Sensei: Transferencia de estado en grupos de objetos distribuidos

L. M. Peña Cabañas, J. Pavón Mestras

Departamento de Sistemas Informáticos y Programación

Universidad Complutense de Madrid

e-mail: luismp@skynet.be, jpavon@sip.ucm.es

Resumen

La construcción de sistemas distribuidos exige cada vez más el poder asegurar fiabilidad, disponibilidad y continuidad de servicio. Una abstracción para lograr este objetivo es la de grupo de objetos que responden conjuntamente a las peticiones de servicio. Para mantener la consistencia entre estos objetos es necesario disponer de mecanismos de transferencia de estado entre los miembros del grupo que soporten la adición de nuevos elementos al grupo y evitar las inconsistencias que pudieran surgir al dividirse el grupo en varias particiones. El sistema de transferencia de estado que aquí se presenta, *Sensei*, está implementado sobre las primitivas de comunicación de *Ensemble*, y permite utilizar distintos algoritmos y protocolos de transferencia de estado, aplicables en sistemas de computación de objetos distribuidos, por ejemplo sobre CORBA.

Palabras clave

Transferencia de Estado, Tolerancia a fallos, Sistemas Distribuidos, Sincronía Virtual, CORBA.

1 Introducción

El auge de la utilización de Internet y de redes corporativas está incrementando notablemente el desarrollo de aplicaciones distribuidas según el modelo cliente-servidor, utilizando entornos de computación de objetos distribuidos como CORBA (OMG, 1998), DCOM (Sessions, 1997) o Java RMI (JavaSoft, 1997). Una limitación actual de este tipo de sistemas es la falta de soporte estándar para construir aplicaciones fiables, tolerantes a fallos. Y este aspecto es fundamental para poder implantar estas tecnologías en dominios de aplicación tan importantes como las transacciones económicas o los servicios de telecomunicación.

Entre las medidas usualmente adoptadas para soportar tolerancia a fallos están la redundancia de los recursos del sistema, la redundancia de los enlaces de comunicaciones entre estos recursos, la detección y el enmascarado de los fallos y la recuperación ulterior a los mismos.

El empleo de redundancia añade nuevos aspectos en el diseño de una aplicación para tratar el *grupo* de elementos replicados (los *miembros* del grupo): necesidad de comunicaciones y sincronización entre los miembros del grupo, tratamiento de errores en los componentes y en los enlaces que los comunican, permitir incluir o excluir miembros en un grupo de objetos replicados, etc. Mediante el modelo de *sincronía virtual* (Schipper y Sandoz, 1993) se genera una abstracción sobre el grupo que permite programar sus miembros en la asunción de que todos reciben los mismos mensajes en el mismo orden (incluso cuando este orden pueda ser flexibilizado) y donde los miembros sólo fallan por caída, y no hubiera problemas de red. Un elemento clave en este modelo es la utilización de un servicio de gestión de los miembros del grupo (*Group Membership Service*, GMS) y un protocolo (*Group Membership Protocol*, GMP) que permita gestionar la consistencia de ese servicio (Birman, 1996). El GMS envía información al grupo según éste va cambiando dinámicamente, de tal forma que los miembros reciben *vistas* (los cambios en el grupo), con la propiedad de *sincronía de vistas* (Babaoglu *et al.*, Sep. 1995): los miembros de un grupo que se mantienen en una vista reciben todos los mismos mensajes. El GMS permite crear grupos donde los miembros se incluyen dinámicamente, y pueden a su vez abandonarlo por decisión propia o por caída del miembro, que el mismo GMS vigila mediante una detección *no* fiable de fallos (Chandra y Toung, 1996). Partiendo de la imposibilidad de disponer de un detector fiables de fallos y de la posibilidad de errores en la red, el GMS se enfrenta a particiones reales y virtuales de la red, que suponen inconsistencias en el mismo GMS y en los grupos que maneja.

Nuestro objetivo es la definición de un servicio CORBA de fiabilidad[1] bajo el modelo de sincronía virtual, creado a

[1] *Aunque la arquitectura CORBA es el objetivo principal, contemplamos también su definición sobre RMI y en DCOM.*

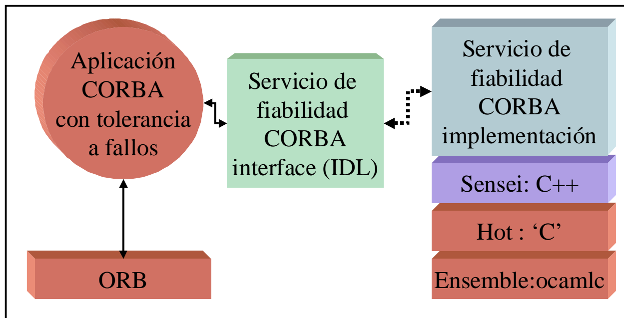

Figura 1: *Servicio CORBA de fiabilidad, implementado sobre Ensemble*

partir de un conjunto de primitivas de comunicaciones multipunto fiables con varios grados de ordenación posibles, y un GMS respetando la sincronía de vistas.

