje que envió. En este caso, el componente no observa la recepción del mensaje, y es necesario que, tras procesar el servicio, invoque la operación *syncCastDomainMessageProcessed*, o el dominio entero queda bloqueado.
- *syncCastDomainMessageProcessed:* completa la sincronización de un servicio que se procesa en envío.

Empleamos un ejemplo para clarificar su uso, basado en el que se utilizó en el capítulo anterior para exponer este patrón de sincronización. La operación *getNumber* devuelve un número secuencial, incrementado cada vez que se invoca sobre cualquier réplica del grupo, de tal forma que nunca se devuelvan dos números iguales. La operación *getNumber* se traduce en un mensaje *GetNumberMessage* que no incluye ninguna información adicional.

El siguiente listado muestra el código asociado a la primera alternativa:

```
public class NumberGenerator : . . .
{
       DomainGroupHandler domain;
       int lastGeneratedNumber;

       . . .

       public int getNumber()
```

```
        {
                domain.syncCastDomainMessage(new GetNumberMessage(), false);
                int result = ++lastGeneratedNumber;
                domain.syncCastDomainMessageProcessed();
                return lastGeneratedNumber;
        }

        public void processCastMessage(in GroupMemberId sender, in Message msg)
        {
                if (msg instanceof GetNumberMessage) {
                        ++lastGeneratedNumber;
                }
        }
}
```

- Al solicitarse un número, se envía el mensaje al grupo, comunicándole al dominio que no se emplea el procesado normal. En el procesado normal, tal como lo define SenseiGMS, todos los mensajes se reciben a través de *processCastMessage*. Si no se emplea procesado normal, el mensaje que una réplica envía no lo recibe.
- Cuando la operación de envío del mensaje vuelve, puede procesarse el servicio, incrementando simplemente el valor de la variable *lastGeneratedNumber* que almacena el último número dado.
- Tras procesarlo, y haber almacenado el valor a devolver al cliente, se comunica al dominio que el mensaje ha sido procesado. Con esta comunicación se indica al dominio que se puede procesar el siguiente mensaje del grupo, si lo hubiera.

El siguiente listado muestra el código asociado a la segunda alternativa.

```
public class NumberGenerator : . . .
{
        DomainGroupHandler domain;
        int lastGeneratedNumber;

        . . .

        public int getNumber()
        {
                domain.syncCastDomainMessage(new GetNumberMessage(), true);
                return lastGeneratedNumber;
        }

        public void processCastMessage(in GroupMemberId sender, in Message msg)
        {
                if (msg instanceof GetNumberMessage) {
```

```
                    ++lastGeneratedNumber;
                }
            }
}
```

- Al solicitarse un número en *getNumber*, se envía el mensaje al grupo, comunicándole al dominio que se emplea el procesado normal.
- El mensaje se recibe entonces a través del procedimiento normal, en *processCastMessage*. No hay diferencia entre la recepción de un mensaje propio o de otra réplica, se incrementa igualmente la variable *lastGeneratedNumber*.
- Cuando *processCastMessage* finaliza, se completa el bloqueo de *syncCastDomainMessage* en *getNumber*. El valor que devuelve al cliente es el que tiene en ese momento *lastGeneratedNumber*.

La segunda alternativa requiere menos código, aunque esa reducción es muy pequeña. Sin embargo, esta segunda solución es incorrecta, pues el manejo de *threads* impide saber si el desbloqueo sobre *syncCastDomainMessage* en *getNumber* se produce inmediatamente. Puede darse el caso de que se procese aún un segundo mensaje *GetNumberMessage* que modificaría el valor de *lastGeneratedNumber*, produciéndose un resultado erróneo. Es posible que la clase defina valores temporales para corregir este error pero, en ese caso, resulta más fácil emplear la primera alternativa.

Puesto que la invocación de *syncCastDomainMessageProcessed* implica que el dominio está bloqueado esperando a que la aplicación procese un mensaje, es un error invocarla en caso contrario, lo que se traduce en la excepción *SyncCastDomainMessageProcessedException*, definida simplemente como:

```
exception SyncCastDomainMessageProcessedException
{
};
```

Debe notarse que, puesto que *DomainGroupHandler* implementa la interfaz *GroupHandler*, también soporta la operación básica de envío de mensajes sin bloqueo: `boolean castMessage(in Message msg)`.

10.8. Implementación

La interfaz pública de SenseiDomains que se ha definido en las secciones anteriores, cubre las siguientes especificaciones:

- `DomainExceptions.idl`: excepciones genéricas del dominio.
- `DomainMessage.idl`: definición básica de mensaje de dominio.
- `Properties.idl`: soporte de propiedades.

- `DomainGroupHandler.idl`: definición de dominio.
- `Monitor.idl`: soporte de concurrencia, mediante réplicas y transacciones.
- `StateTransfer.idl`: soporte de transferencia de estado.
- `SubgroupsHandler.idl`: gestión de componentes.

Además, `InternalDomainMessages.idl` incluye todas las definiciones empleadas internamente por SenseiDomains pero que deben ser definidas a nivel de interfaz. Estas definiciones incluyen los mensajes internos y estructuras empleadas para la transferencia de estado.

SenseiDomains soporta todos los algoritmos de transferencia de estado desarrollados en los capítulos anteriores, a pesar de que el uso de transacciones es sólo compatible con el caso más simple. De la complejidad de estos algoritmos, sumada al amplio conjunto de facilidades asociadas al concepto de dominio, se deduce la dificultad de la implementación de SenseiDomains: tanto en complejidad como en tamaño (o el de las clases que lo implementan), es varias veces[8] superior a SenseiGMS.

El capítulo anterior definió la metodología que SenseiDomains implementa; sin embargo, hay detalles de esta implementación no directamente especificados en la metodología y aspectos de la metodología no recogidos en la implementación. Por ejemplo, la balanza de carga en la elección de coordinadores de transferencia empleando pesos definidos por la aplicación es una facilidad añadida.

Por otra parte, en SenseiDomains la posesión de un monitor no afecta a la transferencia de estado. Como se comprobó, una réplica que realiza una transacción no puede transferir su estado hasta concluir esa transacción. Aunque esta imposibilidad se podría eliminar complicando los algoritmos de tranferencia de estado, se trata de una propiedad conveniente: puesto que la transacción supone la adquisición de un monitor que puede bloquear a otras réplicas, es conveniente que termine esa transacción y ese bloqueo cuanto antes, lo que se facilita al no asignársele la tarea adicional de coordinar una transferencia de estado. El diseño de SenseiDomains permite inhibir la transferencia sobre miembros que realizan una transferencia y, aunque la implementación actual simplemente retrasa el comienzo de esa transferencia hasta que la transacción termina, es factible trasladar esa coordinación a otro miembro del grupo. Sin embargo, no hay ninguna alteración en el proceso de transferencia por el hecho de que un miembro posea un monitor, a pesar de que bloquea igualmente a las demás réplicas.

El motivo de esta diferencia es que, en SenseiDomains, un monitor es un componente más y el dominio no tiene conocimiento del estado de cada componente.

---

[8] La especificación JavaRMI de SenseiDomains define cinco veces más tipos que la de SenseiGMS. La implementación de los algoritmos de SenseiDomains emplea dos veces y media más líneas de código (excluyendo comentarios) que en SenseiGMS.

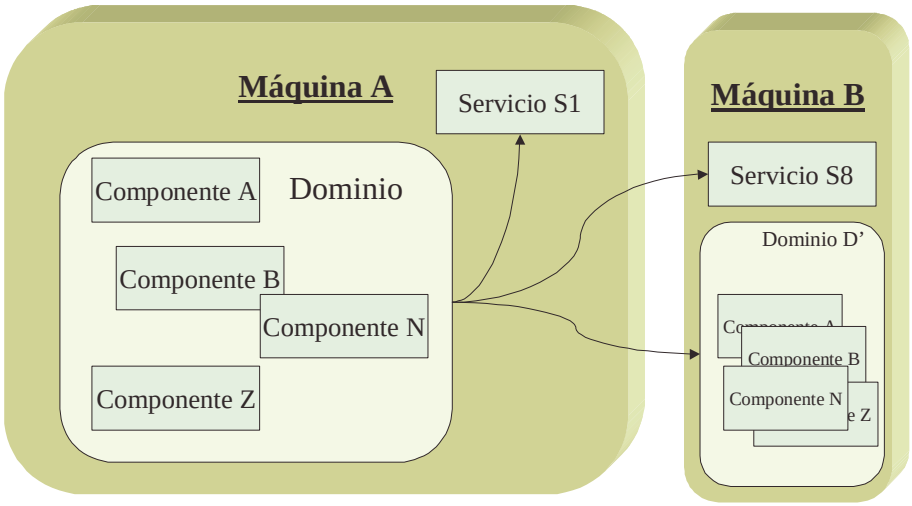

Figura 10.3. Modelo básico de dominio

Con el diseño actual, sería necesario que el dominio verificara su composición de componentes y obtuviera de los identificados como monitores su estado de bloqueo. O bien que la interfaz *DomainGroupHandler* incluyera funcionalidad que permitiera a los monitores informar de su estado pero, al ser una interfaz pública, no habría forma de comprobar la consistencia de los accesos a esa funcionalidad, comprometiendo la consistencia del dominio.

Una alternativa al diseño sería que la implementación de monitores se realizara en el dominio mismo, tal como ocurre con las transacciones, ofreciendo en su interfaz las operaciones *lock* y *unlock*. Esta aproximación emplearía cualquier componente como monitor, de una forma similar a Java, donde todos los objetos son monitores[9]. Sin embargo, la implementación implicaba una fuerte carga de trabajo sobre el dominio, al tener que mantener el estado de cada componente en su faceta de monitor: como un monitor emplea mensajes no transaccionables, su transferencia de estado es más costosa de realizar, coste que se transferiría al dominio, implicando un peor rendimiento general.

Por último, la base de esta metodología es la factorización de un servidor en componentes, para facilitar su replicación. Este servidor reside inicialmente en una máquina dada, aunque puede emplear otros componentes externos como, por ejemplo, un servicio de archivo, que residan en esa misma máquina o en otra cualquiera. Al dividir el servidor en componentes, el modelo del dominio asume, por

[9] Es discutible la relación entre objeto y monitor en Java, pues en puridad el objeto *contiene* un monitor, no *es* un monitor (problema de delegación frente a herencia).

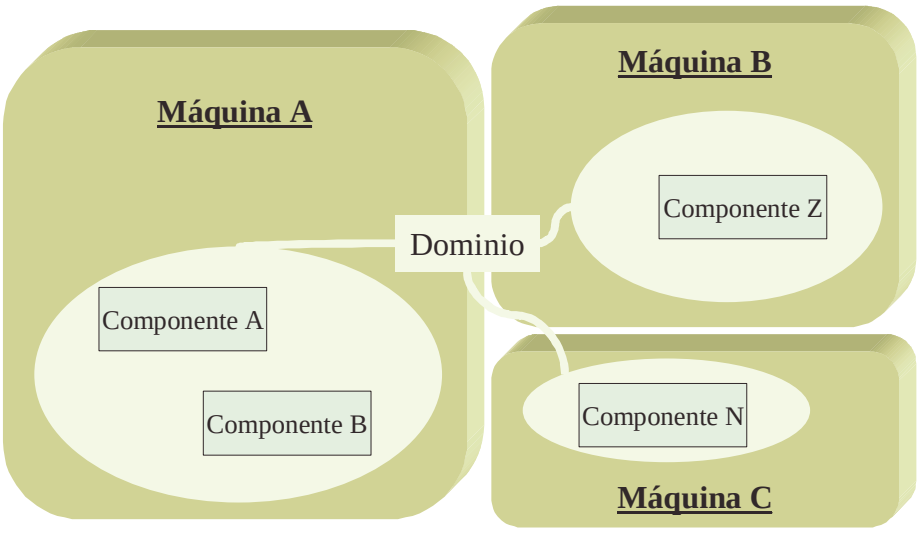

Figura 10.4. Modelo extendido de dominio

extensión, que los componentes residen en una única máquina, y la interacción entre el dominio y los componentes es local. La figura 10.3 muestra este modelo, donde un dominio y sus componentes residen en una máquina dada, pudiendo acceder a otros dominios y servicios o componentes en esa u otras máquinas.

A pesar de esta idea inicial de modelo, el dominio define su interacción con los componentes mediante una interfaz CORBA, por lo que no es preciso que su relación sea local: un dominio puede componerse de componentes esparcidos por diferentes máquinas, como muestra la figura 10.4. En la definición JavaRMI, la relación se define mediante interfaces remotos, por lo que dispone de la misma flexibilidad.

10.9. Conclusiones

SenseiDomains es un sistema que define y soporta un modelo de replicación basado en componentes, facilitando la gestión y empleo de componentes replicados para así poder diseñar servicios tolerantes a fallos a alto nivel. Estos servicios aún están basados en el modelo de sincronía virtual pero se les ocultan los detalles de bajo nivel asociados al modelo, como el manejo de vistas o el uso de primitivas de comunicaciones basadas en mensajes, que se relegan a la implementación de los componentes.

SenseiDomains no implementa directamente el modelo de sincronía virtual, precisa de un substrato de comunicaciones que lo soporte. El soporte que emplea es SenseiGMS, pero limitando sus dependencias a su interfaz, de tal modo que la

implementación pueda migrarse con facilidad a otros sistemas de comunicaciones en grupo, de acuerdo con la filosofía de SenseiGMS. Al igual que éste, SenseiDomains soporta CORBA y JavaRMI, con una interfaz totalmente equivalente entre ambos y una implementación común de los algoritmos.

La metodología que soporta SenseiDomains define una extensa funcionalidad, incluyendo una transferencia de estado flexible, el empleo de propiedades, control de concurrencia mediante monitores, soporte de transacciones, etc. La implementación se realiza de tal manera que no se afecte negativamente, en dificultad de uso o en rendimiento, a un dominio que no necesite una funcionalidad dada.

La factorización de servidores en componentes es especialmente ventajosa si se dispone de implementaciones ya preparadas de esos componentes. SenseiUMA es la parte de este proyecto que define una librería de contenedores replicados, como listas, pilas, etc. Estos componentes deben ser aún manejados conjuntamente, tarea específica del concepto de dominio, que evita, además, los problemas de acceso concurrente mediante transacciones y monitores, implementados en SenseiDomains de acuerdo a la metodología desarrollada.

Además, SenseiDomains implementa facilidades necesarias internamente por esos componentes, fundamentalmente la transferencia de estado y los patrones de sincronización de mensajes.

La metodología expuesta en el capítulo anterior también contempla la generación automática de componentes replicados. Las herramientas necesarias para esta generación son independientes del concepto de dominio y no están integradas en SenseiDomains.

Este capítulo no ha ejemplificado el uso de SenseiDomains. La razón es que el siguiente servicio desarrollado en Sensei, el servicio de localización de miembros, está construido de acuerdo a este modelo de componentes, empleando gran parte de las facilidades soportadas por SenseiDomains. Este servicio se denomina SenseiGMNS, y su funcionamiento e implementación se detallan en el siguiente capítulo, que es por lo tanto un ejemplo real de aplicación de SenseiDomains.

Capítulo 11 - SENSEIGMNS

El modelo de sincronía virtual define un servicio GMS de pertenencia a grupos cuya funcionalidad se basa en dos operaciones: inclusión y exclusión de un miembro en un grupo. SenseiGMS no define ninguna operación pública para la creación de grupos, y esta tarea se delega a SenseiGMNS, que constituye, consecuentemente, una parte integral de Sensei.

El servicio fundamental que SenseiGMNS ofrece es la gestión básica de grupos, permitiendo crearlos o extenderlos, supliendo así la funcionalidad no presente en SenseiGMS. Al emplear este servicio, la aplicación debe decidir si desea crear o extender un grupo, y en este último caso, tiene que suministrar un miembro del grupo en el que quiere incluirse.

Para facilitar esta gestión de grupos, SenseiGMNS implementa también un servicio de directorio, permitiendo manejar los grupos en base a los nombres que se les asigna. En este caso, una aplicación suministra exclusivamente el nombre del grupo en el que quiere incluirse y el servicio decide, en base a la información que mantiene, si se trata de un grupo nuevo que debe crearse o de un grupo que debe extenderse. Este servicio da nombre a SenseiGMNS, acrónimo de las siglas en inglés de *Group Membership Naming Service*.

Además de describir la funcionalidad de SenseiGMNS, este capítulo muestra el empleo de la metodología soportada por SenseiDomains, pues la implementación se realiza empleando componentes dinámicos.

