UNIVERSIDAD POLITÉCNICA DE VALENCIA

Departamento de Sistemas Informáticos y Computación

HIDRA: INVOCACIONES FIABLES Y CONTROL DE CONCURRENCIA

Tesis presentada por:
Francesc Daniel Muñoz i Escoí

Dirigida por:
Dr. José Manuel Bernabéu Aubán

Índice General

Índice de Figuras

Índice de Tablas

ABSTRACT

HIDRA is an architecture that provides support for highly available objects in distributed systems. To detect the failures and reactivations of the nodes that compose a distributed system, a cluster membership protocol is needed. HMM is a cluster membership protocol that is used in HIDRA to assist its components in the reconfiguration tasks that must be taken when a membership change arises. It is the first of the HIDRA components described in this thesis.

The support for high availability is usually based on object replication. Several replication models exist. The coordinator-cohort model is a combination of several characteristics of the passive and active ones. It presents some advantages when it is compared to each one of the other two models, but it requires a distributed concurrency control mechanism and an invocation mechanism that ensures atomicity and consistency. A design of both mechanisms is also presented in this work, allowing a native implementation of this replication model. Thus, HIDRA is the first architecture that provides direct support for the coordinator-cohort replication model, without needing its implementation on top of the active model.

RESUMEN

HIDRA es una arquitectura que proporciona soporte para objetos altamente disponibles en sistemas distribuidos. Para detectar los fallos y reactivaciones de los nodos que componen un sistema distribuido, un protocolo de pertenencia a cluster resulta necesario. HMM es un protocolo de este tipo utilizado en HIDRA para dirigir a sus componentes en las tareas de reconfiguración que deben ser tomadas en caso de que ocurra un cambio en el conjunto de pertenencia. Es el primero de los componentes de HIDRA descrito en esta tesis.

El soporte para alta disponibilidad está normalmente basado en replicación de objetos. Existen múltiples modelos de replicación. El modelo coordinador-cohorte es una combinación de algunas características de los modelos activo y pasivo. Presenta varias ventajas si se compara con cualquiera de los otros dos modelos mencionados, pero requiere un mecanismo de control de concurrencia distribuido y un mecanismo de invocación que garantice atomicidad y consistencia. Un diseño de ambos mecanismos se presenta en este trabajo, permitiendo una implementación nativa de este modelo de replicación. Así, HIDRA es la primera arquitectura que facilita soporte directo para el modelo de replicación coordinador-cohorte, sin necesidad de implementarlo sobre el modelo activo.

RESUM

HIDRA és una arquitectura que proporciona suport per a objectes altament disponibles en sistemes distribuïts. Per a detectar fallades i reactivacions dels nodes que composen un sistema distribuït, un protocol de pertinença a cluster resulta necessari. HMM és un protocol d'aquest tipus utilitzat en HIDRA per a dirigir als seus components en les tasques de reconfiguració que han de ser preses en cas de que ocòrrega un canvi al conjunt de pertinença. És el primer dels components d'HIDRA descrit en aquesta tesi.

El suport per a alta disponibilitat està normalment basat en replicació d'objectes. Existeixen múltiples models de replicació. El model coordinador-cohort és una combinació d'algunes característiques dels models actiu i passiu. Presenta múltiples avantatges si es compara amb qualsevol

dels altres dos models esmentats, però necessita un mecanisme de control de concurrència distribuït i un mecanisme d'invocació que garantitze atomicitat i consistència. Un disseny d'ambdos mecanismes es presenta en aquest treball, permetent una implementació nadiua d'aquest model de replicació. Així, HIDRA és la primera arquitectura que facilita suport directe per al model de replicació coordinador-cohort, sense necessitat d'implantar-lo per damunt del model actiu.

Capítulo 1

Introducción

1.1 Introducción

La continua mejora en las prestaciones ofrecidas por los ordenadores personales, así como la reducción de sus costes de fabricación permiten que se pueda disponer de máquinas con alta capacidad de procesamiento a un coste reducido. Esto, unido a los avances que también se han dado en las redes de área local en la última década, aunque no tan espectaculares como los que han disfrutado los procesadores, permite que actualmente se pueda disponer de un grupo de ordenadores para realizar conjuntamente un determinado servicio. Con ello ha surgido el concepto de *sistema en cluster* que, como cualquier otra variante de un sistema distribuido, presenta algunas características que lo hacen recomendable para mejorar la disponibilidad de los servicios que ofrezca.

En esta tesis se va a describir parte de la arquitectura HIDRA [GMB97b], pensada para ofrecer soporte de alta disponibilidad y fácilmente implementable sobre un sistema en cluster. En concreto, se describirán los componentes de HIDRA relacionados con la gestión de la pertenencia en el conjunto de máquinas que formen el cluster, con la invocación de objetos replicados y con el control de concurrencia en ese protocolo de invocación de objetos.

Existen múltiples alternativas para proporcionar soporte para alta disponibilidad, casi todas ellas basadas en la replicación de componentes. En HIDRA se ha optado por utilizar un modelo orientado a objetos, empleando un ORB como base de la arquitectura y el modelo de replicación coordinador-cohorte para gestionar la replicación de objetos.

En este capítulo se van a presentar algunos conceptos preliminares que será necesario conocer antes de describir en detalle los componentes que forman parte de la arquitectura HIDRA. Así, se empezará en la sección 1.2 presentando el concepto de sistema en cluster, relacionándolo con el de sistema distribuido. A continuación, la sección 1.3 describe qué se entiende por alta disponibilidad y cómo puede conseguirse ésta empleando replicación (sección 1.4). Por último, las dos secciones restantes presentan las contribuciones de esta tesis y la organización de los capítulos que siguen.

1.2 Sistemas en cluster

Los sistemas en cluster son un caso particular de *sistema distribuido* donde se presta especial interés en proporcionar una imagen de *sistema único*, es decir, para un ordenador o un usuario

externo al conjunto de máquinas que formen el cluster, dicho conjunto debe ofrecer la imagen de que en él sólo hay una máquina.

1.2.1 Concepto de sistema distribuido

Definiciones sobre qué es un sistema distribuido pueden encontrarse muchas. Cada una de ellas puede dedicar atención especial a un determinado detalle de este tipo de sistemas, según el contexto en el que aparezca. En [CDK94] se da una definición suficientemente general, que ligeramente adaptada dice lo siguiente: "*un sistema distribuido es una colección de ordenadores autónomos interconectados mediante una red y equipados con unos programas que permiten a estos ordenadores coordinar sus actividades y compartir sus recursos (equipos, programas y datos)*". En particular, esta definición resulta adecuada porque no da preferencia a ningún modelo de programación, únicamente exige la presencia de múltiples máquinas y una red de interconexión y da a entender que los ordenadores empleados deben colaborar hacia un objetivo común. Definiciones similares a ésta podremos encontrarlas en [Gal00], [Sta98] y [Nut97], aunque este último prefiere el término *multicomputador* para hacer referencia a este tipo de sistemas, englobando tanto a los sistemas distribuidos que aparecen en las definiciones citadas anteriormente como a los sistemas multiprocesadores de memoria compartida. La definición que hemos elegido permite englobar tanto a procesadores con memoria compartida como a ordenadores con memoria privada.

El modelo de programación básico en un sistema distribuido, según el mecanismo de intercomunicación de procesos, estará basado en intercambio de mensajes. Sin embargo, es mejor no dejar esto patente en la definición de lo que es un sistema distribuido (véanse las definiciones de [SS94, SG98] para encontrar ejemplos donde se dice explícitamente que no puede compartirse memoria entre los procesadores) ya que sobre este modelo básico se pueden dar implementaciones de memoria compartida distribuida [Cho94, Bat98] que contradirían la definición dada.

Pero la característica más importante de un sistema distribuido, al menos en relación a lo que va a describirse en esta tesis, es su capacidad para tolerar fallos en algunos de sus componentes. A diferencia de lo que ocurre en un *sistema centralizado*, en un sistema distribuido habrá múltiples unidades de cómputo y de gestión de recursos que serán independientes (al menos en cuanto a su probabilidad de fallo). Por tanto, la probabilidad de que falle el sistema en su totalidad será menor. Aun así, debe aplicarse un esfuerzo especial en la gestión del sistema para conseguir que las aplicaciones que se ejecuten en él no adviertan el fallo de algunos de sus componentes. Para ello deberán emplearse técnicas de alta disponibilidad, como la replicación de componentes, que permitirán que el sistema siga comportándose de acuerdo con sus especificaciones incluso cuando alguno de sus elementos falle. En caso de no adoptar estas técnicas, de poco serviría utilizar un conjunto de máquinas para proporcionar un determinado servicio, pues el fallo de una cualquiera de ellas podría tener como resultado que el servicio proporcionado dejase de estar disponible.

Otra ventaja de estos sistemas radica en su *escalabilidad*, es decir, en la posibilidad de aumentar la capacidad de servicio de un sistema de este tipo. Para ello basta adquirir nuevos ordenadores y conectarlos al sistema que ya existía, reconfigurándolo para que estas nuevas unidades pasen a ser también utilizadas. En los sistemas centralizados tradicionales no existía más alternativa que adquirir una máquina más potente que reemplazase a la ya existente.

1.2.2 Concepto de sistema en cluster

Un sistema en cluster es un caso particular de sistema distribuido en el que:

- Se proporciona la imagen de sistema único. Es decir, aquellos componentes externos al sistema (aplicaciones, usuarios y otros sistemas) deben percibir la imagen de que éste está formado por una sola máquina.

  Además, cuando esta imagen es proporcionada por el núcleo del sistema operativo empleado en dicho sistema distribuido, las aplicaciones que se ejecuten en él también percibirán dicha imagen. Esto facilita en cierta medida la tarea de los programadores, que no tendrán que preocuparse por algunos detalles a la hora de desarrollar dichas aplicaciones. De todas formas, esto no siempre resultará recomendable ya que al ocultar dichos detalles y dar una imagen ciertamente diferente a la real, el programador podrá adoptar decisiones de diseño que conduzcan a un peor rendimiento de la aplicación resultante. Sin embargo, sí que resulta agradable a la hora de portar aplicaciones pensadas para otro tipo de sistemas, ya que en ese caso podrán funcionar con el soporte proporcionado.

- Existe una red de interconexión de los ordenadores que compongan el cluster con elevadas prestaciones (retardo de transmisión muy bajo y elevado ancho de banda). Ejemplos de este tipo son las redes *SCI* [Dol96], especificadas en el estándar *ANSI/IEEE 1596-1992 Scalable Coherent Interconnect* o las redes *Myrinet*, creadas por la compañía Myricom, pero estandarizadas posteriormente en la especificación ANSI/VITA 26-1998.

  Como características de las redes SCI cabe citar:

  - Se utilizan mensajes cortos, con 16 bytes de cabecera y 16, 64 o 256 bytes de contenido útil.
  - Ancho de banda entre 1 y 4 Gbits/segundo.
  - Cables de interconexión dobles (bidireccionales). En cada “hilo” sólo se puede transmitir información en un sentido.
  - Protocolos de comunicación propios que dan soporte tanto a modelos de programación de memoria compartida como a intercambio de mensajes.
  - Retardo de transmisión entre 1 y 2 microsegundos en modo memoria compartida y entre 5 y 10 microsegundos en modo de intercambio de mensajes. (Estos son los datos declarados por el fabricante, puede que en mediciones reales sean superiores según el tipo de máquina empleada).
  - Direcciones de nodo de 16 bits. Esto ofrece un máximo de 65536 nodos interconectados en una misma red SCI.

  Por su parte las redes Myrinet ofrecen estas propiedades:

  - Ancho de banda de 2 Gbits/segundo.
  - Cables de interconexión bidireccionales.

  - Servicio de monitorización de máquinas integrado en los propios protocolos de comunicación. Esto facilita la detección de fallos y simplificaría la implementación de un protocolo de pertenencia.
  - Posibilidad de cambios de ruta efectuados por los propios switches Myrinet en caso de caída de alguna máquina conectada a la red.
  - Retardo de transmisión de mensajes entre 13.37 y 21 microsegundos. Estos son resultados obtenidos en algunos benchmarks.

  El uso de una red de estas características tiene como objetivo aumentar el rendimiento del sistema. En la práctica no existe ningún inconveniente que impida utilizar una red de área local convencional, como una Ethernet 100Mb/s para realizar la interconexión.

Existen otras características de estos sistemas, pero en la práctica se derivan de las dos que acabamos de citar y no hay que prestarles excesiva atención para calificar a un sistema distribuido como “cluster” o no. Por ejemplo, cabe resaltar:

- Uso de una red privada. En [Pfi98] se distingue entre *clusters cerrados* y *expuestos*. Para este autor, en un cluster cerrado, todos los ordenadores del cluster están interconectados mediante una red privada de altas prestaciones, que resulta inaccesible para las máquinas que no pertenezcan al cluster. El acceso al sistema se realizará utilizando otra red a la que estarán conectados algunos de los nodos del cluster (aquellos que tengan al menos dos interfaces de red).

  Los clusters expuestos son los que no utilizan una red privada, permitiendo que los mensajes entre nodos del cluster sean transmitidos por la misma red a la que tendrán acceso las máquinas externas a él.

  El objetivo que se pretende conseguir con el uso de una red privada es la mejora de las prestaciones en las comunicaciones internas. Al reducir el tráfico en la red utilizada para las conexiones internas, habrá menos colisiones en el acceso a ella (aunque esto depende del tipo de red utilizada) y la cantidad de información real transferida podrá ser mayor.

- *Anonimia* de los nodos: Esto significa que los nodos que componen el cluster no deben recibir un nombre en los servicios de nominación empleados fuera del cluster. Con ello se evita el acceso individual a cada uno de ellos.

  Esto es una consecuencia de la imagen de sistema único. También puede ayudar a la presentación de anonimia el uso de una red privada interna.

Por ser un caso particular de un sistema distribuido, al igual que en ellos, en un sistema en cluster se podrán plantear ciertos objetivos que no pueden ser conseguidos de igual manera utilizando una sola máquina. De entre ellos ya hemos citado la facilidad para hacer el sistema escalable, pero también podríamos citar el acceso mucho más cómodo a los recursos ubicados en otras máquinas (siempre y cuando también formen parte del cluster), así como la opción de realizar un reparto de carga entre los ordenadores que compongan el cluster de manera que todos ellos deban soportar un conjunto de aplicaciones a ejecutar que esté acorde con sus posibilidades, mejorando así el rendimiento global del sistema.

Pero el objetivo principal, al menos para el contexto de esta tesis, será la mejora de la disponibilidad de las aplicaciones que se ejecuten en este sistema. Este concepto se explica en detalle en la próxima sección.

1.3 Alta disponibilidad

Para entender el concepto de alta disponibilidad es necesario presentar previamente los conceptos de fallo, fiabilidad y disponibilidad que aparecen a continuación. Tras ello se definirá el concepto de alta disponibilidad y se describirá de qué forma es posible obtenerla.

1.3.1 Fallos

Según [Nel90], se da un *fallo* en un sistema cuando éste presenta incapacidad para desarrollar aquellas funciones para las que fue diseñado debido a errores en él o en su entorno, que hayan sido causados por diferentes faltas. A su vez, define la *falta*[1] como una condición anómala y el *error* como la manifestación de una falta en un sistema, donde el estado de un componente diferirá del previsto.

Nuestro objetivo será que un sistema tolere faltas, de manera que nunca presente fallos. Para ello, se podría empezar reduciendo a un mínimo la posibilidad de que se den esas faltas (es decir, que el comportamiento especificado del sistema o de sus componentes pueda extenderse al mayor número posible de situaciones, por lo que se disminuirá la probabilidad de que ocurran condiciones "anómalas"). Sin embargo, existen ciertos tipos de faltas que será imposible evitar: errores del propio usuario, errores de implementación, etc. Ante estas faltas, debería evitarse su conversión en errores. Para ello, los componentes que sufran las faltas deberían ser capaces de evitar que otros componentes relacionados con ellos apreciasen la ocurrencia de la falta.

Como resultado, en [Cri91a] se dice que un sistema es *tolerante a faltas* (o "sin fallos") cuando éste exhibe un comportamiento bien definido en caso de faltas o enmascara las faltas de sus componentes a sus usuarios (es decir, continúa facilitando sus servicios estándar a pesar de la ocurrencia de dichas faltas). Nótese que para que un servicio sea tolerante a faltas, forzosamente deben serlo también todos aquellos servicios de los cuales dependa, es decir, todos aquellos que llegue a necesitar en algún momento.

Como veremos posteriormente, es prácticamente imposible tener un sistema completamente tolerante a faltas. Las técnicas que pueden utilizarse para enmascarar los faltas no son siempre perfectas y algunos fallos sí podrán ser apreciados por otros componentes o por el usuario de nuestro sistema. Por tanto, más que hablar de sistemas tolerantes a faltas, hoy día se prefiere utilizar términos relacionados con la fiabilidad y la disponibilidad de un sistema.

1.3.2 Fiabilidad

Según la definición dada en [Nel90], debe entenderse por *fiabilidad* de un sistema, *F(t)*, la probabilidad condicionada de que éste pueda desarrollar sus funciones correctamente en el instante *t*, sabiendo que era operativo en el instante *t=0*.

[1] El término inglés para referirse a este concepto es "fault" que en nuestro idioma también suele traducirse como "fallo". Se ha preferido utilizar el término "falta", que aunque no sea la traducción habitual evita la ambigüedad al usar la palabra "fallo".

Esta fiabilidad depende por una parte de las faltas que puedan darse en el sistema y por otra de los mecanismos que éste posea para evitar que se manifiesten fallos cuando se den esas faltas.

Para poder clasificar a un sistema como completamente *fiable*, éste debería tener mecanismos que evitasen la aparición de cualquier fallo. Lo que se conoce como "*mecanismos de recuperación automática*".

En la práctica, vuelve a ser altamente improbable tener a un sistema con capacidad para evitar todos los tipos posibles de fallos. Por ello, la fiabilidad de un sistema se suele dar numéricamente como una probabilidad cuyo valor viene dado por la siguiente expresión:

$$F = P(sin\ faltas) + P(funcionamiento\ \ correcto/faltas) * P(faltas)$$

Es decir, la suma de la probabilidad de que el sistema no tenga ninguna falta durante un cierto intervalo de tiempo más la probabilidad de que se dé un funcionamiento correcto en caso de falta multiplicada por la probabilidad de que se dé una falta durante ese intervalo. La probabilidad de que se tenga un funcionamiento correcto en caso de faltas dependerá de los mecanismos de recuperación automática que posea el sistema.

Si nuestro objetivo era tolerar faltas y utilizamos el concepto "fiabilidad" para explicar el grado de tolerancia a faltas que ofrece un sistema determinado ya tendremos al menos una herramienta para determinar la calidad de dicho sistema. Sin embargo, la fiabilidad deja algunas cosas sin medir. En concreto, conoceremos la probabilidad de que nuestro sistema llegue a manifestar algún fallo, pero desconoceremos cuánto tiempo emplearemos en dejar de nuevo al sistema en un estado operativo. Por este motivo, cuando se habla de tolerancia a faltas para componentes software se suele utilizar otro concepto complementario que sí recoge la duración de los periodos no operativos (o de recuperación). Este concepto es la disponibilidad.

1.3.3 Disponibilidad

La *disponibilidad* [Nel90] de un sistema o servicio es la probabilidad de que éste se encuentre operativo en un determinado instante. Para asignar valores numéricos a la disponibilidad deberá emplearse la siguiente expresión:

$$Disponibilidad = \frac{TMEF}{TMEF + TMDR}$$

Donde *TMEF* es el *tiempo medio entre fallos* y *TMDR* es el *tiempo medio de recuperación* (o de reparación).

Nótese que la disponibilidad nos proporciona mayor información sobre la capacidad de prestación de servicios que la fiabilidad. Podríamos tener un sistema más fiable y, en la práctica, menos disponible que otro si la probabilidad de fallo fuera menor pero el tiempo necesario para recuperar el sistema en caso de fallo fuese muy superior.

Nuestro objetivo será tener un sistema altamente disponible. Esto querrá decir que deberá tener una alta fiabilidad y que además, en caso de fallo va a requerir un tiempo mínimo para recuperarse.

Como puede desprenderse de la definición vista arriba, la disponibilidad puede expresarse también como una probabilidad. En [Pfi98] se da una clasificación de diferentes grados de disponibilidad en función del número de "nueves" que va a tener el valor numérico de la disponibilidad. Las clases resultantes aparecen en la tabla 1.1.

Disponibilidad	Tiempo no disponible por año	Clase
90 a 99 %	entre 4 días y un mes	1
99 a 99.9 %	entre 9 horas y 4 días	2
99.9 a 99.99 %	entre 1 y 9 horas	3
99.99 a 99.999 %	entre 5 minutos y 1 hora	4
99.999 a 99.9999 %	entre medio y 5 minutos	5
99.9999 a 99.99999 %	entre 3 y 30 segundos	6

Tabla 1.1: Clases de disponibilidad.

1.3.4 Concepto de alta disponibilidad

Vistos ya todos los conceptos previos que hemos presentado en la secciones anteriores, estamos en condiciones de definir qué se va a entender por *alta disponibilidad*. Para ello vamos a utilizar las clases de disponibilidad presentadas en la tabla 1.1 y diremos que un sistema puede clasificarse como *altamente disponible* cuando la disponibilidad media que ofrezca esté dentro de las clases 5 ó 6. Es decir, ha de tener una disponibilidad mayor a un 99.999%.

1.3.5 Técnicas para mejorar la disponibilidad

La técnica básica para mejorar la disponibilidad de un sistema o aplicación es la replicación de sus componentes, evitando así que exista ningún *punto falible no replicado*. Con ello se tendrán múltiples réplicas de cada componente, ubicadas cada una de ellas en un nodo distinto del sistema distribuido o del cluster. Si se llegase a producir un fallo y cayese alguno de estos componentes, sus servicios serían atendidos por alguna de sus réplicas con lo que el usuario apenas percibiría un leve retraso en la respuesta.

Pero para que esto sea posible se necesitan además algunos servicios complementarios. Uno de ellos es el encargado de monitorizar continuamente el estado de los componentes que formen el sistema o la aplicación altamente disponible. Recibe el nombre de *monitor de pertenencia* [MBG97] y su misión consiste en detectar cuándo un componente ha fallado o cuándo se ha recuperado, notificando al resto cualquiera de estos eventos, permitiendo que así reconfiguren tanto su estado como el conjunto de clientes a atender. El uso de este tipo de servicios es básico para mantener de forma adecuada el estado de un objeto replicado. Así se podrá saber qué réplicas se mantienen operativas en cada momento y se podrá reconfigurar rápidamente el estado del objeto en caso de caída o recuperación de alguna de estas réplicas.

Otra herramienta útil para dar soporte a las técnicas de replicación son los mecanismos necesarios para realizar *actualizaciones de estado* (o "*checkpoints*"). Es decir, en caso de que no todas las réplicas tomen un papel activo para procesar una determinada petición que implique la modificación del estado, la réplica que haya procesado tal petición deberá comunicar posteriormente a las demás qué cambios deberán realizar para tener al final un estado consistente en todas las réplicas. Para realizar estas actualizaciones de estado convendría además utilizar algún tipo de soporte transaccional para proporcionar suficientes garantías de atomicidad, consistencia y aislamiento en los cambios a realizar. La solución adoptada en estos casos para realizar estas actualizaciones de

estado depende en gran medida del modelo de replicación que se esté empleando.

Pero la técnica básica para mejorar la disponibilidad de un servicio no será otra que la replicación de los componentes que proporcionan dicho servicio. En la próxima sección se describirá en detalle qué alternativas de replicación existen, así como las principales ventajas e inconvenientes de cada una de ellas.

1.4 Replicación

La replicación de servidores es una de las técnicas básicas para garantizar su alta disponibilidad. Pero para gestionar un servidor replicado hay que tomar ciertas decisiones que influyen en el comportamiento que ofrecerá ese servidor a sus clientes: cuántas réplicas deben existir, dónde se ubicarán, qué réplicas recibirán las peticiones de los clientes, cómo se garantizará la consistencia del estado de las réplicas, qué tipo de *consistencia* se desea (es decir, qué diferencias entre el estado de las diferentes réplicas podrán ser admitidas), cómo se canalizarán los resultados de una invocación hasta el cliente que la ha iniciado, etc. Todo ello define un *modelo de replicación*.

1.4.1 Modelos

Existe un conjunto de características que definen el modelo de replicación que se está empleando. Estas características y las posibles alternativas que pueden escogerse para cada una de ellas, son las siguientes:

- *Grado*. Indica el número de réplicas que va a mantener el servicio y condicionará el número de fallos que podrán admitirse.

- *Réplicas activas/pasivas*. Se entiende por réplica activa aquella que recibe directamente la petición de un cliente y la sirve, modificando el estado del servidor replicado. Por contra, una réplica pasiva no recibe ni trata las peticiones de los clientes sino únicamente las *actualizaciones de estado* originadas por las réplicas activas.

  El número de réplicas activas y la posibilidad de cambios entre rol activo y pasivo definen al modelo que se esté empleando.

- *Difusión de peticiones*. La petición iniciada por el cliente debe ser dada a conocer (difundida) a todas las réplicas del servidor. Puede ocurrir que la *difusión* se realice de manera *previa* y que la petición llegue directamente a todas las réplicas servidoras o puede que únicamente llegue a una réplica que tras procesar la petición difundirá sus resultados tanto al cliente como al resto de réplicas (réplicas pasivas con *difusión posterior*).

- *Filtrado de respuestas*. En caso de que la petición sea procesada directamente por más de una réplica, cada una de ellas generará una respuesta independiente. No resulta conveniente hacer llegar todas las respuestas al cliente, una tras otra, por lo que debe efectuarse algún tipo de filtrado. Las opciones existentes son: elegir la primera, elegir la que ha sido proporcionada por más réplicas (al estilo de una votación), elegir una cualquiera, etc.

Los modelos de replicación más importantes: activo, pasivo y coordinador-cohorte, se describen en las próximas secciones.

1.4.2 Modelo activo

En el *modelo de replicación activo* [Sch93a], todas las réplicas del servicio son activas, se utiliza difusión previa y algún tipo de filtrado de respuestas (la variante utilizada no importa en exceso). Con ello, este modelo presenta la ventaja de tener un mínimo tiempo de reconfiguración, ya que en caso de caída no es necesario realizar ninguna tarea compleja para restaurar el estado del servicio ni para cambiar el rol de sus réplicas (todas ellas siempre son activas).