El trabajo de Felber (1998) se plantea este mismo objetivo, pero la arquitectura que propone solo soporta funcionalidad básica. Nuestro trabajo (figura 1) trata de ofrecer un marco flexible para la implantación de distintos mecanismos de gestión de la replicación de objetos. La implementación actual se basa en *Ensemble* (Hayden, 1998), un sistema ya desarrollado (aún en evolución) con una funcionalidad muy extensa, lo que nos facilitará una rápida especificación del servicio de fiabilidad. Ensemble quedará oculto a la aplicación CORBA, que usará directamente la interfaz del servicio definido, cuya implementación podrá realizarse posteriormente sin el soporte de Ensemble. A diferencia de *Electra* (Maffeis, 1995), un ORB fiable desarrollado sobre *Horus* (Renesse *et al.*, 1996), un sistema predecesor de Ensemble, nuestro planteamiento es la implementación de la fiabilidad como un servicio CORBA, en vez de extender la funcionalidad del ORB para soportar esa fiabilidad.

La estructura de este artículo incluye una breve descripción de *Sensei*, el conjunto de clases C++ base de nuestro proyecto. A continuación se define el problema de la transferencia de estado y definimos los protocolos que solucionan la transferencia de estado en grupos primarios, no particionables. El caso general de grupos particionables es brevemente delimitado, al estar fuera del alcance de este articulo. Para el caso de sistemas primarios, definimos el interfaz (la parte de transferencia de estado) del servicio CORBA de fiabilidad y el interfaz que debe implementar una aplicacion CORBA que use tal servicio.

2 Sensei

Ensemble está desarrollado en el lenguaje *Ocaml*, lenguaje para el que no se ha definido una correspondencia con OMG/IDL. Hay, sin embargo, una interfaz C denominada *Hot* para acceder a la funcionalidad básica, sobre la cual se ha creado el conjunto de clases C++ *Maestro* (Vaysburd, 1998). En particular, Maestro permite realizar una transferencia de estado entre objetos independiente de la definida en Ensemble, pero los algoritmos que la soportan en su versión actual son erróneos[2]. Por esta razón, hemos creado un nuevo conjunto de clases, con una funcionalidad mayor a la soportada en Maestro, que hemos denominado *Sensei*. Su implementación incluye los protocolos de transferencia de estado que se discuten a continuación y que pueden encontrarse en:

http://bogart.sip.ucm.es/research/sensei.

Sensei define una abstracción de miembros de un grupo como objetos C++, integrando Ensemble a programas C++ y de amplio soporte de transferencia de estado. Es el conjunto de clases sobre el que implementaremos inicialmente nuestro servicio CORBA de fiabilidad.

3 Transferencia de estado

Bajo el modelo de sincronía virtual, cada miembro se programa con la seguridad de que a partir de un estado común, todos reciben los mismos eventos en el mismo orden, evolucionando al mismo estado final. Por tanto, el primer problema a resolver es cómo alcanzar un estado común inicial de todos los miembros: cómo inicializar los miembros que se unen al grupo y cómo consensuar miembros de un grupo que se ha dividido y cuyas particiones han evolucionado paralelamente. Los grupos con fuertes requisitos de consistencia no admiten la partición del sistema (los miembros de las particiones consideradas no primarias pierden su funcionalidad) y sólo presentan el problema de inicialización de miembros; a estos grupos se les denomina de *partición primaria* (Ricciardi *et al.*, 1993).

La transferencia de estado tiene varios problemas que resolver: consensuar cuál es el estado a transferir, quién debe, por tanto, recibirlo (el que no comparta tal estado), 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.

La familia de protocolos que implementa *Sensei* ofrece una gran flexibilidad en las transferencias de estado, flexibilidad que se extiende a la interfaz del servicio CORBA (ver el listado en IDL al final del artículo). Estos protocolos se han desarrollado para cumplir una serie de objetivos:

[2] *Maestro (y Hot) puede trabajar sobre Horus y Ensemble; las premisas sobre las que basa su transferencia de estado son válidas en Horus, para el que se desarrolló inicialmente, pero no en Ensemble*

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

- Facilitar el envío del estado en varias fases, para optimizar 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 funcionalidad* 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 funcionalidad, 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 funcionalidad como servidor. Un grado intermedio se obtiene cuando la transferencia de estado se considera *parcialmente segura*, y se permite el envío de los mensajes considerados seguros (la aplicación debe definir entonces 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 (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. Ejemplos 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 envían al grupo incluyen únicamente una lista ordenada de los miembros del grupo, no información adicional sobre el estado).

El concepto de partición primaria está asociado a la noción de grupos particionados. La aplicación debe poder identificar qué partición o particiones se consideran primaria (si alguna puede considerarse tal), de tal forma que si limita la funcionalidad según el tipo de partición, pueda aglutinar los recursos principales en la partición funcional. La funcionalidad de los miembros puede restringirse también de forma ajena a la definición de particiones primarias, permitiendo una funcionalidad determinada sólo cuando hay quórum en el grupo total, tal como es la aproximación estática. La identificación de particiones primarias está, sin embargo, asociada directamente al GMS, y los protocolos de transferencia de estado recibirán esa información en las vistas.