SenseiDomains se diseñó con independencia del substrato de comunicaciones empleado. SenseiGMNS, sin embargo, está ligado a SenseiGMS y no tiene sentido sin éste. Otros sistemas de comunicaciones en grupo implementan ambas funcionalidades en un solo servicio, por lo que es innecesario el empleo de SenseiGMNS en el caso de utilizar SenseiDomains sobre estos otros sistemas.

11.1. Gestión básica de grupos

Un sistema de comunicaciones en grupo fiables debe soportar la gestión de grupos de réplicas, permitiendo a un servidor crear un nuevo grupo, extenderlo, o excluirse del mismo. El modelo de sincronía virtual no especifica cómo debe realizarse esa gestión; *Ensemble*, por ejemplo, la realiza identificando cada grupo con un nombre y empleando este nombre en la operaciones de creación y extensión de grupos (ambas funciones las implementa con una única función *join*).

La interfaz pública de SenseiGMS no incluye ninguna operación con este propósito, aunque sí se incluye en la interfaz privada, sobre *SenseiGMSMember*:

```
interface SenseiGMSMember : GroupHandler
{
        boolean createGroup();
        boolean addGroupMember(in SenseiGMSMember other);
        boolean joinGroup(in SenseiGMSMember group);
        . . .
};
```

El enfoque tomado es interpretar cada miembro de un grupo como un punto de acceso al grupo. Una réplica debe obtener de alguna manera una referencia a otra réplica para incorporarse al grupo al que esta última pertenece. No es un problema que una réplica pertenezca a varios grupos pues, en ese caso, tiene diferentes referencias *SenseiGMSMember* para cada grupo.

Para que un servidor pueda pertenecer a un grupo, debe implementar la interfaz *GroupMember*; al incorporarse al grupo, obtiene una referencia *GroupHandler* con la que accede a las operaciones en grupo. La relación entre interfaces se muestra en la figura 11.1:

- El servidor implementa la interfaz *GroupMember*, mediante la cual obtiene los eventos de grupo.
- El servidor obtiene una referencia *GroupHandler*, con la que realiza las comunicaciones al grupo.
- La interfaz *GroupHandler* obtenida es una instancia de *SenseiGMSMember*. El anillo de comunicaciones del grupo en Sensei se forma con las instancias *SenseiGMSMember* de sus miembros.

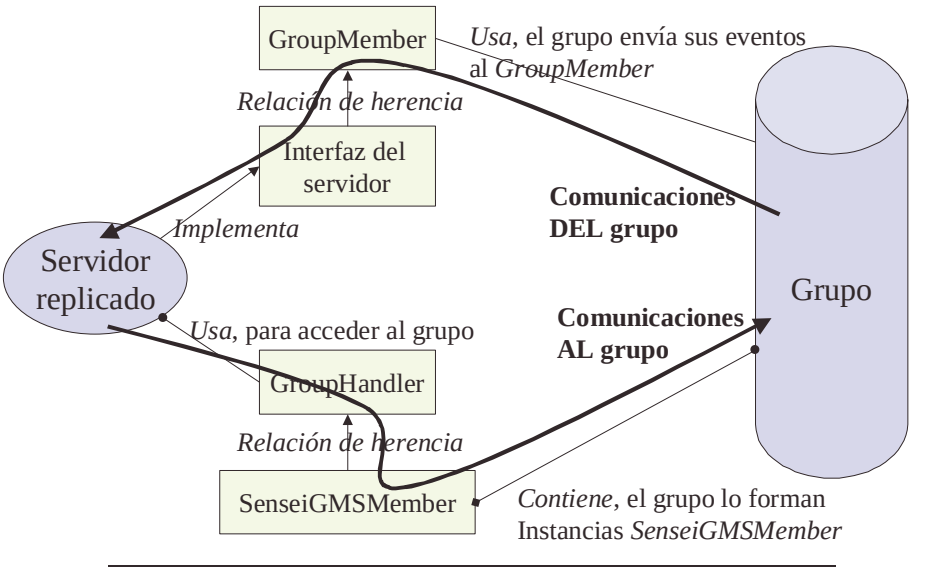

Figura 11.1. Relación de interfaces en *SenseiGMS*

Por lo tanto, todo miembro del grupo debe instanciar la interfaz *SenseiGMSMember*, con la que pasa a formar parte del anillo de comunicaciones de su grupo. Es perfectamente posible que un servidor en una máquina determinada emplee una instancia *SenseiGMSMember* localizada en una máquina diferente, pero introduce un nuevo punto de error, reduciendo la tolerancia a fallos. Por ejemplo, la figura 11.2 visualiza un escenario donde una aplicación replicada con dos servidores mantiene las instancias *SenseiGMSMember* en la misma máquina. Si esta máquina se cae, no sólo se cae el servidor que alberga, sino que el otro servidor pierde igualmente su funcionalidad. La aplicación no era tolerante a fallos en la máquina caída, lo que ha supuesto su pérdida de servicio.

El razonamiento realizado implica que cuando un servidor solicita su inclusión a otro servidor de un grupo dado, éste debe crear una instancia *SenseiGMSMember* que resida en la máquina y proceso del primer servidor. En lugar de extender la interfaz *GroupMember* con una operación *callback* que realice este cometido, SenseiGMNS define una interfaz específica *GroupHandlerFactory*:

```
exception GroupHandlerFactoryException{};
interface GroupHandlerFactory
{
       GroupHandler create() raises (GroupHandlerFactoryException);
};
```

Un servidor que accede a un miembro para unirse a su grupo, suministra una instancia *GroupHandlerFactory* que el miembro destino emplea para crear la instancia *GroupHandler*, la cual resulta ahora local al nuevo miembro.

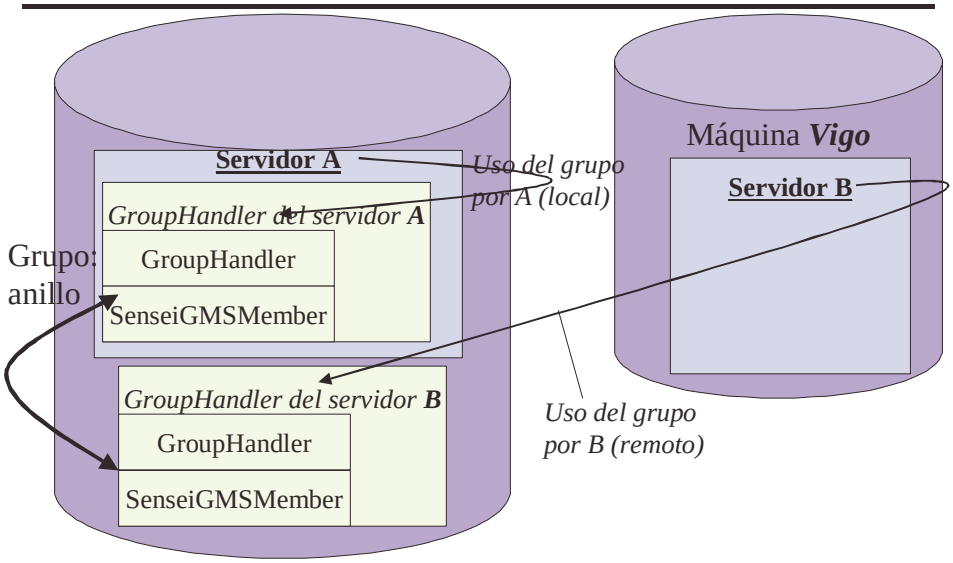

Figura 11.2. Creación errónea de instancias *GroupHandler*

Esta lógica supone que toda aplicación tolerante a fallos debe ser capaz de crear un objeto de tipo *SenseiGMSMember* y de implementar, consecuentemente, los algoritmos de comunicaciones en grupo desarrollados en el capítulo 8, cuando explicamos SenseiGMS. Una alternativa es emplear la funcionalidad de CORBA de transferencia de objetos por valor para este fin: el servidor GMNS puede suministrar el objeto con funcionalidad *SenseiGMSMember*, para lo que definimos la siguiente entidad:

```
valuetype GroupHandlerFactoryCreator
{
       GroupHandlerFactory create(in GroupMember member)
              raises (GroupHandlerFactoryException);
};
```

De esta manera, un miembro accede al servicio SenseiGMNS, obteniendo un objeto *GroupHandlerFactoryCreator*. Su definición es *valuetype* en lugar de *interface*, lo que garantiza que esta referencia *GroupHandlerFactoryCreator* reside localmente en el espacio de memoria del miembro que la solicita[10]. A continuación, este miembro crea un objeto *GroupHandlerFactory* suministrando una referencia a su

[10] Esta funcionalidad está directamente soportada en Java, que permite transferencia no sólo de datos, sino también de código. Así, la interfaz JavaRMI de *SenseiGMNS* define el tipo *GroupHandlerFactoryCreator* como una extensión de *java.util.Serializable*, mientras que todo tipo remoto extiende *java.rmi.Remote*.

interfaz *GroupMember*, y ese objeto será empleado en las operaciones de creación o extensión del grupo.

La interfaz básica de SenseiGMNS es *GroupMembershipBasicService*, definida en `GroupMembershipNamingService.idl` de la siguiente forma:

```
exception InvalidGroupHandlerException{};
exception InvalidGroupHandlerFactoryException{};
interface GroupMembershipBasicService
{
        GroupHandler createGroup(in GroupHandlerFactory theFactory)
                raises (InvalidGroupHandlerFactoryException,
                        GroupHandlerFactoryException);
        GroupHandler joinGroup(in GroupHandler group,
                        in GroupHandlerFactory theFactory)
                raises (InvalidGroupHandlerException,
                        InvalidGroupHandlerFactoryException,
                        GroupHandlerFactoryException);
        GroupHandlerFactoryCreator getFactoryCreator();
};
```

- *createGroup*: crea un grupo a partir de un *GroupHandlerFactory*. La instancia *GroupHandler* que se devuelve se crea de forma local en el miembro que invoca la operación, y no local al servicio GMNS.
- *joinGroup*: extiende el grupo identificado por la referencia *GroupHandler* dada con un nuevo miembro, especificado mediante la referencia *GroupHandlerFactory*.
- *getFactoryCreator*: devuelve la instancia *GroupHandlerFactoryCreator*, con la que es posible crear los objetos *GroupHandlerFactory* empleados en las dos operaciones anteriores.

El escenario asociado resulta complejo para asegurar que el anillo lógico de comunicaciones de Sensei se integra por componentes residentes físicamente en la máquina y proceso que representan. Resumiendo, los pasos requeridos para incluir un miembro en un grupo son:

- El futuro miembro del grupo accede al servicio GMNS y obtiene mediante la operación *getFactoryCreator* un objeto del tipo *GroupHandlerFactoryCreator*, que reside localmente en la máquina y proceso de ese miembro.
- Este miembro emplea el objeto *GroupHandlerFactoryCreator* para crear una instancia de la interfaz *GroupHandlerFactory* que existe, por lo tanto, localmente.
- El miembro accede de nuevo al servicio GMNS, esta vez a la operación *joinGroup*, pasando el objeto *GroupHandlerFactory*.
- El servidor GMNS accede (remotamente) al objeto *GroupHandlerFactory*, para crear el *GroupHandler* en el proceso del futuro miembro, y devuelve la referencia creada a ese miembro, ya perteneciente al grupo.

La interfaz del servicio creado para gestionar los grupos no ofrece, sin embargo, ninguna funcionalidad para obtener referencias a otros miembros, que deben ser localizadas por la aplicación. El servicio de directorio de SenseiGMNS cubre esta funcionalidad.

11.2. Servicio de directorio

El servicio de directorio mantiene referencias a los miembros existentes en un grupo, al que identifica mediante un nombre. Empleando esta funcionalidad, un miembro puede incluirse en un grupo especificando su nombre y el servicio se encarga de crear un nuevo grupo o, si ya contiene referencias a otros miembros en el mismo grupo, de extenderlo.

La utilidad de este servicio no se limita a la gestión de grupos. Un cliente puede emplearlo para obtener la referencia de un miembro de un servidor replicado, a diferencia del servicio mostrado en la anterior sección.

Desde la perspectiva del grupo, cada miembro debe implementar la interfaz *GroupMember*. Pero estos miembros forman, además, parte de una aplicación que ofrece una interfaz específica a sus clientes. El servicio de directorio almacena referencias de los miembros a ambas interfaces y, por ello, se define la interfaz *ReplicatedServer*, sin funcionalidad propia, que debe ser extendida por la interfaz de cliente:

```
interface ReplicatedServer
{
};
```

El servicio de directorio se define mediante la siguiente interfaz:

```
interface GroupMembershipNamingService
{
       GroupHandlerFactoryCreator getFactoryCreator();
       GroupHandler findAndJoinGroup(in string groupName,
                      in GroupHandlerFactory theFactory,
                      in string memberName,
                      in ReplicatedServer clientsReference)
              raises (sensei::middleware::domains::MemberStateException,
                      InvalidGroupHandlerFactoryException,
                      GroupHandlerFactoryException);
       void leaveGroup(in string groupName, in GroupHandler member);
       ReplicatedServersList getGroup(in string groupName)
              raises (sensei::middleware::domains::MemberStateException);
       ReplicatedServer getGroupServer(in string groupName, in string memberName)
              raises(sensei::middleware::domains::MemberStateException);
```

```
        ReplicatedServer getValidGroupServer(in string groupName)
                raises(sensei::middleware::domains::MemberStateException);
};
```

- *getFactoryCreator*: define la misma funcionalidad que en el caso de la operación homónima en *GroupMembershipBasicService*.
- *findAndJoinGroup*: crea o extiende un grupo, al que se identifica por un nombre. El miembro debe suministrar un nombre, una interfaz de cliente y una instancia *GroupHandlerFactory*, usando la misma filosofía mostrada en *GroupMembershipBasicService* para crear los objetos localmente al miembro que extiende o crea el anillo.
- *leaveGroup:* el miembro identificado por la instancia *GroupHandler* asociada es expulsado del grupo.
- *getGroup*: devuelve las referencias de todas las réplicas del grupo especificado, sin comprobarse si las réplicas son aún válidas. Las referencias devueltas son las interfaces de cliente registradas.
- *getGroupServer*: devuelve la referencia a un servidor específico en el grupo dado. La referencia devuelta es la interfaz de cliente registrada.
- *getValidGroupServer*: devuelve la referencia a un servidor específico en el grupo dado, comprobándose previamente que el servidor es todavía válido. La referencia devuelta es la interfaz de cliente registrada.

El servicio GMNS comprueba periódicamente las réplicas que contiene, eliminando aquellas que han caído o hayan sido excluidas del grupo sin haber empleado *leaveGroup*.

Si el servicio básico de SenseiGMNS es esencial para la creación de grupos, este servicio de directorio no lo es. Así, un miembro que se incluye en un grupo sin emplear este servicio de directorio no es registrado y no puede ser utilizado para extender el grupo al que pertenece. Las consecuencias son posibles inconsistencias en aplicaciones que no empleen de forma uniforme el servicio básico o el servicio de directorios de SenseiGMNS.

Para que este servicio sea efectivo, debe ser por sí mismo tolerante a fallos por lo que se ha diseñado como un servidor replicado, diseño que detallamos a continuación como ejemplo de uso de la metodología que SenseiDomains implementa.

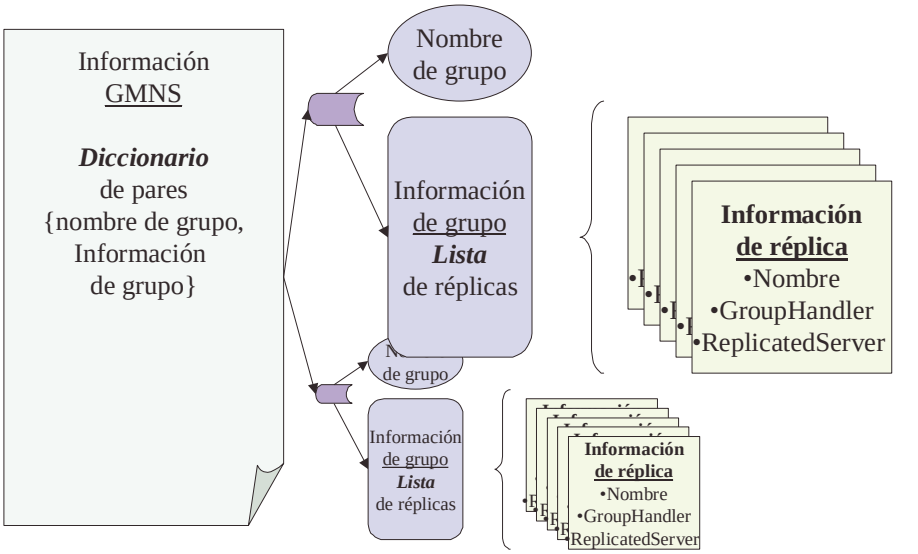

Figura 11.3. Estructura de datos en un servidor GMNS no replicado

11.3. Diseño de SenseiGMNS en componentes dinámicos

La sección 8.3 introdujo las bases de diseño de un servicio de directorio como ejemplo de uso de SenseiGMS. En aquel caso, el diseño se basaba en un único componente *GMNSdata* que almacenaba la información a replicar en el servicio. Este componente no soportaba accesos concurrentes, con lo que varias réplicas no podían actualizarlo simultáneamente.