Por otra parte, sí presenta el inconveniente de que para garantizar la consistencia de todas las réplicas debe asegurarse que todas ellas reciben todas las peticiones en cierto orden (que no tiene por qué ser *orden total* estricto, puede en algunos casos ser simplemente *causal*). Para ello deben emplearse *protocolos de difusión atómica* [HT93, BvR94], es decir, que garanticen que la difusión llegue a todas las réplicas o a ninguna y cuyo coste no es precisamente bajo. Además, aquí se exige que las réplicas del servicio invocado utilicen una codificación basada en un solo hilo de ejecución, puesto que si se da soporte a múltiples hilos de ejecución que puedan dar servicio a múltiples peticiones concurrentemente, podrían aparecer inconsistencias debido al *planificador* a corto plazo que utilice el sistema. Una solución para que desapareciera esta restricción podría ser la utilización de un *planificador determinista* [JPA00] en cada una de las réplicas.

La *adición de una réplica* también plantea problemas, pues al ser todas ellas activas debe asegurarse que ésta recibe inicialmente el mismo estado que todas las demás y que, a partir de ese momento, se integre perfectamente en el grupo y reciba las mismas peticiones que el resto de réplicas. Para ello, la operación de adición también debe ser ordenada como una petición más y exige que, mientras ésta se lleve a cabo, el resto de réplicas no actualicen su estado o se recuerde qué peticiones han servido para suministrarlas posteriormente a la nueva réplica.

Otra dificultad aparece cuando un objeto replicado activamente debe invocar los servicios de cualquier otro objeto. Como todas las réplicas son activas, se generaría una petición por cada una de ellas y esto sobrecargaría excesivamente al objeto que se desa invocar. Para ello, debe procederse a filtrar también las peticiones hacia otros servicios. Lo mismo ocurrirá cuando un objeto de este modelo de replicación decida acceder a un dispositivo de almacenamiento compartido por todas las réplicas (el problema desaparece si cada réplica del objeto accede a una réplica distinta del dispositivo de almacenamiento). La solución será la misma que en el caso anterior: filtrar.

Por último, también hay que considerar la carga introducida por este modelo, ya que cada réplica debe procesar cada petición y requiere tiempo de CPU y uso de otros recursos para lograr dar este servicio. Los otros dos modelos estudiados tienen un coste bastante inferior, como veremos en las próximas secciones.

Como resultado, aunque es el modelo que necesita un menor tiempo de reconfiguración, y ésta es una cualidad muy importante cuando se habla de proporcionar alta disponibilidad, también implica el uso de costosos protocolos de difusión y requiere otras soluciones menores para algunos detalles adicionales.

1.4.3 Modelo pasivo

En el *modelo de replicación pasivo* [BMST93], únicamente existe una réplica activa, llamada *réplica primaria* y múltiples réplicas pasivas, llamadas *réplicas secundarias*. Se utiliza difusión posterior y, gracias a ello, no resulta necesario efectuar un filtrado de respuestas. Su principal inconveniente radica en el tiempo de reconfiguración, pues hay que efectuar un cambio de rol de

una de las réplicas cuando falla la réplica primaria y esto exige además un *protocolo de elección de líder*, con el consiguiente cambio en los clientes, que ahora deberán dirigirse a la nueva réplica primaria que se haya elegido.

A diferencia de lo que sucede en el modelo de replicación coordinador-cohorte, en el modelo pasivo el rol de una determinada réplica es estático y no cambia a menos que se dé un fallo.

En este modelo, la forma de tratar una petición iniciada por un cliente externo es la siguiente. En primer lugar, el cliente dirige su petición a la réplica primaria, que procesa tal petición y actualiza su estado convenientemente. Cuando ya se ha logrado esto, la réplica primaria procede a realizar un *checkpoint* sobre sus réplicas secundarias (en algunos casos, el checkpoint se realiza de manera periódica, por lo que no está ligado a ninguna petición), con lo cual logra que el estado de éstas pase a ser consistente con el suyo. Por último, la réplica primaria devuelve la respuesta al cliente.

Como ventaja principal del modelo cabe citar la baja carga que implica, pues las peticiones sólo son atendidas activamente en una réplica y resulta fácil controlar tanto las peticiones originadas por esta réplica sobre servicios externos como los accesos que realice sobre dispositivos de almacenamiento o de cualquier otro tipo, que no serán intentados por ninguna réplica secundaria. Esto también hace innecesario el uso de un protocolo de difusión atómica, como el empleado en el modelo activo para garantizar cierto orden en la llegada de peticiones a todas las réplicas activas, por lo que el coste también se rebaja en ese apartado.

Sin embargo, no todo son ventajas en este modelo. Aparecerán algunos problemas, al igual que en el modelo activo, cuando un objeto replicado necesite invocar a otro objeto replicado. En este caso el problema no aparece por la necesidad de filtrar peticiones o respuestas sino por la posibilidad de que la réplica primaria del objeto replicado cliente pueda caer tras haber realizado una petición y antes de haber obtenido una respuesta. La solución que deberá proporcionarse en este caso dependerá de la implementación final que se haya realizado de este modelo de replicación. En principio, para solucionarlo bastaría con realizar checkpoints sobre todas las réplicas secundarias cuando un primario pida un servicio a otro objeto replicado, de manera que todas las réplicas secundarias sepan que tal petición se ha iniciado. Cuando se reciba la respuesta habrá que realizar de nuevo un checkpoint para notificar su llegada. Gracias al checkpoint utilizado inmediatamente antes de la invocación, la réplica que deba sustituir a un primario que haya fallado sabrá que tendrá que esperar la respuesta a una invocación iniciada por el antiguo primario. Por otra parte, el objeto replicado que fue invocado deberá tener alguna forma de averiguar cuál es la nueva réplica primaria que ha sustituido a la que haya fallado y deberá retornar la respuesta a esa nueva réplica primaria.

1.4.4 Modelo coordinador-cohorte

En el *modelo de replicación coordinador-cohorte* [BJRA85] únicamente existe una réplica activa y varias réplicas pasivas, al igual que en el modelo pasivo, pero esto es únicamente así si sólo consideramos una petición en particular. El rol activo o pasivo de cada réplica puede variar de una petición a otra. El resto de características es similar al modelo pasivo: no se necesita difusión de peticiones ni filtrado de respuestas.

Cuando una réplica desempeña el papel activo en una petición recibe el nombre de *réplica coordinadora* para esa petición. Si, por contra, desempeña un papel pasivo se la llama *réplica*

cohorte.

En este modelo, a diferencia de lo que ocurría en el pasivo, cada réplica puede comportarse tanto de manera activa como pasiva sin necesidad de ser reconfigurada o promocionada a la otra categoría. Por tanto, aquí se elimina la necesidad de reconfigurar las réplicas y elegir una nueva réplica activa en caso de fallo, como sucedía en el modelo pasivo. Además, por el hecho de que cada petición sólo tenga una réplica activa, se elimina la necesidad de emplear protocolos de difusión atómica para asegurar la consistencia en las invocaciones (no obstante, para lograr esta consistencia habrá que tomar otras medidas adicionales), no se sobrecargan todas las réplicas con el proceso de la petición y no hay que controlar las peticiones que dirija la réplica activa sobre servicios externos, ni filtrar las respuestas hacia el cliente. Por ello, se eliminan los costes más importantes que presentaban los dos modelos anteriores.

Sin embargo, la posibilidad de que múltiples peticiones lleguen concurrentemente y elijan diferentes réplicas coordinadoras introduce problemas de control de concurrencia, que habrá que resolver de manera distribuida y que no estaban presentes en los otros dos modelos. En estos últimos la solución podía ser local a cada réplica, sin considerar para nada a las restantes. En el modelo coordinador-cohorte cabe la posibilidad de que estas múltiples peticiones concurrentes traten de modificar la misma parte del estado del objeto replicado en diferentes réplicas coordinadoras. Obviamente, este tipo de accesos debe evitarse, dando precedencia a una de las dos peticiones e iniciando la otra cuando la primera termine.

Un segundo problema, relacionado con el de control de concurrencia, es el garantizar la *atomicidad* de las actualizaciones. En el caso del modelo pasivo esto no era difícil, porque se podían realizar los checkpoints de manera síncrona antes de devolver el resultado al cliente e iniciar una nueva operación. Sin embargo, aquí podemos tener múltiples peticiones que no tengan *conflictos* entre ellas (que no modifiquen ambas la misma parte del estado del objeto) funcionando concurrentemente y las actualizaciones que efectúen deben terminar en todas las réplicas antes de que se inicie una nueva petición en cualquier otra réplica. La solución de los checkpoints síncronos también es aplicable, pero es preferible utilizar algún tipo de transacción además del sincronismo, para evitar la repetición del servicio de una petición en caso de fallos.

Por lo que respecta a la interacción entre dos servicios replicados bajo este modelo (es decir, una réplica coordinadora de un objeto A pide la realización de cierta operación a otro objeto replicado B), hay que decir que la situación que se plantea en este entorno mantiene idénticas características a las presentadas por el modelo de replicación pasivo. Así, no hay necesidad de filtrado de peticiones o respuestas, pero hay que tener especial cuidado con la caída de la réplica coordinadora cliente. La forma de tratar esa situación es idéntica a la planteada anteriormente para el modelo pasivo.

1.4.5 Otros modelos

Otro ejemplo de modelo de replicación empleado principalmente en el campo de la gestión de bases de datos es el uso de *consenso por quórum* o *votación* [AA92, Her87a, Her87b, JM90]. En esta aproximación, utilizada para bases de datos replicadas, cada operación que se intenta debe conseguir un determinado número de votos de las réplicas (*quórum*) para proceder. Normalmente, las operaciones se dividen en dos categorías: lecturas y escrituras. Cada réplica tiene otorgado cierto número de votos. La regla que suelen seguir la mayor parte de los algoritmos de gestión

de esta área es que el quórum de escritura debe ser superior a la mitad del número total de votos, mientras que el quórum de lectura puede ser inferior a dicha mitad. Aparte, la suma del quórum de lectura y el de escritura debe superar el número total de votos. De esta forma se logra tanto gestionar la replicación como el control de concurrencia asociado a las operaciones de consulta o actualización de la base de datos. Algunas mejoras sobre el modelo original están basadas en una variación dinámica de los quora en caso de fallo, o de la distribución de votos (en caso de mantener los quora fijos) [Her87b, KS93].

Otro modelo especial es el de *replicación perezosa* [LLSG92]. En este caso, cada objeto replicado tiene un *objeto de interfaz* o *front-end*. Para las peticiones de consulta de estado, el objeto de interfaz consulta una réplica y devuelve el resultado tan pronto como está disponible. Para las peticiones de actualización, el objeto de interfaz retorna de inmediato el control a su cliente y propaga posteriormente la actualización de manera perezosa. En este caso, se distinguen tres tipos de operaciones: *forzosas*, las que se sirven entre ellas en el mismo orden en todas las réplicas; *inmediatas*, las que se sirven respecto a cualquier otra operación en el mismo orden en todas las réplicas (orden total); *causales*, las que se sirven siguiendo orden causal. La principal aportación de esta variante de replicación es su eficiencia de servicio, pues el objeto de interfaz puede contestar las peticiones de los clientes de manera rápida, especialmente en el caso de las escrituras. Para las lecturas hay que comprobar qué consistencia espera la petición recibida y contestar a ella cuando la réplica que pueda utilizarse haya recibido todas las actualizaciones que se necesiten.

Si se examinan los trabajos realizados en esta área podrán encontrarse múltiples variantes de los dos modelos de replicación principales (activo y pasivo), más algún otro modelo intermedio como el coordinador-cohorte o las aproximaciones comentadas en esta sección. La elección de un modelo u otro depende de los servicios que deban proporcionarse. Para el caso del modelo coordinador-cohorte la ausencia de implementaciones nativas con este modelo puede deberse a la complejidad de los mecanismos auxiliares que garanticen la atomicidad, consistencia y aislamiento de las actualizaciones de estado. Esta tesis describe el soporte necesario para los mecanismos auxiliares que proporcionan estas garantías.

1.4.6 Arquitecturas de soporte a replicación

Centrándonos en el soporte a replicación, ha habido múltiples aproximaciones para implementar objetos replicados o procesos replicados en un sistema. Las de mayor relevancia son las siguientes:

A1 El sistema operativo distribuido proporciona directamente el soporte necesario. Esta aproximación ha sido seguida en los sistemas *V* [Che88] y *Amoeba* [TvRvS+90], que fueron desarrollados desde un principio incluyendo un *protocolo de comunicación entre grupos* [Che86, KT91] que entregaba las peticiones a las aplicaciones replicadas. Estos sistemas usaron el modelo de replicación activa, ya comentado anteriormente.

En este caso, el soporte para alta disponibilidad está incluido en el sistema operativo que se está usando. Todas las aplicaciones que pueden utilizar este soporte dependen del sistema sobre el que han sido desarrolladas; esto es, no podrán ser migradas fácilmente a otros sistemas operativos.

A2 Una *biblioteca* o *toolkit* facilita el soporte para objetos replicados, los cuales podrán ejecutarse sobre cierto conjunto de sistemas operativos. Existen múltiples ejemplos de este modelo: *Relacs* [BDM95], *Isis* [BvR94], *Arjuna* [PSWL95], *Phoenix* [MFSW95], *Horus* [vRBM96], ...

Normalmente, todos ellos facilitan un soporte a objetos replicados siguiendo el modelo de replicación activo, como ya ocurría en la aproximación A1.

A3 *Protocolos de transporte* que incluyen un servicio de pertenencia y que pueden ser integrados en los niveles de comunicación de un sistema operativo distribuido. Facilitan un servicio de difusión ordenado, que puede ser utilizado por el programador de aplicaciones para desarrollar componentes replicados. *Transis* [DM96] es un ejemplo de un protocolo de transporte de este tipo. Tolera *particiones* de la red. Es decir, que los nodos que forman el sistema se subdividan en varios grupos que permanezcan aislados, al menos temporalmente. *Totem* [MMSA+95] es otro ejemplo de protocolos de esta clase. La principal diferencia entre ellos es que mientras Transis estructura sus nodos de manera jerárquica a la hora de transmitir los mensajes, Totem usa un *anillo* lógico.

A4 Uso de *middleware* [Ber96]; i.e., un nivel de programa situado entre la aplicación y el sistema operativo, que facilita un conjunto de interfaces y protocolos que pueden ser usados para comunicación cliente-servidor independientemente del sistema operativo que se utilice en cada máquina. Un ejemplo de la aproximación middleware es CORBA [OMG99a]. Aunque actualmente no facilita ningún soporte para objetos replicados dentro de su arquitectura estándar, se está considerando la inclusión de un futuro *servicio CORBA* [OMG98a] *de replicación* [OMG98b, OMG99b].

El uso del estándar CORBA conlleva ciertas ventajas. Primero, no depende del sistema operativo que se utilice. Por tanto, podemos construir nuestras aplicaciones distribuidas sobre diferentes sistemas operativos. Segundo, facilita un modelo de programación orientado a objetos, mejorando la modularidad de las aplicaciones resultantes. Tercero, soporta componentes implementados en una gran variedad de lenguajes de programación (*C*, *C++*, *Java*, *Ada95*, *COBOL*, *Smalltalk*, ...). Y, finalmente, CORBA facilita interoperabilidad entre *ORBs* desarrollados por diferentes proveedores.

Sin embargo, sus principales inconvenientes son su ubicación en la arquitectura del sistema distribuido, que normalmente se encuentra fuera del sistema operativo (y que, por tanto, evita que pueda ser usado para incluir objetos replicados en el núcleo del sistema), y su falta de soporte estándar para objetos replicados.

También ha habido aproximaciones que han tratado de combinar el uso de bibliotecas de comunicación entre grupos con CORBA [Maf95]. Aunque es un buen principio, acarrea los inconvenientes presentes en las bibliotecas de comunicación entre grupos (principalmente, el coste de sus protocolos de difusión atómica ordenada).

A5 Modificaciones de un sistema operativo ya existente para incluir soporte a objetos replicados de algún modelo. En este caso, el desarrollo efectuado es altamente dependiente del sistema operativo que se está modificando. En la solución descrita en [BBG+89], las acciones realizadas por un proceso *UNIX* deben ser interceptadas y copiadas en una réplica secundaria

de este proceso (se emplea el modelo de replicación pasivo).

A6 El soporte para objetos replicados está integrado en un lenguaje de programación distribuido. Ejemplos de este tipo son *Argus* (que está basado en transacciones atómicas, pero algunas extensiones fueron realizadas en [Ghe90] para incluir soporte a objetos replicados) y *Drago* [MAAG96], una extensión de Ada95 que gestiona objetos replicados.

Esta aproximación garantiza que las aplicaciones desarrolladas usando estos lenguajes podrán funcionar en varios sistemas operativos. Así, las aplicaciones pueden ser fácilmente migradas a ellos.

A7 Utilizar una solución que toma algunas de las características de las aproximaciones A1 y A4. Esto conlleva adaptar un sistema operativo existente, integrando en su núcleo una capa de middleware con soporte integrado para replicación de objetos. Esto permitirá que se pueda adaptar posteriormente el propio núcleo del sistema incluyendo en sus componentes también objetos replicados que garanticen la alta disponibilidad del propio sistema distribuido. Esta solución fue adoptada en *Solaris MC* [BMK96] y es la que se va a adoptar también en *HIDRA* [GMB97b], objeto principal de esta tesis.

1.5 Contribuciones

Esta tesis describe parte de la arquitectura HIDRA [GMB97a, GMB97b] que proporciona soporte para objetos replicados extendiendo el núcleo de un sistema operativo con un ORB con soporte nativo para este tipo de objetos, lo que permitirá que algunas partes del propio núcleo puedan construirse también como objetos altamente disponibles.

El tipo de sistema donde se implantará la arquitectura HIDRA será un cluster de ordenadores. Como en este caso se pretende dar soporte a replicación de objetos e interesa detectar lo antes posible cualquier variación en la pertenencia de máquinas al sistema, se ha implantado un *protocolo de pertenencia a grupo* [MBG97, MMBG97, MGB00] que comprueba y notifica cualquier cambio que se produzca. Este protocolo es necesario para dirigir la reconfiguración del estado de los objetos replicados en caso de fallo.

Como veremos en el capítulo 3 actualmente ya se han dado muchas soluciones al problema del mantenimiento del conjunto de pertenencia a un grupo. En ellas se pueden distinguir dos fases principales: *formación* y *monitorización*. La fase de formación comprende todos los pasos que deben realizarse desde la detección de un cambio hasta el establecimiento de un nuevo conjunto de pertenencia, mientras que la de monitorización se centra en comprobar periódicamente el buen funcionamiento de los elementos que ya integran el grupo, así como de atender solicitudes de incorporación.

Nuestro monitor de pertenencia HMM tiene unos costes similares a los de los mejores monitores desarrollados en esta área, aunque parte con la ventaja de que su entorno de trabajo es ciertamente favorable (asumiremos que no se darán particiones de la red de interconexión, pues nuestro modelo de cluster tendrá una red donde no existirán pasarelas que intercomuniquen subredes). Su contribución principal radica en que la fase de formación no tiene un coste excesivamente alto y que tolera múltiples fallos simultáneos de los componentes que utilizan el protocolo. En otros algoritmos con costes similares, para tener una fase de formación económica se utilizan

"coordinadores" que dirigen al resto de nodos en caso de variación en la pertenencia. Nuestro protocolo también los utiliza. Sin embargo, en otros algoritmos existe el peligro de que el coordinador falle y entonces sea reemplazado por un suplente. Las cosas se complican demasiado cuando ese suplente también falla; entonces se suele recurrir a reiniciar la formación del grupo entero con una variante del protocolo de formación mucho más cara que la normal. Nuestro algoritmo no presenta ese problema. Se ha introducido el concepto de nodo iniciador que evita fases de formación más costosas en caso de múltiples fallos.

Una segunda aportación de nuestro protocolo de pertenencia es la inclusión de una tercera fase que se inicia en paralelo con la de monitorización y que sirve para notificar los cambios con una secuencia de pasos sincronizados a una lista de componentes registrados. Con ello se abaratan los costes de reconfiguración de nuestros componentes, pues esta sincronización es fácil de conseguir si está dirigida directamente por quien detecte los cambios. Esto es así porque resulta sencillo abortar una reconfiguración de nuestro sistema si durante ella se ha advertido algún otro cambio en el conjunto de máquinas que forman el cluster y que provocará a su vez una nueva reconfiguración.

Otra contribución de esta tesis radica en su soporte para el modelo de replicación coordinador-cohorte. Hasta ahora no ha habido ningún sistema donde este modelo de replicación se haya implantado de forma nativa. En la práctica, otros sistemas proporcionan soporte para este modelo (por ejemplo, Isis [BvR94]) pero lo hacen en una capa situada por encima del modelo de replicación activo. Es decir, para dar soporte al modelo coordinador-cohorte requieren tener implantado por debajo el modelo activo. Esto permite solucionar algunos problemas del modelo activo, pero puede acarrear costes adicionales para el modelo coordinador-cohorte que no habrá manera de eliminar (protocolos de difusión atómica, filtrados de envíos, filtrados de respuestas, etc.). Nuestra solución ha optado por implantar de manera directa este modelo de replicación. Para ello hemos necesitado dos mecanismos básicos: invocaciones fiables a objetos replicados, y control de concurrencia distribuido.

El mecanismo de invocaciones fiables a objeto que presentamos proporciona atomicidad, progreso, mantenimiento de resultados y consistencia. La *atomicidad* resulta necesaria para garantizar que una invocación ha logrado actualizar el estado de todas las réplicas del objeto. Este requerimiento, combinado con un adecuado control de concurrencia, permitirá garantizar la consistencia del estado de esas réplicas. En caso de fallo de alguna réplica o del propio objeto cliente durante una invocación, la propiedad de *progreso* indica que la invocación deberá proseguir, actualizando a las réplicas que queden disponibles. Si ha sido el cliente quien ha caído, y también era un objeto replicado, deberán mantenerse momentáneamente los resultados en las réplicas del objeto invocado. Cuando finalmente otra réplica del cliente repita la invocación, no deberá repetirse su servicio sino que serán localizados los *resultados retenidos* y devueltos al cliente. Cuando el cliente finalmente los obtenga, el mecanismo de invocaciones fiables deberá proceder a eliminar tales resultados retenidos.

Por lo que respecta al mecanismo de control de concurrencia, éste debe llevar un control sobre las operaciones que se invocan sobre las diferentes réplicas de un objeto, dejando proseguir a todas aquellas que no entren en conflicto mutuo. Asumiremos que dos operaciones están en conflicto cuando al menos una de ellas intenta modificar una parte del estado del objeto que es accedida y utilizada por ambas operaciones. Como existen múltiples réplicas del objeto y habrá también

múltiples clientes que podrán iniciar concurrentemente diferentes invocaciones, se ha implantado un mecanismo de control de concurrencia distribuido que tolera el fallo de cualquier componente que participe en el mecanismo.

Obviamente, tanto el mecanismos de invocaciones fiables como el de control de concurrencia son dos nuevas contribuciones realizadas en esta tesis, pues no ha habido hasta la fecha ninguna otra implementación directa del modelo de replicación coordinador-cohorte y estos mecanismos están completamente ligados a dicho modelo de replicación.

1.6 Estructura

El resto de este documento se estructura como sigue. En el capítulo 2 se describe en líneas generales la arquitectura HIDRA, empezando por sus objetivos: dar soporte a alta disponibilidad en un sistema en cluster, utilizando para ello un ORB con soporte nativo a objetos replicados que será incluido, en parte, dentro del núcleo de un sistema operativo de uso general, como por ejemplo *Linux*, o sobre un micronúcleo, como pueda ser *NanOS* [MB97, MGB99a]. Posteriormente se explican los diferentes niveles que componen esta arquitectura, al menos aquellos que tienen cierta relevancia para el soporte necesario a replicación: *transporte no fiable*, monitor de pertenencia, *transporte fiable* y ORB. Dentro de la sección 2.5 se describe en líneas generales el soporte que se ha necesitado para facilitar el modelo coordinador-cohorte y qué partes del ORB ha habido que modificar para ello.

El capítulo 3 está dedicado a la descripción de los monitores de pertenencia a grupo, que forman la base de la arquitectura HIDRA. En primer lugar se describen sus funciones principales y la aplicación de éstas para implantar o reforzar otros componentes del sistema: transporte fiable, *caídas forzosas* en caso de aislamiento, gestión de *protocolos de reconfiguración* de componentes, etc. A continuación se hace un estudio de los protocolos actualmente existentes, dependiendo del entorno y modelo de sistema distribuido en el que funcionan. Finalmente, se presenta el protocolo diseñado e implantado en HIDRA, llamado *HMM*, así como las posibles mejoras y extensiones que podrían introducirse.

El capítulo 4 se centra en los mecanismos que deben emplearse para realizar invocaciones sobre objetos replicados, particularmente para el caso de aquéllos que sigan el modelo coordinador-cohorte. En la primera sección se empezará analizando en detalle qué ocurre dentro de este modelo de replicación cuando se invoca un objeto, qué pasos se siguen y qué alternativas existen para ordenar tales pasos. Una vez descrito el funcionamiento general se estudiarán las garantías que debería ofrecer cualquier mecanismo de invocación que pretenda ser fiable. La sección 4.3 describe el mecanismo utilizado en HIDRA para garantizar la fiabilidad, atomicidad y progreso de las invocaciones a objetos replicados: el mecanismo de invocación fiable a objeto, o *protocolo IFO* (o *ROI*, en inglés). La próxima sección describe todos los posibles casos de fallo que pueden llegar a darse en este protocolo y cómo son solucionados éstos. Para terminar, se ofrece una última sección donde se compara el trabajo expuesto con otros protocolos empleados en otros sistemas.