4 Transferencia de estado en grupos de partición primaria

En estos grupos, existen dos formas posibles de realizar la transferencia de estado; una primera donde los miembros con estado coordinan la transferencia hacia el miembro o

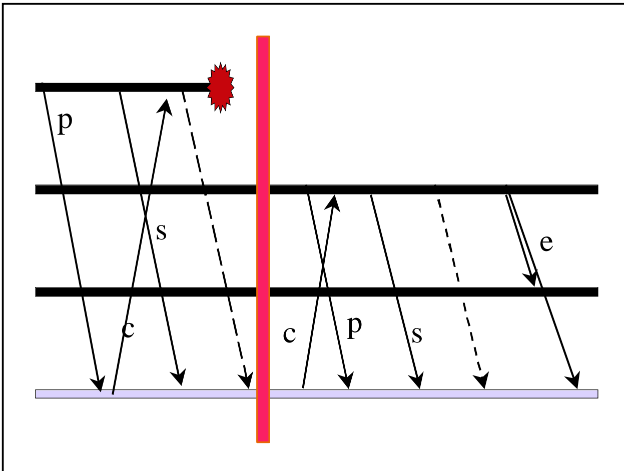

Figura 2: *Protocolo de transferencia push.*

(p): transferir información del protocolo
(s): transferir estado
(c): coordinar estado
(e): vista de transferencia

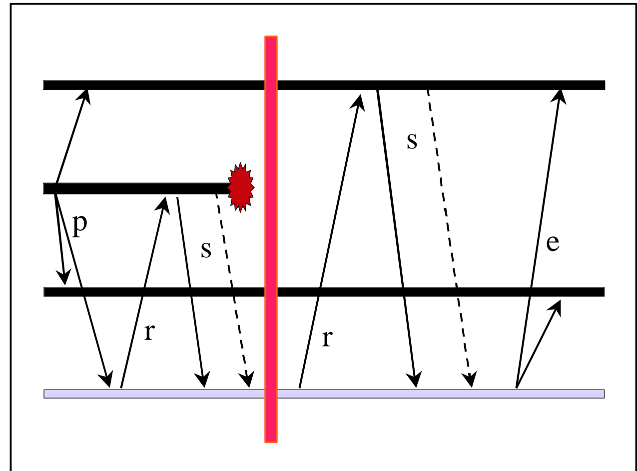

Figura 3: *Protocolo de transferencia pull.*

(p): transferir información del protocolo
(s): transferir estado
(r): solicitar estado
(e): vista de transferencia

miembros sin estado (transferencia *push*) y una segunda donde los miembros sin estado solicitan la transferencia de estado (transferencia *pull*).

4.1 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 2:

1. *Selección de un coordinador* para cada miembro sin estado, siguiendo las pautas definidas por la aplicación (usando pesos, balanza de carga, etc.).

2. El coordinador envía al nuevo miembro la lista de miembros con estado y los miembros que coordinan (*mensaje p*). Si se han definido *propiedades*, este mensaje incluye las de los miembros con estado.

3. Si el nuevo miembro ha definido alguna propiedad, las envía al coordinador en un mensaje *c*. Si la selección del coordinador precisa de las propiedades del nuevo miembro (como su localización física), las propiedades del miembro deben enviarse como multicast a todos los miembros del grupo en el inicio del protocolo, y este solo se iniciará una vez que las propiedades se hayan recibido.

4. El coordinador transfiere el estado en uno o varios *mensajes s*.

5. Al finalizar la transferencia, el coordinador realiza un *multicast* de un *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. En particular, si se cae durante una transferencia, el miembro al que coordinaba puede seleccionar un nuevo coordinador (si ya recibió del anterior el mensaje *p*) y enviarle automáticamente un mensaje *c* de coordinación de la transferencia, de tal forma que esta no tenga que reiniciarse.

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 más adelante, al definir la interfaz de la aplicación.

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.

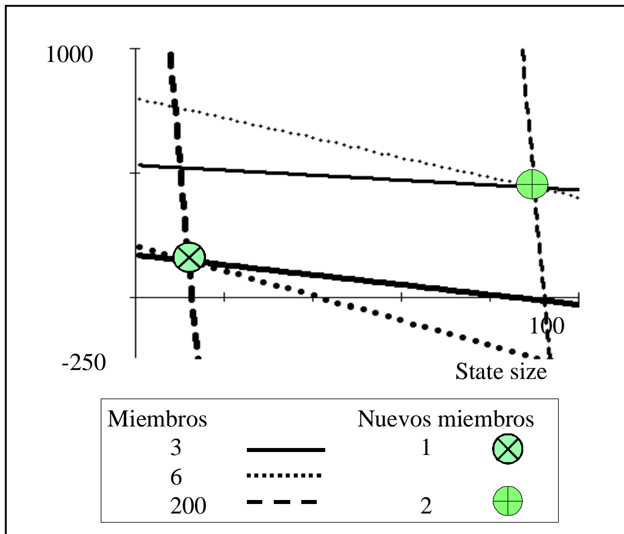

Figura 4: *Comparativa entre protocolos push y push-one step, sin soporte de propiedades*