El diseño real de un servidor GMNS replicado lo hemos realizado extendiendo el modelo del mismo servidor no replicado. Sobre este servicio no tolerante a fallos, una implementación lógica se basaría [figura 11.3] en un contenedor tipo *diccionario* que contendría como claves los nombres de los grupos y como valores la información asociada a éstos. La información necesaria para un grupo puede, a su vez, representarse con un contenedor tipo *lista* donde cada elemento es la información de una réplica, compuesta por sus referencias *GroupHandler* y *ReplicatedServer*, y el nombre asignado.

Con esta estructura de datos, el diccionario estaría inicialmente vacío; al crearse un nuevo grupo, se crea una nueva instancia del tipo lista empleado para almacenar la información de las réplicas de un grupo, y se introduce un par {nombre, lista} en el diccionario. A continuación, se crea una instancia de la clase empleada para almacenar la información de una réplica, que se inserta en la lista

dada. Por consiguiente, el diccionario es el único componente inicial, todos los demás se crean dinámicamente.

Al transformar el servidor en uno replicado bajo CORBA, sería necesario definir primero la entidad que almacena la información de una réplica, para lo que empleamos la *struct MemberInfo*:

```
struct MemberInfo
{
	GroupHandler handler;
	string name;
	ReplicatedServer publicReference;
};
typedef sequence<MemberInfo> MemberInfoList;
```

El contenedor que guarda las instancias *MemberInfo* es una lista, por lo que se podría implementar con el contenedor replicado estándar *ReplicatedList* de SenseiUMA. Sin embargo, *SenseiUMA*, a diferencia de los demás sistemas en *Sensei* se define de forma diferente en CORBA y JavaRMI, pues en su implementación bajo JavaRMI intenta integrarse con tipos ya definidos, como *java.util.iterator*, etc. Por esta razón, SenseiGMNS se implementaría también diferentemente en ambas arquitecturas, en contra de lo deseado. Además, implicaría una dependencia de SenseiGMNS en SenseiUMA, dependencia que preferimos evitar para minimizar el código estrictamente necesario en la implementación de SenseiGMNS.

Como consecuencia, definimos un contenedor del tipo *lista replicada* especialmente para este caso. El principal beneficio asociado es que en lugar de emplear tipos genéricos, la lista se define como un contenedor de instancias concretas *MemberInfo*:

```
interface SetMemberInfo : ExtendedCheckpointable, SetMemberInfoObservable
{
	boolean add(in MemberInfo member)
		raises (sensei::middleware::domains::MemberStateException);
	boolean remove(in MemberInfo handler)
		raises (sensei::middleware::domains::MemberStateException);
	MemberInfo get()
		raises (sensei::middleware::domains::MemberStateException);
	MemberInfoList toArray()
		raises (sensei::middleware::domains::MemberStateException);
};
```

El tipo de contenedor es *set*, una lista que no acepta valores duplicados. Puesto que este contenedor no tiene un uso general, se definen sólo las operaciones requeridas: insertar y borrar un elemento, que devuelven un valor booleano que indica el éxito de la aplicación, y dos operaciones de acceso a los elementos contenidos. Este acceso es también específico y, en lugar de definirse una interfaz

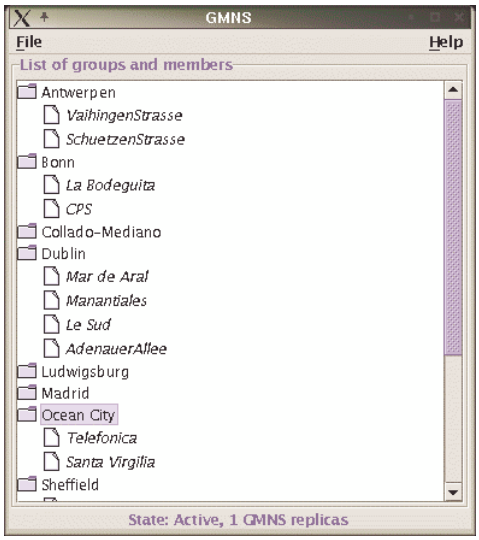

Figura 11.4. Interfaz gráfica de *SenseiGMNS*

que itere sobre el contenido, al estilo de *java.util.Iterator* en Java, se emplean dos operaciones básicas: obtener un elemento y obtener todos los elementos.

La definición de esta interfaz es *ExtendedCheckpointable*, luego el soporte de transferencia de estado es básico y la transferencia se efectúa en un solo paso. Además, la interfaz extiende *SetMemberInfoObservable*, que se define como:

```
interface SetMemberInfoObservable
{
       void addObserver(in SetMemberInfoObserver observer);
       boolean removeObserver(in SetMemberInfoObserver observer);
};
```

La funcionalidad asociada es permitir observar al contenedor dado. Puesto que este contenedor está replicado, cualquiera de los servidores GMNS puede actualizarlo, y esas actualizaciones deben reflejarse en la interfaz gráfica del servicio [figura 11.4], lo que admite dos soluciones: interrogar periódicamente a los contenedores sobre su contenido o recibir eventos cuando se actualizan. La segunda aproximación es más complicada, pero ventajosa. De hecho, SenseiUMA soporta también esta segunda opción en los contenedores definidos.

La observación se realiza aplicando el patrón de observación [Gamma95], tambien conocido como patrón de publicación/subscripción: el contenedor se define como *observable*, permitiendo que todo elemento registrado, implementando una interfaz *Observer*, reciba los eventos de cambios. Esta última interfaz se define para este caso específico como:

```
interface SetMemberInfoObserver
{
        void addDone(in SetMemberInfoObservable observable, in MemberInfo member);
        void removeDone(in SetMemberInfoObservable observable, in MemberInfo member);
};
```

La lista informa, por lo tanto, de cualquier adición o borrado de elementos, junto con el elemento dado.

La misma lógica general se emplea para el diccionario, que se define mediante la interfaz *MapStringSet*:

```
exception InvalidSetMemberInfo{};
typedef sequence<string> stringList;

interface MapStringSetObserver
{
        void clearDone(in MapStringSetObservable observable);
        void putDone(in MapStringSetObservable observable,
                        in string groupName,
                        in SetMemberInfo set);
        void removeDone(in MapStringSetObservable observable,
                        in string groupName,
                        in SetMemberInfo set);
};

interface MapStringSetObservable
{
        void addObserver(in MapStringSetObserver observer);
        boolean removeObserver(in MapStringSetObserver observer);
};

interface MapStringSet : ExtendedCheckpointable, MapStringSetObservable
{
        SetMemberInfo put(in string groupName, in SetMemberInfo set)
                raises (InvalidSetMemberInfo,
                        sensei::middleware::domains::MemberStateException);
        SetMemberInfo remove(in string groupName)
                raises (sensei::middleware::domains::MemberStateException);
        SetMemberInfo get(in string groupName)
                raises (sensei::middleware::domains::MemberStateException);
        stringList getKeys()
                raises (sensei::middleware::domains::MemberStateException);
};
```

11.3.1. Implementación de los componentes

Tras haber definido la interfaz de los componentes de SenseiGMNS, es necesario especificar los mensajes que las réplicas intercambian y cómo se transfiere el estado.

MemberInfo se definió en OMG/IDL como *struct*: no tiene comportamiento, sólo almacena los datos asociados. En el caso del componente *SetMemberInfo*, soporta dos operaciones: *add*, *remove*. En lugar de definir dos mensajes, empleamos uno solo, especificado como:

```
valuetype SetMemberInfoChangedMessage : DomainMessage
{
       public MemberInfo info;
       public boolean add;
};
```

Este mensaje contiene la información del miembro que se añade o elimina, y se define como una especialización de *DomainMessage*. No precisa de un tratamiento especial durante las transferencias o transacciones, por lo que la implementación no debe modificar el valor del campo *unqueuedOnST* (*false*) de *DomainMessage*.

El componente se define como *ExtendedCheckpointable*, luego es estado se transfiere en un único paso; en este caso, la información a transferir incluye los datos de cada uno de los componentes de la lista:

```
valuetype SetMemberInfoState : State
{
       public sequence<MemberInfo> members;
};
```

La implementación del componente la realizamos en Java; la clase asociada necesita dos constructores distintos:

```
SetMemberInfoImpl(int subgroupId, DomainGroupHandler groupHandler);
SetMemberInfoImpl(DynamicSubgroupInfo dynamicInfo, DomainGroupHandler groupHandler);
```

- Constructor *estático*. Empleado en caso de registrar el componente estáticamente en el dominio, precisa de una identidad de componente predefinida y del dominio a emplear. También se utiliza para crear componentes dinámicos en demanda, al conocerse ya su identidad.
- Constructor *dinámico*. Empleado para crear dinámicamente un componente, precisa de la información a enviar a otras réplicas.

El componente implementa una lista, por lo que emplea internamente una lista para almacenar los miembros; el tipo seleccionado es *java.util.Set* (implementado mediante *java.util.HashSet*). El estado inicial de esta lista no incluye ningún miembro, y se implementan las operaciones de transferencia como:

```
public synchronized State getState()
{
        return factory.createSetState(getMembers());
}

MemberInfo[] getMembers()
{
        int size = set.size();
        MemberInfo members[] = new MemberInfo[size];
        if (size>0) {
                int i=0;
                Iterator it = set.iterator();
                while(it.hasNext()) {
                        members[i++]=(MemberInfo) it.next();
              }
        }
        return members;
}

public synchronized void setState(State state)
{
        set.clear();
        MemberInfo members[]=((SetMemberInfoState)state).members;
        int size = members.length;
        for (int i=0;i<size;i++) {
                addMemberInfo(members[i]);
        }
}

synchronized boolean addMemberInfo(MemberInfo memberInfo)
{
        boolean ret = set.add(memberInfo);
        if (ret) {
                observersHandler.informPut(memberInfo);
        }
        return ret;
}

public synchronized void assumeState()
{
        set.clear();
}
```

- *assumeState*: elimina cualquier elemento de la lista.

- *setState*: añade cada uno de los elementos dados en la lista de entrada. Para ello emplea un método *addMemberInfo*, también empleado en la adición posterior de miembros que, además de incluirlo en la lista, informa del evento a los posibles observadores.
- *getState*: emplea un objeto *factory*, capaz de crear el objeto *middleware State*, tanto para JavaRMI, como para CORBA, con la información del miembro. Esta información la obtiene de *getMembers*, que itera sobre los elementos de la lista, devolviéndolos en un *array*.

El componente *SetMemberInfo* es un miembro de grupo, por lo que debe reaccionar a los seis eventos del grupo. Sólo es relevante el procesado de los mensajes multipunto:

```
public void processPTPMessage(int parm1, Message parm2){}
public void changingView(){}
public void installView(View parm1){}
public void memberAccepted(int id, GroupHandler parm2, View parm3)
{
	memberId=id;
}
public void excludedFromGroup()
{
	freeResources();
}
public void processCastMessage(int sender, Message message)
{
	if (message instanceof SetMemberInfoChangedMessage) {
		synchronized(this) {
			SetMemberInfoChangedMessage msg =
				(SetMemberInfoChangedMessage) message;
			if (msg.add)
				addMemberInfo(msg.info);
			else
				removeMemberInfo(msg.info);
		}
	}
}
```

En caso de recibir un mensaje *SetMemberInfoChangedMessage*, se añade o elimina el miembro que contiene, según sea el mensaje. Para ello, se emplea el método *addMemberInfo*, ya mostrado anteriormente, o *removeMemberInfo* que, de la misma manera, elimina el miembro de la lista e informa a los observadores:

```
synchronized boolean removeMemberInfo(MemberInfo memberInfo)
{
	Iterator it = set.iterator();
```

```
        while(it.hasNext()) {
                MemberInfo content = (MemberInfo) it.next();
                if (ObjectsHandling.areEquivalent(memberInfo.handler, content.handler))
                {
                        it.remove();
                        observersHandler.informRemove(memberInfo);
                        return true;
               }
        }
        return false;
}
```

En CORBA, dos referencias a un mismo miembro pueden ser distintas, por lo que es necesario emplear métodos CORBA específicos para realizar la comparación entre referencias. El objeto *ObjectsHandling* efectúa esta comparación de modo transparente para CORBA y JavaRMI, permitiendo que el mismo código sea válido en ambas plataformas.

La interfaz pública del componente se implementa a continuación, sin incluir el tratamiento de errores. El código es una implementación directa del patrón de sincronización de mensajes en el caso de tener que devolver un valor al cliente:

```
public boolean add(MemberInfo info) throws MemberStateException
{
        boolean result=false;
        factory.castSetMessage(info, true);
        result=addMemberInfo(info);
        factory.syncMessageCompleted();
        return result;
}
public synchronized MemberInfo get() throws MemberStateException
{
        Iterator it=set.iterator();
        return it.hasNext()? (MemberInfo) it.next(): null;
}
public boolean remove(MemberInfo info) throws MemberStateException
{
        boolean result=false;
        factory.castSetMessage(info, false);
        result=removeMemberInfo(info);
        factory.syncMessageCompleted();
        return result;
}
public synchronized MemberInfo[] toArray()
{
        int size = set.size();
```

```
        MemberInfo[] ret = new MemberInfo[size];
        set.toArray(ret);
        return ret;
}
```

Por consiguiente, las operaciones *get* y *toArray* se implementan sin comunicaciones al grupo, y las operaciones *add* y *remove* implican el envío de un mensaje y su procesado en el mismo *thread* de envío, devolviendo el resultado de la operación al cliente. El código necesario para implementar el patrón de observación es irrelevante y no lo incluimos para no extender innecesariamente esta explicación; el apéndice D contiene el código completo de esta clase.

El componente *MapStringSet* sigue la misma lógica. Define los mismos constructores, mismo tratamiento de eventos, etc. A continuación se muestra la definición de los mensajes y el estado:

```
valuetype MapStringSetPutMessage : DomainMessage
{
        public string key;
        public SubgroupId value;
};

valuetype MapStringSetRemoveMessage : DomainMessage
{
        public string key;
};

struct MapStringSetEntry
{
        string key;
        SubgroupId value;
};

valuetype MapStringSetState : State
{
        public sequence<MapStringSetEntry> groups;
};
```

En este caso se definen dos mensajes distintos para las operaciones *put* y *remove*, pues la información asociada es diferente.

Es importante destacar que en las comunicaciones al grupo no se envía el componente *SetMemberInfo* que contiene, sino la identidad de ese componente replicado (en negrita en el anterior listado), tal como se ha registrado en el dominio. Este tratamiento no es una optimización, sino una necesidad, pues cada réplica debe contener referencias de los componentes que instancia, no de los componentes en la réplica que envía la información.

Ésta es precisamente la segunda razón por la que SenseiUMA es diferente en JavaRMI y CORBA. Java soporta nativamente patrones de serialización y es posible especificar la manera en que un determinado objeto se serializa. Al serializar objetos del tipo *GroupMember*, se accede al dominio para obtener su identidad de componente, que es la información que se envía. Bajo CORBA, el componente podría verificar explícitamente si el objeto a enviar implementa la interfaz *GroupMember*, pero esta solución resulta insuficiente, al no cubrir objetos complejos que incluyan en su composición componentes replicados.