El capítulo 5 trata sobre los mecanismos de control de concurrencia, así como de su aplicación al modelo de replicación coordinador-cohorte utilizado en nuestro sistema. Se empieza con una introducción general de los objetivos de estos mecanismos. Después se describen los mecanismos de control de concurrencia más comúnmente utilizados en entornos distribuidos: cerrojos, vota-

ciones, serialización, objetos protegidos, etc. Para comparar la calidad de diferentes mecanismos de control de concurrencia se presenta también el concepto de potencia expresiva. Dados estos conocimientos previos, se pasa seguidamente a describir el mecanismo utilizado en HIDRA, que recibe el nombre de *HCC*. Se presentan sus componentes básicos, objetos auxiliares necesarios, relación con el protocolo IFO, comportamiento en caso de fallos y una comparativa con otros mecanismos existentes.

Finalmente, en el capítulo 6 se presentan las conclusiones sobre el trabajo desarrollado en esta tesis. En primer lugar se da un breve repaso a todas las tareas completadas y las contribuciones que éstas conllevan. Para terminar se dan a conocer las futuras líneas de trabajo que van a derivarse de esta tesis.

Capítulo 2

La arquitectura HIDRA

2.1 Introducción

HIDRA [GMB97a, GMB97b] es una arquitectura que tiene como objetivo ofrecer soporte para objetos altamente disponibles (*objetos replicados*). Esto implica que el tipo de soporte que ofrecerá la arquitectura presupondrá un modelo de programación orientado a objetos y que existirán múltiples máquinas donde podrán ubicarse las múltiples réplicas de éstos. Para dar el soporte a un modelo de programación orientado a objetos en un entorno distribuido, como es el caso (pues las diferentes máquinas donde residan las réplicas de cada objeto, deberán estar interconectadas), una de las bases más asentadas la proporciona el estándar CORBA. Este estándar no depende del lenguaje de programación que se emplee para implantar los objetos a intercomunicar, soportando actualmente una gran variedad de ellos. Con ello queda claro que en nuestro soporte va a figurar un ORB para facilitar las herramientas de intercomunicación. Para dar un soporte mejor a los objetos replicados, convendría que éstos fueran reconocidos nativamente por el núcleo de este ORB y no que fueran gestionados como grupos de objetos nativos que pueden seguir siendo invocados individualmente, tal como ha ocurrido en la versión adoptada de la especificación para tolerancia a fallos dentro de *CORBA*, iniciada en 1998 [OMG98b, OMG99b] y que ha tenido su texto final durante el año 2000 [OMG00] aunque sobre dicha edición todavía se pueden realizar ampliaciones o revisiones. Con esto no se está haciendo una crítica a lo que marca el estándar CORBA, simplemente justificamos nuestra elección. En nuestro caso, la interoperbilidad con otros OO.RR.BB. no era importante, pero sí lo es el rendimiento del ORB, aunque se aparte de lo que dicte el estándar. Además, la especificación oficial contempla algunos modelos de replicación, pero no todos. Por ejemplo, el modelo coordinador-cohorte defendido en esta tesis no se soporta.

Para que todo funcione correctamente se necesitarán además otros componentes y servicios. Un ejemplo de ellos es el monitor de pertenencia que comprueba continuamente el funcionamiento de las máquinas del cluster e informa a los componentes preconfigurados sobre cualquier cambio que haya habido en el conjunto de nodos que integran ese cluster. Esta herramienta servirá para decidir en qué momento debe reconfigurarse el estado de un objeto replicado, bien porque alguna de sus réplicas ha fallado o bien porque existen nuevos nodos donde podrán ser ubicadas sus nuevas réplicas en caso de ser necesario. Aparte, nuestro ORB ofrece para ciertos tipos de sus objetos una gestión de *cuenta de referencias* que permite saber cuántos clientes pueden acceder a él y, en caso de no existir ninguno, notifica al propio objeto acerca de esa situación (generando una

notificación de no referencia) para que adopte las medidas oportunas. Normalmente, esto último conllevará la destrucción voluntaria del objeto replicado.

En este capítulo se van a describir todos estos componentes de la arquitectura HIDRA, así como la interrelación existente entre ellos. La sección 2.2 describe cada uno de los componentes que forman la arquitectura, o al menos aquellos que guardan relación directa con el soporte necesario para objetos replicados. En la sección 2.3 se presenta el modelo de fallos que va a utilizarse en los diferentes niveles de la arquitectura HIDRA y cómo se ha dado soporte a tales modelos. Posteriormente, la sección 2.4 describe qué es un ORB según el estándar CORBA y qué diferencias presenta nuestro componente de intercomunicación con lo establecido en CORBA. Por una parte, encontraremos algunos detalles del estándar que no están implantados en nuestro ORB como pueda ser el protocolo IIOP, el uso de un adaptador de objetos, el soporte a invocaciones dinámicas o el uso de interceptores para modificar la información enviada entre los núcleos. Por otra, nuestro ORB añade soporte especial que no está contemplado en el estándar, como los objetos replicados nativos o la búsqueda de basura mediante cuenta de referencias. La última sección se centra en el propio soporte como tal, estudiando qué partes del ORB han debido ser modificadas para incluir la gestión de los objetos replicados.

2.2 Visión general

Los componentes principales de la arquitectura HIDRA aparecen en la figura 2.1 y se describen en los apartados que siguen. Tal como se muestra en la figura, estos componentes deben ubicarse parcialmente en el núcleo del sistema operativo que se utilice como base, ya que uno de nuestros objetivos es que el soporte a objetos replicados pueda ser también usado para construir algunos servicios internos del sistema operativo. En cualquier caso, ese objetivo depende de los que forman el contenido principal de esta tesis y su consecución se deja como trabajo futuro.

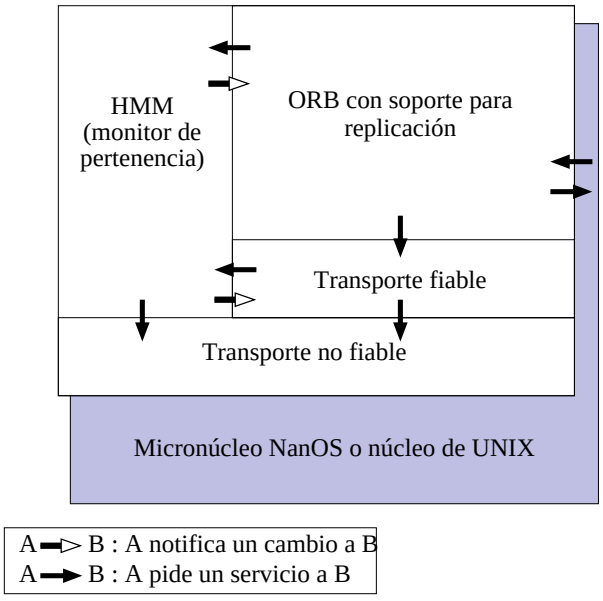

Figura 2.1: Componentes de la arquitectura HIDRA.

2.2.1 Transporte no fiable

En el nivel más bajo del soporte que necesita nuestra arquitectura de alta disponibilidad encontramos un *transporte no fiable*. Por tal elemento entendemos aquél que no establece ningún tipo de conexión ni efectúa ningún esfuerzo para conseguir que la información sea entregada en su destino.

El tipo de protocolo que habrá que emplear para implementar este transporte depende del tipo de interconexión interna que utilice el cluster.

En un cluster cerrado, la red de interconexión interna no puede ser accedida por ningún nodo ajeno al cluster. Suele ser algún tipo especial de red de altas prestaciones como las *SCI* (*Scalable Coherent Interconnect*), las *Myrinet* o las *Gigabit Ethernet*. En este caso, los nodos del cluster deben estar conectados tanto a la red interna como a una *red externa* que permita conectarles con nodos externos que puedan requerir sus servicios.

Por el contrario, en lo que puede definirse como un cluster expuesto, la red de interconexión interna está compartida con el resto de máquinas que haya en ese entorno. Es decir, tanto los nodos externos al cluster como los que pertenecen a él comparten una misma red.

Pues bien, si se está utilizando un cluster cerrado podrá implementarse un protocolo especial para este nivel no fiable, o adaptar el que facilite el sistema operativo para trabajar con la red interna. Ese esfuerzo es justificable en este caso puesto que con esta red interna especial se pretende reducir el tiempo de comunicación entre los diferentes nodos y tendrá sentido utilizar protocolos especiales para mejorar las prestaciones. Sin embargo, si se utiliza un cluster expuesto puede utilizarse tranquilamente el protocolo estándar no fiable de *ARPANET* (*UDP*/*IP*, o *User Datagram Protocol*/*Internet Protocol* [Pos80]), puesto que en ese caso las prestaciones que alcancemos dependerán más del tipo de red y de su carga que del protocolo empleado.

En nuestra arquitectura asumiremos que el tipo de cluster a emplear será cerrado, por lo que tendrá sentido emplear un protocolo especial para lograr este transporte no fiable.

2.2.2 Monitor de pertenencia

El *monitor de pertenencia* es el componente de HIDRA que comprueba qué máquinas están funcionando dentro del cluster. Esta información es importante puesto que en nuestro ORB se registra la ubicación de cada una de las réplicas de los diferentes objetos. En caso de caída de alguna máquina hay que modificar en ciertos casos las referencias a objeto para que se adapten a la nueva situación, hay que reconfigurar las invocaciones en marcha y hay que recalcular el número de referencias clientes que apuntan a estos objetos (Todos estos puntos se aclararán posteriormente en las secciones 2.4.3 y 2.5.2 y en el capítulo 4).

El monitor se basa en un fichero de configuración en el que están registradas todas las máquinas que podrán pertenecer al cluster y debe poder gestionar tanto la puesta en marcha de alguno de estos nodos como la caída de cualquiera de ellos. Una característica añadida a nuestro monitor y que no siempre está presente en los componentes de este tipo es la incorporación de un *protocolo* para guiar los pasos *de reconfiguración* de cada nodo. Este protocolo admite un número variable de pasos y garantiza que todos los nodos del cluster realicen a la vez cada uno de los pasos. Para ello, se espera a que todos los nodos activos vayan confirmando la terminación del paso actual antes de iniciar el paso siguiente. Existe un nodo especial más prioritario que controla la recepción de estas confirmaciones. Cuando finalmente todos los nodos confirman la finalización,

este coordinador envía un mensaje para que de nuevo todos a la vez empiecen el paso siguiente. Mientras la reconfiguración se está llevando a cabo, los monitores también ejecutan el protocolo normal de monitorización de estado, por lo que si se da una nueva caída o se añade una nueva máquina al cluster, la nueva situación se detecta de inmediato.

Puede que existan múltiples formas de detectar la caída o adición de nodos a un cluster, pero el uso de un protocolo distribuido es necesario por varias razones. Una alternativa habría sido simplemente confiar en los protocolos de transporte, esperando a que éstos sean incapaces de entregar cierto mensaje a un determinado nodo destino para declararlo inhábil. Esto plantea el problema de que inicialmente sólo el nodo que pretendía comunicarse con el que ha fallado ha detectado ese fallo. Esto conllevaría que el nodo detector podría reconfigurar su estado ante tal situación, pero los demás nodos no harían lo mismo hasta que intentasen comunicarse con él. El uso del protocolo de pertenencia permite que todos los nodos tomen las mismas decisiones consensuadamente y que reaccionen ante ellas de la misma manera. De esta forma se establece una buena base para asegurar la consistencia del estado del cluster y todas las aplicaciones y objetos que se estén ejecutando en él.

Si nuestro monitor de pertenencia asegura el consenso en la toma de decisiones sobre el conjunto de máquinas que componen el cluster, aparece otra situación bastante conveniente. Si el grupo ha decidido que una máquina ha fallado (y ésta realmente no lo ha hecho, pero su capacidad de trabajo actual no le permite responder de ninguna forma las nuevas peticiones), el protocolo de transporte fiable asume que la máquina ya no está disponible y ni siquiera se le intentará enviar nada mientras no dé ninguna señal de actividad. Es decir, que al tomar una decisión de exclusión, todos los nodos activos la acatan y el nodo en cuestión ha quedado realmente fuera del cluster. Si realmente no había fallado deberá intentar posteriormente reintegrarse en el cluster. Aquí la ventaja reside en que todos los nodos adoptan la misma decisión y se garantiza que todas las aplicaciones tendrán que adaptarse a la nueva situación. Si no existiera un protocolo de pertenencia, puede que algunas aplicaciones siguieran contando con el nodo mientras que otras no y se podría llegar a generar un error si estas aplicaciones interactuasen.

Una descripción detallada del protocolo de pertenencia puede encontrarse en la sección 3.4.

2.2.3 Transporte fiable

El único requerimiento que tendrá el protocolo de *transporte fiable* es que aquellos mensajes que hayan sido enviados sean siempre entregados a su destino, excepto cuando este último o el emisor hayan caído.

La ubicación de este componente en la arquitectura se emplaza junto al protocolo de transporte no fiable (puede que utilice parte de sus servicios, al menos en la implementación para un cluster cerrado, por ello en la figura 2.1 de la página 20 aparece sobre él). Si no se utiliza un protocolo estándar, se podrán usar los servicios proporcionados por el monitor de pertenencia, aprovechando sus informes para abortar la comunicación con las máquinas que se notifique que han fallado.

Los servicios del protocolo de transporte fiable son utilizados por el ORB, el mecanismo de intercomunicación que también facilitará soporte para objetos replicados y que se describe seguidamente.

2.2.4 ORB

Por último, el componente principal de la arquitectura va a ser un ORB con soporte para alta disponibilidad. En él se incluye toda la gestión de objetos replicados. La función estándar de un ORB, según la arquitectura CORBA, es la gestión de las invocaciones entre los diferentes objetos que se hallen en un sistema distribuido. Para ello se necesita un ORB en cada *dominio* a intercomunicar. Estos dominios pueden ser procesos o nodos enteros. La implementación del ORB puede realizarse de diferentes maneras, el estándar no obliga a utilizar ninguna de ellas en particular: procesos dedicados, bibliotecas, componentes de un sistema operativo, etc.

Para que la invocación de objetos pueda tener lugar se necesita, al igual que en el caso de una llamada remota a procedimiento, un *stub cliente* que ofrezca la misma interfaz que el objeto remoto que va a invocarse y que mantenga algún tipo de *referencia a objeto* que permita al núcleo del ORB localizar al objeto destino y encauzar la invocación hacia él. Por tanto, se precisa también algún mecanismo para que los dominios clientes puedan obtener referencias a los objetos ubicados en otros dominios. Normalmente esto se consigue gracias a un *servicio de nominación* [OMG98a, Capítulo 3] donde puede asociarse un nombre a un objeto al registrar una referencia a éste y se puede obtener después una referencia si se conoce tal nombre.

Una vez el ORB ha identificado al objeto destino y ha averiguado en qué nodo se encuentra, se envían algunos mensajes a tal nodo, donde su respectivo ORB recogerá la petición y la hará llegar a su *stub servidor* que en esta arquitectura recibe el nombre de *esqueleto*.

CORBA presenta la característica de que tanto el stub cliente como el esqueleto servidor pueden ser generados automáticamente utilizando un compilador de interfaces. Para ello, el programador ha debido declarar tales interfaces de objeto utilizando *IDL* (o *Interface Definition Language*) [OMG99a, Capítulo 3].

Nuestro ORB va a seguir todos los principios comentados en los párrafos anteriores, pero además incluirá servicios de *cómputo de referencias*. Esto permitirá que un objeto sepa cuando ya no va a ser invocado y pueda autoeliminarse en ese caso.

En la sección 2.4 se describe con mayor detalle qué es un ORB.

2.3 Modelo de fallos

Describamos qué tipos de fallos se han llegado a distinguir en el estudio de sistemas distribuidos, para ver qué modelos son los más convenientes para cada uno de los niveles de la arquitectura HIDRA. En [Sch93b], se citan como posibles *modelos de fallos* los siete siguientes:

1. *Fallo parada* [SS83]. Un procesador falla parando. Una vez ha parado, el procesador permanecerá así. El hecho de que un procesador haya fallado será detectable para el resto de procesadores.

2. *Caída*. Un procesador falla parando. Una vez ha parado, el procesador permanecerá así. El hecho de que un procesador haya fallado puede que no sea detectable para el resto de procesadores.

3. *Caída y enlace*. Un procesador falla parando. Una vez ha parado, el procesador permanecerá así. Un enlace falla perdiendo mensajes, pero no retarda, duplica ni corrompe mensajes.

4. *Omisión de recepciones*. Un procesador falla recibiendo sólo un subconjunto de los mensajes dirigidos hacia él o parando y permaneciendo parado.

5. *Omisión de envíos*. Un procesador falla transmitiendo sólo un subconjunto de los mensajes que intentaba enviar o parando y permaneciendo parado.

6. *Omisión general*. Un procesador falla recibiendo sólo un subconjunto de los mensajes dirigidos hacia él o transmitiendo sólo un subconjunto de los mensajes que intentaba enviar o parando y permaneciendo parado.

7. *Fallo bizantino*. Un procesador falla exhibiendo un comportamiento arbitrario.

La numeración seguida en la lista anterior también sirve para graduar la *severidad* de cada modelo de fallos (Excepto para los modelos 4 y 5, que pueden considerarse con el mismo grado de severidad). Cuanto menor sea el número, menor será la severidad de tal modelo. Es decir, menor será el conjunto de fallos que podrá admitir dicho modelo. Por tanto, el modelo en el que se pueden dar los fallos menos graves es el primero que hemos citado y debería ser el proporcionado al ORB de nuestra arquitectura. Pero para conseguir esto habrá que utilizar algunos servicios de bajo nivel que lo hagan posible. Veamos cuáles son y en qué niveles de nuestra arquitectura se encuentran.

Para ello será necesario en primer lugar detallar el modelo de sistema que se está utilizando como base. Este modelo corresponde a un sistema distribuido parcialmente síncrono que reúne las siguientes características:

- Los relojes físicos de los diferentes nodos no estarán sincronizados. Tampoco se conoce la cota superior en la deriva que tengan los relojes debido a las posibles diferencias en su cadencia de actualización.

- Se asumirá que todos los nodos se encuentran en una misma red local, sin necesidad de pasarelas ni puentes para lograr la interconexión. Se asumirá también que no podrá haber particiones en dicha red.

  Esto se explica en parte por asumir que los nodos de nuestro cluster utilizarán una red interna privada de altas prestaciones y en dicha red no tiene sentido que aparezcan estos elementos. Esta suposición es empleada por el protocolo de pertenencia ya que en este entorno las soluciones que deberá proporcionar son algo más sencillas que en un entorno que admita particiones.

  Sin embargo, no resultaría excesivamente difícil adaptar nuestro monitor de pertenencia para que pudiese funcionar en un entorno particionable, ofreciendo garantías similares a las proporcionadas actualmente. Para ello bastaría con seguir un modelo de tratamiento de particiones basado en partición primaria (se explicará en la sección 3.3.1, página 40), tal y como ya se hizo en un monitor de pertenencia anterior [MMBG97].

- El tiempo máximo de transmisión de un mensaje vendrá determinado por las características físicas de la red y por los protocolos utilizados para realizar el transporte. Ambas cosas serán conocidas en nuestro entorno, por lo que el tiempo máximo de transmisión también podrá conocerse (en la práctica, esto nos permitirá decir que está acotado).

En principio, el modelo de fallos con el que debería trabajar el protocolo de transporte no fiable que constituye el nivel más bajo de la arquitectura HIDRA correspondería al modelo de omisión general, ya que los protocolos de comunicación empleados en este nivel no van a garantizar para nada ni la entrega ni la emisión de los mensajes que hayan solicitado los procesos.

Sin embargo, sobre el nivel de transporte no fiable tenemos un monitor de pertenencia que ofrecerá algunas garantías más y que servirá como base para implantar un protocolo de transporte fiable donde se eliminarán tanto los fallos de omisión como los fallos de recepción como los fallos de enlace.

Con ello, sobre el nivel de transporte fiable ya tendríamos un modelo de fallos que bien sería el modelo de caída o bien el modelo de fallo parada. El propio monitor de pertenencia nos asegura que el modelo resultante para el ORB sea el modelo de fallo parada. Para ello se toman las siguientes medidas:

- La misión principal de este monitor es detectar las caídas e incorporaciones de nodos al cluster, notificando todas las variaciones a los nodos que prosigan en él. Con ello puede asumirse que cualquier caída será detectable por todos los nodos, con lo que pasaríamos del modelo de caída al modelo de fallo parada, al menos por lo que respecta a la detección. Ahora lo que falta ver es si puede conseguirse que los procesadores que fallen realmente paren y se queden parados.

- En la práctica los nodos podrán recuperarse tras haber fallado, violando de esta forma uno de los principios del modelo de fallo parada: aquellos procesadores que fallen, permanecerán siempre así.

  Sin embargo, para identificar a un procesador nuestro sistema utiliza *identificadores por encarnación*. Para construir estos identificadores se añade al *identificador fijo* de cada nodo, que está preconfigurado en cierto fichero, un sufijo que incluye el número de encarnación del nodo. Este número de encarnación se incrementa cada vez que el nodo solicita su inclusión en el conjunto de pertenencia del cluster.

- Cuando el monitor de pertenencia supone que un nodo A ha fallado y todos los nodos del cluster llegan a ese acuerdo, tal *nodo* se declara *fallido* y ningún nodo del sistema aceptará ya más mensajes de ese nodo A. Por tanto, aunque realmente no haya fallado, para volver a integrarse en el cluster y poder establecer nuevas comunicaciones con el resto de sus nodos, el nodo A deberá solicitar su inclusión en el cluster.

  Por lo dicho en el punto anterior, cuando ese nodo A se reintegre pasará a tener un identificador diferente, con lo que a efectos del modelo de fallos será visto como un nodo diferente. Además, el nodo que se asumió que había fallado ya no podrá integrarse nunca más en el cluster y a efectos del modelo de fallos corresponderá a un nodo parado que permanecerá indefinidamente en dicho estado.

Por tanto, puede verse como gracias al monitor de pertenencia y al protocolo de transporte fiable se logra que el ORB de la arquitectura HIDRA trabaje con el modelo de fallo parada, garantizándose que no habrá fallos en los enlaces de comunicación y que todos los nodos que fallen lo hagan parando.

Esta afirmación que acabamos de hacer en el párrafo anterior no contradice en ningún momento los resultados de imposibilidad para los servicios de pertenencia dados en [CHTCB96]. Veamos por qué. Según [CHTCB96] el problema de la pertenencia a grupo es irresoluble en un sistema distribuido asíncrono, incluso si se utiliza el modelo de particiones basado en partición primaria, no hay fallos en el transporte, se permite el rechazo de los nodos sospechosos y los nodos sólo fallan por caída. Para ello, los autores de dicho artículo establecen una equivalencia entre el problema de pertenencia a grupo y el problema de consenso, que ya fue demostrado como imposible de solucionar en dicho entorno.

Sin embargo, el modelo de sistema empleado en HIDRA no presenta las mismas características que el adoptado en el desarrollo teórico de [CHTCB96]. Existe una diferencia que hace que nuestro sistema no sea tan general como el mencionado y que permite que sí sea solucionable el problema de pertenencia. En nuestro caso el sistema no es totalmente asíncrono porque nuestro transporte sí tiene un tiempo máximo de transmisión y en base a eso se puede lograr la sincronía necesaria entre los diferentes nodos como para poder adoptar decisiones en cuanto a la pérdida de mensajes. Esa acotación del tiempo de transmisión viene determinada por el tipo de red de altas prestaciones empleado, del que se conocerá el tiempo máximo de retardo de transmisión. Además, también conoceremos la implementación del protocolo de transporte no fiable (y del fiable también, aunque aquí no es relevante). Con ello se puede determinar con exactitud el tiempo máximo de transmisión de los mensajes, que estará en función del retardo de transmisión y del número de reintentos que utilice el protocolo. Por tanto, sí será posible decidir en nuestro entorno cuándo se ha perdido un mensaje. Aparte, los fallos de los enlaces podrán ser descartados (es decir, poco nos importará que haya fallado el enlace o el nodo que use el enlace, en la práctica para nuestro algoritmo será un fallo del nodo), con lo que nuestro algoritmo de pertenencia sí proporcionará unos resultados que serán aceptables en el sistema que hemos supuesto.

2.4 Intercomunicación

Como ya se ha comentado en la introducción, el objetivo principal de esta arquitectura es ofrecer soporte para objetos replicados. Dadas sus ventajas a la hora de efectuar un reparto de carga y de aprovechar convenientemente los nodos del sistema, se ha optado por utilizar el modelo de replicación coordinador-cohorte.

Dadas las características ofrecidas por el estándar CORBA, se ha optado por utilizar un ORB para proporcionar los servicios de intercomunicación entre los diferentes nodos que compongan el sistema. A consecuencia de ello, cualquier comunicación entre las réplicas del objeto o entre sus clientes y estas réplicas deberá efectuarse a través del ORB. No obstante, nuestro ORB no es completamente estándar, ya que no cumple estrictamente con la definición del protocolo de intercomunicación *IIOP* (*Internet Inter-ORB Protocol*) que define el formato de los mensajes a intercambiar, ni con la restricción de que el núcleo de este componente no debe dar soporte directo a objetos replicados.

Vamos a centrar así el contenido de esta sección en la descripción de qué debería ser un ORB y qué diferencias plantea nuestro modelo de intercomunicación con el propuesto en el estándar CORBA.

2.4.1 Los OO.RR.BB. según el estándar CORBA

Si se sigue el estándar CORBA, las componentes relacionadas con la invocación de objetos que podrán encontrarse en esta arquitectura aparecen en la figura 2.2. En ella se muestran dichos componentes mediante rectángulos rellenos, mientras que los usuarios de sus servicios aparecen con rectángulos con esquinas redondeadas.

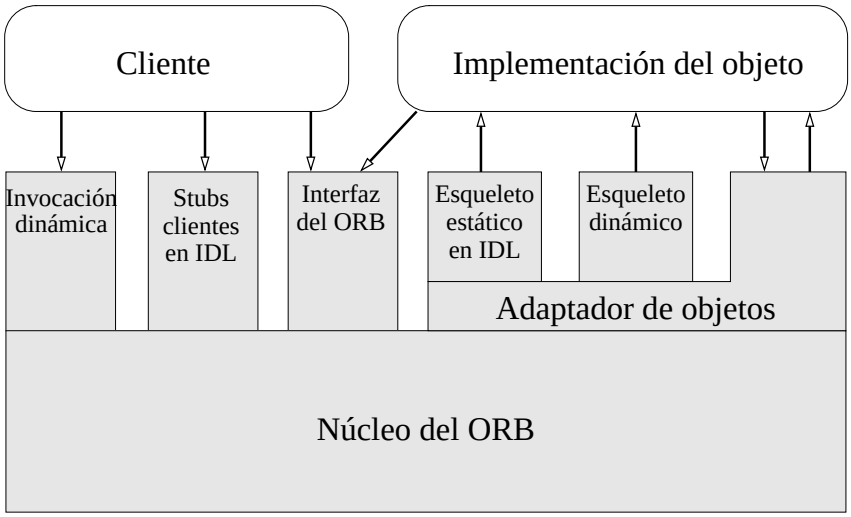

Figura 2.2: Componentes de un ORB.