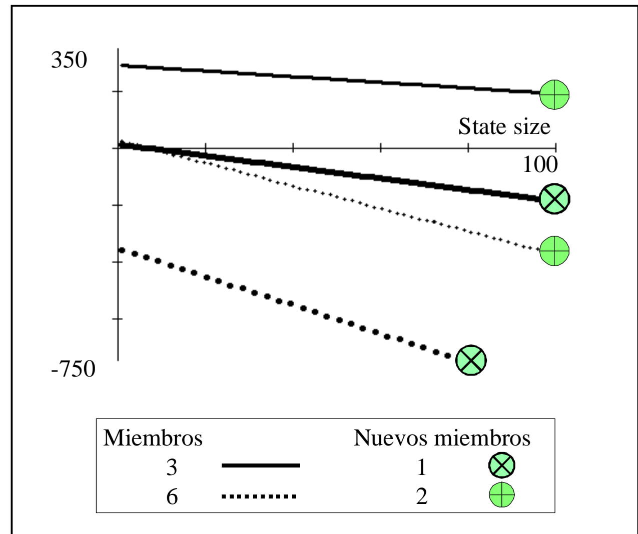

Figura 5: *Comparativa entre protocolos push y push-one step, con soporte de propiedades*

4.2 Transferencia *pull*

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

1. Un miembro con estado es seleccionado determinísticamente para enviar en un *mensaje multicast 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.

2. Cada miembro sin estado selecciona un coordinador (no soportándose balanza de carga), al que le envía un *mensaje r*.

3. El coordinador envía entonces el estado en *mensajes s*.

4. Al concluir la transferencia, el miembro sin estado envía un *mensaje multicast e*, indicando así su promoción a miembro con estado. En este mensaje, incluye sus propiedades. Al igual que en el protocolo *push*, si estas propiedades son necesarias para la selección del coordinador, deberán enviarse en un mensaje *multicast* inicial al comienzo del protocolo.

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.

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*.

4.3 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 *multicast*, que incluye asimismo las propiedades de los miembros con estado. Cada miembro sin estado debe realizar a su vez un *multicast* de sus propiedades.

A pesar del nombre, el protocolo *push* aun puede usarse para transferencias en un sólo paso, y como se ve en la siguiente comparativa, ésta es la solución a emplear en la mayoría de los casos.

4.4 Comparativa entre protocolos *push* y *push-one step*

Las figuras 3 y 4 comparan los protocolos *push* y *push-one step* para pequeños estados, asumiendo que el coordinador elegido no se cae durante la 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

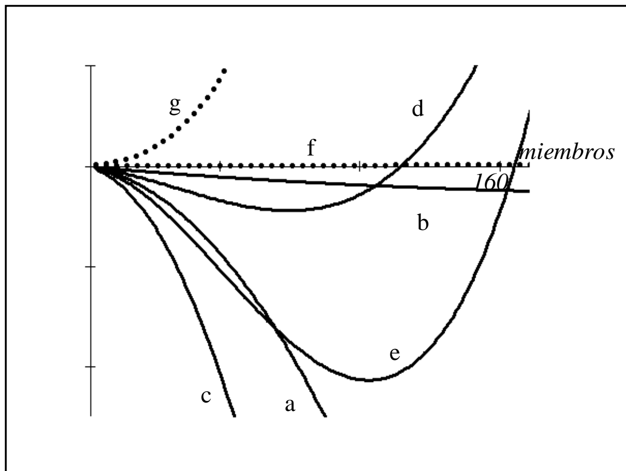

Figura 6: *Comparativa entre protocolos push y pull*

(a):un nuevo miembro
(b):la tercera parte de los miembros son nuevos
(c):un nuevo miembro, usando propiedades
(d):muchos miembros nuevos (la sexta parte), con frecuentes caídas de coordinadores
(e):como (d),usando propiedades
(f):como (a),con soporte multicast
(g):como (e),con soporte multicast

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 (dos contadores enteros). Se observa que el protocolo en un solo paso se comporta mejor para estados pequeños, degradándose rápidamente según aumenta el numero 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 *multicast*, 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 *multicast* por cada nuevo miembro, lo que se traduce en un peor rendimiento respecto al protocolo normal.
- Los gráficos se han realizado suponiendo que no hay soporte *multicast*; con éste, el protocolo en un solo paso resultará siempre preferible al protocolo *push*.

Por lo tanto, sin soporte *multicast*, 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 el protocolo *push* también permite transferencias en un solo paso (se envía un único mensaje *s*).

4.5 Comparativa entre protocolos *push* y *pull*

La figura 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 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 *multicast*; por esta misma razón, un aumento en el tamaño de los mensajes (como cabeceras mayores) implica un peor rendimiento del protocolo *pull*.
- La inclusión simultanea 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 *multicasts* necesarios bajo protocolo *pull*, lo que se traduce en un peor rendimiento de este protocolo según el tamaño de las propiedades aumenta.
- Con soporte multicast, es favorable el empleo del protocolo *pull*.

Resumiendo, sin soporte *multicast*, 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 *multicast*, el protocolo *pull* tiene mejor rendimiento.

4.6 GMS con soporte de transferencia de estado

El GMS puede incluir soporte de transferencia de estado, incluyendo en cada vista una indicación de cuales son los miembros con estado. En este caso, cuando un nuevo miembro recibe el estado, debe comunicárselo al GMS para que éste pueda instalar una nueva vista. Este soporte podría además extenderse para incluir asimismo las propiedades de los miembros.