Este componente implementa un diccionario o *mapa*, por lo que emplea internamente el mismo tipo para almacenar elementos; *java.util.Map* (implementado mediante *java.util.HashMap*). El apéndice D incluye el código completo de esta clase, y sólo mostramos aquí el tratamiento de componentes replicados en el método *put*, que incluye para un nombre de grupo un componente *SetMemberInfo*, y en el método de recepción de mensajes, *processCastMessage*, ambos sin comprobación exhaustiva de errores:

```
public SetMemberInfo put(String group, SetMemberInfo set)
	throws InvalidSetMemberInfo, MemberStateException
{
	SetMemberInfo result=null;
	int subgroupId = groupHandler.getSubgroupId(set);
	if (subgroupId==Consts.INVALID_SUBGROUP) {
		throw new InvalidSetMemberInfo();
	}
	factory.castMapPutMessage(group, subgroupId);
	result = doPut(group, set);
	factory.syncMessageCompleted();
	return result;
}

public void processCastMessage(int sender, Message message)
{
	if (message instanceof MapStringSetPutMessage) {
		MapStringSetPutMessage msg = (MapStringSetPutMessage) message;
		GroupMember subgroup = groupHandler.getSubgroup(msg.value);
		SetMemberInfo set = GMNSnarrower.toSetMemberInfo(subgroup);
		if (set!=null) {
			doPut(msg.key, set);
		}
	}
	. . .
}

SetMemberInfo doPut(String group, SetMemberInfo set)
{
```

```
        SetMemberInfo ret = null;
        Object obj = null;
        synchronized(this){obj=map.put(group, set);}
        if (obj!=null) {
                ret = GMNSnarrower.toSetMemberInfo(obj);
                observersHandler.informRemove(group, ret);
        }
        observersHandler.informPut(group, set);
        return ret;
}
```

- Al procesar una operación *put*, se comprueba que el componente insertado es válido (que esté registrado en el dominio). En la estructura de datos interna se guarda la referencia al componente replicado, aunque la información que se envía al grupo es su identidad.
- Al recibir el mensaje asociado a la operación *put*, se obtiene el componente replicado a partir de la identidad de componente recibida.
- Ambos métodos delegan en la operación *doPut* para insertar efectivamente el par {nombre, componente} que, además, comunica el evento a los observadores.

11.3.2. Integración de componentes

Al introducir información relativa a un nuevo grupo, el servidor GMNS debe crear un componente dinámico *SetMemberInfo*, cuya creación debe propagarse a cada réplica GMNS, donde se instancia la réplica de ese componente dinámico. El servidor GMNS que crea el grupo emplea un código que en esencia es:

```
SetMemberInfoImpl setImpl = new SetMemberInfoImpl(Consts.noInfo, domainHandler);
SetMemberInfo set = setImpl.theSetMemberInfo();
map.put(groupName, set);
set.add(new MemberInfo(groupHandler,memberId, clientReference));
```

- Se crea el componente empleando el constructor dinámico, que precisa de la información que se envía a las demás réplicas sobre el componente que se crea. En ese caso concreto no se envía ninguna información. Cuando este componente se crea, también lo hacen sus réplicas en los otros servidores GMNS.
- La segunda línea en el anterior listado es necesaria por la diferenciación CORBA entre servidor y *servant*[11] [OMG98]. Simplemente activa el servidor, obteniendo una referencia válida al mismo. En el caso de JavaRMI, *exporta* el servidor para ser públicamente accesible.

[11] En esencia, *servant* es la implementación en un lenguaje determinado de la funcionalidad del servidor. Es preciso activar (encarnar) el *servant* para obtener el servidor.

- Se inserta el *set* en el diccionario, con el nombre dado al grupo. Puesto que el componente diccionario está replicado, sus réplicas insertan simultáneamente el *set* con el mismo nombre. El *set* que insertan no es el existente en este servidor, sino el que haya sido creado en el respectivo servidor.
- Finalmente, se inserta la información del grupo en el *set*. De nuevo, esta operación se propaga a las demás réplicas, de acuerdo con la implementación de *SetMemberInfo*, con lo que la información es consistente en cada réplica.

Cada réplica del servicio GMNS debe ser capaz de crear componentes dinámicos, por lo que es necesario implementar la interfaz *DynamicSubgroupsUser*:

```
class MyDynamicSubgroupsUser extends DynamicSubgroupsUserBaseImpl
{
   public MyDynamicSubgroupsUser() throws Exception
   {
   }
   public GroupMember acceptSubgroup(int id, DynamicSubgroupInfo info)
   {
      try{return new SetMemberInfoImpl(id, domainHandler).theSetMemberInfo();}
      catch(Exception ex){return null;}
   }
   public GroupMember subgroupCreated(int creator, int id, DynamicSubgroupInfo info)
   {
      try{return new SetMemberInfoImpl(id, domainHandler).theSetMemberInfo();}
      catch(Exception ex){return null;}
   }
   public void subgroupRemoved(int remover, int id, DynamicSubgroupInfo info)
   {
   }
}
```

- Cuando se recibe cualquiera de los dos eventos de creación de componente, se crea un objeto *SetMemberInfo*. Puesto que es el único tipo de objeto que se crea, no hace falta que el parámetro *DynamicSubgroupInfo* contenga información específica.
- No es preciso hacer nada cuando el componente se elimina (una implementación C++ hubiera precisado la destrucción del *servant*).

Por último, la instancia de la anterior clase debe registrarse en el dominio, que debe crearse con las características deseadas. También deben registrarse en el dominio los componentes estáticos:

```
(A)   domainHandler = new DomainGroupHandlerImpl().theDomainGroupHandler();
(B)   domainHandler.setBehaviourMode(
         BehaviourOnViewChanges.MembersOnTransferExcludedFromGroup);
      domainHandler.setDynamicSubgroupsUser(
```

```
        new MyDynamicSubgroupsUser().theDynamicSubgroupsUser());
      domainHandler.setDomainGroupUser ( . . . );
(C)   GroupMonitorImpl monitorImpl =
        new GroupMonitorImpl(Consts.MONITOR_SUBGROUP, domainHandler);
      MapStringSetImpl mapImpl =
        new MapStringSetImpl(Consts.MAP_SUBGROUP, domainHandler);
      map = mapImpl.theMapStringSet();
(D)   monitorImpl.register();
      mapImpl.register();
```

- La única operación en el grupo A crea el dominio.
- Las siguientes tres líneas en el grupo B definen las características del dominio, incluyendo el registro del gestor de componentes dinámicos que permite emplear este tipo de componentes.
- A continuación, se crean los componentes estáticos, que en este caso es un monitor, empleado para las transacciones, y un diccionario.
- Finalmente, las dos operaciones del grupo D registran los componentes en el dominio.

11.3.3. Implementación de los algoritmos

Tras haberse implementado los componentes replicados y creado el dominio para gestionarlos dinámicamente, podemos centrarnos en la lógica de aplicación. La operación fundamental del servidor GMNS es la introducción de un servidor en un grupo, que tal vez sea necesario crear.

Para esta operación definimos dos operaciones básicas:

- *joinGroup*: incluye un miembro en un grupo. Si el grupo no existe, la operación devuelve *false*.
- *createGroup*: crea un grupo e inserta el miembro especificado. Si el grupo ya existe, la operación devuelve *false*.

Con este soporte, la operación *findAndJoinGroup* se escribe, sin soporte de errores, como:

```
. . .
boolean stop=true;
while(!stop) {
       stop=joiner.joinGroup(groupName, factory, memberId, reference);
       if (!stop) {
            stop=creator.createGroup(groupName, factory, memberId, reference);
       }
}
. . .
```

Primero se intenta extender el grupo; si no se consigue, se intenta crearlo pero, durante este intervalo, otro servidor puede haber creado el mismo grupo, con lo que la creación puede fallar también, en cuyo caso se repite el proceso. La alternativa a este algoritmo es emplear un monitor que bloquee el acceso al diccionario a otras réplicas, impidiendo que creen el mismo grupo. Sin embargo, la operación *join* es particularmente lenta, y esta aproximación supone bloquear a todos los servidores GMNS en cada operación, incluso aquellos que pretenden acceder a grupos diferentes.

La operación *createGroup* debe emplear transacciones para evitar que dos servidores creen el mismo grupo al mismo tiempo:

```
domainHandler.startTransaction(monitor);
SetMemberInfo group = map.get(groupName);
if (group==null) {
         ret=createGroupSafely(groupName, factory, memberId, reference);
}
else if (group.get()==null) {
       ret=addMemberSafely(group, factory, memberId, reference);
}
else {
  ret=false;
}
domainHandler.endTransaction();
```

- La creación del grupo se protege con una transacción en torno al monitor registrado.
- Se obtiene del diccionario la información asociada al grupo específico, que es una referencia nula si el grupo no existe.
- Si el grupo no existe, se crea. Si existe pero está vacío, simplemente se inserta el nuevo miembro (un grupo no se elimina automáticamente cuando queda vacío). Y si existe y tiene elementos, la operación falla y devuelve *false*.

La operación *joinGroup* no precisa de transacciones: un componente replicado admite accesos concurrentes sin problemas de consistencia, siempre que la semántica de la operación lo permita. Si dos réplicas insertan simultáneamente dos elementos en una lista, la semántica de esta operación implica que la lista termina con dos elementos. Pero si esta operación se realiza sobre un diccionario, y las dos réplicas emplean la misma clave, uno de los valores se pierde.

De esta manera, la operación más lenta, *join*, no bloquea a las réplicas. Esta operación obtiene la información del grupo y, si éste existe, solicita un miembro del grupo. Este miembro se emplea, a continuación, para incluir al nuevo miembro en su grupo. Si la operación tiene éxito, el nuevo miembro pasa a formar parte de ese grupo:

```
SetMemberInfo group=map.get(groupName);
if (group==null) {
        ret=false;
}
else {
        MemberInfo toJoin = group.get();
        if (toJoin == null) {
                ret = false;
        }
        else {
                ret = utilJoinGroup(toJoin.handler);
        }
}
```

Este código sólo muestra la lógica principal, puesto que el proceso de inclusión de un miembro en un grupo puede fallar por problemas de comunicaciones y el servidor GMNS debe entonces repetir el proceso, o devolver un error válido al cliente.

Tampoco se explican en este capítulo otros algoritmos de SenseiGMNS, por ser irrelevantes en cuanto a su aplicación de la metodología, incluyendo:

- Métodos para que el mismo servidor SenseiGMNS forme un grupo por sí mismo, publicando su información en puertos TCP específicos o en direcciones configurables.
- Control de referencias inválidas, pertenecientes a miembros excluidos de grupos. Sólo uno de los servidores GMNS realiza este control, pues no tiene sentido que todas las réplicas verifiquen las mismas referencias.
- Persistencia de la información, de tal manera que si todos los servidores GMNS se caen, no se pierdan los datos de los grupos existentes y sus miembros.
- Interfaz gráfica, que muestra los eventos producidos por los componentes mediante el patrón de observación.

11.4. Conclusiones

SenseiGMNS completa la funcionalidad del modelo de sincronía virtual no presente en SenseiGMS, permitiendo la creación y extensión de grupos. También presenta funcionalidad extendida para soportar la gestión de grupos mediante nombres.

La implementación es un ejemplo de empleo de la metodología soportada por SenseiDomains. La aplicación de esta metodología pretende facilitar el diseño de servidores replicados mediante el empleo de componentes, permitiendo que el

diseñador se enfoque en la lógica de la aplicación y no en los métodos de aplicar esa lógica.

Sin embargo, al mostrar cómo se ha diseñado SenseiGMNS, la implementación de su lógica de aplicación no ha sido inmediatamente posible. Primero, ha sido necesario desarrollar los componentes que la sustentan, tarea que conlleva un considerable esfuerzo. No obstante, debe tenerse en cuenta que, con el soporte de SenseiUMA, esos componentes habrían estado ya disponibles.

Pero incluso con el empleo de componentes replicados ya disponibles, es preciso que la aplicación realice una serie de pasos que permitan su posterior uso, tales como crear un dominio, registrar los componentes estáticos e instruir al dominio sobre cómo actuar con componentes dinámicos. Una vez concluidos estos pasos, es posible dedicarse a la lógica de la aplicación, empleando componentes replicados como si fueran *normales*, aunque compartidos por todas las réplicas.

El esfuerzo adicional para manejar el dominio es, sin embargo, muy inferior al necesario para programar una aplicación tolerante a fallos sin soporte de componentes. Procesos como la optimización que hemos realizado en el algoritmo para bloquear réplicas el mínimo tiempo posible son evidentemente factibles sin el empleo de componentes, diseñando, posiblemente, una lógica de aplicación muy diferente, pero la ventaja de esta aproximación es su semejanza a un caso no replicado y, consecuentemente, la posibilidad de emplear los mismos algoritmos. Esta lógica implica, además, una menor curva de aprendizaje, lo que resulta en una implementación más rápida para analistas no experimentados en aplicaciones tolerantes a fallos, así como en una más fácil migración de servidores no replicados a su equivalente replicado.

Capítulo 12 - CONCLUSIONES FINALES

Este último capítulo recoge las principales conclusiones de la memoria, describiendo las aportaciones realizadas y las que consideramos que son nuestras próximas líneas de investigación y trabajo futuro.

12.1. Aportaciones realizadas

El dominio del trabajo que esta memoria presenta son las técnicas de desarrollo de aplicaciones tolerantes a fallos mediante la replicación de sus recursos. En el núcleo de estas técnicas está el modelo de sincronía virtual, con el cual es posible programar aplicaciones en grupo bajo la abstracción de que no hay errores de comunicaciones entre los miembros del grupo y que todos los miembros ven los mismos eventos en el mismo orden.

Este modelo precisa de un sistema de comunicaciones fiables en grupo, y debe soportar un conjunto de propiedades y facilidades, como es el servicio de pertenencia a grupo, que posibilita el empleo de grupos dinámicos. En especial, debe soportar la transferencia de estado a los miembros que se incluyen en el grupo, de tal forma que todas las réplicas mantengan un estado consistente; sin embargo, esta facilidad está débilmente especificada y su soporte en los sistemas actuales de comunicaciones fiables en grupo suele ser insuficiente o, incluso, inexistente.

El desarrollo de las aplicaciones replicadas se facilita enormemente con el modelo de sincronía virtual, pero implica un diseño de bajo nivel basado en el envío de mensajes entre réplicas. Aunque es posible enmascarar la interfaz ofrecida con orientación a objetos, la lógica de la replicación sigue basada en un envío de mensajes. Ésto implica diseños muy ligados a cada programa, dificultando el empleo de técnicas como la reutilización de componentes.

La especificación del servicio CORBA de tolerancia a fallos soluciona todos estos problemas, pero impone un modelo muy restrictivo, tanto para la transferencia de estado, como para las aplicaciones que lo emplean, de tal forma que resulta poco práctico y efectivo. Sin embargo, debe destacarse que este servicio soporta tanto replicación activa como pasiva, pero ambas con el mismo modelo y con un especial énfasis sobre la pasiva, lo que puede explicar su poca idoneidad para replicación activa.

En este escenario, Sensei analiza, por un lado, la transferencia de estado, aportando soluciones para aplicaciones no particionables, esto es, aplicaciones en grupo que, en caso de sufrir problemas de comunicaciones que la dividan en subgrupos, permiten que sólo uno de los subgrupos, en el mejor de los casos, permanezca operativo. Por otro lado, se centra en la metodología necesaria para factorizar una aplicación tolerante a fallos en componentes replicables, facilitando, tanto la reutilización de esos componentes, como el proceso de su replicación. El resultado que se persigue es la equiparación de las aplicaciones replicadas con sus homólogas sin soporte de replicación, lo que permite un diseño considerablemente simplificado.

Las aportaciones que esta memoria realiza se dividen en tres grupos. El primer grupo se centra en la investigación sobre la transferencia de estado:

- Consideramos la transferencia de estado como un proceso *sobre* el modelo de sincronía virtual, lo que permite implantar las soluciones obtenidas sobre cualquier sistema de comunicaciones en grupo. Analizamos, entonces, las condiciones que deben verificar un grupo y sus mensajes para permitir transferencias de estado *simples*, siendo simples aquellas que pueden efectuarse con el flujo normal de mensajes, sin precisar bloqueos o tratamientos especiales. El resultado es que únicamente las aplicaciones muy sencillas soportan estas transferencias.
- Desarrollamos algoritmos genéricos que posibilitan la transferencia basada en mensajes para todo tipo de aplicaciones. Estos algoritmos soportan transferencias realizadas en varios pasos y se basan en el bloqueo de los mensajes del grupo mientras haya una transferencia en curso.
- Estudiamos las soluciones obtenidas en el marco del modelo de sincronía virtual. La primera conclusión es que este modelo debe extenderse para comunicar los eventos de bloqueo de vistas; en caso contrario no es posible el bloqueo de mensajes, lo que imposibilita el proceso de transferencia. La segunda conclusión

es que toda transferencia debe cancelarse y reiniciarse en caso de cambios de vistas. Debido al impacto negativo que esta reiniciación puede tener en el rendimiento de un grupo, proponemos soluciones alternativas, describiendo su impacto en el modelo de sincronía virtual.