Veamos para qué sirve cada uno de estos componentes:

- ***Núcleo del ORB***. Este componente es el que intercomunica los dominios cliente y servidor. Para ello recibe las peticiones efectuadas por los stubs clientes, consulta la referencia a objeto que estos guardan y en base a ella averigua dónde se encuentra el objeto a invocar. Utilizando el protocolo IIOP, se encarga de hacer llegar la petición al nodo donde se encuentre el dominio del objeto invocado. Una vez allí pasa la invocación al adaptador de objetos, que localizará el esqueleto correspondiente.

  Tal como se muestra en la figura, el núcleo del ORB no es directamente accesible. Para usar sus servicios deben utilizarse otros componentes que interactúen con él como la interfaz de invocación dinámica, los stubs clientes y la interfaz del ORB en el dominio cliente y el esqueleto servidor, el esqueleto dinámico, el adaptador de objetos y la interfaz del ORB en el dominio donde resida la implementación del objeto.

  El estándar también define la presencia de *interceptores* [OMG99a, Capítulo 18] como un mecanismo que permite modificar el tratamiento que da el núcleo del ORB a los mensajes que debe intercambiar entre diferentes dominios. Para ello se definen dos tipos de interceptores. El primero es el *interceptor de peticiones* mediante el cual se pueden modificar algunos atributos que definen una petición CORBA (es decir, la información que transita entre los diferentes núcleos de ORB para representar una invocación a objeto). Estas modificaciones pueden realizarse tanto en el dominio cliente como en el servidor, pero en el nivel del núcleo del ORB. El segundo tipo corresponde a los *interceptores de mensajes*. Mediante ellos se pueden cambiar los contenidos de los mensajes del protocolo IIOP que son inter-

cambiados entre los núcleos. Estas modificaciones podrán consistir en añadir o eliminar parte del contenido, añadir referencias a objeto, filtrarlas, cambiarlas, etc.

- ***Interfaz del ORB***. El ORB ofrece una interfaz pública estándar que puede utilizarse tanto en los programas que compongan al cliente como al objeto implementado (es decir, en el dominio servidor del objeto) para acceder a sus servicios. Mediante esta interfaz se puede obtener la lista de servicios ofrecidos por el ORB (como mínimo, debería existir un servicio de nominación, cuyo servidor podrá usarse para registrar objetos u obtener sus referencias), obtener una referencia a algún adaptador de objetos que permita registrar objetos en él, inicializar el ORB, convertir referencias a cadenas, convertir cadenas a referencias, etc.

- ***Stub cliente***. El stub cliente es generado automáticamente por un *compilador de interfaces*. Para ello el programador ha debido escribir un fichero de interfaz en el que se especifique, siguiendo la sintaxis del lenguaje IDL, qué operaciones ofrece el objeto, qué argumentos tiene cada una de ellas, qué *excepciones* puede generar cuando se invoque cada operación y de qué otras interfaces hereda la que se está especificando.

  La misión del stub cliente, al igual que ocurrió en la descripción de las llamadas remotas a procedimiento, es ofrecer al programa cliente la misma interfaz del objeto remoto que se está invocando y llevar a cabo el aplanado y desaplanado de argumentos y resultados para facilitar la construcción de los mensajes que transmitan la petición y la respuesta entre los dominios cliente y servidor.

- **Interfaz de *invocación dinámica***. La generación de stubs clientes debe realizarse, como hemos comentado previamente, utilizando un compilador de interfaces. Una vez obtenido el código del stub cliente siguiendo ese paso, ya se puede utilizar para implementar el código del programa cliente que nos interese.

  En algunos casos esto no podrá utilizarse, bien porque desconocemos qué tipo de objeto deberá invocarse y por tanto no hemos tenido acceso al código de su stub, o bien porque no nos interesaba incluir dicho código y aumentar de esa forma el tamaño del ejecutable del programa cliente. Para gestionar estas situaciones en las que no está disponible el stub cliente o no nos interesa utilizarlo, CORBA define una interfaz estándar de invocación dinámica. Mediante esta interfaz podrá invocarse, en tiempo de ejecución, cualquier método de cualquier objeto. Para ello, los únicos requerimientos serán haber obtenido una referencia al objeto a invocar, conocer el nombre de la operación a invocar y conocer la signatura de dicha operación. Con esta información, y sin necesidad de ningún stub, se construirá dinámicamente la petición para que el ORB la haga llegar a su destino.

  Toda la información que hemos dicho que resulta necesaria en este punto puede obtenerse de un *repositorio de interfaces*.

- ***Adaptador de objetos***. El adaptador de objetos es el componente de la arquitectura CORBA que maneja el registro de implementaciones de objetos en el ORB. Es decir, asocia objetos CORBA con el programa o el código que los implementa. También se utiliza para, una vez identificado el destino de una invocación, activar al código servidor de éste en caso de que todavía no hubiese sido utilizado (si el ORB admite *activación dinámica*).

Guarda fuerte relación con los esqueletos, pues estos últimos son los componentes que encauzan la invocación hasta el código que realmente implementa al objeto. De hecho, el adaptador identifica al esqueleto correspondiente y transfiere a éste todo el contexto de la invocación en curso.

Otras tareas que debe realizar el adaptador se centran en la gestión de los *hilos de ejecución* necesarios para atender las invocaciones que vayan llegando desde otros dominios.

- ***Esqueleto estático***. El esqueleto estático también se obtiene como resultado de la compilación del fichero con la interfaz IDL del objeto. Es el equivalente a un stub servidor en el campo de las invocaciones a objeto. Por tanto, es el componente que atendiendo a la identidad de la operación invocada seleccionará el código que debe ejecutarse, realizando previamente el desaplanado de la información recibida como argumentos y completando posteriormente el aplanado de los resultados y argumentos de salida que deban devolverse al cliente.

- ***Esqueleto dinámico***. Al igual que ocurría con los stubs clientes, en el dominio servidor existe una variante dinámica del esqueleto que normalmente genera el compilador de interfaces. Esta variante recibe el nombre de esqueleto dinámico.

  La implementación del esqueleto dinámico depende en gran medida del adaptador de objetos que se utilice. Por tanto, no existen mecanismos estándar para construirlos. Sin embargo, la interfaz que ofrece sí es estándar.

Los usuarios de todos estos componentes serán los dominios cliente y servidor, que no tienen por qué encontrarse en la misma máquina. Estos dominios utilizan las interfaces ofrecidas por los componentes descritos, según lo ya visto en la figura 2.2. Hay que hacer notar que las únicas interfaces estándar son las ofrecidas por: invocación dinámica, esqueletos dinámicos e interfaz del ORB. El resto de interfaces (las del adaptador de objetos y las de los stubs y esqueletos estáticos) dependen del implementador del ORB. El estándar CORBA no exige que un ORB ofrezca servicios de invocación y esqueletos dinámicos.

2.4.2 Elementos ausentes en nuestro ORB

El objetivo principal a la hora de utilizar un ORB en nuestra arquitectura no era obtener interoperabilidad con otros OO.RR.BB., sino utilizarlo como mecanismo de intercomunicación interno para dar soporte a objetos replicados cuyas réplicas únicamente podrán encontrarse en los nodos del cluster en el que se haya instalado HIDRA. Por tanto, no se han tenido en cuenta algunos detalles del estándar CORBA que hubiesen complicado innecesariamente el resultado obtenido.

De esta manera, los mensajes que intercambia el núcleo del ORB cuando debe hacer llegar una petición o respuesta a un nodo o dominio remoto, no siguen estrictamente el protocolo IIOP marcado por el estándar. En cualquier caso, esto no es tan grave como pueda parecer en un principio, pues aunque se siga el protocolo IIOP también tiene que resolverse el problema de convertir las referencias a objeto a un formato que sea comprensible por los diferentes OO.RR.BB. a intercomunicar. Esto no es nada sencillo y en nuestro caso también debería resolverse si se quisiera proporcionar acceso a clientes que utilicen un ORB diferente al nuestro. No hay soluciones generales para ello.

Otra parte del estándar CORBA que no ha sido necesario implantar en nuestro ORB es la relacionada con invocaciones dinámicas, tanto en el dominio cliente (interfaz de invocación dinámica) como en el servidor (esqueletos dinámicos). Esto exigía la implementación y gestión de un repositorio de interfaces. Aunque la posibilidad de realizar invocaciones dinámicas es atractiva, en nuestro sistema no compensaba el esfuerzo necesario para introducir tal soporte.

2.4.3 Elementos añadidos en nuestro ORB

Aparte de los componentes requeridos por el estándar CORBA, nuestro ORB incluye soporte directo en su núcleo para objetos replicados siguiendo el modelo coordinador-cohorte y un protocolo de cuenta de referencias clientes que permite detectar cuándo un objeto ya no puede ser accedido.

El soporte para replicación se explicará en detalle en la sección 2.5; veamos ahora para qué resulta necesario un *protocolo de cuenta de referencias*.

El objetivo principal de la cuenta de referencias es averiguar cuándo han dejado de haber referencias para un objeto determinado dentro del sistema. Cuando esto suceda, ese objeto ya no podrá ser invocado por nadie, pues ya no existen clientes que puedan invocarle y ni siquiera existe una referencia a él en un servidor de nombres desde donde otros dominios puedan obtener esas referencias y utilizarlas. En nuestro caso, esta cuenta de referencias es mantenida y gestionada por las distintas copias del núcleo del ORB que existen en cada uno de los nodos del cluster donde se ha instalado HIDRA. Cuando la cuenta para un objeto (ya sea replicado o no) llega a cero, el núcleo realiza una *notificación de no referencia* a tal objeto. Normalmente este objeto decidirá liberar todos los recursos que le han sido asignados (memoria, ficheros, cerrojos, etc.) y autodestruirse.

Esta técnica se encuentra dentro del área de las soluciones al problema de la *recolección de basura*; es decir, a la búsqueda de componentes software que ya no son utilizables porque no hay referencias que permitan acceder a ellos. Esto es especialmente importante en un sistema altamente disponible, pues en este ámbito los objetos suelen tener una duración prolongada y pueden acaparar bastantes recursos. Sin el uso de estos mecanismos, resultaría complicado saber cuándo pueden liberarse estos recursos y cuándo pueden destruirse los objetos.

2.5 Soporte para replicación

En el caso de nuestra arquitectura HIDRA, el soporte necesario para gestionar objetos replicados está incluido en el propio núcleo del ORB. Gracias a esto, los objetos replicados son soportados directamente y sus clientes únicamente deben mantener una referencia para poder invocar tales objetos. El modelo de replicación al que se ha ofrecido soporte en HIDRA es el coordinador-cohorte por presentar excelentes propiedades para repartir la carga y no aumentarla en exceso, puesto que solamente una réplica procesa cada una de las peticiones efectuadas sobre el objeto y pueden atenderse múltiples peticiones simultáneamente y en diferentes réplicas.

En esta sección se describirá en primer lugar el funcionamiento de este modelo de replicación, pasando a continuación a describir la gestión de referencias que efectuamos y los problemas adicionales que habrá que resolver y que constituyen el núcleo de esta tesis.

2.5.1 Modelo coordinador-cohorte

Como ya se comentó en la sección 1.4.4, este modelo de replicación es intermedio a los modelos activo y pasivo. Al igual que en el pasivo, sólo una réplica (llamada réplica coordinadora) atiende cada petición, efectuando checkpoints sobre el resto de réplicas (llamadas cohortes) antes de que se devuelva el resultado al cliente. De manera similar a lo que ocurre en el modelo activo, no es necesaria una reconfiguración del rol de cada réplica en caso de que caiga alguna de ellas, ya que todas las réplicas son capaces de trabajar adoptando uno cualquiera de los roles. De hecho, el rol tomado depende de qué réplica elija cada cliente como coordinadora de sus peticiones.

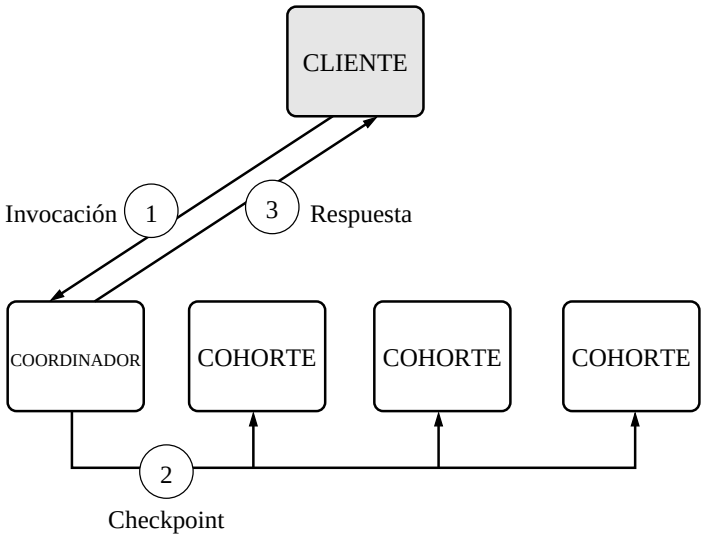

Figura 2.3: Pasos a seguir en una invocación del modelo coordinador-cohorte.

En la figura 2.3 se muestra el papel de cada réplica y los pasos seguidos para atender una invocación. En primer lugar (paso 1 en la figura), el cliente elige una réplica cualquiera del objeto a invocar y realiza sobre ella la invocación. Ésta pasa a ser la réplica coordinadora para esa petición. Esta réplica coordinadora procesa localmente la petición efectuada y poco antes de retornar el resultado al cliente realiza un checkpoint sobre las réplicas cohortes para que éstas actualicen su estado (paso 2 en la figura). Una vez ya ha finalizado el checkpoint, la réplica coordinadora devuelve el resultado a su cliente (paso 3 de la figura) y la invocación termina.

Las operaciones que acabamos de describir son las que actualizan el estado del objeto replicado que se ha invocado y exigen el retorno de algún resultado. No obstante, existen dos tipos de operaciones más simples: las que únicamente consultan el estado y las que no requieren ningún resultado (unidireccionales).

En el caso de las *operaciones de consulta* el cliente no quiere modificar el estado del objeto replicado, sino únicamente obtener información derivada de él. Por ello, en estas invocaciones el paso 2 correspondiente a los checkpoints no va a resultar necesario.

En el caso de las *operaciones unidireccionales* (u *operaciones "oneway"*, si seguimos la terminología CORBA), el cliente no espera ningún resultado ni argumento de salida de la invocación realizada, por lo que no esperará respuesta por parte del servidor invocado. En estos casos, la simplificación consiste en que no va a ser necesario realizar el paso 3 que correspondía al envío de la respuesta desde el coordinador al cliente. El checkpoint sí va a ser necesario, pues lo único

que se realiza en una operación unidireccional es algún tipo de actualización del estado.

2.5.2 Gestión de referencias

La gestión de referencias a objetos realizada en HIDRA se describe en detalle en [GMB99b, GMB99a]. El núcleo del ORB utilizado en HIDRA se ha estructurado en un par de niveles. El nivel superior proporciona los servicios necesarios para efectuar el aplanado y desaplanado de referencias a objeto, mientras que el nivel inferior es el que se encarga de la creación de referencias y *puntos de entrada* (dirección y puerto al que apuntará la referencia y hacia donde se dirigirán las invocaciones), asociación de éstas a objetos, mantenimiento del número de referencias clientes ligadas a un objeto y localización de los objetos en el sistema.

Para la gestión de objetos, basta con que el punto de entrada mantenga un *identificador de objeto* (u *OID*). El OID está construido de manera que permite localizar fácilmente al objeto. Integra el identificador de nodo, el número de encarnación del cluster (el número de veces que éste ha sido reconfigurado) en el que fue creado el objeto, el número de puerto donde atenderá invocaciones y un número de versión.

Para los objetos replicados debería existir un punto de entrada por cada réplica, pero un solo tipo de referencia cliente que apunte a tal objeto. Para solucionar esta cuestión, se introduce el concepto de *punto de entrada principal* que es el que llevará el control de la cuenta de referencias clientes del objeto replicado y el que normalmente estará apuntado por el *localizador* incluido en estas referencias (Al menos para el modelo de replicación pasiva).

Para llevar el control del número de referencias clientes que existen sobre un determinado objeto, tanto su punto de entrada como sus referencias clientes mantienen un contador.

El objetivo principal del protocolo de cuenta de referencias es averiguar en qué momento ya no existen referencias clientes sobre un determinado objeto. En el caso de los objetos replicados debe tenerse en cuenta que para crear una nueva réplica debe tranferirse a ese nuevo dominio una referencia cliente. En base a esa referencia cliente, nuestro soporte permitirá la creación de la nueva réplica, pero la referencia cliente seguirá existiendo asociada a la nueva réplica (hasta que explícitamente sea descartada). Una descripción detallada del protocolo empleado puede encontrarse en [GMB99b].

2.5.3 Problemas a resolver

Centrándonos de nuevo en el soporte al modelo coordinador-cohorte, se aprecian en él ciertos problemas que deben comentarse y resolverse. Todos ellos aparecen debido a la posibilidad de que múltiples clientes realicen simultáneamente diversas peticiones y escojan réplicas coordinadoras distintas. Esta situación se muestra esquemáticamente en el figura 2.4.

Como puede observarse en la figura, cada uno de los dos clientes ha elegido a un coordinador distinto. Y ambos coordinadores procesan la petición independientemente. El primer problema que puede plantearse depende del tipo de operaciones que hayan solicitado los clientes. Si alguna de ellas actualiza el estado del objeto, entonces habrá problemas pues la otra no sabrá nada sobre esa actualización y puede que devuelva resultados inconsistentes a su cliente.

Por tanto, hay que distinguir en primer lugar entre las *operaciones en conflicto* y las operaciones que puedan ejecutarse concurrentemente. Dos operaciones estarán en *conflicto* si ambas

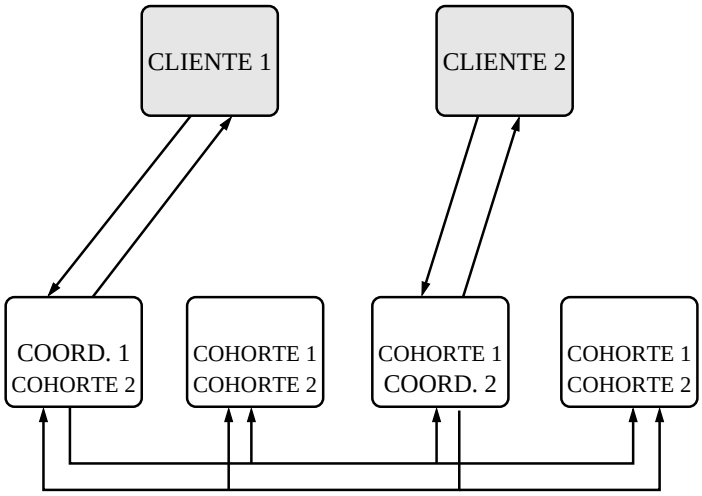

Figura 2.4: Peticiones concurrentes en el modelo coordinador-cohorte.

acceden a una misma parte del estado del objeto y al menos una de ellas modifica dicho estado. De aquí surge la necesidad de evitar que aquellas operaciones que estén en conflicto puedan ejecutarse simultáneamente en diferentes réplicas del objeto. Como resultado, será necesario introducir algún mecanismo de control de concurrencia distribuido para evitar estas situaciones. Este mecanismo se llama *HCC* y se describe en el capítulo 5.

Sin embargo, no basta con tener sólo un mecanismo de control de concurrencia. Para evitar conflictos debemos asegurarnos de que una invocación ha actualizado por completo todas las réplicas antes de darla por terminada y dar paso a alguna de las que estaba en conflicto con ella. En caso contrario, el control de concurrencia no serviría para nada. Por ello, hay que introducir un mecanismo de invocación de objetos replicados que asegure lo siguiente:

- La identificación de cada invocación realizada. De esta manera se podrán detectar reintentos de una misma invocación y proporcionar un servicio más rápido para éstos. Estos reintentos podrán darse en caso de fallo de la réplica coordinadora o de la réplica cliente que inició la invocación.

- La detección del instante en que todas las réplicas han sido actualizadas, para dejar paso a las siguientes invocaciones en conflicto, según dictamine el mecanismo de control de concurrencia.

- Los resultados que va a proporcionar la invocación se pasarán también en los checkpoints para que los mantengan temporalmente las réplicas cohortes. A esta copia de los resultados se la llama *resultados retenidos*. De esta manera, si la réplica coordinadora falla entre el instante en que se realiza el checkpoint y el momento en que se envía el mensaje de respuesta al cliente, el cliente puede reintentar la invocación sobre otra réplica y obtener de inmediato tal resultado.

- La detección del instante en que los resultados de la invocación hayan llegado al cliente. De esta manera se sabrá cuándo podrán liberarse los resultados retenidos por parte de las réplicas del objeto invocado.

Para proporcionar todas estas garantías, se ha implantado un mecanismo de invocaciones fiables a objetos que se explicará en el capítulo 4.

2.6 Conclusiones

La arquitectura HIDRA define una serie de componentes que serán necesarios para proporcionar soporte a objetos replicados en un sistema distribuido. Algunos de estos componentes podrán incluirse dentro del núcleo del sistema operativo, con lo que podrían utilizarse posteriormente para extender dicho núcleo con el objetivo de dar soporte a un sistema con imagen única, tendencia que se está teniendo cierta aceptación actualmente para los sistemas en cluster.

Otras arquitecturas con soporte a replicación ya han sido citadas de forma muy general en la sección 1.4.6 por lo que no vamos a repetir dicho estudio comparativo aquí.

HIDRA podrá ofrecer soporte a varios modelos de replicación, pero los componentes descritos en esta tesis se centran únicamente en el modelo de replicación coordinador-cohorte. En este modelo se necesitará implantar un mecanismo de invocaciones fiables, que garantice la atomicidad y consistencia de éstas, y un mecanismo de control de concurrencia. Así mismo, cualquier modelo de replicación es conveniente que tenga por debajo un monitor de pertenencia que de manera rápida pueda informar al soporte de replicación acerca de los cambios que haya habido en la configuración del sistema: caídas de nodos debidas a fallos, o bien incorporación de nuevos nodos. Estos tres componentes se describirán en los próximos capítulos.

Capítulo 3

Monitor de pertenencia

3.1 Introducción

Un *monitor de pertenencia* es el componente encargado de controlar qué procesos o nodos están activos en un momento determinado dentro de cierto sistema distribuido. Para el caso de HIDRA, donde se busca dar soporte a objetos replicados, el monitor de pertenencia es particularmente importante porque será utilizado para dar un modelo de fallos con el que sea sencillo implantar el resto del sistema y para proporcionar un soporte que coordine la reconfiguración del sistema en caso de variación dentro del conjunto actual de pertenencia. Obviamente, su función principal consiste en monitorizar el estado de los nodos que forman parte del sistema, tanto para detectar fallos como para aceptar la inclusión de aquellas máquinas que arranquen cuando el sistema ya está en funcionamiento.

La dificultad que puede encontrarse a la hora de implantar un monitor de este tipo radica en que hay que asegurar que todos los nodos del sistema adopten las mismas decisiones, por lo que puede establecerse cierta correspondencia entre los algoritmos de pertenencia a grupo y los algoritmos de *consenso*. De hecho, sin este requerimiento los monitores de pertenencia se reducirían a un pequeño fragmento en el protocolo de red que comprobaría si el destino de los envíos está contestando o no y actualizaría la información que mantendría para saber el estado de los demás nodos del sistema. Sin embargo, esto no sería muy aconsejable en un sistema distribuido puesto que cada nodo podría tener una idea distinta acerca del estado de los demás, con lo que las aplicaciones distribuidas existentes podrían comportarse de una manera poco predecible al no tener ninguna garantía acerca del estado de cada uno de sus componentes.

En este capítulo se va a describir el protocolo de pertenencia utilizado en nuestra arquitectura HIDRA. Para ello vamos a empezar en la sección 3.2 describiendo las funciones principales que se van a exigir a este componente de la arquitectura. En la sección 3.3 se explicarán las diferentes soluciones que se han venido dando en otros sistemas para este mismo problema, catalogándolas principalmente por el grado de sincronía asumido en ellos. Posteriormente, en la sección 3.4 se describirá el protocolo empleado en HIDRA, estudiando su coste y comparando su funcionalidad con la ofrecida por algunos otros algoritmos. Para terminar, la sección 3.5 propone otras líneas de estudio que podrán ser iniciadas en HIDRA en relación a los algoritmos de pertenencia.

3.2 Funciones principales

La función básica de un protocolo de pertenencia en un sistema distribuido es comprobar periódicamente (y normalmente con un período lo más corto posible para permitir que el sistema reaccione con rapidez a los cambios que pueda haber) el estado de las máquinas o los procesos que integren dicho sistema. El uso de este tipo de protocolos se extendió gracias a la implantación de algunos soportes para el modelo de replicación activa, los cuales precisaban conocer el conjunto actual de procesos que constituía el grupo para implantar de esta manera los protocolos de difusión atómica ordenada de mensajes. Con ese objetivo inicial, los primeros protocolos de este estilo se centraron en comprobar el estado de un conjunto de procesos. Sin embargo, estos algoritmos tienen sentido para otras muchas tareas relacionadas con el funcionamiento "normal" de un sistema distribuido por lo que parece más aconsejable implantar un protocolo de pertenencia que controle el estado de las múltiples máquinas que compongan el sistema. Adaptar posteriormente el resultado proporcionado por un algoritmo de este estilo para conocer el estado de un conjunto de procesos no resulta demasiado difícil.