Este soporte no supone ninguna mejora para el protocolo *push* (figura 6), pero sí para el protocolo *pull* (figura 7) que, al no precisar del primer mensaje *multicast p*, resulta claramente superior al primer protocolo. Una comparación entre el protocolo *push* sin soporte del GMS, y el protocolo *pull* con soporte clarificaría cuando la inclusión de tal funcionalidad en el GMS resulta útil. No obstante, tal

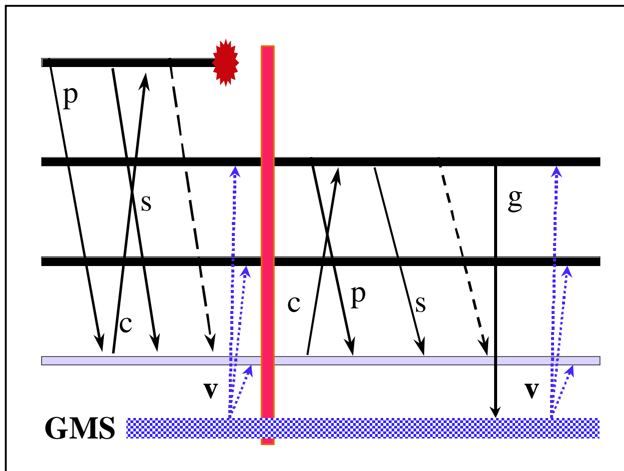

Figura 7: Protocolo push con soporte de transferencia en el GMS

(p): transferir información del protocolo
(s): transferir estado
(c): coordinar estado
(g): notificación al GMS del nuevo estado
(v): nueva vista

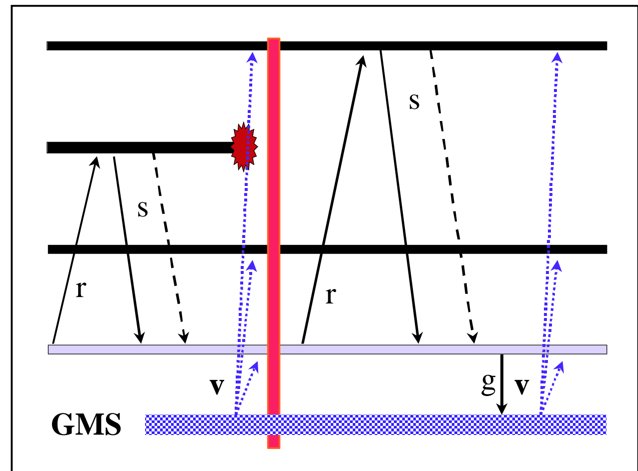

Figura 8: *Protocolo pull con soporte de transferencia en el GMS*

(r):solicitar estado
(s): mensaje de estado
(g): notificación al GMS del nuevo estado
(v): nueva vista

comparativa dependerá de la implementación particular de las vistas en el GMS. Así, si cada vista implica un mensaje *multicast* 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 como JavaGroups (Ban, 1998) -ambos sobre Ensemble- utilizan transferencia *pull*; el primero realiza la asunción errónea[3] de que la vista incluye los miembros más antiguos con los rangos más bajos, por lo que el miembro sin estado siempre se dirige al miembro con menor rango. En JavaGroups[4], 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.

5 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 (distintos miembros pueden presentar estados inconsistentes), y el GMS admitirá dos niveles de soporte:

1. Asociar a cada miembro un identificador único de estado, que es enviado en las vistas.

2. Implementar la interfaz definida por Babouglu *et al.* (1995) 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 implementa un protocolo análogo al definido por Amir *et al.* (1997) para colectar la información requerida en el caso de que el GMS no preste ningún soporte; a partir de esta información, es posible definir la interfaz de *sincronía virtual extendida*, pero, dado que el objetivo perseguido es un protocolo de transferencia de estado automático, no se hace necesario exportar la interfaz hacia la aplicación.

6 Interfaz con la aplicación en CORBA

Los protocolos definidos hasta el momento muestran el modo en que la información de estado es transferida entre los miembros. Estos protocolos se encuentran a un nivel inferior de la aplicación, que sólo precisa conocer la forma en que la transferencia se realiza; de hecho, el protocolo de

[3] *La última versión de Maestro corrige este comportamiento.*
[4] *JavaGroups permite distintas implementaciones de la transferencia de estado, mediante los denominados StateTransferFunclets. El comportamiento descrito es el desarrollado por la implementación básica.*

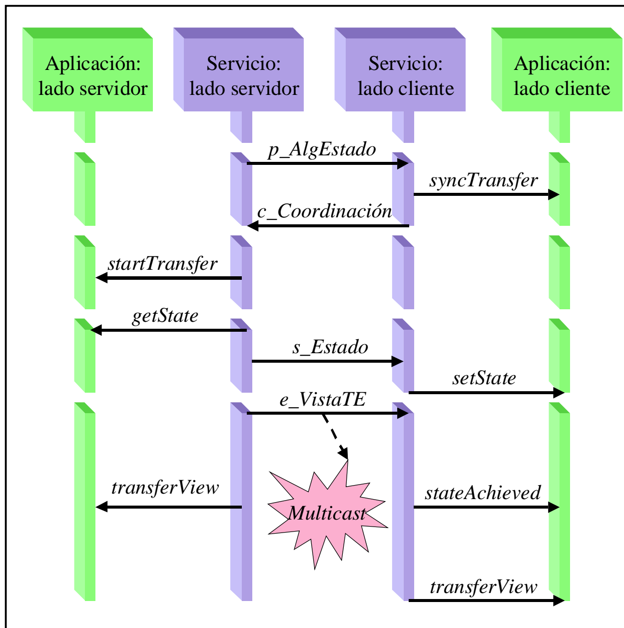

Figure 9: *Transferencia de estado sobre protocolo push*

transferencia debe ser independiente del protocolo de bajo nivel (*push, pull, push-one step*) que se utilice.