- Diseñamos protocolos de alto y bajo nivel (interfaz de aplicación e intercambio de mensajes) que posibilitan una transferencia de estado cuya principal característica es la flexibilidad. Admiten transferencias secuenciales, de tipo *push* o *pull*, delegación en la elección del coordinador sobre la aplicación, empleo de propiedades de miembros, transferencias concurrentes e interrupciones y reanudaciones de las mismas, al soportar un canal de comunicación bidireccional entre los miembros que participan en la transferencia.
- Analizamos los diferentes protocolos de bajo nivel, estudiando los beneficios de unos u otros según el tipo de aplicación, topología de la red, frecuencia de errores, etc. La conclusión básica es que el protocolo *push* resulta más eficiente salvo que el grupo disponga de soporte hardware de mensajería multipunto. Es, por otra parte, más complejo, por lo que resulta más difícil de implementar.
- Especificamos la interfaz de aplicación de estos protocolos en JavaRMI y OMG/IDL. Además, integramos esta interfaz con el servicio CORBA de tolerancia a fallos, extendiendo su funcionalidad.

La segunda línea de investigación con el sistema desarrollado, Sensei, la constituye el estudio de técnicas de diseño de aplicaciones tolerantes a fallos:

- Analizamos los patrones de implementación, proponiendo soluciones específicas para el problema de sincronización de mensajes.
- Estudiamos el proceso de replicación de una aplicación con dos resultados. Primero, describimos el proceso de cómo automatizar la replicación de una aplicación. Esta automatización se realiza a partir de la descripción de su interfaz y de una implementación sin soporte de replicación de esta interfaz. Segundo, comprobamos la dificultad de automatizar la replicación de una aplicación compleja y la necesidad de factorizarla en componentes, cuya replicación ya sí es automatizable.
- Desarrollamos una metodología basada en componentes. La principal motivación es la delegación de la lógica de replicación sobre esos componentes, permitiendo que el diseño de una aplicación tolerante a fallos pueda centrarse en la lógica de esa aplicación.
- Estudiamos el ciclo de vida de los componentes replicados, con la pretensión de equipararlos a los componentes *normales*, no replicados. El resultado es la necesidad de permitir componentes replicados dinámicos: no existen desde el principio, son creados dinámicamente en una máquina y su existencia es propagada a las demás réplicas. Describimos las posibilidades de implementación, centrándonos en el empleo de *dominios*, que permiten gestionar conjuntamente los componentes replicados de una aplicación,

- Estudiamos los problemas de concurrencia sobre los componentes replicados, proponiendo una solución basada en *monitores*, cuya interfaz e implementación detallamos. Proponemos el empleo de *transacciones* como la forma de asegurar la consistencia del grupo en caso de problemas al actualizar varios componentes en un miembro, y describimos su implementación en base al empleo de dominios.
- Analizamos los impactos de las soluciones propuestas sobre el modelo de sincronía virtual. El empleo de transacciones no es compatible con este modelo, por lo que analizamos las consecuencias de esa incompatibilidad, describiendo las aplicaciones que, por lo tanto, no pueden emplearlas.
- Identificamos los contenedores (de datos/objetos) como las estructuras replicables genéricas de una aplicación, y proponemos el diseño de una librería básica de contenedores replicados con el objetivo de facilitar su reutilización. Esta librería constituye la parte de este proyecto denominada SenseiUMA.

El resultado que ofrece esta metodología es la utilización efectiva de componentes, dinámicos o estáticos, que simplifica el desarrollo de aplicaciones tolerantes a fallos. Sin embargo, la transparencia no es total, y debe aún incluirse cierta lógica de replicación. La situación es, en cierto modo, comparable a la de las aplicaciones distribuidas CORBA: con estas aplicaciones, la abstracción que se busca es la de una interacción local de componentes, aun cuando esos componentes son remotos y la transparencia de acceso no resulta total, lo que supone un trabajo inicial para localizar esos componentes y una problemática posterior mayor en su acceso, por los posibles problemas de comunicaciones. A pesar de esta falta de transparencia total, consideramos que la metodología desarrollada facilita extraordinariamente el diseño e implementación de las aplicaciones tolerantes a fallos.

La tercera aportación que Sensei realiza es un soporte completo de las dos líneas de investigación mencionadas, permitiendo comprobar su validez teórica con la práctica (la figura 12.1 muestra los distintos componentes de Sensei y sus interacciones):

- Implementamos un sistema de comunicaciones fiables en grupo. Este sistema, SenseiGMS, implementa sobre CORBA y JavaRMI el modelo de sincronía virtual con una interfaz orientada a objetos. Su funcionalidad se limita a la definida en este modelo de tal forma que pudiera ser fácilmente reemplazable por otros sistemas más eficientes. Hemos diseñado también el algoritmo de paso de testigo que emplea SenseiGMS, soportando exclusivamente mensajes con orden total.
- Desarrollamos *VirtualNet*, una herramienta que permite probar la corrección de SenseiGMS, así como la de las aplicaciones tolerantes a fallos que lo empleen. Soporta el desarrollo de topologías virtuales y permite comprobar la reacción de las aplicaciones a los problemas de comunicaciones.

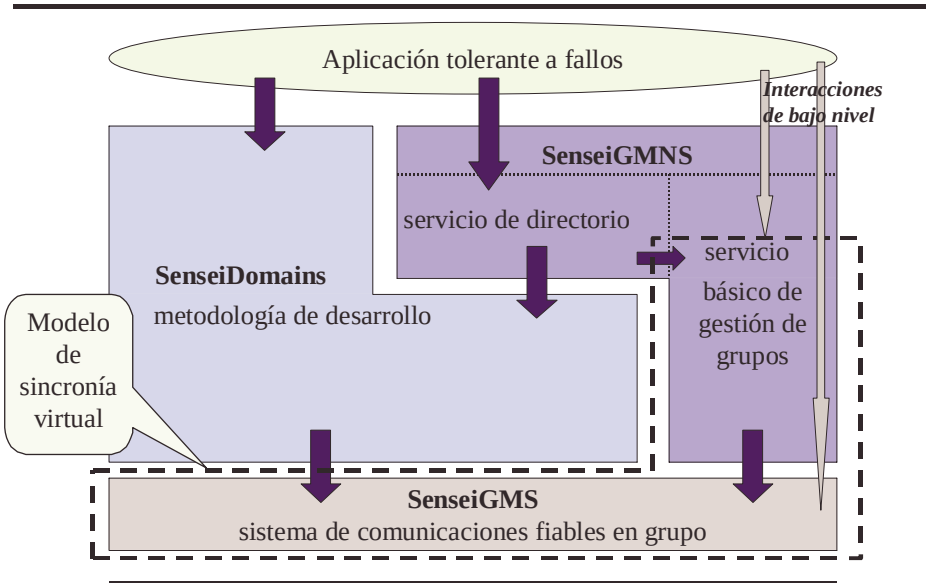

Figura 12.1. Componentes de Sensei

- Implementamos la metodología desarrollada previamente, así como los protocolos de trasferencia de estado. Esta implementación, *SenseiDomains*, soporta aplicaciones CORBA y JavaRMI, al igual que SenseiGMS, que es el substrato de comunicaciones que emplea. Una de las principales características de este sistema es que, si bien soporta toda la funcionalidad de la metodología, las aplicaciones sólo observan la que necesitan, redundando en una mejor eficiencia y una implementación de la aplicación más simple.
- Finalmente, hemos desarrollado SenseiGMNS, un servicio de directorio para SenseiGMS. Sin embargo, la principal aportación de SenseiGMNS es que su diseño es un ejemplo de aplicación de la metodología desarrollada y de SenseiDomains por extensión.

12.2. Trabajo Futuro

Nuestro trabajo futuro sobre Sensei tiene dos vertientes: consolidación del trabajo realizado y evolución con nuevas propuestas.

Dentro de la consolidación, tenemos pendiente la substitución del substrato de comunicaciones empleado, SenseiGMS, por otros sistemas de comunicaciones en grupo. El esfuerzo asociado no se realizaría para incrementar la eficiencia del sistema, sino para verificar la corrección de la implementación de la metodología. Aunque SenseiDomains emplea únicamente la interfaz pública de SenseiGMS, sólo la práctica permitirá comprobar las deducciones erróneas que hayamos podido hacer

sobre el comportamiento del modelo de sincronía virtual. Hasta el momento, hemos podido ya encapsular *Ensemble* bajo la interfaz de SenseiGMS, aunque limitados a la implementación CORBA. Además, como inicialmente la interfaz de SenseiGMS se diseñó bajo la influencia de *Ensemble*, puede resultar más interesante realizar la substitución con otro sistema diferente, como *Transis*.

Para consolidar la metodología desarrollada debemos, también, validarla con un número suficiente de aplicaciones, preferiblemente implementadas fuera de nuestro grupo de trabajo, para obtener así respuestas independientes. Un aspecto que ya consideramos mejorable en la metodología es su tratamiento de la transferencia de estado con respecto a múltiples componentes. Transferir primero el estado de los componentes más antiguos ofrece un resultado válido, pero ineficiente; la transferencia simultánea de estados de varios componentes puede, obviamente, provocar errores de consistencia, salvo que la aplicación pueda anticiparlos y obtener así una eficiencia mayor.

Consideramos también un problema de consolidación el empleo de los monitores replicados. Su interfaz no define operaciones de guarda (*wait,notify*), cuya implementación no precisa, sin embargo, de un gran esfuerzo. No obstante, su objetivo en la metodología es la definición de regiones críticas, permitiendo la sincronización de acceso a recursos. Las operaciones de guarda se usan, en el caso de aplicaciones *multithread*, para la colaboración entre múltiples *threads*, mientras que la metodología desarrollada no prevé la colaboración entre componentes.

La principal línea de investigación actual es SenseiUMA, el diseño de una librería de componentes estándar replicados. Este diseño lo realizamos basándonos en los contenedores existentes en Java (*java.util*), pero su contenido no está aún completado. También hemos debido postergar su implementación, pues bajo CORBA es sólo posible perdiendo gran parte de la transparencia, ya que la aplicación debe poder especificar cómo transferir el estado de cada elemento en el contenedor. Esta implementación sí que está siendo posible con JavaRMI, pero debemos investigar las mejoras que hacen especialmente interesante el empleo efectivo de los componentes.

Finalmente, el trabajo desarrollado considera grupos no particionables, por lo que se podría considerar la principal evolución de Sensei su aplicabilidad sobre aplicaciones tolerantes a fallos que permitan particiones. Esta línea de investigación es, sin embargo, sumamente complicada. Los algoritmos de transferencia de estado desarrollados son exclusivos para sistemas no particionables y lo mismo ocurre con la metodología. Debe destacarse que en sistemas que admiten varios grupos activos, es precisa la intervención de la aplicación para decidir cómo mezclar coherentemente sus estados cuando se reúnan de nuevo esos grupos, y esta intervención implica un uso no transparente de los componentes. Sin embargo, este mezclado de estados es, de hecho, más sencillo cuando se emplean componentes básicos. Por ejemplo, un componente replicado *lista* que se divide en dos grupos, pero mantiene un historial de sus cambios, puede luego realizar, *en muchos casos*, un mezclado coherente de los datos en base a esos cambios.

En cualquier caso, esta línea de investigación supone un cambio de criterio completo con respecto al trabajo desarrollado hasta el momento, que busca una simplificación del proceso de replicación, haciendo que su lógica resulte lo más transparente posible para la aplicación. Una aplicación que soporte particionado debe imbuir desde un principio toda la lógica necesaria para soportar estados inconsistentes, perdiendo, consecuentemente, interés en esa transparencia. Por lo tanto, consideramos actualmente que el soporte de aplicaciones particionadas en la metodología tiene un interés meramente teórico.

Apéndice A. Especificación CORBA de SenseiGMS

A.1. GroupMembershipService.idl

```
#ifndef VIRTUAL_SYNCHRONY_IDL
#define VIRTUAL_SYNCHRONY_IDL

module sensei
{
  module middleware
  {
    module GMS
    {
      typedef long GroupMemberId;
      const GroupMemberId INVALID_GROUP_MEMBER_ID = 0;

      struct View
      {
        long viewId;
        GroupMemberIdList members;
        GroupMemberIdList newMembers;
        GroupMemberIdList expulsedMembers;
      };
```

```
      valuetype Message
      {
      };

      interface GroupHandler
      {
        GroupMemberId getGroupMemberId();
        boolean isValidGroup();
        boolean leaveGroup();
        boolean castMessage(in Message msg);
        boolean sendMessage(in GroupMemberId target, in Message msg);
      };

      interface GroupMember
      {
        void processPTPMessage(in GroupMemberId sender, in Message msg);
        void processCastMessage(in GroupMemberId sender, in Message msg);
        void memberAccepted(in GroupMemberId identity, in GroupHandler handler,
                      in View theView);
        void changingView();
        void installView(in View theView);
        void excludedFromGroup();
      };
    };
  };
};

#endif  VIRTUAL_SYNCHRONY_IDL
```

A.2. InternalSensei.idl

```
#ifndef INTERNAL_SENSEI_IDL
#define INTERNAL_SENSEI_IDL

#include "GroupMembershipService.idl"

module sensei
{
  module middleware
  {
    module GMS
    {
```

```
interface SenseiGMSMember;

struct Token
{
  GroupMemberId creator;
  long id;
};

struct InternalViewId
{
  long id;
  long installing;
};

typedef sequence <SenseiGMSMember> SenseiGMSMemberList;

struct InternalView
{
  InternalViewId viewId;
  SenseiGMSMemberList members;
  GroupMemberIdList memberIds;
  GroupMemberIdList newMembers;
};

struct MessageId
{
  long view;
  long id;
};

typedef sequence <Message> MessageList;

struct AllowTokenRecoveryAnswer
{
  boolean validCommunication;
  boolean recoveryAllowed;
  MessageId lastMessage;
};

interface SenseiGMSMember : GroupHandler
{
  boolean receiveToken(in GroupMemberId sender, in Token theToken);
  boolean receiveView(in GroupMemberId sender, in InternalView view,
                in Token viewToken);
  boolean receiveMessages(in GroupMemberId sender, in MessageList messages,
                in MessageId msgId);
```

```
        boolean confirmMessages(in GroupMemberId sender, in MessageId msgId);
        boolean receivePTPMessages(in GroupMemberId sender, in MessageList messages,
                        in MessageId msgId);
        AllowTokenRecoveryAnswer allowTokenRecovery(in GroupMemberId sender,
                        in InternalViewId myViewId);
        boolean recoverToken(in GroupMemberId sender);
        boolean addGroupMember(in SenseiGMSMember other);
        boolean createGroup();
        boolean joinGroup(in SenseiGMSMember group);
      };
    };
  };
};

#endif  INTERNAL_SENSEI_IDL
```

Apéndice B. Especificación CORBA de SenseiDomains

B.1. DomainExceptions.idl

```
#ifndef DOMAIN_EXCEPIONS_IDL
#define DOMAIN_EXCEPIONS_IDL

module sensei
{
  module middleware
  {
    module domains
    {
      enum MemberStateExceptionReason
      {
        MemberNotJoined,
        MemberJoined,
        MemberWithoutState,
        MemberExcluded
      };
      exception MemberStateException
      {
        MemberStateExceptionReason reason;
```

```
      };
    };
  };
};

#endif DOMAIN_EXCEPIONS_IDL
```