Estas tareas adicionales, además de las de comprobación del estado de las máquinas, son aquellas relacionadas con la implantación de un determinado modelo de fallos (estos modelos ya fueron descritos en la sección 2.3 que empieza en la página 23) y los ofrecidos en general para reconfiguración de aplicaciones y otros componentes del propio sistema. En las próximas secciones se estudiará con más detalle cada una de estas funciones.

3.2.1 Detección de caídas e incorporaciones

La principal misión del protocolo de pertenencia será justamente controlar cuándo una de las máquinas del cluster ha caído, es decir, ha dejado de responder a las peticiones que efectúan el resto de las máquinas, y cuándo una máquina se ha reactivado y ha pedido su inclusión en el cluster.

La detección de caídas no es una tarea trivial en sistemas asíncronos. De hecho, existen resultados que prueban la imposibilidad de tal detección en ese tipo de sistemas [CHTCB96]. En HIDRA, aunque no se realiza una sincronización de los relojes de los diferentes nodos que componen el sistema, este resultado de imposibilidad no resulta aplicable. Esto se debe a que sí es posible acotar el tiempo máximo de transmisión de los mensajes ya que por el tipo de red de interconexión utilizado y por sus protocolos de transporte, se puede conocer dicho límite. Teniendo dicho tiempo de transmisión acotado se puede tener el suficiente grado de sincronismo entre los nodos como para determinar cuándo se han perdido mensajes.

Además, se necesita que el algoritmo garantice que las decisiones que se tomen sean adoptadas por todos los miembros del sistema. Si la máquina que se sospecha que ha caído no lo ha hecho realmente, en la práctica el sistema deberá comportarse como si tal máquina fuera inaccesible. Para ello se necesitará el auxilio de los protocolos de transporte que se utilizarán para intercomunicar los nodos que compongan el cluster.

La detección de incorporaciones no resulta tan complicada. Basta con que el nodo que desee incorporarse siga el protocolo establecido y comunique al resto que ya vuelve a estar activo.

Nótese que estamos asumiendo que las máquinas pueden caer y posteriormente, tras ser reparadas o reactivadas, solicitar de nuevo su reincorporación al cluster. Esto no está contemplado en

el modelo de *fallo parada* que se pretende garantizar en HIDRA. Por tanto, necesitamos de nuevo la ayuda del protocolo de pertenencia y de los protocolos de transporte para poder implantar tal modelo de fallos. La solución no es excesivamente complicada. Basta con modificar la identidad de una máquina cada vez que ésta se reactive. Con ello, para el sistema cada reincorporación de un mismo nodo será vista como la incorporación de una máquina distinta. Esto tiene pocas repercusiones en la implantación de las aplicaciones que funcionarán sobre nuestro sistema en cluster, por lo que es admisible este tipo de modelo.

3.2.2 Implementación de protocolos de transporte fiable

Cada vez que haya algún cambio en el *conjunto de pertenencia*, (es decir, en el conjunto de máquinas que compongan el cluster actual) el monitor deberá notificárselo al componente de nuestra arquitectura encargado de implantar un transporte fiable. Por tal *transporte fiable* entendemos aquél que no pierda mensajes, ni los entregue en más de una ocasión y que garantice que cualquier mensaje enviado será entregado a menos que su destinatario caiga entre ambos eventos (el de emisión y el de entrega del mismo mensaje).

Nótese que esto también tiene influencia en el modelo de fallos que se estará proporcionando a los niveles superiores (en concreto, al ORB, que es el primer componente de nuestra arquitectura a la hora de utilizar los servicios de este protocolo de transporte fiable). Si no existiera tal protocolo, el modelo de fallos pasaría a ser uno de caída y enlace, omisión de envíos, omisión de recepciones o bien omisión general, pues estos cuatro modelos guardan relación con los problemas que encontremos en los enlaces de intercomunicación.

Por tanto, puede observarse que estos dos primeros requerimientos que se han planteado al protocolo de pertenencia han tenido como objetivo ofrecer un modelo de fallos con la mínima severidad posible. En concreto, el modelo de fallos que utilizará el ORB y cualquier otra aplicación que funcione por encima del transporte fiable será un modelo de fallo parada. Esto facilitará las cosas a la hora de desarrollar el resto de componentes de nuestra arquitectura HIDRA, principalmente por lo que respecta a los protocolos que deberá utilizar el ORB para reconfigurar su estado.

3.2.3 Caída forzosa

Cuando un nodo deja de responder durante cierto tiempo al intercambio de mensajes que requiere nuestro protocolo, los demás consideran que ha fallado. Aunque no lo haya hecho, nuestro protocolo le obliga a solicitar de nuevo su inclusión, utilizando un identificador diferente (su mismo identificador fijo pero con un nuevo número de encarnación).

Para obligarle a solicitar su inclusión lo único que resulta necesario es ignorar todos aquellos mensajes enviados por un nodo considerado fallido. Para ello se utiliza la ayuda del protocolo de transporte fiable que está implantado de esta manera.

Nótese que este tipo de *caída forzosa* resulta necesario para poder implantar el modelo de fallo parada que deseamos proporcionar al ORB que incluirá el soporte para objetos replicados.

3.2.4 Gestión de protocolos de reconfiguración

Una vez haya sido cambiado el conjunto de pertenencia, bien por una inclusión o una detección de fallo (simples o múltiples), el estado del sistema debería reconfigurarse para adaptarse a la nueva situación. Para ello, el protocolo de pertenencia tendrá registrados algunos componentes que deseen ser informados sobre tales cambios.

En nuestro algoritmo se deja que estos componentes que actúan como clientes del protocolo de pertenencia puedan solicitar en qué orden desean ser informados. Para ello, nuestro algoritmo establece una serie de *pasos de reconfiguración*, asegurando que durante dicha reconfiguración todos los nodos que compongan el sistema se encontrarán siempre en el mismo paso y que transitarán al siguiente todos ellos (o al menos, los que tengan el siguiente paso pendiente) a la vez. Además, para aumentar su flexibilidad se permite que diferentes nodos puedan tener un número diferente de pasos solicitados. Cuando un nodo haya terminado los pasos que solicitó, abandonará el proceso de reconfiguración y los demás podrán continuar con él.

Los principales clientes de nuestro soporte para protocolos de reconfiguración son el transporte fiable y el ORB, que de hecho necesita varios pasos para rehacer las cuentas de referencias en caso de que la reconfiguración se deba a la parada de un nodo. Aparte, nuestro monitor de pertenencia admite que otros componentes del sistema, incluso aplicaciones que se ejecuten a nivel de usuario, puedan registrarse y ser notificados en un determinado paso (o en varios de ellos).

3.3 Protocolos existentes

Antes de describir el monitor de pertenencia desarrollado para HIDRA conviene estudiar qué características puede presentar un protocolo de este tipo y ver otros protocolos desarrollados para otros sistemas. En las próximas secciones se detallará esta caracterización de protocolos y se darán algunos ejemplos de los protocolos más relevantes, según el tipo de entorno en el que deban funcionar.

3.3.1 Caracterización

Para caracterizar un monitor de pertenencia pueden estudiarse múltiples propiedades que debe cumplir o bien el entorno o bien el propio protocolo de pertenencia. Para realizar este estudio nos centraremos en éstas: sincronía del sistema; precisión, viveza y seguridad del algoritmo empleado; identificación de miembros; gestión de particiones; simetría del algoritmo empleado; acuerdos sobre cambios; procedimiento de inicio.

Sincronía del sistema

Cuando consideramos la *sincronía* del sistema, podemos establecer dos extremos opuestos:

- *Sistemas síncronos.* En un sistema de este tipo se asume que todos los relojes de todos los nodos se encuentran sincronizados con un alto grado de precisión y que todos ellos avanzan con la misma *cadencia*. Además, los enlaces de comunicación deben tener un *tiempo* máximo *de transmisión* acotado.

- *Sistemas asíncronos.* En este caso se asume que los relojes de cada nodo no están sincronizados e incluso que no todos ellos progresan con la misma cadencia. El tiempo máximo de transmisión de un mensaje por los enlaces de comunicación no está acotado.

En [CHTCB96] se demuestra que en sistemas totalmente asíncronos resulta imposible implantar un protocolo de pertenencia con garantías de adoptar decisiones aceptadas por todos sus miembros. Es más, se indica que esto seguirá siendo imposible aunque se adopte el modelo de partición primaria y se eliminen forzosamente a los nodos sospechosos de haber fallado.

Sin embargo, no todos los sistemas distribuidos suelen adherirse completamente a alguno de los dos modelos presentados anteriormente. En la práctica, la mayor parte de los sistemas no son síncronos, pero tampoco totalmente asíncronos. Por tanto, dependiendo de qué tipo de sincronía se introduzca, sí se podrán implantar protocolos de pertenencia en sistemas que no sean totalmente síncronos. Por ejemplo, bastaría con determinar con certeza cuál puede ser el tiempo máximo de transmisión de un mensaje en la red y replicar ésta para que no se den fallos en los enlaces, junto a la exigencia a los sistemas operativos de que se proporcione suficiente memoria para el almacenamiento temporal de los mensajes para que no se produzcan pérdidas cuando el sistema supere cierto grado de carga.

En nuestro caso, para realizar el estudio dividiremos a los sistemas analizados en las dos clases siguientes: síncronos y parcialmente síncronos. Los síncronos serán aquéllos que se adapten exactamente a la definición dada anteriormente. Los *parcialmente síncronos* serán aquéllos que o bien sean capaces de sincronizar sus relojes con una precisión aceptable o bien puedan tener enlaces de la suficiente calidad como para garantizar un tiempo acotado en la transmisión de los mensajes.

Precisión, viveza y seguridad

El mecanismo de notificación de fallos que utilice un algoritmo de pertenencia cumplirá las propiedades de precisión, viveza y seguridad, según [BG93] si cumple:

- *Precisión*: El mecanismo será preciso si sólo notifica cambios reales de la pertenencia a sus clientes. En otras palabras, todos los cambios en la pertenencia deben ocurrir antes de ser notificados, pero puede haber cambios que nunca se notifiquen.

  Normalmente esta propiedad no puede ser cumplida por los sistemas parcialmente síncronos, donde muchas veces se dan notificaciones falsas que la propiedad de precisión no permite.

- *Viveza*: El mecanismo será vivo si todos los cambios en la pertenencia son eventualmente notificados a los clientes de este servicio. Aquí, un monitor puede notificar cambios que nunca hayan ocurrido realmente, pero sí debe garantizar que los cambios ocurridos sean notificados.

- *Seguridad*: El mecanismo será seguro si en un momento determinado todos los miembros operativos están de acuerdo en el conjunto actual de pertenencia y todos los procesos que se asumen fallidos no pueden comunicarse con los operativos.

Aunque es interesante que un monitor de pertenencia cumpla las tres propiedades, en sistemas parcialmente síncronos resulta difícil garantizar la precisión. Sin embargo, esto puede compensarse si el mecanismo de notificación de cambios utilizado sí cumple las propiedades de viveza y seguridad. Si no cumple al menos esas dos, serían inútiles sus servicios.

Identificación de miembros

Dependiendo del modelo de fallos que se quiera proporcionar en el sistema, la identificación de los miembros que puedan pertenecer al conjunto de interés puede realizarse de dos maneras distintas:

- *Identificadores fijos*: La identidad de una máquina nunca cambia.
- *Identificadores por encarnación*: Cada vez que se considera que la máquina ha fallado y cuando ésta se recupera y se reintegra en el grupo, se modifica su identificador de nodo. Esto puede lograrse sin más que añadir un sufijo con el número de encarnación a un identificador fijo preconfigurado.

La utilización de identificadores por encarnación tiene la ventaja de que facilita enormemente la detección de *mensajes obsoletos* (aquellos que se han emitido en una configuración anterior del grupo) o el rechazo de los mensajes emitidos por una máquina que se considera que ha fallado. Aparte, sirve también para implantar el modelo de fallo parada.

Gestión de particiones

Una *partición* se da en un grupo cuando al menos dos subgrupos disjuntos y no vacíos permanecen aislados, esto es, sin poderse intercomunicar. Por su parte, un grupo está *libre de particiones* si dos nodos operativos cualesquiera del grupo pueden intercomunicarse siempre.

Existen tres modelos de monitores de pertenencia según su gestión de particiones:

- *Ausencia de particiones* [Cri91a]. Este modelo asume que los grupos siempre estarán libres de particiones. Para garantizar esto, se necesita replicar los enlaces de intercomunicación o tener una red cuya topología garantice la ausencia de particiones aunque aparezcan fallos en algunos enlaces.
- *Partición primaria* [RSB93]. Aquí se reconoce que las particiones pueden darse, pero sólo uno de los subgrupos resultantes puede continuar. Los nodos incluidos en los demás subgrupos estarán forzados a fallar y a reintentar su inclusión en el *grupo principal* (aquél que tienen la mayoría de los nodos), al igual que los nuevos nodos. Los protocolos que siguen este modelo evitan el problema de la fusión de aquellos subgrupos que han permanecido aislados temporalmente.
- *Particionable* [FKM$^+$95]. Este modelo admite particiones y permite que cualquier subgrupo resultante pueda proseguir y reintegrarse en el grupo principal posteriormente.

Simetría

Un protocolo de pertenencia a grupo es *simétrico* (o totalmente distribuido) si todos los monitores ejecutan el mismo algoritmo y en tal algoritmo únicamente existe un rol. Por contra, un protocolo de pertenencia es *centralizado* si algún miembro del grupo gestiona el comportamiento de los demás, coordinando la aceptación de los cambios que se deban introducir.

Los protocolos centralizados normalmente requieren un número inferior de mensajes para desarrollar su trabajo ya que la mayor parte de las decisiones son tomadas por un nodo coordinador, difundidas por éste y aceptadas por el resto.

Acuerdo sobre cambios

Cuando el conjunto de pertenencia cambia, normalmente no todos los monitores son capaces de detectar dicho cambio al mismo tiempo. El acuerdo sobre el cambio se consigue cuando todos los miembros lo han adoptado. La forma de hacerlo dependerá del grado de simetría del algoritmo empleado por los monitores de pertenencia.

Se distinguen dos tipos de acuerdo [RFJ93]:

- *Acuerdo fuerte*. Un protocolo de pertenencia mantiene un tipo de acuerdo fuerte si todos los miembros operativos ven y han visto la misma secuencia de conjuntos de pertenencia desde que han sido incluidos en el grupo. Obsérvese que este tipo de acuerdo exige que los mensajes de notificación de cambio en el grupo guarden *orden total*.

- *Acuerdo eventual*. Cuando se usa este tipo de acuerdo, el protocolo requiere que cuando se dé algún cambio en el conjunto de pertenencia, eventualmente todos los miembros del grupo lleguen a la misma visión del conjunto de pertenencia actual. Esto no excluye que dos miembros distintos tengan dos secuencias de conjuntos diferentes. Por ejemplo, un miembro operativo A puede haber detectado el fallo y reinserción de un miembro C, mientras que para otro miembro B, el nodo C nunca falló.

El tipo de acuerdo que se adopte en un protocolo de pertenencia dependerá del tipo de aplicación que sea el principal usuario de los servicios del monitor. Existen algunas aplicaciones que pueden soportar los acuerdos eventuales y de esa forma logran simplificar bastante las tareas que desarrolla el monitor de pertenencia.

Procedimiento de inicio

Con el nombre de *procedimiento de inicio* denotaremos la forma en la que un nodo puede integrarse en el grupo al que va a pertenecer. Esto está condicionado por el conocimiento que tiene cada nodo sobre los demás posibles integrantes del grupo, en base a lo cual estableceremos las posibles categorías de procedimientos de inicio.

En [HS95] se distinguen dos tipos de inicio:

- *Inicio colectivo*: En este tipo de inicio cada nodo conoce el identificador y la dirección de cada uno de los demás posibles miembros del grupo al cual debe pertenecer. De esta manera resulta sencillo que cuando cada máquina arranque pueda enviar mensajes únicamente a los

nodos preconfigurados dentro del grupo y no haya que tomar excesivas precauciones para garantizar la seguridad de éste.

Por contra, presenta el problema de que el conjunto máximo de nodos debe estar preconfigurado y resulta complicado añadir nuevas máquinas a tal conjunto sin modificar algunos ficheros o bases de datos.

- *Inicio individual*: En este segundo tipo de inicio cada nodo desconoce la identidad y dirección del resto. Así el procedimiento de inicio resulta ligeramente más complejo que en el caso anterior pues debería comprobarse que efectivamente se está entrando en el grupo donde se quería y no en algún otro que emplee ese mismo algoritmo.

  Normalmente en el inicio individual cada nodo empieza construyendo un conjunto de pertenencia donde únicamente se encuentra él y difunde algunos mensajes con la esperanza de que otros nodos puedan bien integrarse en su grupo o bien permitir que él se integre en el que ya está formado.

  Este tipo de inicio tiene sentido en aquellos grupos que puedan crecer dinámicamente y donde no haya grandes restricciones para mantener la seguridad dentro del grupo.

3.3.2 Ejemplos

En la tabla 3.1 se presentan las principales características de algunos monitores de pertenencia. En sus columnas y contenido se han utilizado las siguientes abreviaturas: “Sin.” para referirse a la sincronía con los posibles valores S (sistema síncrono) y PS (sistema parcialmente síncrono); “Pre.” hace referencia a la propiedad de precisión; “Viv.” a la de viveza; “Seg.” a la de seguridad; “Ide.” indica el tipo de identificador para los nodos, puede ser fijo o PE (por encarnación); “Par.” se refiere a la tolerancia a las particiones que realice el protocolo, sus posibles valores son “Aus” para indicar la ausencia de particiones, “Pri” para indicar un modelo de partición primaria y “Par” para indicar un sistema particionable; “Sim.” indica la simetría del algoritmo, sus posibles valores son “Sim” para algoritmos simétricos y “Cen” para algoritmos centralizados o con al menos un miembro coordinador; “Acu.” se refiere al tipo de acuerdo que utiliza el algoritmo, puede ser “Fu” para acuerdo fuerte o “Ev” para acuerdo eventual; “Ini.” indica el tipo de procedimiento de inicio a emplear, sus posibles valores son “Ind” para inicio individual y “Col” para inicio colectivo.

En el resto de esta sección se van a describir los algoritmos más interesentes que aparecen en la tabla 3.1, siguiendo un orden cronológico.

Algoritmos síncronos de Cristian [Cri91a]

Los tres algoritmos síncronos presentados en [Cri91a] se caracterizan por suponer que existe un servicio de difusión atómico utilizable por estos algoritmos, aparte de suponer un sistema distribuido en el que todos sus nodos tienen sus relojes perfectamente sincronizados.

El protocolo necesario para realizar inserciones de nuevos miembros es idéntico para los tres monitores presentados y funciona del siguiente modo. El proceso que intenta incorporarse al grupo difunde un *mensaje NEW_GROUP* que es contestado mediante una difusión de un *mensaje PRESENT* por cada uno de los miembros ya activos en el grupo. Como la entrega de las difusiones

Algoritmo	Sin.	Pre.	Viv.	Seg.	Ide.	Par.	Sim.	Acu.	Ini.
Cristian [Cri91a]	S	Sí	Sí	Sí	Fijo	Aus	Sim	Fu	Ind
Delta-4 [RVR93]	PS	Sí	Sí	Sí	Fijo	Aus	Cen	Fu	Col
Totem [AMMS$^+$93]	PS	No	Sí	Sí	Fijo	Par	Cen	Fu	Col
Débil [RFJ93]	PS	No	Sí	No	PE	Par	Cen	Ev	Ind
Fuerte [RFJ93]	PS	No	Sí	Sí	PE	Par	Cen	Fu	Ind
Isis S-GMP [RB94]	PS	No	Sí	Sí	PE	Pri	Cen	Fu	Col
TTP [KG94]	S	Sí	Sí	Sí	Fijo	Aus	Sim	Fu	Col
Transis [DMS96]	PS	No	Sí	Sí	PE	Par	Sim	Fu	Ind
HMM [MGB00]	PS	No	Sí	Sí	PE	Aus	Cen	Fu	Col

Tabla 3.1: Principales características de algunos protocolos de pertenencia.

es atómica, transcurrido un cierto tiempo límite todos los nodos válidos han debido responder y todos ellos tendrán la misma idea del conjunto actual de pertenencia.

Este mismo protocolo de inserción de nodos se utiliza cuando se detecta algún fallo para reconstruir el conjunto de pertenencia.

Las diferencias surgen a la hora de comprobar la estabilidad del grupo actual, o lo que es lo mismo, a la hora de detectar los fallos de los miembros activos. Para esta tarea se presentan tres alternativas que dan nombre a cada uno de los algoritmos:

- **Difusiones periódicas**. En este caso, cada cierto tiempo todos los miembros difunden un mensaje PRESENT. En base a él se construye la imagen actual del grupo.
- **Lista de espera**. Los miembros activos se organizan lógicamente en un anillo y el miembro con el identificador más alto periódicamente envía un mensaje a uno de sus vecinos, que irá rotando por el anillo. Como todos los relojes están perfectamente sincronizados, cada nodo sabe cuándo debe recibir el mensaje periódico. Si no lo hace, asume que su emisor ha fallado y reinicia el protocolo tal como se explicó arriba.
- **Vigilancia de vecinos**. Igual que en el algoritmo anterior, los nodos se organizan en un anillo lógico siguiendo el orden creciente de identificadores. Periódicamente cada nodo envía un mensaje a uno de sus vecinos (por ejemplo, al que tenga el identificador menor, cerrando el ciclo). Si el mensaje no llega en el tiempo esperado, se asume que su emisor ha fallado y se reinicia el protocolo.

Nótese que estos algoritmos son relativamente fáciles de describir y seguir, pero no tan fáciles de implantar. La principal dificultad estriba en la asunción de que hay un protocolo de difusión atómica por debajo del protocolo de pertenencia. No todos los sistemas pueden facilitar este soporte.

Algoritmo MGS del sistema Delta-4 [RVR93]

El protocolo de pertenencia *MGS* del sistema *Delta-4* [RVR93] desarrollado en la *Univ. Técnica de Lisboa* está pensado para aplicaciones de tiempo real que utilicen también objetos replicados.

El entorno donde se utiliza es una red local donde se asume que no van a haber particiones. El sistema es parcialmente síncrono, pues se apoya principalmente en la limitación del tiempo máximo de transmisión de los mensajes y no sincroniza los relojes de los diferentes nodos ni los monitores que en ellos funcionan.

El protocolo de pertenencia está integrado en los protocolos de red que utiliza el sistema, donde se utilizan las difusiones como el principal mecanismo de intercomunicación. Así, cuando se inicia una difusión se registra cuáles son sus destinos y se espera un mensaje de confirmación de éstos. Si alguna de estas confirmaciones no llega, el emisor le repetirá unas cuantas veces el envío y si sigue sin responder, sospechará que ha fallado e iniciará el protocolo para realizar un cambio en el conjunto de pertenencia.

Cuando un nodo ha detectado que otro ha fallado o que uno nuevo intenta incorporarse al grupo difunde un mensaje solicitando efectuar el cambio en el conjunto de pertenencia. Los nodos receptores deben contestarle otorgándole el permiso a realizar tal cambio, pero si ya existe algún cambio que ha sido iniciado pero todavía no ha concluido responderán negativamente. Puede darse el caso de que algunos respondan afirmativamente y otros no y que haya contención a la hora de efectuar la reserva para iniciar el cambio. Esto se deshace dando mayor prioridad a uno de los dos contendientes.

Asumiendo que el iniciador ha obtenido respuesta afirmativa por parte del resto de nodos activos, procedería seguidamente a notificar el nuevo conjunto de pertenencia y a esperar confirmación por parte de los demás. Si algún nodo falla mientras se está realizando esto, el protocolo termina en primer lugar la instalación del conjunto que estaba en curso (arrancando nuevos coordinadores que terminan el trabajo) y posteriormente procede a la instalación del nuevo conjunto reiniciando el protocolo desde el principio.

Algoritmo de Totem [AMMS$^+$93]

El algoritmo de pertenencia [AMMS$^+$93, AMMS$^+$95] utilizado en el sistema *Totem* está pensado para sistemas parcialmente síncronos con redes particionables, al igual que los algoritmos de Rajkumar, Fakhouri y Jahanian que se verán seguidamente. Además, utiliza identificadores fijos y procedimiento de inicio colectivo.

En Totem, tanto para lograr las difusiones fiables como para monitorizar el estado de los nodos que integran el grupo se organizan estos nodos en un anillo lógico y se hace circular por él un *testigo* (o *token*). El grupo tiene un nodo que realiza el papel de *representante* del grupo a la hora de aceptar inclusiones de nuevos nodos o fusiones de subgrupos que han podido intercomunicarse.

El algoritmo de pertenencia es invocado cuando se detecta una pérdida del testigo o la recepción de un mensaje externo a los actuales miembros del grupo. Un nodo que arranca forma en primer lugar un grupo donde el único miembro será él mismo y después difunde un *mensaje ATTEMPT JOIN*. Únicamente los representantes procesan los mensajes externos.

Cuando un representate recibe un mensaje externo, inicia la difusión de un mensaje ATTEMPT JOIN, advirtiendo su intención de formar un anillo mayor y después cambia al estado de recolección.

En el estado de recolección los representantes de cada anillo van aceptando mensajes ATTEMPT JOIN de otros representantes hasta que transcurre un cierto tiempo. Hecho esto, se difunde un *mensaje JOIN* que contiene los identificadores de todos los representantes localizados y se

pasa al estado de confirmación.

En el estado de confirmación, los representantes se ponen de acuerdo acerca del conjunto de representantes que participarán en la formación del nuevo anillo. Los mensajes JOIN contienen dos conjuntos de representantes: los considerados activos y los que se considera que han fallado. Si todos los JOIN contienen la misma información, el estado de confirmación termina con ese acuerdo. Si por el contrario, algunos mensajes difieren, los nodos que han emitido aquellos mensajes con menor información deben reemitirlos con la información actualizada hasta que todos lleguen a emitir lo mismo. Puede darse el caso de que no se logre el acuerdo dentro del tiempo establecido, en cuyo caso aquellos representantes cuyos conjuntos eran minoritarios pasan a engrosar el conjunto de representantes fallidos.