Nuestro diseño del servicio CORBA de fiabilidad define un interfaz con la aplicación, con un conjunto de funciones específicas para la transferencia de estado. La implementación de este servicio sí debe hacer uso de los protocolos definidos, seleccionando el más apropiado dependiendo de los recursos del sistema. De esta forma, si se dispone de soporte *multicast*, el servicio de fiabilidad se implementará usando protocolo *pull*. La excepción a esta transparencia se encuentra en la selección automática entre protocolos *push* y *push-one step*. La comparativa mostró que el segundo protocolo es preferible cuando hay soporte *multicast* o cuando el estado es realmente pequeño y no se precisa soporte de propiedades. Sin embargo, esta información no está disponible inicialmente (se conoce durante el protocolo, cuando se crea el mensaje *s*), por lo que la aplicación debe decidir cuando se usa el protocolo *push-one step*.

La figura 9 muestra la transferencia sobre protocolo *push*. Se puede ver que este protocolo se usa exclusivamente en el servicio, para la comunicación entre las partes transmisora y receptora del estado La interacción entre el sistema de transferencia de estado y la aplicación se realiza principalmente mediante primitivas *getState* y *setState*, (el listado IDL de la parte del servicio relativa a la transferencia de estado se incluye al final del articulo) llamadas automáticamente por el sistema de transferencia al detectar un miembro sin estado en el grupo. Junto a estas primitivas, una serie de notificaciones permiten a la aplicación (tanto al lado que recibe como al que transfiere el estado) controlar el estado de la transferencia:

- *syncTransfer*: el servicio llama al lado cliente (receptor del estado) de la aplicación cuando la transferencia se inicia o se reinicia tras una transferencia previa cancelada, para que pueda enviar al lado servidor (emisor del estado) información sobre su propio estado.

- *startTransfer*: con la información de estado recibida del lado cliente, el servicio le comunica al lado servidor de la aplicación que va a iniciarse una transferencia. La aplicación devuelve información sobre la transferencia a efectuar.

- *stateAchieved*: el servicio de transferencia le comunica a un miembro que los demás miembros del grupo le consideran ya como miembro con estado.

- *transferView:* información de vista extendida con la información de estado. Si el GMS presta soporte a la transferencia, esta información esta incluida en las vistas que el GMS instala.

- *transferMemberDown*: notifica a la aplicación que el otro miembro de la transferencia se ha caído. La cancelación de una transferencia se notifica igualmente con esta llamada.

Se define asímismo un objeto llamado *coordinación de fase* para controlar la coordinación de la transferencia, que dirige el comportamiento del servicio en cuanto al número de veces que serán invocados los métodos *getState, setState* y la forma de retomar una transferencia cancelada. Este objeto es definido por la aplicación, y el único interfaz que presenta al servicio de transferencia de estado es una función que define cuando la transferencia ha finalizado.

Este objeto es transmitido entre los lados clientes y servidor (en *syncTransfer, startTransfer, getState, setState*), y el servicio continuara invocando *getState* y *setState* hasta que la aplicación devuelva un objeto de *coordinación de fase* que indique el final de la transferencia. Una implementación típica viene dada por dos contadores, uno definiendo el numero de partes en que se ha dividido el estado y otro indicando la transferencia actual; en cada llamada a *getState*, la aplicación incremente el segundo contador. Si una transferencia es cancelada, el lado cliente o el servidor, a través de *syncTransfer* y *startTransfer*, pueden definir como se reinicia aquella.

De los requisitos de la funcionalidad de transferencia que se listaron en la sección 3ª, los siguientes no han sido aún tratados:

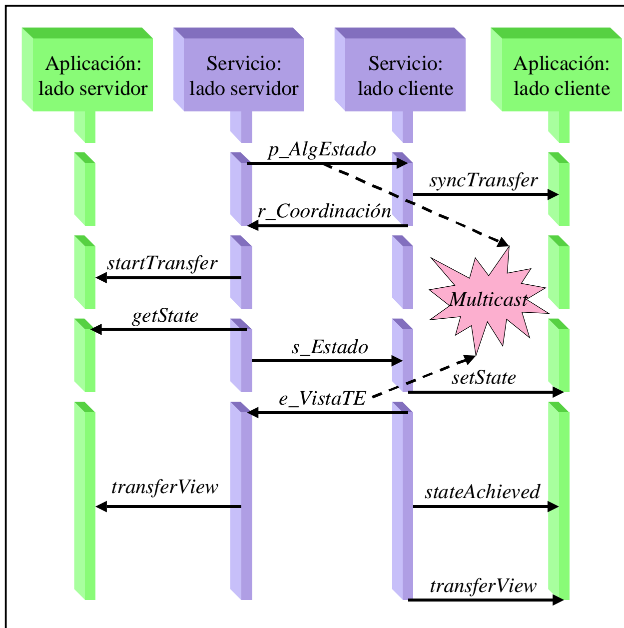

Figura 10: *Transferencia sobre protocolo pull*

- *Propiedades*: los protocolos se han definido con soporte del envío de propiedades. Su definición se realiza mediante las primitivas del sistema de transferencia *getProperty, setProperty*.