B.2. StateTransfer.idl

```
#ifndef STATE_TRANSFER_IDL
#define STATE_TRANSFER_IDL

#include "GroupMembershipService.idl"

module sensei
{
  module middleware
  {
    module domains
    {
      typedef sensei::middleware::GMS::GroupMember   GroupMember;
      typedef sensei::middleware::GMS::GroupMemberId GroupMemberId;

      abstract valuetype State {};
      typedef GroupMemberId Location;
      typedef sequence <Location> Locations;

      interface Checkpointable : GroupMember
      {
        State getState();
        void setState (in State s);
      };

      typedef string Name;
      typedef string Value;
      struct Property
      {
        Name nam;
        Value val;
      };
      typedef sequence <Property> Properties;

      valuetype PhaseCoordination
```

```
{
  public boolean transferFinished;
};

interface ExtendedCheckpointable : Checkpointable
{
  void assumeState ();
};

interface BasicStateHandler : GroupMember
{
  void assumeState ();
  void startTransfer (in Locations joiningMembers,
                inout PhaseCoordination phase);
  State getState (inout PhaseCoordination phase);
  void setState (in State s, in PhaseCoordination phase);
  void stopTransfer (in Locations joiningMembers, in boolean transferFinished);
};

interface StateHandler : BasicStateHandler
{
  void syncTransfer (in Location coordinator, inout PhaseCoordination phase);
  void interruptTransfer (inout PhaseCoordination phase);
  void continueTransfer (in Locations joiningMembers,
                inout PhaseCoordination coordinator,
                in PhaseCoordination joining);
};

struct CoordinatorInformation
{
  Location statusMember;
  Locations currentCoordinations;
};

typedef sequence <CoordinatorInformation> CoordinatorInformationList;

interface CoordinatorElector
{
  Location getCoordinator (in Location loc, in CoordinatorInformationList info);
};

enum BehaviourOnViewChanges
{
  MembersOnTransferExcludedFromGroup,
  StatelessMembersDoNotBelongToGroup,
  StatelessMembersBelongToGroup
```

```
      };
    };
  };
};

#endif STATE_TRANSFER_IDL
```

B.3. Properties.idl

```
#ifndef PROPERTIES_IDL
#define PROPERTIES_IDL

#include "DomainExceptions.idl"
#include "StateTransfer.idl"

module sensei
{
  module middleware
  {
    module domains
    {
      struct MemberProperties
      {
        GroupMemberId member;
        Properties props;
      };

      typedef sequence <MemberProperties> MemberPropertiesList;

      exception PropertiesDisabledException
      {
      };

      interface PropertiesListener
      {
        void propertiesUpdated (in Location loc);
      };

      interface PropertiesHandler
      {
        void enableProperties() raises (MemberStateException);
        boolean arePropertiesEnabled();
        void setPropertiesListener(in PropertiesListener listener)
               raises (PropertiesDisabledException);
```

```
        MemberPropertiesList getAllProperties ()
               raises (MemberStateException, PropertiesDisabledException);
        MemberPropertiesList getPropertyForAllMembers (in Name nam)
               raises (MemberStateException, PropertiesDisabledException);
        Properties getMemberProperties (in Location loc)
               raises (MemberStateException, PropertiesDisabledException);
        Value getMemberProperty (in Location loc, in Name n)
               raises (MemberStateException, PropertiesDisabledException);
        Properties getProperties() raises (PropertiesDisabledException);
        Value getProperty(in Name n) raises (PropertiesDisabledException);
        void setProperties (in Properties props)
               raises (MemberStateException, PropertiesDisabledException);
        void addProperties (in Properties props)
               raises (MemberStateException, PropertiesDisabledException);
        void removeProperties (in Properties props)
               raises (MemberStateException, PropertiesDisabledException);
      };
    };
  };
};

#endif PROPERTIES_IDL
```

B.4. SubgroupsHandler.idl

```
#ifndef SUBGROUPS_HANDLER_IDL
#define SUBGROUPS_HANDLER_IDL

#include "DomainExceptions.idl"
#include "StateTransfer.idl"

module sensei
{
  module middleware
  {
    module domains
    {
      typedef long SubgroupId;
      typedef sequence<SubgroupId> SubgroupIdsList;
      const long EVERY_SUBGROUP = 0;
      const long MAX_STATIC_SUBGROUP_ID = 65535;

      enum SubgroupsHandlerExceptionReason
```

```
{
  SubgroupIdAlreadyInUse,
  InvalidStaticSubgroupId,
  InvalidDynamicSubgroupId,
  DynamicBehaviourNotRegistered
};

exception SubgroupsHandlerException
{
  SubgroupsHandlerExceptionReason reason;
};

valuetype DynamicSubgroupInfo
{
};

valuetype DynamicSubgroupInfoAsString : DynamicSubgroupInfo
{
  public string info;
};

interface DynamicSubgroupsUser
{
  GroupMember acceptSubgroup(in SubgroupId id, in DynamicSubgroupInfo info);
  GroupMember subgroupCreated(in GroupMemberId creator, in SubgroupId id,
                in DynamicSubgroupInfo info);
  void subgroupRemoved(in GroupMemberId remover, in SubgroupId id,
                in DynamicSubgroupInfo info);
};

interface SubgroupsHandler
{
  void setDynamicSubgroupsUser(in DynamicSubgroupsUser user)
         raises (MemberStateException);
  void registerSubgroup(in SubgroupId uniqueSubgroupId, in GroupMember subgroup)
         raises (MemberStateException, SubgroupsHandlerException);
  void castSubgroupCreation(in DynamicSubgroupInfo info)
         raises (MemberStateException, SubgroupsHandlerException);
  SubgroupId createSubgroup(in DynamicSubgroupInfo info,
                in GroupMember subgroup)
         raises (MemberStateException, SubgroupsHandlerException);
  boolean removeSubgroup(in SubgroupId id, in DynamicSubgroupInfo info)
         raises (MemberStateException, SubgroupsHandlerException);

  SubgroupIdsList getSubgroups();
  GroupMember getSubgroup(in SubgroupId id);
```

```
        SubgroupId getSubgroupId(in GroupMember subgroup);
      };
    };
  };
};

#endif SUBGROUPS_HANDLER_IDL
```

B.5. DomainMessage.idl

```
#ifndef DOMAIN_MESSAGE_IDL
#define DOMAIN_MESSAGE_IDL

#include "SubgroupsHandler.idl"

module sensei
{
  module middleware
  {
    module domains
    {
      typedef sensei::middleware::GMS::Message  Message;
      typedef long MessageId;

      valuetype DomainMessage : Message
      {
        public SubgroupId subgroup;
        public boolean unqueuedOnST;
        public boolean unTransactionable;
        public boolean waitReception;
        public MessageId  id;
      };
    };
  };
};

#endif DOMAIN_MESSAGE_IDL
```

B.6. Concurrency.idl

```
#ifndef CONCURRENCY_IDL
```

```
#define CONCURRENCY_IDL

#include "DomainMessage.idl"
#include "StateTransfer.idl"

module sensei
{
  module middleware
  {
    module domains
    {
      typedef sensei::middleware::GMS::GroupMemberIdList GroupMemberIdList;

      exception MonitorException{};

      interface Monitor
      {
        void lock() raises (MemberStateException);
        void unlock() raises (MonitorException, MemberStateException);
      };

      interface GroupMonitor : Monitor, ExtendedCheckpointable
      {
      };

      exception TransactionException{};

      interface TransactionsHandler
      {
        void startTransaction(in Monitor locker)
               raises (MonitorException, TransactionException, MemberStateException);
        void endTransaction()
               raises (MonitorException, TransactionException, MemberStateException);
      };
    };
  };
};

#endif CONCURRENCY_IDL
```

B.7. DomainGroupHandler.idl

```
#ifndef DOMAIN_GROUP_HANDLER_IDL
```

```
#define DOMAIN_GROUP_HANDLER_IDL

#include "Concurrency.idl"
#include "Properties.idl"

module sensei
{
  module middleware
  {
    module domains
    {
      typedef sensei::middleware::GMS::GroupHandler GroupHandler;

      enum DomainExpulsionReason
      {
        GMSLeaveEvent,
        WrongPropertyAllowance,
        WrongStaticSubgroupsComposition,
        WrongCoordinatorElectionPolicy,
        WrongBehaviourMode,
        WrongDynamicPolicy,
        WrongSubgroupsTypes,
        SubgroupError
      };

      exception SyncCastDomainMessageProcessedException
      {
      };

      interface DomainGroupUser
      {
        void domainAccepted(in GroupMemberId id);
        void domainExpulsed(in DomainExpulsionReason reason);
        void stateObtained(in boolean assumed);
        void offendingSubgroup(in SubgroupId subgroup, in string reason);
      };

      interface DomainGroupHandler : GroupHandler, GroupMember, PropertiesHandler,
               SubgroupsHandler, TransactionsHandler
      {
        void setBehaviourMode(in BehaviourOnViewChanges mode)
               raises (MemberStateException);
        void setCoordinatorElector(in CoordinatorElector elector)
               raises (MemberStateException);
        void setDomainGroupUser(in DomainGroupUser user)
               raises (MemberStateException);
```

```
        GroupMemberIdList getStatefulMembers()
               raises (MemberStateException);
        boolean syncCastDomainMessage(in DomainMessage message,
                       in boolean normalProcessing)
               raises (MemberStateException);
        void syncCastDomainMessageProcessed()
               raises(MemberStateException, SyncCastDomainMessageProcessedException);
      };
    };
  };
};

#endif DOMAIN_GROUP_HANDLER_IDL
```

B.8. InternalDomainMessages.idl

```
#ifndef INTERNAL_DOMAIN_MESSAGES_IDL
#define INTERNAL_DOMAIN_MESSAGES_IDL

#include "DomainGroupHandler.idl"

module sensei
{
  module middleware
  {
    module domains
    {
      enum StateTransferType
      {
        STT_StateHandler,
        STT_BasicStateHandler,
        STT_ExtendedCheckpointable,
        STT_Checkpointable,
        STT_StatelessTransfer
      };

      struct SubgroupProperties
      {
        SubgroupId id;
        DynamicSubgroupInfo info;
        StateTransferType type;
      };
      typedef sequence <SubgroupProperties> SubgroupsInfo;
```

```
struct ReplicaInformation
{
  GroupMemberId replicaId;
  GroupMemberIdList replicasToCoordinate;
};
typedef sequence <ReplicaInformation> ReplicasInformation;

valuetype MessageP : DomainMessage
{
  public long viewId;
  public BehaviourOnViewChanges behaviour;
  public boolean coordinatorElectorRegistered;
  public boolean dynamicSubgroupsAllowed;
  public long nextDynamicSubgroupId;
  public SubgroupsInfo subgroupsInformation;
  public MemberPropertiesList props;
  public ReplicasInformation replicasInfo;
};

struct SubgroupCoordination
{
  SubgroupId id;
  PhaseCoordination phase;
};
typedef sequence <SubgroupCoordination> SubgroupCoordinations;

valuetype MessageC : DomainMessage
{
  public long viewId;
  public SubgroupCoordinations coordinations;
  public boolean containOwnProperties;
  public Properties props;
};

valuetype MessageE : DomainMessage
{
  public long viewId;
  public GroupMemberId member;
  public Properties props;
};

struct SubgroupState
{
  SubgroupId id;
  PhaseCoordination phase;
  State subState;
```

```
      };
      typedef sequence <SubgroupState> SubgroupStates;

      valuetype MessageS : DomainMessage
      {
        public long viewId;
        public SubgroupStates states;
      };

      valuetype MessageMemberProperties : DomainMessage
      {
        public Properties props;
      };

      valuetype MessageCreateSubgroup : DomainMessage
      {
        public DynamicSubgroupInfo info;
        public long creatorId;
      };

      valuetype MessageRemoveSubgroup : DomainMessage
      {
        public SubgroupId subgroupToRemove;
        public DynamicSubgroupInfo info;
      };

      valuetype MonitorMessage : DomainMessage
      {
        public boolean lock;
      };

      valuetype EndOfTransactionMessage : DomainMessage
      {
        public sequence<Message> actions;
      };

      valuetype MonitorState : State
      {
        public GroupMemberIdList locks;
      };

    };
  };
};

#endif INTERNAL_DOMAIN_MESSAGES_IDL
```

Apéndice C. Especificación CORBA de SenseiGMNS

C.1. GroupMembershipNamingService.idl

```
#ifndef GROUP_MEMBERSHIP_NAMING_SERVICE
#define GROUP_MEMBERSHIP_NAMING_SERVICE

#include "GroupMembershipService.idl"
#include "DomainExceptions.idl"

module sensei
{
  module middleware
  {
    module GMNS
    {
      typedef sensei::middleware::GMS::GroupHandler  GroupHandler;
      typedef sensei::middleware::GMS::GroupMember   GroupMember;

      exception InvalidGroupHandlerException{};

      exception InvalidGroupHandlerFactoryException{};
```

```
exception GroupHandlerFactoryException{};

interface GroupHandlerFactory
{
  GroupHandler create() raises (GroupHandlerFactoryException);
};

valuetype GroupHandlerFactoryCreator
{
  GroupHandlerFactory create(in GroupMember member)
         raises (GroupHandlerFactoryException);
};

interface GroupMembershipBasicService
{
  GroupHandlerFactoryCreator getFactoryCreator();
  GroupHandler createGroup(in GroupHandlerFactory theFactory)
         raises (InvalidGroupHandlerFactoryException,
                 GroupHandlerFactoryException);
  GroupHandler joinGroup(in GroupHandler group,
                 in GroupHandlerFactory theFactory)
         raises (InvalidGroupHandlerException,
                 InvalidGroupHandlerFactoryException,
                 GroupHandlerFactoryException);
};


interface ReplicatedServer
{
};
typedef sequence<ReplicatedServer> ReplicatedServersList;


interface GroupMembershipNamingService
{
  GroupHandlerFactoryCreator getFactoryCreator();
  GroupHandler findAndJoinGroup(in string groupName,
                 in GroupHandlerFactory theFactory,
                 in string memberName,
                 in ReplicatedServer clientsReference)
         raises (sensei::middleware::domains::MemberStateException,
                 InvalidGroupHandlerFactoryException,
                 GroupHandlerFactoryException);

  void leaveGroup(in string groupName, in GroupHandler member);
  ReplicatedServersList getGroup(in string groupName)
```

```
                raises (sensei::middleware::domains::MemberStateException);
        ReplicatedServer getGroupServer(in string groupName, in string memberName)
                raises(sensei::middleware::domains::MemberStateException);
        ReplicatedServer getValidGroupServer(in string groupName)
                raises(sensei::middleware::domains::MemberStateException);
      };
    };
  };
};

#endif  GROUP_MEMBERSHIP_NAMING_SERVICE
```

C.2. MemberInfo.idl

```
#ifndef MEMBER_INFO
#define MEMBER_INFO

#include "GroupMembershipNamingService.idl"

module sensei
{
  module middleware
  {
    module GMNS
    {
      struct MemberInfo
      {
        GroupHandler handler;
        string name;
        ReplicatedServer publicReference;
      };
      typedef sequence<MemberInfo> MemberInfoList;
    };
  };
};

#endif  MEMBER_INFO
```

C.3. SetMemberInfo.idl

```
#ifndef SET_MEMBER_INFO
#define SET_MEMBER_INFO
```

```
#include "DomainGroupHandler.idl"
#include "MemberInfo.idl"

module sensei
{
  module middleware
  {
    module GMNS
    {
      typedef sensei::middleware::domains::DomainMessage DomainMessage;
      typedef sensei::middleware::domains::ExtendedCheckpointable
               ExtendedCheckpointable;
      typedef sensei::middleware::domains::State State;

      interface SetMemberInfoObservable;

      interface SetMemberInfoObserver
      {
        void addDone(in SetMemberInfoObservable observable, in MemberInfo member);
        void removeDone(in SetMemberInfoObservable observable, in MemberInfo member);
      };

      interface SetMemberInfoObservable
      {
        void addObserver(in SetMemberInfoObserver observer);
        boolean removeObserver(in SetMemberInfoObserver observer);
      };

      interface SetMemberInfo : ExtendedCheckpointable, SetMemberInfoObservable
      {
        boolean add(in MemberInfo member)
               raises (sensei::middleware::domains::MemberStateException);
        boolean remove(in MemberInfo handler)
               raises (sensei::middleware::domains::MemberStateException);
        MemberInfo get()
               raises (sensei::middleware::domains::MemberStateException);
        MemberInfoList toArray()
               raises (sensei::middleware::domains::MemberStateException);
      };

      valuetype SetMemberInfoChangedMessage : DomainMessage
      {
        public MemberInfo info;
        public boolean add;
      };
```

```
      valuetype SetMemberInfoState : State
      {
        public sequence<MemberInfo> members;
      };
    };
  };
};

#endif  SET_GROUP_HANDLER
```

C.4. MapStringSet.idl

```
#ifndef MAP_STRING_SET
#define MAP_STRING_SET

#include "SetMemberInfo.idl"

module sensei
{
  module middleware
  {
    module GMNS
    {
      typedef sensei::middleware::domains::SubgroupId SubgroupId;

      exception InvalidSetMemberInfo{};

      interface MapStringSetObservable;

      interface MapStringSetObserver
      {
        void clearDone(in MapStringSetObservable observable);
        void putDone(in MapStringSetObservable observable, in string groupName,
                     in SetMemberInfo set);
        void removeDone(in MapStringSetObservable observable, in string groupName,
                     in SetMemberInfo set);
      };

      interface MapStringSetObservable
      {
        void addObserver(in MapStringSetObserver observer);
        boolean removeObserver(in MapStringSetObserver observer);
      };
```

```
      typedef sequence<string> stringList;

      interface MapStringSet : ExtendedCheckpointable, MapStringSetObservable
      {
        SetMemberInfo put(in string groupName, in SetMemberInfo set)
               raises (InvalidSetMemberInfo,
                       sensei::middleware::domains::MemberStateException);
        SetMemberInfo remove(in string groupName)
               raises (sensei::middleware::domains::MemberStateException);
        SetMemberInfo get(in string groupName)
               raises (sensei::middleware::domains::MemberStateException);
        stringList getKeys()
               raises (sensei::middleware::domains::MemberStateException);
      };

      valuetype MapStringSetPutMessage : DomainMessage
      {
        public string key;
        public SubgroupId value;
      };

      valuetype MapStringSetRemoveMessage : DomainMessage
      {
        public string key;
      };

      struct MapStringSetEntry
      {
        string key;
        SubgroupId value;
      };

      valuetype MapStringSetState : State
      {
        public sequence<MapStringSetEntry> groups;
      };
    };
  };
};

#endif  MAP_STRING_SET
```

C.5. GroupsCheckerState.idl

```
#ifndef GROUPS_CHECKER_STATE
#define GROUPS_CHECKER_STATE

#include "StateTransfer.idl"

module sensei
{
  module middleware
  {
    module GMNS
    {
      typedef sensei::middleware::domains::State State;
      typedef sensei::middleware::GMS::GroupMemberId GroupMemberId;

      valuetype GroupsCheckerState : State
      {
        public GroupMemberId checker;
      };
    };
  };
};

#endif  GROUPS_CHECKER_STATE
```

Apéndice D. Ejemplos de Componentes Replicados

Este apéndice incluye el código de algunas clases relevantes en *SenseiGMNS*, siguiendo la explicación del capítulo 11. Estas clases implementan los componentes replicados empleados; en concreto, *SetMemberInfoImpl.java* define el comportamiento del componente *SetMemberInfo* y *MapStringSetImpl.java* implementa el componente *MapStringSet*.