Una vez se ha llegado al acuerdo dentro del estado de confirmación, uno de los representantes (elegido con cierto criterio uniforme. Por ejemplo, aquél con el identificador de nodo más bajo) crea un testigo nuevo para recolectar la identidad de todos los miembros del nuevo anillo. Para ello, el testigo inicia el recorrido por el antiguo anillo al que pertenecía ese representante y todos los nodos van añadiendo su identificador a una lista contenida en él. Una vez el testigo llega al representante, éste lo transmite al siguiente representante, que lo hará rotar por su anillo y lo transmitirá al siguiente representante, y así sucesivamente. Finalmente, el testigo da una vuelta completa al nuevo anillo y ya contiene la identidad de todos los nodos que lo componen. Se vuelve a transmitir el testigo para que dé otra vuelta completa y de esta forma cada nodo obtiene la lista de todos los integrantes del grupo.

Para el caso de las caídas, el nodo que esperaba el testigo que no ha llegado pasa a ser representante, eliminando del grupo al nodo que debería haberlo transmitido. Tras esto se reinicia el protocolo descrito anteriormente.

Algoritmos de Rajkumar, Fakhouri y Jahanian [RFJ93]

Los algoritmos fuerte y débil descritos en [RFJ93] asumen un sistema distribuido parcialmente síncrono donde el tiempo de transmisión de los mensajes no está limitado y se pueden dar fallos por rendimiento o por omisión en las transmisiones. El único grado de sincronía introducido corresponde a la cadencia de progreso en los relojes de cada nodo, cuyas diferencias deben ser inapreciables.

En ambos protocolos el grupo actual está estructurado en un anillo lógico y un miembro es elegido como coordinador. Para proceder a la detección de fallos, cada miembro emite periódicamente un mensaje a cada uno de sus dos vecinos en el anillo.

Cuando se detecta un fallo en el protocolo débil, el detector informa sobre el cambio al coordinador. Entonces, este coordinador difunde un *mensaje NEW_GROUP* que es aceptado por todos los demás miembros, formando de nuevo un anillo y reiniciando las comprobaciones periódicas. Ya que pueden perderse tanto los informes de los detectores como algunos mensajes en las difusiones del coordinador. No hay ninguna garantía de que todos los miembros estén siempre de acuerdo sobre cuál es el conjunto actual de pertenencia, por eso se dice que en este algoritmo se emplea el acuerdo eventual.

En el protocolo fuerte el algoritmo anterior se extiende con una segunda fase. Cuando el líder recibe la notificación de fallo, difunde un *mensaje PTC* (*prepare to commit*). Los demás miembros deben contestarlo con un *mensaje ACK* o *NAK*. Si no llega ningún NAK y llegan todos los ACK, el

coordinador difunde el nuevo conjunto de pertenencia en un *mensaje COMMIT* que será aceptado por todos los miembros.

Como ambos protocolos utilizan un coordinador que centraliza algunas fases, ambos deben gestionar su fallo. En ese caso, otro miembro es elegido como nuevo coordinador y retoma el resto de pasos que faltaban completar, eliminando al coordinador antiguo.

Algoritmo de Isis [RB94]

El algoritmo *S-GMP* (*Strong Group Membership Protocol*) del sistema *Isis* está orientado a sistemas asíncronos, donde los relojes de cada nodo no están sincronizados y el tiempo de transmisión de los mensajes no está acotado. El protocolo de pertenencia va a colaborar con el de transporte para facilitar un modelo de fallo parada, lo que va a provocar que se utilicen identificadores por encarnación para referirse a los nodos. Este protocolo de transporte también garantizará que los canales realicen sus entregas en orden FIFO.

Este algoritmo es asimétrico pues a uno de los nodos del grupo le otorga un papel gestor, forzando a que todos los demás respeten sus decisiones. De esta manera, cualquier variación advertida por el gestor es difundida de inmediato por éste mediante un mensaje SUBMIT. En este mensaje únicamente se especifican las variaciones detectadas (qué nodos han fallado o qué nodos acaban de entrar; téngase en cuenta que se utiliza inicio colectivo y que todos los nodos conocen la identidad de cualquier posible miembro del grupo). Aparte, se asume un mecanismo de detección externo que no se describe en este protocolo de pertenencia.

En cuanto un nodo recibe el *mensaje SUBMIT* lo contesta con un *ACK*, indicando que se ha enterado del cambio propuesto. Transcurrido un cierto tiempo, el gestor comprueba cuántas contestaciones ha recibido. Si hay una mayoría de ellas, procede a iniciar una segunda fase, emitiendo un *mensaje COMMIT* para que todos los miembros instalen la nueva imagen del grupo. Si por contra, el gestor no recibió una mayoría de reconocimientos, asumirá que ha quedado en un subgrupo minoritario y que se ha dado una partición, abortando.

Si en lugar de una caída se ha detectado la adición de uno o más miembros al grupo. Antes del mensaje de COMMIT, se envía un *mensaje STATE-XFER* en el que se indica a los nuevos nodos cuál es el conjunto de pertenencia actual.

En caso de que el nodo que falle sea el propio gestor, el protocolo aumenta ligeramente con una fase previa en la que el *nodo iniciador* (aquél que ha detectado el fallo del gestor) envía un *mensaje INTERROGATE* que será contestado por cada miembro con información acerca de su estado local: conjunto de pertenencia, número de reconfiguraciones realizadas, último cambio confirmado, etc. Con esta información, el iniciador sabrá qué cambios deberá notificar al resto de nodos para formar el nuevo grupo de pertenencia, que será establecido con las dos rondas de mensajes que ya hemos descrito anteriormente.

Algoritmo de TTP [KG94]

El *Time-Triggered Protocol* (*TTP*) desarrollado en la *Universidad Técnica de Viena* ofrece una amplia variedad de servicios y garantías orientados a sistemas de tiempo real. Junto a los servicios de pertenencia a grupo podemos encontrar servicios de sincronización de relojes, transporte fiable y con tiempo de transmisión acotado, gestión de interferencias y gestión de cambios de modo operacional.

Se asume que el sistema utiliza tanto nodos como enlaces replicados, con lo que la tolerancia a fallos empieza aquí ya dentro de los propios equipos. Con esto se consigue adoptar un modelo de ausencia de particiones.

El propio protocolo TTP establece las bases para modelar el sistema como síncrono, gracias a la acotación de los tiempos de transmisión y a la sincronización de los relojes físicos.

Otra característica importante de este protocolo es su forma de regular el acceso a la red de interconexión. Ya que todos los nodos tienen sus relojes sincronizados, se aprovecha esto para establecer una división temporal en el acceso a la red. Cada nodo que pueda pertenecer al sistema distribuido tiene asignado un determinado intervalo de acceso que se va repitiendo periódicamente. Con ello se garantiza la acotación del tiempo de transmisión y se facilita bastante el diseño del protocolo de pertenencia.

Este protocolo de pertenencia se basa en el uso que cada nodo realice de su intervalo de acceso a la red. Si el nodo no lo utiliza será dado de baja en el grupo, por lo que al menos debe enviar un mensaje en cada intervalo para indicar que sigue activo. Por el contrario, si el nodo no pertenecía al grupo y quiere incorporarse a él, deberá emitir un mensaje en su intervalo e indicar en él, utilizando un campo de bits que está presente en todos los mensajes, que va a funcionar normalmente a partir de esta ronda. Además de esto, deberá esperar a que su mensaje sea reconocido por algún otro nodo. En caso contrario, su solicitud de reincorporación se entiende que ha sido rechazada, cosa que sólo ocurriría si se hubiese partido la red y que rara vez sucederá debido a que ésta se encuentra replicada.

La identidad de cada nodo en este protocolo se guarda en un campo de bits presente en todos los mensajes que envían los nodos. Por tanto, es uno de los pocos sistemas en los que se utiliza identificación fija para estas labores. En la práctica, ya que se proporciona el modelo de fallo parada, esta identidad fija debe acompañarse por algún contador adicional que permita al nodo reincorporarse con una nueva identidad. Sin embargo, esto se realiza fuera del subprotocolo de pertenencia, por lo que aquí consideraremos que los identificadores son fijos.

Por último, el protocolo de inicio es colectivo, pues antes de que el sistema pueda funcionar debe conocerse cuántas máquinas podrán llegar a formar parte de él. De otra forma sería imposible realizar la división del tiempo de transmisión en los intervalos necesarios para cada posible nodo. Esto también facilita en gran medida el desarrollo del protocolo de pertenencia que hemos explicado arriba.

Algoritmo de Transis [DMS96]

El algoritmo utilizado en el sistema *Transis* está pensado para sistemas parcialmente síncronos, particionables y con inicio individual. Es uno de los pocos algoritmos simétricos que podremos encontrar para sistemas con estas características.

Se asume que la detección de fallos es llevada a cabo por el propio protocolo de comunicaciones que sirve de base al monitor de pertenencia y que le notifica todas las sospechas de fallo que haya podido detectar.

Cada monitor mantiene un conjunto de nodos sobre los que se sospecha que han podido fallar y un conjunto de nodos activos. Cada mensaje del protocolo de pertenencia siempre incluye ambos conjuntos. Para que un cambio en el grupo pueda tener lugar, todos los miembros deben llegar a un acuerdo sobre ambos conjuntos. Para ello, ante cada cambio detectado, cada nodo difunde un

mensaje dando a conocer sus nuevos conjuntos.

Los detalles sobre cómo se logra el acuerdo pueden encontrarse en [DMS96].

3.4 HMM: Un monitor para HIDRA

HMM [MGB00, MGGB00] es el protocolo de pertenencia diseñado para la arquitectura HIDRA. Está pensado para trabajar sobre la red interna de un cluster cerrado de ordenadores. Utiliza identificadores por encarnación, un procedimiento de inicio colectivo y un modelo de ausencia de particiones.

Seguidamente se describirá el modelo de sistema sobre el que va a trabajar, los componentes de la arquitectura HIDRA con los que interactúa y el algoritmo utilizado por los monitores ubicados en cada uno de los nodos.

3.4.1 Entorno de uso

Este protocolo de pertenencia está pensado para un sistema distribuido en el que se asume que los relojes de cada nodo progresan con la misma cadencia y donde el tiempo de transmisión de los mensajes estará acotado. Esto no tiene por qué ser difícil de conseguir si existe una LAN privada para los nodos del cluster, donde los mensajes únicamente podrían perderse debido a la falta de buffers de recepción o algún tipo de interferencia. No se require que los nodos del sistema tengan sus relojes sincronizados. Aun así, el modelo de sistema utilizado puede catalogarse como *parcialmente síncrono*.

Debido al tipo de red de interconexión que se empleará, sus reducidas dimensiones y la ausencia de puentes o pasarelas para lograr dicha interconexión, tampoco resulta necesario contemplar el caso de que la red se pueda particionar. De cualquier forma, adaptar el algoritmo para que admita particiones no será complicado.

Las máquinas tendrán asociados unos identificadores fijos a los que se añadirá un contador de encarnaciones que se incrementará cada vez que la máquina a la que está asociado sea reiniciada o reincorporada al grupo. Todos los nodos del sistema conocerán los identificadores y direcciones de todos los demás nodos que podrán formar parte de él. Por ello, nuestro protocolo puede catalogarse como un monitor con inicio colectivo.

No habrá forma de distinguir entre las caídas reales de las máquinas y su pérdida de respuestas debido a un periodo en el que tengan que servir una carga elevada o a la rotura de un enlace de comunicación. Por tanto, por debajo de nuestro protocolo de pertenencia el modelo de fallos, según [Sch93b], corresponderá al de *omisión de recepciones*. Si seguimos los modelos de fallos propuestos en [Cri91b], nuestro sistema tendría *fallos de omisión* y *fallos de caída con amnesia*. En el modelo propuesto por Cristian, las caídas de los nodos no presuponen que dicho nodo nunca se reinicie. Esto se adapta mejor al modelo existente por debajo de nuestro protocolo de pertenencia. Sin embargo, los servicios del protocolo de pertenencia permitirán desarrollar un protocolo de transporte fiable que eliminará los fallos de omisión, retardo, duplicidad y corrupción de mensajes y que gracias al uso de identificadores por encarnación logrará alcanzar un modelo de *fallo parada*, que es el que podrán utilizar las aplicaciones y el resto de componentes de HIDRA que trabajen por encima de nuestro protocolo.

3.4.2 Componentes relacionados

Tal como se describió en la sección 2.2 que empieza en la página 20 y donde se explican los componentes que definen la arquitectura HIDRA, existen algunas partes de esta arquitectura que deben interactuar con el monitor de pertenencia.

Así tenemos en primer lugar un protocolo de transporte no fiable que será el utilizado por el monitor de pertenencia. Este protocolo de transporte no asegura que los mensajes emitidos se entreguen a sus destinos, ni que en caso de lograr la entrega, ésta respete el orden de emisión. Las pérdidas pueden deberse a roturas o desconexiones de enlaces, falta de buffers de recepción libres en el nodo destino o a la corrupción del contenido del mensaje. Este va a ser el único componente que deberá utilizar el protocolo de pertenencia para poder ser implantado en HIDRA.

Como usuarios de los servicios de pertenencia tendremos a un protocolo de transporte fiable y el ORB con soporte a replicación. El protocolo de transporte fiable garantizará la entrega de todos los mensajes emitidos. No se espera implantar ningún protocolo de difusión atómica ordenada a este nivel.

El ORB necesita los servicios del protocolo de pertenencia para reconfigurar su estado en caso de que ocurra algún cambio en el conjunto de nodos miembros del cluster. Si hay una caída de algún nodo, el ORB debe realizar algunos pasos de reconfiguración para rehacer sus cuentas de referencias. Estas cuentas son utilizadas para determinar cuándo un objeto deja de tener clientes y, por tanto, pueden ser liberados los recursos que tuviese asignados y destruir finalmente tal objeto. Aparte, esta reconfiguración del ORB debe realizarse una vez se ha completado la reconfiguración del protocolo de transporte fiable y debe llevarse a cabo en un conjunto de pasos que deben ser completados síncronamente. Es decir, ningún nodo empezará un determinado paso hasta que todos los nodos hayan completado satisfactoriamente el paso anterior. Por ello, nuestro protocolo de pertenencia mantiene una serie de fases en las que se realizan las notificaciones sobre los cambios en el conjunto de pertenencia, garantizando la compleción síncrona de cada fase antes de iniciar la siguiente.

3.4.3 Algoritmo utilizado

Para poder describir el algoritmo utilizado en nuestro monitor de pertenencia hay que explicar en primer lugar los diferentes roles que puede adoptar un nodo que ejecute dicho algoritmo, los diferentes estados que componen el autómata seguido y los tipos de mensajes que van a intercambiarse en cada uno de esos estados. En las próximas secciones se explican estos detalles.

Roles

HMM utiliza un algoritmo asimétrico en el que cada nodo puede seguir diferentes roles a lo largo de su ejecución. En cada momento existirá un nodo coordinador que dirigirá a todos los demás e impondrá sus decisiones. De esta forma se reduce el número de mensajes a intercambiar entre los diferentes nodos que componen el grupo.

Veamos cuáles son estos roles:

- *Maestro*. Es el único nodo que puede procesar los mensajes de solicitud de inclusión y el que dirige la reconfiguración en caso de que se haya dado un cambio en el conjunto de pertenencia.

Para llegar fácilmente a un acuerdo sobre quién debe ser el nodo maestro, éste se elige como aquél nodo con un identificador fijo más bajo de entre todos los que formen una configuración estable del grupo. Es decir, puede darse el caso de que al añadir un nuevo nodo, éste tenga el identificador fijo más bajo de entre todos los nodos activos, pero no adoptará el papel de maestro hasta que el maestro actual no haya reconfigurado el grupo y le haya dado entrada en él.

- *Esclavo*. Los nodos esclavos son todos aquellos nodos activos que, por no tener el identificador más bajo, participarán en las tareas de reconfiguración y monitorización pero siempre aceptando las decisiones que adopte el nodo maestro.

- *Iniciador*. Un nodo iniciador es aquél que durante las labores de monitorización ha detectado que el nodo que estaba monitorizando ha dejado de mostrar actividad. Por tanto, asume que tal nodo ha fallado e inicia la parte del protocolo dedicada a obtener el conjunto de nodos activos. Una vez obtenida la lista de nodos activos, se la pasa al nodo maestro y éste iniciará la reconfiguración del sistema.

- *Indefinido*. Cuando un nodo recibe un mensaje del iniciador preguntando acerca de su estado, pasa a adoptar el rol indefinido a la espera de que se elija un nodo maestro y éste inicie una reconfiguración. El propio iniciador notificará al nuevo maestro acerca de la elección efectuada. Después, el primer mensaje enviado por el maestro para iniciar la reconfiguración conseguirá que el resto de nodos cambie su papel indefinido por el de esclavo.

El rol de nodo maestro sólo puede ser llevado a cabo por un nodo dentro de una determinada configuración del sistema. Sin embargo, como pueden darse caídas múltiples, puede darse el caso de que haya múltiples nodos iniciadores dentro de la misma configuración del sistema. Veamos ahora qué estados pueden distinguirse en el autómata que es ejecutado en cada nodo.

Estados

En la figura 3.1 se presenta el diagrama de estados que sigue el autómata utilizado en el algoritmo HMM. Discutiremos seguidamente para qué sirve cada uno de estos estados.

- *Inicial*. En este estado, el nodo todavía no ha recibido ningún mensaje de ningún otro nodo y piensa que es el único miembro del cluster. Periódicamente, difundirá mensajes JOIN mientras continúe en este estado. Si el mensaje es contestado por otros miembros activos, se pasará al estado *pasos*.

- *Pasos*. Todos los nodos entran en este estado cuando algún cambio en el conjunto de pertenencia haya sido detectado. Esto ocurre cuando el nodo maestro ha recibido un mensaje JOIN de un nuevo miembro o cuando un nodo iniciador ha detectado uno o más fallos.

  Varios pasos independientes pueden distinguirse en este estado. En el primero de ellos, el nuevo conjunto de pertenencia es difundido por el nodo maestro a todos los demás. Cada nodo notifica los cambios a los componentes interesados y una vez concluido esto, responde al maestro. El resto de pasos sigue una estrategia similar: son iniciados por el maestro cuando éste advierte que todos los nodos participantes en el anterior paso han contestado

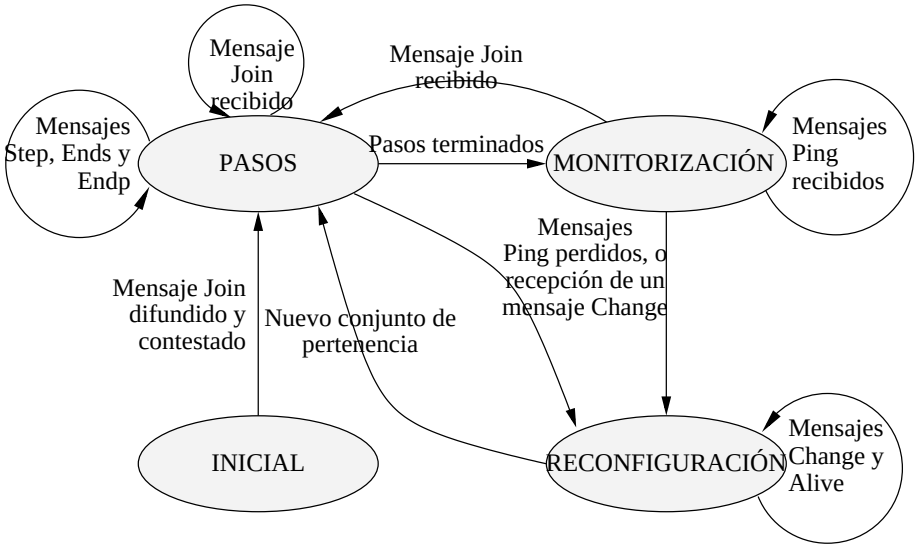

Figura 3.1: Estados y transiciones en el algoritmo HMM.

terminando tal paso, los nodos esclavos reciben el mensaje del maestro indicando que deben empezar el nuevo paso, notifican los cambios a los componentes interesados y responden posteriormente al maestro indicando que ya han terminado.

Cada nodo puede tener un número diferente de pasos de reconfiguración, por lo que existen dos tipos de mensajes de contestación al maestro, ambos indican que el paso actual ha terminado, pero uno de ellos sirve para además notificar al maestro que en dicho esclavo ya no deben realizarse más pasos de reconfiguración.

Cuando un nodo termina todos sus pasos de reconfiguración pasa de inmediato al estado de monitorización. Cuando el maestro advierte que todos los nodos han terminado todos sus pasos de reconfiguración, él también pasa al estado de monitorización.

Hay que advertir que el trabajo realizado en el estado de monitorización también se emprende en el estado de pasos una vez ha finalizado el paso 0 y todos los nodos conocen el nuevo conjunto de pertenencia.

Otras posibles transiciones de estado pueden ser causadas por la recepción de un mensaje JOIN en el nodo maestro que provocará el reinicio de este estado pasos, y la pérdida de alguna secuencia de mensajes PING utilizados para efectuar la monitorización y que conducirían al nodo detector a adoptar el papel de iniciador y transitar hacia el estado de reconfiguración.

- *Monitorización.* En este estado, se sigue exclusivamente con la monitorización de las máquinas actualmente activas. Para ello, en el estado anterior y una vez ya se conocía el conjunto de pertenencia, se estructuraban los nodos en un anillo lógico siguiendo un orden creciente de identificadores fijos de nodo y cada nodo emite periódicamente un mensaje PING a su vecino superior.

  Este estado puede abandonarse en dos casos diferentes. Bien porque el maestro activo recibe un mensaje JOIN y pasa al estado pasos desde donde obligará a todos los demás nodos a ir también al estado pasos, o bien porque un nodo advierte que no han llegado varios mensajes

PING de su vecino inferior y pasa a adoptar el papel de iniciador y entrar en el estado de reconfiguración.

- *Reconfiguración*. Cuando un nodo advierte que no le han llegado varios mensajes PING consecutivos, pasa a ser iniciador y difunde un mensaje CHANGE. Todos los nodos que contesten este mensaje dentro de un determinado tiempo son considerados nodos activos y pasarán a formar el nuevo conjunto de pertenencia. Nótese que puede haber varias caídas simultáneas y no basta con la eliminación del nodo que no ha enviado los PINGs para formar el nuevo conjunto.

  Este nuevo conjunto se incluye en un mensaje SETMEM que se envía al nuevo nodo maestro para que inicie una nueva ronda de pasos.

Veamos ahora qué contenido y qué misión tienen cada uno de los mensajes utilizados en estos estados.

Mensajes

Todos los mensajes utilizados en el protocolo HMM utilizan un mismo formato, presentado en la figura 3.2. El contenido de cada campo del mensaje puede variar según su tipo y será descrito seguidamente, junto a la misión que cumple cada mensaje en el protocolo.

```
typedef enum { ALIVE, CHANGE, ENDP, ENDS, JOIN, NEWMEM,
               PING, SETMEM, STEP } kind;
struct msg {
  seqnum_t     msg_seqnum;   // Configuration number.
  kind         msg_kind;     // Type of message.
  node_id_t    msg_sender;   // Message sender.
  other_t      msg_contents; // Message contents.
}
```

Figura 3.2: Declaración del tipo mensaje utilizado en HMM.

Los campos que encontramos en un *mensaje de pertenencia* son, por este orden, los siguientes:

1. *Número de configuración*: Guarda el número de veces que el conjunto de pertenencia ha tenido que ser reconfigurado. Sirve para detectar mensajes obsoletos cuando se intercambien mensajes entre los diferentes monitores.

2. *Tipo de mensaje*: Valor numérico que indicará el tipo de mensaje utilizado, de entre los nueve disponibles.

3. *Nodo emisor*: Guarda el identificador del nodo emisor. Incluye también su *número de encarnación*, aunque el algoritmo de pertenencia utiliza sólo el identificador fijo. El número de encarnación se guarda para informar posteriormente a los componentes interesados en los cambios.

4. *Contenido*: La información guardada en este campo depende del tipo de mensaje.

Los mensajes intercambiados serán de uno de los siguientes tipos:

- *ALIVE*. Este mensaje es enviado por todos aquellos nodos que hayan recibido un mensaje CHANGE emitido por un iniciador. Es usado en el estado de reconfiguración.

  No se utiliza el campo `msg_contents` y se recoge el valor del campo `msg_sender` para incluirlo en el nuevo conjunto de pertenencia que está formando el iniciador.

- *CHANGE*. Cuando un nodo pasa a ser iniciador al detectar el fallo de su vecino con identificador inferior, difunde este mensaje CHANGE que deberá ser contestado con un ALIVE por todos los nodos activos. Se utiliza en el estado de reconfiguración.

  Tampoco se utiliza el campo `msg_contents`. El valor de `msg_sender` se almacena para conocer la identidad del iniciador al que se debe contestar con un ALIVE.

- *ENDP*. Un nodo esclavo utiliza un mensaje ENDP cuando ha notificado al componente que lo pidió previamente acerca del nuevo conjunto de pertenencia y además, éste era el último paso de reconfiguración pedido por cualquier componente local. Este mensaje debe enviarse hacia el nodo maestro quien, por otra parte, ha debido enviar antes un mensaje STEP indicando que debía iniciarse este paso de reconfiguración.

  El envío de un ENDP implica que este nodo ya no participará en los siguientes pasos de reconfiguración, si es que hay alguno más.

  El campo `msg_contents` se utiliza para guardar el número del paso de reconfiguración que se está reconociendo.

- *ENDS*. Es un mensaje similar al anterior. La única diferencia es que en este caso el paso de reconfiguración no es el último que va a darse en el nodo emisor, por lo que el maestro enviará seguidamente otro mensaje STEP cuando esté en condiciones de empezar el próximo paso. El contenido es idéntico al presentado en el mensaje anterior.

- *JOIN*. Estos mensajes son difundidos por nodos externos al cluster cuando arrancan e intentan incorporarse a él. Se emiten dentro del estado inicial. Los campos `msg_contents` y `msg_seqnum` no son utilizados.

- *NEWMEM*. Este mensaje es difundido por el maestro a sus esclavos cuando empieza el estado pasos. El campo `msg_contents` se utiliza para mantener la lista de los identificadores completos (es decir, con el número de encarnación acompañando al número fijo de nodo) de todos los nodos que forman la nueva configuración del grupo. Los nodos esclavos deberán contestar este mensaje, una vez hayan efectuado las notificaciones oportunas, mediante un mensaje ENDS con número de paso cero.