- *Elección del coordinador*: *Sensei* implementa automáticamente balanza de carga y pesos mediante el uso de propiedades. De esta forma se define una propiedad específica para la definición de los pesos de cada miembro, y se realiza balanza de carga teniendo en cuenta estos pesos. Si la aplicación realizara la elección del coordinador, debe definirse esta característica en la creación del grupo, mediante un *flag* especial; además, debe definir un método especial (selectTransferCoordinator) que será invocado por el sistema de transferencia de estado en cada nueva transferencia.

- *Limitar la funcionalidad del grupo en la transferencia*: *Sensei* permite agrupar los mensajes por grupos, definiendo para cada grupo cuando deben bloquearse durante las transferencias.

- *Inicialización del grupo*: un miembro puede invocar la primitiva *achieveState* en el sistema de transferencia de estado para que el servicio de fiabilidad pase a considerarlo como miembro con estado. Evidentemente, no debe haber otros miembros con estado en el grupo.

Por último, es posible definir también el grado de concurrencia en la transferencia: si un miembro con estado puede transferir el estado simultáneamente a mas de un miembro. Sin concurrencia, los protocolos de transferencia quedan bloqueados si el miembro con estado que debe realizar una determinada transferencia está ya ocupado en otra.

La figura 10 muestra la transferencia sobre protocolo *pull*. Aunque la implementación del servicio emplea ahora un juego distinto de mensajes, el interfaz con la aplicación se mantiene idéntico.

Con protocolo *push-one step* el interfaz con la aplicación se simplifica, eliminándose la necesidad de los métodos *startTransfer, syncTransfer, stateAchieved* y *transferMemberDown*.

7 Conclusiones

Hay dos conceptos básicas en el desarrollo de los protocolos de transferencia de estado: flexibilidad y eficiencia. Este artículo describe unos protocolos que cumplen ambos requisitos, soportando un sencillo pero potente paradigma para la transferencia de estado.

La idea básica que soporta la flexibilidad de la transferencia es la división del estado en varias partes. La aplicación puede decidir también como manejar transferencias interrumpidas, si estas pueden ser concurrentes, o qué miembro con estado es más apropiado para realizar una transferencia específica. También se pueden definir propiedades, como una característica independiente del concepto de estado del grupo.

La eficiencia se logra mediante la definición de unos protocolos que usan el mínimo ancho de banda posible, y la selección del protocolo particular que mejor se adapta a las necesidades de la aplicación y los recursos del sistema (*push, pull, push-one step*).

La implementación de *Sensei* permite realizar transferencia de estado en grupos de partición primaria, con toda la flexibilidad que habíamos requerido (incluyendo transferencia de propiedades), sin precisar soporte por parte del GMS. También hemos implementado el protocolo necesario para el soporte de transferencia de estado en sistemas particionables (identificación de miembros en subvistas y su agrupación en conjuntos de subvistas y vistas), y trabajamos en la automatización del proceso de unión de estados.