D.1. SetMemberInfoImpl.java

```
package sensei.GMNS;

import sensei.middleware.GMS.*;
import sensei.middleware.GMNS.*;
import sensei.middleware.domains.*;
import sensei.middleware.util.ObjectsHandling;
import sensei.util.Error;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

class SetMemberInfoImpl extends SetMemberInfoBaseImpl
```

```
{
  private SetMemberInfoImpl(DomainGroupHandler groupHandler) throws Exception
  {
    this.groupHandler=groupHandler;
    set=new HashSet();
    factory = new GMNSFactory(groupHandler);
    observersHandler = new SetMemberInfoObserversHandler(thisSetMemberInfo);
  }

  public int getSubgroupId()
  {
    return subgroupId;
  }

  private void setSubgroupId(int subgroupId)
  {
    this.subgroupId=subgroupId;
    factory.setSubgroupId(subgroupId);
  }

  public SetMemberInfoImpl(int subgroupId, DomainGroupHandler groupHandler)
       throws Exception
  {
    this(groupHandler);
    setSubgroupId(subgroupId);
  }

  public SetMemberInfoImpl(DynamicSubgroupInfo dynamicSubgroupInfo,
                      DomainGroupHandler groupHandler)
    throws SubgroupsHandlerException, MemberStateException, Exception
  {
    this(groupHandler);
    subgroupId = groupHandler.createSubgroup(dynamicSubgroupInfo, thisSetMemberInfo);
    setSubgroupId(subgroupId);
  }

  public void register()
       throws MemberStateException, SubgroupsHandlerException, Exception
  {
    groupHandler.registerSubgroup(subgroupId, thisSetMemberInfo);
  }

  public static void register(DynamicSubgroupInfo dynamicSubgroupInfo,
       DomainGroupHandler groupHandler)
    throws MemberStateException, SubgroupsHandlerException, Exception
  {
```

```
  groupHandler.castSubgroupCreation(dynamicSubgroupInfo);
}

void freeResources()
{
  if (set!=null) {
    try{deactivate();}catch(Exception ex){}
    set.clear();
    set=null;
    groupHandler=null;
    factory=null;
  }
}

public boolean add(MemberInfo info) throws MemberStateException
{
  boolean result=false;
  try {
    factory.castSetMessage(info, true);
    result=addMemberInfo(info);
    factory.syncMessageCompleted();
  }
  catch(Exception ex) { Error.unhandledException(Consts.AREA, ex); }
 return result;
}

public synchronized MemberInfo get() throws MemberStateException
{
  Iterator it=set.iterator();
  return it.hasNext()? (MemberInfo) it.next(): null;
}

public boolean remove(MemberInfo info) throws MemberStateException
{
  boolean result=false;
  try {
    factory.castSetMessage(info, false);
    result=removeMemberInfo(info);
    factory.syncMessageCompleted();
  }
  catch(Exception ex) { Error.unhandledException(Consts.AREA, ex); }
  return result;
}

public synchronized MemberInfo[] toArray()
{
```

```
  int size = set.size();
  MemberInfo[] ret = new MemberInfo[size];
  set.toArray(ret);
  return ret;
}

public synchronized void addObserver(SetMemberInfoObserver observer)
{
  observersHandler.addObserver(observer, getMembers());
}

public boolean removeObserver(SetMemberInfoObserver observer)
{
  return observersHandler.removeObserver(observer);
}

public synchronized State getState()
{
  return factory.createSetState(getMembers());
}

MemberInfo[] getMembers()
{
  int size = set.size();
  MemberInfo members[] = new MemberInfo[size];
  if (size>0) {
    int i=0;
    Iterator it = set.iterator();
    while(it.hasNext())
      members[i++]=(MemberInfo) it.next();
  }
  return members;
}

public synchronized void setState(State state)
{
  set.clear();
  MemberInfo members[]=((SetMemberInfoState)state).members;
  int size = members.length;
  for (int i=0;i<size;i++)
    addMemberInfo(members[i]);
}

public synchronized void assumeState()
{
  set.clear();
```

```
}

public void processPTPMessage(int parm1, Message parm2){}
public void changingView(){}
public void installView(View parm1){}
public void memberAccepted(int id, GroupHandler parm2, View parm3)
{
  memberId=id;
}
public void excludedFromGroup()
{
  freeResources();
}

public void processCastMessage(int sender, Message message)
{
  if (message instanceof SetMemberInfoChangedMessage) {
    synchronized(this) {
      SetMemberInfoChangedMessage msg = (SetMemberInfoChangedMessage) message;
      if (msg.add)
        addMemberInfo(msg.info);
      else
        removeMemberInfo(msg.info);
    }
  }
}

synchronized boolean addMemberInfo(MemberInfo memberInfo)
{
  boolean ret = set.add(memberInfo);
  if (ret)
    observersHandler.informPut(memberInfo);
  return ret;
}

synchronized boolean removeMemberInfo(MemberInfo memberInfo)
{
  Iterator it = set.iterator();
  while(it.hasNext()) {
    MemberInfo content = (MemberInfo) it.next();
    if (ObjectsHandling.areEquivalent(memberInfo.handler, content.handler)) {
      it.remove();
      observersHandler.informRemove(memberInfo);
      return true;
    }
  }
```

```
    return false;
  }

  Set set;
  int subgroupId, memberId;
  DomainGroupHandler groupHandler;
  GMNSFactory factory;
  SetMemberInfoObserversHandler observersHandler;
};
```

D.2. MapStringSetImpl.java

```
package sensei.GMNS;

import sensei.middleware.GMS.*;
import sensei.middleware.GMNS.*;
import sensei.middleware.domains.*;
import sensei.util.Error;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

class MapStringSetImpl extends MapStringSetBaseImpl
{
  private MapStringSetImpl(DomainGroupHandler groupHandler) throws Exception
  {
    this.groupHandler=groupHandler;
    map=new HashMap();
    factory = new GMNSFactory(groupHandler);
    observersHandler = new MapStringSetObserversHandler(thisMapStringSet);
  }

  public int getSubgroupId()
  {
    return subgroupId;
  }

  private void setSubgroupId(int subgroupId)
  {
    this.subgroupId=subgroupId;
    factory.setSubgroupId(subgroupId);
  }

  public MapStringSetImpl(int subgroupId, DomainGroupHandler groupHandler)
```

```
      throws Exception
{
  this(groupHandler);
  setSubgroupId(subgroupId);
}

public MapStringSetImpl(DynamicSubgroupInfo dynamicSubgroupInfo,
             DomainGroupHandler groupHandler)
  throws SubgroupsHandlerException, MemberStateException, Exception
{
  this(groupHandler);
  subgroupId = groupHandler.createSubgroup(dynamicSubgroupInfo, thisMapStringSet);
  setSubgroupId(subgroupId);
}

public void register()
      throws MemberStateException, SubgroupsHandlerException, Exception
{
  groupHandler.registerSubgroup(subgroupId, thisMapStringSet);
}

public static void register(DynamicSubgroupInfo dynamicSubgroupInfo,
             DomainGroupHandler groupHandler)
      throws MemberStateException, SubgroupsHandlerException, Exception
{
  groupHandler.castSubgroupCreation(dynamicSubgroupInfo);
}

void freeResources()
{
  if (map!=null) {
    try{deactivate();}catch(Exception ex){}
    map.clear();
    groupHandler=null;
    factory=null;
  }
}

public SetMemberInfo put(String group, SetMemberInfo set) throws
      InvalidSetMemberInfo, MemberStateException
{
  SetMemberInfo result=null;
  try  {
    int subgroupId = groupHandler.getSubgroupId(set);
    if (subgroupId==Consts.INVALID_SUBGROUP)
      throw new InvalidSetMemberInfo();
```

```
    factory.castMapPutMessage(group, subgroupId);
    result = doPut(group, set);
    factory.syncMessageCompleted();
  }
  catch(Exception ex){Error.unhandledException(Consts.AREA, ex);}
  return result;
}

public SetMemberInfo remove(String group) throws MemberStateException
{
  SetMemberInfo result=null;
  try {
    factory.castMapRemoveMessage(group);
    result = doRemove(group);
    factory.syncMessageCompleted();
  }
  catch(Exception ex){Error.unhandledException(Consts.AREA, ex);}
  return result;
}

public SetMemberInfo get(String group)
{
  SetMemberInfo ret = null;
  try {
    ret = (SetMemberInfo) map.get(group);
  }
  catch(Exception ex){Error.unhandledException(Consts.AREA, ex);}
  return ret;
}

public synchronized String[] getKeys()
{
  String ret[] = null;
  try {
    int size=map.size();
    ret=new String[size];
    for (Iterator it=map.keySet().iterator();it.hasNext();)
      ret[size]=(String) it.next();
  }
  catch(Exception ex){Error.unhandledException(Consts.AREA, ex);}
  return ret;
}

public void addObserver(MapStringSetObserver observer)
{
  observersHandler.addObserver(observer);
```

```
}

public boolean removeObserver(MapStringSetObserver observer)
{
  return observersHandler.removeObserver(observer);
}

public State getState()
{
  int size = map.size();
  MapStringSetEntry entries[] = new MapStringSetEntry[size];
  if (size>0) {
    int i=0;
    Iterator it = map.entrySet().iterator();
    while(it.hasNext()) {
      Map.Entry entry = (Map.Entry) it.next();
      try {
        int subgroup=groupHandler.getSubgroupId((SetMemberInfo)entry.getValue());
        entries[i++]=new MapStringSetEntry((String) entry.getKey(),subgroup);
      }
      catch(Exception ex) {Error.unhandledException(Consts.AREA, ex); }
    }
  }
  return factory.createMapState(entries);
}

public void setState(State state)
{
  map.clear();
  MapStringSetEntry entries [] = ((MapStringSetState)state).groups;
  int size = entries.length;
  for (int i=0;i<size;i++) {
    try {
      GroupMember subgroup = groupHandler.getSubgroup(entries[i].value);
      if (subgroup !=null) {
        SetMemberInfo set = GMNSnarrower.toSetMemberInfo(subgroup);
        doPut(entries[i].key, set);
      }
    }
    catch(Exception ex) {Error.unhandledException(Consts.AREA, ex); }
  }
}

public void assumeState()
{
  map.clear();
```

```
}

public void processPTPMessage(int parm1, Message parm2){}
public void changingView(){}
public void installView(View parm1){}
public void memberAccepted(int id, GroupHandler parm2, View parm3)
{
  memberId=id;
}
public void excludedFromGroup()
{
  freeResources();
}
public void processCastMessage(int sender, Message message)
{
  try {
    if (message instanceof MapStringSetPutMessage) {
      MapStringSetPutMessage msg = (MapStringSetPutMessage) message;
      GroupMember subgroup = groupHandler.getSubgroup(msg.value);
      if (subgroup!=null) {
        SetMemberInfo set = GMNSnarrower.toSetMemberInfo(subgroup);
        if (set!=null)
          doPut(msg.key, set);
      }
    }
    else if (message instanceof MapStringSetRemoveMessage) {
      doRemove(((MapStringSetRemoveMessage) message).key);
    }
  }
  catch(Exception ex) {Error.unhandledException(Consts.AREA, ex); }
}

SetMemberInfo doPut(String group, SetMemberInfo set)
{
  SetMemberInfo ret = null;
  Object obj = null;
  synchronized(this){obj=map.put(group, set);}
  if (obj!=null) {
    ret = GMNSnarrower.toSetMemberInfo(obj);
    observersHandler.informRemove(group, ret);
  }
  observersHandler.informPut(group, set);
  return ret;
}

SetMemberInfo doRemove(String group)
```

```
  {
    SetMemberInfo ret=null;
    Object obj = null;
    synchronized(this){obj=map.remove(group);}
    if (obj!=null) {
      ret=GMNSnarrower.toSetMemberInfo(obj);
      observersHandler.informRemove(group, ret);
    }
    return ret;
  }

  synchronized void doClear()
  {
    map.clear();
    observersHandler.informClear();
  }

  Map map;
  int subgroupId, memberId;
  DomainGroupHandler groupHandler;
  GMNSFactory factory;
  MapStringSetObserversHandler observersHandler;
};
```

GLOSARIO

BOA *Basic Object Adapter* (adaptador básico de objetos). Término relacionado con *CORBA*.

CORBA *Common Object Request Broker Architecture*, arquitectura distribuida del *OMG*.

DCOM *Distributed Component Object Model*, arquitectura distribuida de *Microsoft*

FIFO *First In, First Out* (primero que entra, primero que sale). Término empleado para referirse a un orden específico en mensajes.

GMNS *Group Membership Naming Service*, servicio de gestión de grupos basado en nombres.

GMP *Group Membership Protocol*, protocolo del servicio de gestión de grupos. Término relacionado con el modelo de sincronía virtual.

GMS *Group Membership Service*, servicio de gestión a grupos, también traducido en esta memoria por servicio de pertenencia a grupo. Término relacionado con el modelo de sincronía virtual.

HTML *HyperText Markup Languaje*, lenguaje hipertexto basado en etiquetas.

HTTP *Hipertext Transfer Protocol*, protocolo empleado en la transferencia de páginas HTML.

IDL *Interface Definition Language* (lenguaje de definición de interfaz).

IIOP *Internet Inter-ORB Protocol*, protocolo para la comunicación entre *ORB*s sobre Internet. Término relacionado con *CORBA*.

IOGR *Interoperable Object Group Reference*, referencia de grupos de objetos entre ORBs. Término relacionado con *CORBA*.

IP *Internet Protocol*, protocolo de red.

JMS *Java Messaging Service* (servicio de mensajería Java).

JSP *Java Server Pages*, páginas de servidor Java.

OMG *Object Management Group*.

ORB *Object Request Broker*, gestor de objetos remotos.

OSI *Open Systems Interconnections*, modelo abierto de interconexión de sistemas. Término relacionado con sistemas distribuidos.

POA *Portable Object Adapter*, adaptador de objetos *portable*, en cuanto a su independencia del *ORB*. Término relacionado con *CORBA*.

RMI *Remot Method Invocation* (invocación remota de métodos). Es el gestor de objetos distribuidos en Java.

RPC *Remot Procedure Call* (llamadas a procedimientos remotos).

SOAP *Simple Object Access Protocol*. Protocolo simple de acceso a objetos.

STL *Standar Template Library* (librería estándar de plantillas). Término relacionado con C++.

TCP *Transfer Control Protocol* (protocolo de control de transferencia).

UDP *User Datagram Protocol*, protocolo de datagrama de usuario.