  Cuando un nodo recibe o envía un mensaje NEWMEM se activa el envío de mensajes PING hacia su vecino superior según el nuevo conjunto de pertenencia que deberá haber instalado.

  El campo `msg_seqnum` de este mensaje incorpora el número de secuencia de reconfiguración que debe establecerse en cada nodo. Este número de secuencia se utiliza para descartar mensajes obsoletos generados en reconfiguraciones anteriores.

- *PING*. Cuando los nodos están en el estado de pasos o en el de monitorización, envían periódicamente un mensaje PING a su vecino con identificador superior en el anillo lógico que se ha construido tras recibir el mensaje NEWMEM.

  Este mensaje no utiliza el campo `msg_contents`.

- *SETMEM*. Una vez el nodo iniciador ha difundido un mensaje CHANGE y ha transcurrido un cierto tiempo, construye el nuevo conjunto de pertenencia en función de las respuestas recibidas en forma de mensajes ALIVE. Este conjunto de pertenencia es enviado al nuevo nodo maestro, escogido como aquél con identificador fijo más bajo de entre los nodos activos, en un mensaje SETMEM.

```
 1:  algorithm hmm;
 2:  type
 3:      state_t = ( INIT, STEPS,
 4:                  MONITORING, RECONF );
 5:      role_t = ( MASTER, SLAVE,
 6:                  BEGINNER, UNKNOWN );
 7:  var
 8:      stage : state_t;      (* Estado actual.           *)
 9:      thisid : node_t;      (* Ident. del nodo local.   *)
10:      step : integer;       (* Número de paso.          *)
11:      masterid : node_t;    (* Ident. del nodo maestro. *)
12:      members : nodeset_t;  (* Conjunto de pertenencia. *)
13:      seqnum : seqnum_t;    (* Número de configuración. *)
14:      role : role_t;        (* Papel de este nodo.      *)
15:  begin
16:      stage := INIT;
17:      masterid := thisid;
18:      seqnum := -1;
19:      step := 0;
20:      while true do
21:         case stage of
22:         INIT:         members := empty_set;
23:                       add_member( members, thisid );
24:                       role := MASTER;
25:                       st_init;
26:         STEPS:        st_steps;
27:         MONITORING:   st_monitoring;
28:         RECONF:       st_reconf;
29:         esac;
30:  end;
```

Figura 3.3: Autómata principal del protocolo HMM.

Este mensaje hace que se pase del estado de reconfiguración al de pasos. Su campo `msg_contents` tiene el mismo contenido que en los mensajes NEWMEM: la lista de los identificadores de nodo que formen el conjunto actual.

- *STEP*. El mensaje STEP es difundido por el nodo maestro a sus esclavos cuando un nuevo paso de reconfiguración debe ser iniciado dentro del estado pasos. El campo `msg_contents` contiene el número de paso que va a iniciarse.

Los *mensajes obsoletos* pueden detectarse y descartarse utilizando el campo `msg_seqnum` de cada mensaje. Para ello, el nodo receptor compara este campo con el número de secuencia local que fija el nodo maestro cada vez que entra en el estado de pasos y que es adoptado de inmediato por todos los nodos esclavos. Para aceptar un mensaje, el número de secuencia local y el que lleva el mensaje deben coincidir. En cualquier otro caso, el mensaje es rechazado, a menos que sea un mensaje NEWMEM con el que explícitamente se modifica el número de secuencia local.

Algoritmo

Una vez presentados los diferentes papeles, estados y mensajes utilizados en el protocolo, iniciaremos su descripción detallada. Para ello se ha descompuesto el algoritmo en función de los estados por los que deba pasar un nodo.

Para empezar, se presenta en la figura 3.3 el algoritmo principal donde se declaran las variables globales y se dan a conocer los nombres de los subalgoritmos asociados a cada estado. Además de esto, existen algunos temporizadores que no van a mostrarse en los textos de los algoritmos y que se utilizan para detectar pérdidas de los mensajes STEP, ENDS y ENDP, reenviándolos si fuera necesario.

Como puede observarse en las líneas 16 a 25, el nodo empieza en el estado inicial y adopta de inmediato el papel de maestro. El número de secuencia se establece en un valor negativo de forma que en el primer incremento que se realice en el estado de pasos adquiera el valor cero o el valor propuesto por el maestro que haya activo en ese momento. El conjunto de pertenencia únicamente contiene al nodo local y el número de paso de reconfiguración se fija a cero.

El resto de estados se describe a continuación:

Estado inicial: La figura 3.4 muestra el algoritmo utilizado en el estado inicial. En él, el nodo local construye un mensaje JOIN y lo difunde (líneas 6 a 8). Posteriormente, instala un temporizador (línea 9) que se agota cuando ha transcurrido el periodo utilizado para enviar los mensajes JOIN. En la línea 10 tenemos el inicio de un buble que sólo terminará cuando haya transcurrido el periodo de los mensajes JOIN (y, por tanto, haya que enviar el siguiente) o se haya recibido una contestación en forma de mensaje NEWMEM por parte del nodo maestro que ya existiese en el cluster. Hay que advertir que la instrucción de espera situada en la línea 11 únicamente se supera cuando se ha recibido un mensaje o ha vencido el temporizador. En este último caso la ejecución de este subalgoritmo termina, volviendo al principal descrito anteriormente, pero como el estado no ha cambiado se volverá a entrar en éste y se volverá a difundir el mensaje JOIN.

Si la espera ha terminado debido a la recepción de un mensaje (línea 13), HMM pasa a comprobar qué tipo de mensaje se ha recibido. En este estado únicamente interesan dos tipos de mensaje: los JOIN enviados por otros nodos que también están intentando formar un cluster y los NEWMEM que envía el maestro actual aceptando al nodo que intentaba incorporarse al grupo. Este segundo caso es comprobado en la línea 14 y conlleva la ejecución de las

líneas 16 a 20. En ellas, el nodo local cambia su rol al de esclavo (recuérdese que inicialmente todos los nodos adoptan el papel de maestro), se almacena en la variable `masterid` la identidad del nodo maestro que acaba de responder y se cambia el estado de pasos. Posteriormente se encola de nuevo el mensaje para que sea procesado en el nuevo estado hacia el que se transitará y también se fija la variable `elapsed` para que no sea necesario entrar de nuevo en el bucle de espera.

```
 1:  algorithm st_init;
 2:  var
 3:      theMsg :  msg; elapsed :  boolean;
 4:  begin
 5:      elapsed := FALSE;
 6:      theMsg.msg_kind = JOIN;
 7:      theMsg.msg_sender = thisid;
 8:      bcast( theMsg );
 9:      install_timer( join_time );
10:      while not elapsed do begin
11:          wait_for event;
12:          case event of
13:          recv( theMsg ):
14:              if theMsg.msg_kind = NEWMEM
15:              then begin
16:                  role := SLAVE;
17:                  masterid := theMsg.msg_sender;
18:                  stage := STEPS;
19:                  requeue_message;
20:                  elapsed := TRUE;
21:              end else if ( theMsg.msg_kind = JOIN )
22:              then if ( theMsg.msg_sender > thisid )
23:                  then begin
24:                      stage := STEPS;
25:                      add_member( members,
26:                          theMsg.msg_sender );
27:                  end else begin
28:                      stage := INIT;
29:                      elapsed := TRUE;
30:                  end;
31:          join_timeout:
32:              elapsed := TRUE;
33:          esac;
34:      end;
35:  end;
```

Figura 3.4: Algoritmo del estado inicial.

Por contra, si el mensaje recibido es un JOIN y no hay ningún mensaje NEWMEM en la cola de recepción, se pasa a procesar ese JOIN. Esta situación se dará cuando todavía no haya ningún cluster formado. Aquí, el receptor de estos mensajes JOIN compara el identificador

del emisor del mensaje con el suyo propio. Si el local es más bajo y no se ha recibido ningún mensaje JOIN de algún nodo con identificador más bajo (esto es lo que mantiene ahora la variable `role`, véase la condición de las líneas 22 y 23), este nodo incluye el identificador del otro en el conjunto de pertenencia. De momento se establece que habrá que cambiar al estado de pasos cuando se agote el periodo del mensaje JOIN difundido por el nodo local, pero como no se modifica el valor de la variable `elapsed` se admiten recepciones de otros mensajes JOIN dados por otros nodos que también estén arrancando ahora.

Si la comprobación de las líneas 22 y 23 falló eso indicará que se ha recibido un mensaje JOIN de un nodo con identificador más bajo que el local y ese nodo externo debería ser el nuevo maestro (o al menos se sabe que el nodo local no podrá ser el maestro, si lo será o no el emisor del mensaje que se acaba de recibir poco importa). Por tanto, en las líneas 28 y 29 se vuelve a solicitar la reentrada en el estado inicial (Puede que durante el periodo de espera se hayan recibido otros mensajes JOIN con identificadores mayores y que se hubiera anotado que había que pasar al estado de pasos. Eso habrá que deshacerlo ahora.) y se pasará de inmediato a abandonar el bucle de espera.

Estado de pasos: El algoritmo seguido en el estado de pasos se muestra en la figura 3.5. Una vez un nodo ha entrado en este estado, pasa a estructurar el conjunto de pertenencia en un anillo lógico siguiendo un orden creciente de los identificadores fijos de nodo y se inicia el envío periódico de mensajes PING en la línea 5. Cada nodo envía un mensaje de este tipo al vecino con identificador superior.

Las líneas 6 a 40 contienen la comprobación del rol que ejercita el nodo local y el algoritmo que seguirá si es el nodo maestro. Por cada paso que se inicie, el nodo maestro debe construir un mensaje STEP o NEWMEM y difundirlo a los nodos miembros actuales. Para ello se utiliza el procedimiento `multicast` facilitado por el protocolo de transporte no fiable.

En la línea 8 se verifica si el nodo maestro se encuentra en el primer paso o ya ha superado éste. Si todavía está en el paso cero, deberá incrementar el número de secuencia y construir y emitir un mensaje NEWMEM donde depositará la información referente al conjunto actual de pertenencia. Esto aparece en las líneas 10 a 12. Si por el contrario, el paso cero ya ha sido superado, deberá emitirse un mensaje STEP en cuyo contenido se especificará que número de paso se desea iniciar (líneas 14 y 15). En las líneas 17 a 19 aparecen las instrucciones comunes a ambos tipos de envío.

Posteriormente, en la línea 20 el maestro notifica localmente a los componentes que así se hayan registrado acerca del nuevo conjunto actual de pertenencia. El procedimiento `notify_step` realiza esta labor de notificación consultando en una tabla qué componente debe invocarse en cada paso de reconfiguración.

Cuando estas tareas han concluido, el maestro pasa a esperar un evento (línea 21). Si todos los miembros han contestado con mensajes ENDS o ENDP, el caso `all_answers_gotten` es tomado. En él simplemente se incrementa el número de paso y se vuelve a entrar en este mismo estado. Nótese que para aceptar cualquier mensaje, a excepción de los mensajes JOIN, el número de secuencia del mensaje debe coincidir con el número de secuencia mantenido en el monitor del nodo receptor.

```
 1:algorithm st_steps;
 2:var
 3:  theMsg :  msg;
 4:begin
 5:  enable_pings;
 6:  if role = MASTER
 7:  then begin
 8:    if step = 0
 9:    then begin
10:      theMsg.msg_kind = NEWMEM;
11:      theMsg.msg_contents=members;
12:      seqnum := seqnum + 1;
13:    end else begin
14:      theMsg.msg_kind = STEP;
15:      theMsg.msg_contents = step;
16:    end;
17:    theMsg.msg_seqnum = seqnum;
18:    theMsg.msg_sender = thisid;
19:    multicast( theMsg );
20:    notify_step;
21:    wait_for event;
22:    case event of
23:    all_answers_gotten:
24:      step := step + 1;
25:    all_steps_concluded:
26:      step := 0;
27:      stage := MONITORING;
28:    join_received:
29:      add_member( members,
30:        msg_sender );
31:      step := 0;
32:    ping_timeout:
33:      step := 0;
34:      role := BEGINNER;
35:      stage := RECONF;
36:    change_received:
37:      step := 0;
38:      role := UNKNOWN;
39:      stage := RECONF;
40:    end;
41:  end else begin (*Esclavo.*)
42:    wait_for event;
43:    case event of
44:    newmem_received:
45:      set_members( received_msg,
46:        members, seqnum );
47:      notify_step;
48:      send_ends_or_endp;
49:      step := step + 1;
50:    step_received:
51:      notify_step;
52:      send_ends_or_endp;
53:      step := step + 1;
54:    join_received:(*Ignorar.*);
55:    ping_timeout:
56:      step := 0;
57:      role := BEGINNER;
58:      stage := RECONF;
59:    change_received:
60:      step := 0;
61:      role := UNKNOWN;
62:      stage := RECONF;
63:    end;
64:    if steps_concluded
65:    then begin
66:      step := 0;
67:      stage := MONITORING;
68:    end;
69:  end;
70:end;
```

Figura 3.5: Algoritmo del estado de pasos.

Cuando todos los miembros han enviado su mensaje ENDP ya no son necesarios más pasos de reconfiguración (líneas 25 a 27) y se pasa al estado de monitorización.

Si se recibe un mensaje JOIN, la reconfiguración actual se da por terminada y se inicia de inmediato una nueva. Para ello basta con poner a cero el número de paso (línea 31), añadir al emisor del mensaje JOIN en el grupo de pertenencia (líneas 29 y 30) y volver a entrar al estado de pasos con lo que se iniciará la nueva reconfiguración.

Otros motivos para abandonar la reconfiguración actual son las pérdidas de mensajes PING y la llegada de un mensaje CHANGE enviado por un iniciador. En ambos casos la salida se debe al fallo de una o más máquinas integrantes del conjunto actual de pertenencia.

Cuando se detecta la pérdida de un mensaje PING (líneas 32 a 35), en la práctica nuestro algoritmo incrementa un contador de mensajes perdidos (que se pone a cero cada vez que

se recibe un PING) y si llega a cierto valor prefijado se pasa a ejecutar las instrucciones que aparecen en las líneas 33 a 35. Lo que debe hacerse aquí es simplemente adoptar el papel de iniciador y cambiar al estado de reconfiguración. Además, el valor del contador de pasos de reconfiguración se pone a cero para iniciar adecuadamente la nueva entrada en este estado cuando esto deba ocurrir.

La llegada de un mensaje CHANGE emitido por otro iniciador (líneas 36 a 39) provocará también el cambio hacia el estado de reconfiguración. La única diferencia respecto al caso anterior aparece en el cambio de papel que debe realizar el maestro. En este caso ha de pasar al estado indefinido.

Si el nodo estaba desempeñando el papel de esclavo, deberá utilizar las instrucciones que se encuentran entre las líneas 41 y 69. Aquí se empieza esperando directamente un evento externo, sin emitir ningún mensaje (línea 42).

Los posibles eventos aparecen seguidamente. Para empezar, tenemos la recepción de un mensaje NEWMEM. Esto conllevará la modificación del conjunto actual de pertenencia y del número de secuencia (líneas 45 y 46). A continuación se notifica el cambio a los componentes interesados en este paso de reconfiguración y se contesta al maestro con un mensaje ENDP o ENDS, dependiendo de si era el último paso registrado o no, respectivamente. Finalmente se incrementa el número de paso.

El proceso de un mensaje STEP es muy similar al anterior. La única diferencia consiste en que en lugar de utilizar el procedimiento `set_members` para modificar el conjunto de pertenencia y el número de secuencia, aquí se verifica que el número de paso asociado al mensaje recibido concuerde con el número de paso que mantiene el monitor en su variable `step`. Si es así, el monitor ejecutará las líneas 51 a 53, efectuando las mismas acciones que en las líneas 47 a 49 del mensaje NEWMEM.

Los mensajes JOIN no deben ser procesados por un nodo esclavo, por lo que no son atendidos. Los casos de pérdida de mensajes PING (líneas 55 a 58) y de llegada de un mensaje CHANGE (líneas 59 a 62) se procesan exactamente igual que en el caso de esos mismos eventos para un nodo maestro, por lo que no hace falta volverlos a explicar aquí.

Finalmente, en las líneas 64 a 68 se comprueba si el número de paso actual indica que ya han concluido los pasos de reconfiguración en el nodo local. De ser así, se cambia al estado de monitorización.

Estado de monitorización: El algoritmo seguido en el estado de monitorización se muestra en la figura 3.6. Debe tenerse en cuenta que el procedimiento `enable_pings` que aparecía en la línea 5 del estado de pasos creó un hilo de ejecución que periódicamente enviaba mensajes PING a su vecino superior. Ese hilo de ejecución todavía funciona en este estado.

En las líneas 3 a 25 hay un bucle que sólo termina cuando se realiza un cambio de estado. En este bucle se van esperando (línea 5) y procesando eventos.

El primer evento importante es la recepción de un mensaje NEWMEM (líneas 7 a 9). Este mensaje sólo puede ser recibido por un nodo cuyo rol sea esclavo y conducirá a éste hacia el estado de pasos, reencolando el mensaje para procesarlo posteriormente en ese nuevo estado. Nótese que no resulta necesario efectuar un cambio del rol del nodo.

```
 1: algorithm st_monitoring;
 2: begin
 3:   while stage = MONITORING do
 4:   begin
 5:     wait_for event;
 6:     case event of
 7:     newmem_received:
 8:       stage := STEPS;
 9:       requeue_message;
10:     ping_timeout:
11:       role := BEGINNER;
12:       stage := RECONF;
13:     join_received:
14:       if role = MASTER
15:       then begin
16:         add_member( members,
17:           msg_sender );
18:         stage := STEPS;
19:       end;
20:     change_received:
21:       step := 0;
22:       role := UNKNOWN;
23:       stage := RECONF;
24:     esac;
25:   end;
26: end;
```

Figura 3.6: Algoritmo del estado de monitorización.

Si se detecta una secuencia de fallos en la recepción de los mensajes PING emitidos por el vecino inferior (líneas 10 a 12), el nodo local adopta el papel de iniciador y se pasa al estado de reconfiguración.

```
 1: algorithm st_reconf;
 2: begin
 3:   disable_pings;
 4:   members := empty_set;
 5:   if role = BEGINNER
 6:   then begin
 7:     broadcast_change;
 8:     add_member( members,
 9:         thisid );
10:     install_timer(reconf_time);
11:     repeat
12:       wait_for event;
13:       case event of
14:       alive_received:
15:         add_member( members,
16:             msg_sender );
17:       change_received:
18:         reply_alive;
19:       reconf_timeout:
20:         masterid := getMaster(
21:             members );
22:         send( masterid,
23:             setmem_msg );
24:       setmem_received:
25:         role := MASTER;
26:         stage := STEPS;
27:         set_members( members,
28:                 msg_contents );
29:       newmem_received:
30:         role := SLAVE;
31:         stage := STEPS;
32:         requeue_message;
33:       esac;
34:     until stage <> RECONF;
35:   end else begin
36:     reply_alive;
37:     repeat
38:       wait_for event;
39:       case event of
40:       change_received:
41:         reply_alive;
42:       setmem_received:
43:         role := MASTER;
44:         stage := STEPS;
45:         set_members( members,
46:                 msg_contents );
47:       newmem_received:
48:         role := SLAVE;
49:         stage := STEPS;
50:         requeue_message;
51:       esac;
52:     until stage <> RECONF;
53:   end;
54: end;
```

Figura 3.7: Algoritmo del estado de reconfiguración.

Si se recibe un mensaje JOIN y el nodo es el maestro del grupo (líneas 13 a 19), el emisor

de tal mensaje es añadido al grupo y se cambia al estado de pasos para iniciar la reconfiguración. Nótese que si el nodo que recibe este tipo de mensajes no es el maestro no debe efectuar ninguna acción en respuesta a tal recepción.

Por último, los mensajes CHANGE (líneas 20 a 23) se procesan exactamente igual que en el estado anterior.

Estado de reconfiguración: La figura 3.7 muestra el algoritmo empleado en el estado de reconfiguración. Para empezar (línea 3) ya que se está realizando una reconfiguración del conjunto de pertenencia, debe inhabilitarse el envío y recepción de mensajes PING, cosa que se consigue mediante el procedimiento `disable_pings` que para al temporizador y al hilo correspondientes. Además, independientemente del papel que se esté llevando a cabo, el nodo vacía su conjunto de pertenencia en la línea 4.

El resto del código depende del rol asignado al nodo. Si es un iniciador, ejecuta las instrucciones entre las líneas 6 y 34, mientras que en otro caso (papel indefinido) ejecutaría el código de las líneas 35 a 53. Debe tenerse en cuenta que pueden darse múltiples fallos simultáneos y que a consecuencia de ello puede haber múltiples iniciadores en una misma reconfiguración.

Para el caso de los iniciadores, éstos empiezan difundiendo un mensaje CHANGE (línea 7). En las líneas 8 y 9 añaden el identificador del nodo local al conjunto de pertenencia que se había vaciado previamente. A continuación, en la línea 10 se instala un temporizador para fijar un tiempo límite para la recepción de respuestas ante la difusión del mensaje CHANGE realizada previamente.

Entre las líneas 11 y 34 tenemos un bucle que terminará cuando el estado sea diferente al de reconfiguración, cosa que podrá suceder a medida que vayan dándose eventos, los cuales son esperados en la línea 12 y procesados en las siguientes. Así, si se recibe un mensaje ALIVE, el emisor del mensaje es añadido al conjunto de pertenencia que se está formando (líneas 15 y 16). Si otro iniciador ha emitido un mensaje CHANGE, se contesta con el correspondiente mensaje ALIVE en la línea 18. Con ello, cualquier iniciador podrá recolectar todas las respuestas de los nodos activos.

En las líneas 20 a 23 encontramos el proceso a realizar cuando ha finalizado el plazo de recepción de mensajes ALIVE. En ese caso se consulta el conjunto actual de pertenencia que acaba de formarse para elegir entre sus miembros al nuevo maestro. Esto se realiza mediante la función `getMaster` que devuelve el identificador del nodo más bajo de entre todos los activos. Una vez hecho esto, se le envía un mensaje SETMEM con el nuevo conjunto de pertenencia. No es necesario realizar un cambio de estado. Cuando el nuevo maestro difunda su mensaje NEWMEM, o si es el propio nodo iniciador el maestro, cuando éste reciba el SETMEM que se ha enviado a sí mismo, se cambiará al estado de pasos y se adoptará el papel apropiado.

En la línea 25 empieza el caso de la recepción de un mensaje SETMEM. Esto puede deberse a que otro iniciador ha podido recoger las respuestas antes que éste y ha visto que el nodo local era el que debía elegirse como nuevo maestro. Por ello, el papel cambia a maestro y el estado será el de pasos. Antes de abandonar este caso se copia el contenido del mensaje para formar el conjunto de pertenencia que deberá gestionar este nuevo maestro.

En las líneas 30 a 32 se procesa el caso de la recepción de un NEWMEM. Esto sólo podrá darse cuando algún otro iniciador haya terminado antes la fase de reconfiguración y haya enviado el mensaje SETMEM al correspondiente maestro y éste haya difundido el mensaje que se acaba de recibir. Obviamente, el nodo que recibe este NEWMEM debe abandonar el estado de reconfiguración y acatar los pasos de reconfiguración que dicte el nuevo maestro. Para ello cambia al estado de pasos y adopta el papel de esclavo, reencolando el mensaje que acaba de recibir para procesarlo acto seguido en el estado al que se dirige.

El proceso que debe seguirse en un nodo con papel indefinido es muy similar al descrito para el iniciador. Las diferencias consisten en que ahora no hay necesidad de ir formando un conjunto de pertenencia, ni difundir un mensaje CHANGE al empezar tal proceso. Lo único que se deberá hacer será contestar de inmediato con un mensaje ALIVE al nodo que nos envió un CHANGE y nos obligó a pasar al estado de reconfiguración. El proceso de los mensajes CHANGE, SETMEM y NEWMEM recibidos por un nodo indefinido es exactamente igual al visto para el caso del iniciador, por lo que no vale la pena repetir su descripción.

3.4.4 Identificadores

En la sección 2.3 se dijo que este algoritmo de pertenencia podría utilizarse para facilitar un modelo de fallos de parada. Para ello resultaba necesario que los *identificadores de nodo* que empleasen los componentes de nuestro sistema cambiasen con cada "encarnación" del nodo; es decir, que se incrementase un *número de encarnación* cada vez que el nodo falle y se recupere, y que se añada dicho número de encarnación como un sufijo del *identificador fijo* de esa máquina.

Nuestro algoritmo no necesita utilizar estos números de encarnación para funcionar correctamente, pero ya que otros componentes que dependen de él van a utilizarlos es conveniente integrar su soporte en los servicios proporcionados por el monitor de pertenencia. Para ello basta con que cada monitor tenga acceso a almacenamiento estable donde pueda guardar entre cada una de sus ejecuciones el número de encarnación adecuado. La idea es que cuando el monitor de pertenecia se inicie, deberá leer el número de encarnación, incrementarlo (el valor resultante es el que deberá mantenerse hasta que falle de nuevo) y guardar inmediatamente el nuevo valor en almacenamiento estable.

Con esto se resuelve el problema de que cada nodo mantenga su propio número de encarnación, pero eso no es todo. Para que los componentes que dependan de la información de pertenencia puedan trabajar de manera adecuada no sólo necesitan conocer el identificador de nodo local (que ahora incluye el número de encarnación), sino también los identificadores de nodo del resto de máquinas del cluster. Dichos identificadores también necesitan el número de encarnación de la máquina a la cual se refieran.

Hay dos formas de solucionar esta segunda cuestión. La primera sería confiar enteramente en las decisiones adoptadas por el monitor de pertenencia, que ha de conocer forzosamente cuando una máquina cae y cuando no. Así, para los nodos remotos se podría calcular qué número de encarnación deberán tener y llevar el control de manera independiente en cada máquina. Sin embargo, esto entraña algunos riesgos y es mejor no emplear esta solución. Por ejemplo, puede darse el caso de que un nodo se haya recuperado durante un tiempo muy breve y que no haya logrado integrarse en el cluster durante ese reintento. Localmente, su monitor de pertenencia

habría incrementado su número de encarnación, pero el resto de nodos no sabría nada sobre esto.