Paralelamente, hemos iniciado su integración con CORBA, con una definición inicial del servicio CORBA de fiabilidad. El siguiente listado muestra su especificación en la parte relativa a transferencia de estado.

```
//AUXILIAR DEFINITIONS
//following definitions are left unspecified

struct GroupMemberID {};
struct PropertyID {};
interface GroupMember {};
typedef sequence <GroupMemberID> ListMemberIDs;

//interface to be implemented by the user-defined
//properties
interface Property
{
  //returns the ID of the property
  PropertyID getPropertyID();
};

//interface to be implemented by any chunk state
interface State {}

//kind of transfer views
enum KindOfTransferView
  {WITHOUT_STATE, TRANSFERRING_STATE, STATE};

//the installed view containing the state
//information
struct TransferView
{
  //kind of view
  KindOfTransferView kind;
  //members having state
  ListMemberIDs stateMembers;
};

exception InvalidMember{};
exception InvalidMemberID{};
exception InvalidPropertyID{};

//the transfer is driven by TransferPhases, to be
//defined by the application
interface TransferPhase
{
  //When the transfer has finished, the next
  //method must return true
  boolean finished();
};

//GroupDefinition includes the group
//characteristics
struct GroupDefinition
{
  //only the state transfer functionality is
  //listed
  ...
  //transferInOneStep is true if the state is
  //transferred in one chunk
  boolean transferInOneStep;

  //true when the application selects the
  //oordinator on each transfer
  boolean appSelectsCoordinator;
};


//STATE TRANSFER SYSTEM
//interface implemented by the State Transfer
//System
interface StateTransferSystem
{
  //the specified member achieves state, if its
  //group doesn't contain any member with state
  void achieveState(in GroupMember member)
       raises (InvalidMember);

  //sets the specified property for the member,
  //overwritting the property if already set
  void setProperty(in GroupMember member,
                   in Property property)
       raises (InvalidMember);

  //returns the property associated to the
  //specified member, having the specified
  //property ID.
  //The call is blocked until the member sends its
  //properties
  Property getProperty(in GroupMemberID member,
                       in PropertyID property)
      raises (InvalidMemberID, InvalidPropertyID);
};


//STATE TRANSFER APP.
//interface to be implemented by GroupMembers
//using the state transfer mechanisms
interface StateTransferApplication
{
  //Downcall to the server side of the application
  //when a transfer is going to start, to allow
  //any pre-processing that the application can
  //need to perform, and to synchronize the
  //transfer with the client.
  //The client list contains just one member if
  //the application doesn't allow for concurrent
  //calls to more than one member.
  //This method is not called under transfers in
  //one step
  TransferPhase startTransfer
               (in ListMemberIDs clients,
                in TransferPhase phase);

  //Downcall to the client side of the application
  //when a transfer is going to start, meaning a
  //communication channel between the client and
  //the server.
  //If there was already a transfer and the
  //previous /coordinator felt down, this method
  //is called with the identity of the new
  //coordinator and the last received phase; the
  //application must return the next expected
  //phase.
  //This method is not called under transfers in
  //one step
  TransferPhase syncTransfer
               (in GroupMemberID coordinator,
                in TransferPhase lastPhase);

  //The application -server- returns the state and
  //the next transfer phase.
  //If this phase flags the end of the transfer,
  //there won't be anymore calls to the method
  //until a new transfer be started.
  State getState (inout TransferPhase phase);

  //Downcall to the application -client- to
  //transfer the state
   void setState(in State state,
                 in TransferPhase phase);

  //Downcall to the application -client- to
  //flag the transfer end.
  //If the transfer is done in one step, this
  //method is not called, its functionality is
```

```
    //included on setState (a call to setState
    //transfers the state and finished the
    //transfer). It can still be called after a call
    //to achieveState()
    void stateAchieved();

    //Installs a new state view on the member, with
    //the list of state members.
    void transferView (in TransferView view);

    //Downcall to the aplication when a member
    //in the transfer has felt down, flagging as
    //well if the transfer is cancelled.
    //On the client side, it's cancelled when no
    //state members are left; in this case, a view
    //without state will be eventually installed.
    //On the server side, if no clients are left in
    //this transfer, it's cancelled
    void transferMemberDown(in GroupMemberID member,
                            in boolean cancelled);

    //if the application will elect the coordinator
    //on each transfer, this election is done
    //through this method
    GroupMemberID selectTransferCoordinator
        (in ListMemberIDs stateMembers);
};
```

Listado 1: *interfaz IDL con la funcionalidad de transferencia de estado del servicio de fiabilidad CORBA*

Referencias

Amir Y., Chockler G., Dolev D., Vitemberg R., *Efficient State Transfer in Partitionable Environments*, February 1997.

Baboglu O., Davoli R., Montresor A., *Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specifications*, University of Bologna, Technical Report UBLCS-95-18, November 1995.

Babaoglu O., Bartoli A., Dini G., *On Programming with View Synchrony*, University of Bologna, Technical Report UBLCS-95-15, September 1995.

Ban B., *JavaGroups - Group Communication Patterns in Java*, Cornell University, July 1998.

Birman, K., *Building Secure and Reliable Network Applications*, Manning Publications, 1996.

Chandra T.D., Toung S., *Unreliable Failure Detectors for Reliable Distributed Systems*, Journal of the ACM, Mar 1996, pp 256-267.

Felber P., *The CORBA Object Group Service. A Service Approach to Object Groups in CORBA*, PhD Thesis, École Polythecnique Fédérale de Lausanne, Laussane, 1998. Number 1867.

Hayden M., *The Ensemble System*, Cornell University Technical Report, TR98-1662, January 1998.

JavaSoft, *Java Remote Method Invocation – Distributed Computing for Java*, White Paper, Sun Microsystems, 14 Oct. 1997.

Maffeis S., *Adding Group Communication and Fault-Tolerance to CORBA*, Proceedings of the USENIX Conference on Object-Oriented Technologies, Monterey, CA, June 1995.

OMG, *The Common Object Request Broker: Architecture and Specification*, Revision 2.2, Feb. 1998. Más información sobre CORBA y OMG en http://www.omg.com.

Renesse R., Birman K., Maffeis S.: *Horus, a flexible Group Communication System*, Communications of the ACM, April 1996.

Ricciardi A., Schiper A., Birman K., *Understanding Partitions and the "No Partition" Assumption*, IEEE Proc Fourth Workshop on Future Trends of Distributed Systems, Lisbon, September 22-24, 1993.

Sessions R., *COM and DCOM: Microsoft's Vision for Distributed Objects*, John Wiley & Sons, 1997.

Schipper A. , Sandoz A., *Understanding the power of the virtually-synchronous model*, Proc of the 5th European Workshop on Dependable Computing, Lisbon, February 17-19, 1993.

Vaysburd A., *Building Reliable Interoperable Distributed Objects with the Maestro Tools*, Cornell University Technical Report, TR98-1678, May 1998.