XML *eXtended Markup Language*, lenguaje extendido de marcado (basado en etiquetas).

BIBLIOGRAFÍA

[Allaramaju01]
Professional Java Server Programming J2EE, 1.3 Edition
S. Allaramaju, C. Buest, M. Wilcox, S. Tyagi, R. Johnson, G. Watson., A. Williamson, J. Davies, R. Naggapan, A. Longshaw, P. Sarang, T. Jewell, A. Toussaint
Wrox Press Ltd., September 2001

[Amir95]
The Totem Single-Ring Ordering and Membership Protocol
Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, P. Ciarfella
ACM Transactions on Computer Systems, Vol. 13, No. 4, November 1995. pag 311-342

[Amir98]
The Spread Wise Area Group Communication System
Y. Amir, J. Stanton
Technical Report CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University

[Amoeba]
http://www.cs.vu.nl/pub/amoeba/

[Arjuna]
http://arjuna.ncl.ac.uk/

[Babaoglu95]
Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specifications
O. Baboglu, R. Davoli, A. Montresor
University of Bologna, Technical Report UBLCS-95-18, November 1995

[Babaoglu96]
Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems
O. Babaoglu, A. Bartoli, G. Dini
Technical Report. Department of Computer Science, University Of Bologna, May 1996

[Ban98]
JavaGroups - Group Communication Patterns in Java
B. Ban
Cornell University, July 1998

[Bast]
http://lsewww.epfl.ch/bast/

[Berson96]
Client/Server Architecture
A. Berson
McGraw Hill, 2nd Edition, March 1996

[Birman85]
Replication and Fault-Tolerance in the ISIS System
K. Birman
10th ACM Symposium on Operating Systems Principles, 79-86. Operating Systems Review, 19, 5. December 1985

[Birman87]
Reliable Communication in the Presence of Failures
K. Birman, T. Joseph
ACM Transactions on Computer Systems, February 1987, Vol 5, No 1, pag 47-76

[Birman96]
Building Secure and Reliable Network Applications
K. Birman
Manning Publications, 1996

[Birman98]
Bimodal multicast
K. Birman, M. Hayden, O. Ozkasap, M. Budiu, Y. Minski
Technical Report TR98-1665, Cornell University

[Birman99]
Bimodal Multicast
K. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, Y. Minsky.
ACM Transactions on Computer Systems. (May 1999). Volume 17, No. 2

[Birrel84]
Implementing Remote Procedure Calls.
A.D. Birrell, B.J. Nelson
ACM Transactions on Computer Systems, Vol 2, No 1, pag 39-59, February 1984

[Budhiraja93]
Distributed Systems
N. Budhiraja, K. Marzullo, F. Schneider, S. Toueg
Addison-Wesley, 2nd Edition, 1993. Chapter 8: The Primary-Backup Approach, pag 169-197

[Cactus]
http://www.cs.arizona.edu/cactus/index.html

[Chang84]
Reliable broadcast protocols
J.M. Chang, N.F. Maxemchuk
ACM Trans Comput. Systems, 2, 3 (August 1984), 251-273

[Chou97]
Beyond Fault Tolerance
T. C.K. Chou
IEEE Computer, April 1997

[Christian91]
Understanding Fault-Tolerant Distributed System
F. Christian
Communications of the ACM, February 1991, Vol 34, No 2, pag 56-78

[Chung98]
DCOM and CORBA side by side, Step by Step, and Layer by Layer
P.E. Chung, Y. Huang, S. Yajnik, D. Liang, J.C. Shih, Ch. Wang, Y. Wang
http://www.cs.wustl.edu/~schmidt/submit/Paper.html, 1998

[Curtis97]
Java, RMI and CORBA
D. Curtis
http://www.omg.org, 1997

[Dolev96]
The Transis Approach to High Availability Cluster Communication
D. Dolev, D. Malki
Communications of ACM, 39,4, April 1996

[Dwoning98]
Java RMI
T.B. Dwoning
IDG Books Worldwide, February 1998

[Dwork88]
Consensus in the presence of partial synchrony
C. Dwork, N. A. Lynch, L. Stockmeyer
Journal of the ACM, 35(2):288-323, April 1988

[Electra]
http://www.softwired.ch/people/maffeis/electra.html

[Ensemble]
http://www.cs.cornell.edu/Info/Projects/Ensemble/index.html

[Fidge96]
Fundamentals of Distributed System Observation
C. J. Fidge
IEEE Software November 1996

[Freeman99]
JavaSpaces Principles, Patterns and Practice
E. Freeman, S. Hupfer, K. Arnold
Addison-Wesley Pub Co, June 1999

[Friedman95]
Strong and Weak Virtual Synchrony in Horus
R. Friedman, R. Renesse
Cornell University Technical Report TR95-1537, August 1995

[Gamma95]
Design Patterns
E. Gamma, R. Helm, R. Johnson, J. Vlissides
Addison Wesley Longman, 1995

[Garbinato97]
Using the Strategy Design Pattern to Compose Reliable Distributed Protocols
B. Garbinato, R. Guerraoui
Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS'97), Portland (Oregon), June 1997

[Gelernter85]
Generative Communications in Linda
D. Gelernter
ACM Transactions on Programming Languages and Systems, Jauary 1985, Vol. 7, N. 1, pp 80-112

[Gopalan98]
A Detailed Comparison of CORBA, DCOM and Java/RMI
R. Gopalan
http://www.execpc.com/~gopalan/misc/compare.html, 1998

[Gray93]
Transaction Processing: Concepts and Techniques
J.Gray, A. Reuter
Morgan Kaufmann Publishers, San Matero, CA, 1993

[Guerraoui95]
Transaction model vs Virtual Synchrony model: bridging the gap
R. Guerraoui, A. Schiper
Proc. International Workshop "Theory and Practice in Distributed Systems", Springer Verlag, LNCS938, 1995

[Hayden98]
The Ensemble System Cornell University
M. Hayden
Technical Report, TR98-1662, January 1998

[Horus]
http://www.cs.cornell.edu/Info/Projects/Horus/index.html

[Ibus]
http://www.softwired-inc.com/products/ibus/

[Isis]
http://www.cs.cornell.edu/Info/Projects/Isis/index.html

[JavaGroups]
http://www.cs.cornell.edu/Info/People/bba/javagroups.html

[JGroup]
http://www.cs.unibo.it/projects/jgroup/

[Jiménez99]
TransLib: An Ada95 Object Oriented Framework for Building Transactional Applications
R. Jiménez, M Patiño, S. Arévalo, F. Ballesteros
Special Issue on Developing Fault-Tolerant Systems with Ada95, Intl. Journal on Computer Systems: Science and Engineering, 1999

[Juric00]
Java 2 RMI and IDL comparison
M.B. Juric, I. Rozman
Java Report, February 2000

[Kaashoek91]
Group communication in the Amoeba distributed operating system
M.F. Kaashoek, A.S. Tanenbaum
In Proceedings of the IEEE 11th International Conference on Distributed Computing Systems (Arlington, Texas, 1991). IEEE, New York, 882-891

[Lamport82]
Fail-stop processors: An approach to designing fault-tolerant computing systems
L. Lamport, R.E. Shostak, M.C. Pease
ACM Transactions on Programming Languages and Systems (TOPLAS), 1982, Vol 4, No 3: pag 382-401

[Leffler89]
The Design and Implementation of the 4.3BSD UNIX Operating System
S.J. Leffler, M. McKusick, M. Karels, J. Quaterman
Addison-Welsey, 1989

[Little00]
Integrating Group Communication with Transactions for Implementing Persistent Replicated Objetcs
M. Little, S. Shrivastava
LNCS 1752, 2000, pag 238-253

[Maestro]
http://simon.cs.cornell.edu/Info/Projects/Ensemble/Maestro/Maestro.html

[Maffeis97]
Constructing Reliable Distributed Communication Systems with CORBA
S. Maffeis D. Schmidt
IEEE Communications Magazine 14(2) , February 1997

[Maguire93]
Writing Solid Code
S. Maguire
Microsoft Press, May 1993

[Malki94]
Uniform Actions in Asynchronous Distributed Systems
D. Malki, K. Birman, A. Ricciardi, A. Schiper
Proc. Of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC), Los Angeles, August 14-17, 1994

[Malloth96]
Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks
C. Malloth
PhD Thesis No. 1557, Swiss Federal Institute of Technology of Lausanne (Switzerland) (Ecole Polytechnique Fédérale de Lausanne). September 1996

[Marzullo96]
NILE: Wide-area computing for high energy physics
K. Marzullo, M. Ogg, A. Ricciardi, A. Amoroso, F.A. Calkins, E. Rothfus
In Proc. 7th ACM SIGOPS European Workshop, pag 49-54, Connemara, Ireland, 2-4 September 1996. ACM

[Montresor98]
The Jgroup Reliable Distributed Object Model
A. Montresor
Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, Helisinki, Finland, June 1999

[Moser95]
The Totem System
L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, T. P. Archambault
Proceedings of the 25th International Symposium on Fault Tolerant Computing, Pasedena, CA (June 1995), 61-66

[Moss85]
Nested Transactions: An Approach to Reliable Distributed Computing
J. Moss
MIT Press, Cambridge, MA, 1985

[Neumann56]
Probabilistic logics and the synthesis of reliable organisms from unreliable components
J. von Neumann
Automata Studies, Princeton University Press, 1956, pag 43-98

[Nile]
http://www.nile.cornell.edu/

[OMG00]
Fault Tolerant CORBA Specification, v1.0
OMG Document ptc/2000-04-04. *http://www.omg.org/cgi-bin/doc?ptc/2000-04-04*

[OMG97]
CORBAservices: Common Object Services Specification
OMG, 1997

[OMG98]
The Common Object Request Broker: Architecture and Specification
OMG, February 1998

[OMG99]
CORBA/IIOP 2.3.1 Specification, formal/99-10-07
OMG, October 1999

[Oram01]
Peer-to-Peer. Harnessing the Power of Disruptive Technologies
Oram A. (editor)
O'Reilly, March 2001

[Orfali97]
Instant CORBA
R. Orfali, D. Harkey, J. Edwards
Wiley Computer Publishing, 1997

[Parrington95]
The Design and Implementation of Arjuna
G.D. Parrington, S.K. Shrivastava, S.M. Wheater, M.C. Little
USENIX Computing Systems Journal, Vol 8, No 3, 1995

[Patiño00]
Scalable Replication in Database Clusters
M. Patiño, R. Jimenez, B. Kemme, G. Alonso
Proc. Of the Int. Conf. On Distributed Computing DISC'00, (LNCS 1914) , Toledo, October 2000

[Pavón00]
Conditions for the State Transfer on Virtual Synchronous Systems
J. Pavón, LM. Peña
Proceedings of The 10th International Conference on Computing and Information ICI' 2000, Kuwait, Noviembre 18-21, 2000 (LNCS series)

[Peña00]
Replicación de Objetos Distribuidos
LM. Peña , J. Pavón
1er Taller de Trabajo en Ingeniería del Software basada en Componentes Distribuidos, IScDIS'2000, Valladolid, 9 Noviembre 2000

[Peña97]
Fiabilidad y Calidad de Servicio en la Arquitectura CORBA
LM. Peña
Actas de las V Jornadas de Concurrencia, Vigo, Junio 1997, pp. 381-388

[Peña99]
Sensei: Transferencia de Estado en Grupos de Objetos Distribuidos
LM. Peña, J. Pavón
*Computación y Sistemas Vol. 2, No. 4, pp.191-201, Abril-Junio 1999 (*versión extendida de la ponencia presentada en el *Simposio Español de Informática Distribuida, SEID'99, Santiago de Compostela, 24-26 Febrero 1999)*

[Peterson89]
Preserving and using context information in interprocess comunication
L. L. Peterson, N.C. Buchholz, R.D. Schlichting
ACM Trans Comput. System 7, 3 (August 1989), 217-246

[Philippsen97]
JavaParty – Transparent Remote Objects in Java
M. Phillipsen, M. Zenger
Concurrency: Practice and Experience, 9(11):1125-1242, 1997

[Phoenix]
http://lsewww.epfl.ch/projets/phoenix/index.html

[Rajagopalan89]
A token-based protocol for reliable, ordered multicast communication
B. Rajagopalan, P.K. McKinley
In Proceedings of the IEEE 8th Symposium on Reliable Distributed Systems (Seattle, Wash., 1989). IEEE, New York, 84-93

[Relacs]
http://www.cs.unibo.it/projects/relacs.html

[Renesse96]
Horus, a flexible Group Communication System
R. van Renesse, K. Birman, S. Maffeis
Communications of the ACM, April 1996

[Ricciardi93]
Understanding Partitions and the "No Partition" Assumption
A. Ricciardi, A. Schiper, K. Birman
IEEE Proc Fourth Workshop on Future Trends of Distributed Systems, Lisbon, September 22-24, 1993

[RMP]
http://research.ivv.nasa.gov/RMP/

[Rodrigues92]
xAMp: a Multi-primitive Group Communications Service
L. Rodrigues, P. Veríssimo
Proceedings of the 11th Symposium On Reliable Distributed Systems, October 1992, Houston, Texas

[Sabel94]
Simulating Fail-Stop in Asynchronous Distributed Systems
L. Sabel, K. Marzullo
Proceedings of the 13th Symposium on Reliable Distributed Systems, October 1994

[Schiper96]
From Group Communication to Transactions in Distributed Systems
A. Schiper, M. Raynal
Communications of the ACM, Vol. 39 No. 4, April 1996. pag 84-87

[Schlichting93]
A Communication Substrate for Fault-tolerant Distributed Programs
S. Mishra, L. Peterson, R. Schlichting
Distributed System Engineering, vol. 1, pp. 87-103, December 1993

[Schmidt95a]
Object interconnections. Introduction to Distributed Object Computing
D. Schmidt, S. Vinoski
SIGS C++ Report Magazine, Vol 7, No 1, January 1995

[Schmidt95b]
Object interconnections. Comparing Alternative Server Distributed Programming Techniques.
D. Schmidt, S. Vinoski
SIGS C++ Report Magazine, Vol 7, No 8, October 1995

[Schmidt97]
Object interconnections. OMG Event Object Service
D. Schmidt, S. Vinoski
SIGS C++ Report Magazine, Vol 9, No 2, February 1997

[Schneider90]
Implementing fault-tolerant services using the state machine approach: A tutorial
F. Schneider
ACM Computing Surveys, 22(4):299-319, December 1990

[Schneider93]
Distributed Systems
F. Schneider

Addison-Wesley, 2nd Edition, 1993. Chapter 7: Réplication Management using the State-Machine Approach, pag 169-197

[Sensei]
http://grasia.fdi.ucm.es/sensei

[Sessions97]
COM and DCOM: Microsoft's vision for Distributed Objects
R. Sessions
John Wiley & Sons, 1997

[Spinglass]
http://www.cs.cornell.edu/Info/Projects/Spinglass/index.html

[Spread]
http://www.spread.org/

[SSL]
http://home.netscape.com/eng/security

[Stevens90]
UNIX Network Programming
W.R Stevens
Englewood Cliffs, NJ: Prentice Hall, 1990

[SUN99]
Java Messaging Service
SUN, version 1.0.2, November 1999

[Tanembaum91]
The Amoeba Distributed Operating System-A Status Report
A, Tanenbaum, M. Kaashoek, R. van Renesse, H. Bal
Computer Communications, vol. 14, pp. 324-335, July/August 1991

[Totem]
http://beta.ece.ucsb.edu/totem.html

[TLS]
http://www.ietf.org/rfc/rfc2246.txt

[Transis]
http://www.cs.huji.ac.il/labs/transis/index.html

[Vaysburd98]
Building Reliable Interoperable Distributed Objects with the Maestro Tools
Cornell University Technical Report, TR98-1678, May 1998

[Vogels98]
The Design and Architecture of the Microsoft Cluster Service
W. Vogels, D. Dumitriu, K. Birman, R. Garnache, M. Massa, R. Short, J. Vert, J. Barrera, J. Gray
Proceedings of FTCS'98, June 1998, Munich, Germany

[W3C00]
Extensible Markup Languaje (XML) 1.0 (Second Edition)
http://www.w3.org/TR/REC-xml

[W3C01]
SOAP Version 1.2 Part 0: Primer
http://www.w3.org/TR/soap12-part0/

[White75]
A High-Level Framework for Network-Based Resource Sharing
J.E. White
RFC 707, December 1975

[xAmp]
http://www.navigators.di.fc.ul.pt/xAMp/xAMp.html