La segunda variante únicamente exige que en los envíos utilizados dentro del algoritmo de pertenencia, cada nodo utilice siempre su identificador completo. Es decir, no sólo su identificador fijo, sino éste junto al número de encarnación. Aunque el algoritmo de pertenencia únicamente necesita el identificador fijo, el número de encarnación recibido podrá guardarse y servirá para informar posteriormente a los componentes que así lo soliciten sobre el identificador completo que va a tener dicha máquina en la configuración actual del cluster. Como ya explicamos al describir los campos de un mensaje de pertenencia, HMM utiliza esta segunda variante.

3.4.5 Coste

Si consideramos el número de mensajes que es necesario intercambiar entre los diferentes nodos que componen el sistema, nuestro algoritmo presenta el coste mostrado en la tabla 3.2.

Fase	Número de mensajes	
	Con difusión	**Punto a punto**
Incorporación	1	N
Formación tras incorporación	N	2 * N - 1
Formación tras caída	1 + 2 * N	4 * N - 2
Reconfiguración	(P - 1) * N	(P - 1) * 2 * (N - 1)
Monitorización	N	N

Tabla 3.2: Número de mensajes intercambiados en las diferentes fases del protocolo.

En esta tabla se asume que existen N nodos activos y para el caso de la reconfiguración se realizan P pasos de reconfiguración. Se han dado valores en dos columnas diferentes distinguiendo dos casos. En el primero se asume que la red permite la realización de difusiones en una sola emisión, mientras que en el segundo esto no resulta posible y hay que implantarlas enviando un mensaje independiente a cada nodo.

Las fases que se distinguen en el protocolo de pertenencia son las siguientes. En la fase de *incorporación* se considera el número de mensajes que debe emitir un nuevo nodo para darse a conocer al grupo. En el caso de nuestro algoritmo basta con difundir un mensaje que pueda recibir el maestro actual del grupo.

La fase de *formación* de grupo incluye todos aquellos mensajes que deban intercambiar los nodos para modificar el conjunto de pertenencia una vez se ha detectado una incorporación o una caída. Para el caso de una caída, correspondería a la difusión de un mensaje CHANGE por parte del iniciador, más las N - 1 respuestas dadas por todos los nodos activos excepto el iniciador, más el envío del mensaje SETMEM del iniciador al maestro elegido. A esto, además, hay que sumarle los mensajes que se necesitan para el caso de una incorporación normal. El caso de una incorporación corresponde a la primera ronda de mensajes del estado de pasos. En ella el maestro difunde un mensaje NEWMEM y cada nodo, excepto él mismo, contesta con un mensaje ENDS o ENDP.

La fase de *reconfiguración* incluye los mensajes intercambiados por los nodos para efectuar la notificación a todos los componentes interesados en la información sobre el conjunto de perte-

nencia. En nuestro caso corresponde al número de mensajes STEP emitidos por el maestro, tantos como números de pasos de reconfiguración menos uno, más las respuestas que deben dar los nodos esclavos a tales mensajes: tantas como rondas haya, multiplicadas por los N - 1 nodos que deban contestar.

Finalmente, la fase de *monitorización* incluye los mensajes que son intercambiados en cada ronda de monitorización. En nuestro caso serán N mensajes, pues cada nodo emite un PING que será recibido por su vecino superior.

Como puede observarse, el coste en mensajes de nuestro algoritmo siempre es lineal respecto al número de nodos activos en cada una de las fases estudiadas.

3.4.6 Comparación con otros algoritmos

En la tabla 3.3 se muestra el número de mensajes intercambiados en las fases de formación tras incorporación, formación tras caída y monitorización empleados por los principales algoritmos de pertenencia que ya habíamos presentado en la sección 3.3.2 que empezaba en la página 42. En esta tabla se asume que toda la comunicación va a realizarse punto a punto, sin posibilidad de efectuar difusiones en un único envío. No tiene sentido presentar los valores para la fase de incorporación, pues la práctica totalidad de los algoritmos emplean los mismos mensajes en ella, ni la de reconfiguración, pues en la mayor parte de los casos se considera que esta tarea se desarrolla fuera del algoritmo de pertenencia. En algunos casos, los mensajes necesarios en la fase de monitorización tampoco se han presentado. Esto es debido a que en los trabajos correspondientes no se dice nada acerca de la técnica empleada para detectar los fallos, sino solamente sobre las tareas necesarias para llegar a un consenso sobre ellos.

Protocolo	**Número de mensajes**		
	Formación tras incorp.	**Formación tras caída**	**Monitorización**
Difusiones periódicas [Cri91a]	N^2 - N	N^2 - N	N^2 - N
Lista de espera [Cri91a]	N^2 - N	N^2 - N	N
Vigilancia de vecinos [Cri91a]	N^2 - N	N^2 - N	N
Delta-4 [RVR93]	4N - 4	4N - 4	2N - 2
Totem [AMMS$^+$93]	6N $\rightarrow$ $2N^2$ + 2N	6N	N
Débil [RFJ93]	N	N	2N
Fuerte [RFJ93]	3N - 2	3N - 2	2N
Isis [RB94]	3N - 2	3N - 3 $\rightarrow$ 5N - 5	?
TTP [KG94]	N $\rightarrow$ N^2	N $\rightarrow$ N^2	N $\rightarrow$ N^2
HMM [MGB00]	2N - 1	4N - 2	N

Tabla 3.3: Número de mensajes utilizados en los principales algoritmos.

Como puede observarse, existen algunos algoritmos en los que el número de mensajes en cada fase puede variar entre un caso mejor y uno peor. Por ejemplo, en el algoritmo seguido por Totem en su fase de formación tras incorporación, el número de mensajes depende del número de nodos que decidan incorporarse a la vez. Si únicamente se realiza una incorporación tendremos un coste

de 6N mensajes. Por el contrario, si todos los nodos empezaran a la vez, el coste necesario para integrar a todos ellos en un mismo grupo sería de $2N^2 + 2N$ mensajes. Algo similar ocurre en Isis para el caso de las caídas, en las cuales el caso peor se da cuando cae el nodo coordinador y el mejor cuando caiga cualquier nodo o conjunto de nodos en el que no esté incluido el nodo coordinador. En TTP el coste de cada fase depende del número de rondas que necesite el nodo que caiga o se incorpore para obtener acceso a la red. Si tiene suerte y obtiene el acceso en la siguiente ronda, el coste será únicamente de N mensajes. Por el contrario, si su ronda acaba de pasar, deberá esperar a que el resto de nodos realicen todas sus difusiones en cada una de sus rondas hasta que vuelva a recuperar el turno.

Con todo esto, el protocolo HMM es de los que menos mensajes requieren en cada ronda de monitorización (al igual que el de lista de espera y vigilancia de vecinos de [Cri91a] y el de Totem). También tiene muy buen comportamiento en la formación tras una incorporación, donde únicamente es superado por el algoritmo débil de Rajkumar, Fakhouri y Jahanian [RFJ93], pero este algoritmo utiliza acuerdo eventual y es difícilmente aplicable a entornos similares al nuestro. Su variante fuerte, que es la que garantiza un acuerdo fuerte al igual que nuestro algoritmo, requiere una ronda más de mensajes que en nuestro caso. Por último, en cuanto a los mensajes necesarios en la fase de formación tras una caída, nuestro protocolo está en una posición intermedia.

3.5 Trabajo futuro

Aparte de los protocolos ya mencionados en este capítulo, otros trabajos sobre pertenencia se han dedicado a resolver este mismo problema en redes de área extensa [MFSW95, ACDK98], donde las soluciones suelen ser jerárquicas. En nuestro caso, esto no es directamente aplicable al entorno que hemos descrito y donde se intenta implantar nuestra arquitectura HIDRA. Sin embargo, en un futuro, cuando el desarrollo para un cluster ya esté completado, sí tendría sentido construir estructuras *jerárquicas de clusters* y consultar el conjunto de pertenencia entre múltiples redes interconectadas, llegando así a un entorno similar al mencionado anteriormente.

Otra línea de investigación podría centrarse en la inclusión no sólo de las máquinas del cluster en el grupo a monitorizar, sino también las máquinas clientes. Esto facilitaría las cosas a la hora de controlar las cuentas de referencias mantenidas por clientes externos, cosa que ahora debe efectuarse con representantes internos de tales clientes.

Por tanto, nuestros algoritmos de pertenencia pueden expandirse en un futuro, siguiendo dos aproximaciones. La primera sería desarrollar un algoritmo jerárquico que pudiera tratar los casos de agrupaciones de clusters. La segunda sería una ligera modificación del algoritmo actual donde se permitieran dos clases de nodos miembros, los clientes y los servidores y donde también se eliminaría la restricción de trabajar únicamente con la red privada interna del cluster para expandirlo a la red que interconecte al cluster con sus clientes. En estos dos nuevos entornos podrían darse fácilmente particiones, por lo que los algoritmos resultantes deberían admitirlas. Todo ello deberá resolverse en trabajos futuros.

Capítulo 4

Invocación de objetos

4.1 Introducción

La invocación de objetos replicados depende en gran medida del modelo de replicación que se haya adoptado en el sistema sobre el que se esté trabajando. En el caso de HIDRA se ha adoptado el modelo coordinador-cohorte por lo que en un principio la invocación solamente debe llegar a una de las réplicas que constituyan el objeto a invocar. Posteriormente esa réplica ya se encargará de realizar las actualizaciones de estado sobre el resto, siempre y cuando resulten necesarias (es decir, sólo en el caso de que la invocación haya modificado tal estado) y contestará al cliente que inició tal invocación.

Pero en el modelo coordinador-cohorte hay algunos problemas adicionales que deben contemplarse a la hora de proporcionar el soporte a estas invocaciones a objeto. Estas dificultades surgen del hecho de que en este modelo cada cliente puede elegir a un coordinador diferente, lo cual puede repercutir positivamente en el rendimiento del sistema, al lograrse una mayor concurrencia en el servicio de las peticiones de los clientes, pero esa misma concurrencia debe controlarse de alguna manera y ahí es donde reside el problema. Por tanto, el mecanismo de invocación empleado debe proporcionar los medios necesarios para detectar cuándo una invocación ha completado sus acciones en las réplicas invocadas, independientemente de que la respuesta haya llegado al cliente o no, pues esto es lo que se necesitará saber para dar entrada a las invocaciones generadas por otros clientes que estén en conflicto con la invocación que se estaba siguiendo. Por tanto, como se ve, el protocolo que se utilice para soportar las invocaciones a objetos replicados deberá proporcionar cierto soporte para poder construir con él un mecanismo de control de concurrencia distribuido que sea adecuado para este modelo de replicación. Éste es el tema principal de este capítulo.

Además de la detección de la terminación de la invocación en el dominio servidor, pueden exigirse otras garantías adicionales a este soporte para invocaciones sobre objetos replicados. Algunas de estas garantías serán similares a las proporcionadas por las *transacciones sencillas* y *anidadas* en el área de las bases de datos. Por ello, también es recomendable un estudio de esta área de investigación pues de ella se podrán obtener algunas ideas aplicables a nuestras soluciones.

En esta primera versión de nuestro soporte se ha optado por exigir una estructuración de los servicios replicados que evite la aparición de interbloqueos, por lo que no vamos a entrar en el estudio de este tipo de problemas. Sin embargo, en un futuro próximo se van a eliminar estas restricciones y se dotará a HIDRA del soporte necesario para abortar invocaciones en curso, así

como detectar y resolver los interbloqueos que puedan surgir.

El soporte implantado actualmente en HIDRA recibe el nombre de *protocolo IFO* o *protocolo de invocaciones fiables a objeto* [MGB98b, MGB98c] y se basa en el uso de algunos objetos auxiliares que permiten la detección de la terminación de la invocación en el dominio servidor, así como la recepción de la respuesta en el dominio cliente. Con ello, tanto en un dominio como en otro se pueden detectar casos de fallo que es posible corregir con un conjunto reducido de acciones.

El contenido de este capítulo se estructura como sigue. En la sección 4.2 se explica qué es una invocación de un objeto replicado según el modelo coordinador-cohorte, así como las garantías que debe proporcionar un mecanismo que dé soporte a tales invocaciones. La sección 4.3 describe en detalle el protocolo IFO, estudiando los roles desempeñados por los dominios participantes, los objetos auxiliares empleados, el protocolo en sí y las ventajas e inconvenientes frente a otras soluciones empleadas en otros sistemas. La sección 4.4 se centra en el estudio de los posibles casos de fallo que podrán encontrarse durante la ejecución de una invocación, así como la forma en que son detectados y corregidos. Finalmente, la sección 4.5 describe otros trabajos que guardan cierta relación con el presentado en este capítulo.

4.2 Invocación en el modelo coordinador-cohorte

Las invocaciones en el modelo coordinador-cohorte son en principio iguales a las que se dan en el modelo pasivo, por lo que podría emplearse un soporte similar al propuesto en esos casos. En [BMST93] se dan unos cuantos ejemplos de la forma de organizar las invocaciones en el modelo pasivo.

Sin embargo, no todo es tan fácil como puede parecer en un principio. Tanto el modelo pasivo como el coordinador-cohorte presentan la característica de tener sólo una réplica activa, por lo que resulta bastante sencillo implantar el código de tales réplicas con soporte a múltiples hilos de ejecución, de manera que cada hilo atienda una petición de un cliente diferente y todos ellos puedan sincronizar sus acciones en base a mecanismos locales. Pero en el modelo coordinador-cohorte existe una dificultad adicional y es que aquí cada cliente puede elegir una réplica coordinadora diferente. Con ello, aunque para una determinada petición sólo tengamos una réplica activa, un conjunto de peticiones iniciadas en un determinado intervalo pueden trabajar con un conjunto de réplicas activas bastante grande. Hay que sincronizar la ejecución de tales peticiones en ese conjunto de réplicas y para ello se necesitará, además, un mecanismo de control de concurrencia distribuido.

Vistas las diferencias respecto al modelo pasivo, pasemos ahora a ver en detalle las características que presenta una invocación en este modelo de replicación.

4.2.1 Características

Al igual que en el modelo pasivo, en una invocación de este tipo únicamente se trabaja activamente con una sola réplica, conocida aquí como *réplica coordinadora*. Esta réplica debe procesar localmente la petición efectuada por el cliente y, una vez terminada ésta, si ha habido modificaciones en el estado del objeto, deberá enviar las actualizaciones de estado necesarias hacia el resto de réplicas del objeto, conocidas como *réplicas cohortes*.

De esta manera, como cada cliente puede tener una réplica coordinadora diferente y una misma réplica puede adoptar ambos papeles, eligiendo entre uno y otro según hacia donde haya dirigido su petición un cliente, en este modelo de replicación no resulta necesario efectuar promociones de las réplicas en caso de fallo. En el caso del modelo pasivo, esto debía hacerse para poder reemplazar a la *réplica primaria*.

A diferencia del modelo activo, por tener en el coordinador-cohorte sólo una réplica activa por cada invocación no van a ser necesarios los *filtrados de peticiones* ni los *de respuestas*. En el modelo activo, se necesitaba un filtrado de peticiones para que un objeto replicado bajo este modelo pudiera invocar los servicios de algún objeto externo. Debe tenerse en cuenta que en tal modelo surgirían tantas peticiones como réplicas hubiese y que debe darse algún soporte para que el receptor de tales invocaciones únicamente procese una de ellas y devuelva el resultado a todas las réplicas que le hayan invocado. Una tarea nada sencilla de resolver. Otra alternativa sería conseguir que todas las operaciones invocadas por un objeto replicado activo fueran *idempotentes*, es decir, que siempre devolvieran la misma respuesta y provocaran los mismos cambios, independientemente del número de veces que fueran utilizadas.

El filtrado de respuestas tiene sentido en el otro extremo de la comunicación. Es decir, en la devolución de respuestas al cliente que envió tal petición. Nuevamente, el objeto replicado activo devolverá tantas respuestas como réplicas existan. Vuelve a ser necesario un soporte que filtre tales respuestas y entregue una sola al cliente. El criterio y modo de realizar la elección de la respuesta a proporcionar dependerá básicamente del modelo de fallo que se haya asumido en el sistema [Sch93a]. Por ejemplo, con un modelo de fallo parada puede entregarse la primera respuesta e ignorar todas las demás, mientras que con un modelo de hasta k fallos bizantinos, debería haber al menos 2k+1 réplicas y retornar la primera respuesta que haya sido repetida k+1 veces.

Lo que sí va a ser necesario proporcionar en las invocaciones del modelo coordinador-cohorte será algún mecanismo de *detección de terminación* en los dominios servidores para facilitar la implantación de un mecanismo de control de concurrencia distribuido si se pretende que éste tenga una granularidad de operación. Esta será una de las garantías a exigir al mecanismo de invocaciones fiables que se propone en este capítulo.

4.2.2 Garantías a proporcionar

En el sistema HIDRA, el mecanismo de invocaciones fiables que aquí se presenta debe proporcionar una serie de garantías que conlleven la buena terminación de tales invocaciones. Estas garantías son:

- *Progreso*. Una invocación no podrá ser abortada una vez haya modificado el estado de alguna réplica. Si se sigue estrictamente el modelo presentado anteriormente, tal réplica será la coordinadora para una invocación que implique un cambio en el estado del objeto.

  Esto implica que una vez la réplica coordinadora haya visto modificado su estado, todas las demás réplicas deberían conseguir que su estado coincida con el de la coordinadora para esa misma petición. Además, esto debe darse incluso si falla alguna réplica mientras se está sirviendo la petición; a menos que sea la coordinadora quien falle y todavía no haya iniciado ninguna actualización de estado sobre las demás.

- *Transparencia* para el cliente. El programador de un objeto cliente no debe observar ninguna diferencia entre el código que debe utilizar para invocar a un objeto simple (no replicado) y el que utiliza para invocar a un objeto replicado utilizando el mecanismo IFO. Para ello, todo el soporte a invocaciones fiables será implantado dentro del ORB y generado automáticamente cuando se inicie una invocación sobre el objeto replicado.

  El programador del objeto replicado sí tendrá que codificar de manera distinta a tales objetos que a los objetos simples, pero no así quien desarrolle el código de un cliente.

- *Resultados retenidos*. Las réplicas servidoras de cada petición deben almacenar temporalmente los resultados que la operación invocada deba proporcionar. Esto se mantendrá hasta que se sepa con certeza que el cliente ha recibido tales resultados. De esta manera se pueden atender de inmediato, sin necesidad de reejecutarlas desde el inicio, aquellas repeticiones de la invocación que haya habido que generar debido al fallo de la réplica coordinadora durante el período comprendido entre la realización de la primera actualización de estado ligada a la invocación y la devolución de su resultado al cliente. Nótese que en estos casos el cliente, una vez detectado el fallo de la réplica coordinadora, reintentará la operación eligiendo otra réplica coordinadora. Esta segunda réplica coordinadora, si ha sabido algo acerca del anterior intento, habrá obtenido una copia de los resultados proporcionados por la coordinadora que falló, ligándolos a un identificador de invocación. Cuando se recibe el reintento etiquetado con la misma identificación, la nueva coordinadora obtendrá los anteriores resultados y, sin necesidad de reejecutar la operación, contestará de inmediato al cliente.

  Con este soporte se evita tener que reprocesar las peticiones, con el peligro que esto conllevaría para la integridad y consistencia de la información mantenida por el objeto (a menos que las operaciones fueran idempotentes, cosa que no se va a exigir en ningún momento). Además, se reduce el tiempo necesario para atender los reintentos de ejecución de una operación en caso de fallo.

- *Aislamiento relajado*. En HIDRA, por la organización en niveles que se exige a la hora de estructurar los objetos replicados, no deben aparecer situaciones de interbloqueo. Por tanto, en esta primera versión de HIDRA no tiene sentido abortar invocaciones. Como resultado de ello, no es necesario garantizar la propiedad de aislamiento de los cambios parciales que vaya produciendo cada invocación. Es decir, es posible que una invocación vea y utilice aquellos cambios en el estado que hayan ocasionado otras invocaciones que todavía no hayan finalizado. Sin embargo, esto no representa ningún problema porque también se está garantizando la propiedad de progreso y todas estas invocaciones finalizarán correctamente, sin *abortos*.

  Debe advertirse que lo dicho en el párrafo anterior implica una relajación del aislamiento para cadenas de invocaciones, que en un entorno transaccional deberían protegerse mediante transacciones anidadas. Es en ese entorno donde se relaja el aislamiento porque se permitirá que otras invocaciones consulten los cambios realizados por invocaciones situadas al final de una cadena antes de que las que estén al inicio de ella (y que incluían a estas últimas) hayan finalizado. Sin embargo, como ya hemos dicho, esto es admisible porque sí garantizamos el progreso y buena terminación de cualquier invocación que haya afectado a más de una réplica.

Además, si consideramos invocaciones aislada, si no se desea permitir la visión y utilización de dichos cambios se podrá impedir utilizando el mecanismo de control de concurrencia facilitado. Es decir, en el ámbito de las invocaciones aisladas (no incluidas en una cadena) se da la posibilidad tanto de relajar el *aislamiento* como de respetarlo, según convenga.

- *Detección de terminación*. Cuando la invocación haya sido completada por todas las réplicas servidoras, nuestro protocolo ha de detectar tal situación para poder facilitar la base necesaria para construir un mecanismo de control de concurrencia distribuido con granularidad de operación.

  Igualmente, debe detectarse cuándo el cliente ha recibido la respuesta proporcionada por la réplica coordinadora para poder liberar en ese momento los resultados retenidos en las réplicas servidoras.

- Tolerancia a fallos. Una invocación fiable debe ser capaz de detectar el fallo de cualquiera de los agentes y objetos auxiliares involucrados en ella, reaccionando de manera apropiada a cada una de estas situaciones de fallo. Los procedimientos a seguir en caso de fallo se describen en la sección 4.4 que empieza en la página 87.

4.3 Protocolo IFO

El protocolo IFO tiene como principal objetivo el proporcionar un mecanismo que dé las garantías presentadas en la sección anterior cuando se invoquen objetos replicados que sigan el modelo coordinador-cohorte. Para ello cada *agente* o *dominio* participante debe desempeñar un papel bien definido en el protocolo, que se describirá en la sección 4.3.1. Además, como se verá posteriormente cada agente genera una serie de objetos auxiliares que participan activamente en el protocolo y que sirven para detectar tanto las situaciones de fallo como las de terminación de cada una de sus fases.

Dentro del protocolo IFO distinguiremos diversas variantes. El protocolo básico se describe en la sección 4.3.3. Esta variante asume un cliente simple, es decir, no replicado y que se invoca una operación que modifica el estado del objeto y devuelve argumentos de salida o resultados, es decir, que necesita un mensaje de respuesta para el cliente.

Otras variantes se describen posteriormente. La primera de ellas corresponde a una situación muy similar a la anterior, pero con el cliente replicado. Se asume aquí que la réplica coordinadora del cliente realiza la invocación sobre un objeto externo. Veremos cómo funciona el protocolo IFO para este caso, comentando las diferencias entre esta variante y el protocolo básico descrito previamente.

La segunda variante corresponde al caso de operaciones unidireccionales. Es decir, operaciones que modifican el estado del objeto pero no devuelven ningún resultado ni argumento de salida y, por tanto, no necesitan un mensaje de respuesta para el cliente.

La última situación que convendrá describir corresponde a las operaciones de sólo lectura. Nuestro soporte podrá conocer qué operaciones son de sólo lectura porque en la especificación utilizada para generar la información de control de concurrencia se podrá indicar qué operaciones tienen esta característica. En cualquier caso, estas operaciones ofrecen la particularidad de que no

necesitan realizar ningún envío de actualización de estado sobre las réplicas cohortes. Veremos cómo se gestionan tales invocaciones en la sección 4.3.6.

4.3.1 Agentes

Los roles que se pueden distinguir para cada uno de los agentes o dominios participantes en una invocación sobre un objeto replicado que siga el modelo coordinador-cohorte son los siguientes:

- *Cliente*. Es el dominio que inicia la invocación. En el caso más general, el cliente queda suspendido tras iniciar la *petición* hasta que recibe la respuesta que le proporciona el coordinador. Sin embargo, puede darse el caso de invocar *operaciones unidireccionales*, donde no se espera *respuesta* ni *argumentos de salida* (como máximo, se podrán utilizar *argumentos de entrada*). En este último caso, el cliente realiza la petición pero continúa de inmediato con su ejecución, sin necesidad de suspenderse.

- *Coordinador*. Es el dominio donde se halla ubicada la réplica coordinadora. Esta es la réplica que recibe directamente la petición efectuada por el cliente y la única que la procesa y actualiza su estado realizando cada una de las tareas asociadas con tal operación. Antes de responder al cliente y sólo si ha habido modificaciones en el estado, esta réplica realizará el envío de uno o más *mensajes de actualización de estado* al resto de las réplicas del objeto, o réplicas cohortes, con el fin de que en base a las indicaciones dadas por esta réplica coordinadora, modifiquen su estado para hacerlo consistente con el de esta primera réplica.

- *Cohorte*. Un dominio cohorte es cada uno de los que mantienen una réplica cohorte que recibe las actualizaciones de estado originadas en la réplica coordinadora. Junto a estas actualizaciones de estado, también recibirá una copia de los resultados que retendrá hasta que haya detectado la correcta recepción de éstos por parte del cliente.

  Cuando las réplicas cohortes participen en una invocación fiable a objeto, se utilizará para acceder a ellas un tipo de referencia especial para la llamada que se está realizando en ese momento. Debe observarse que todas las réplicas que componen un objeto del modelo coordinador-cohorte tienen dos comportamientos distintos, según el rol que adopten. Cuando trabajan como cohortes, las invocaciones realizadas sobre ellas son para completar actualizaciones de estado y dichas invocaciones deben partir desde la réplica coordinadora y llegar a todas las demás. En esa referencia habrá múltiples elementos localizadores y tendrán que emplearse todos ellos a excepción del localizador que apunte a la réplica que inicia esa llamada. Detalles sobre este tipo de referencias y sobre cómo se utilizan y qué características ofrecen pueden encontrarse en [Gal01].

  El soporte final que proporcionará ese tipo de referencias todavía se está estudiando. Puede que se considere oportuno dar soporte a múltiples invocaciones de checkpoint para una misma IFO. En ese caso se garantizará ordenación FIFO de dichas invocaciones. En cualquier caso, nuestro protocolo asume que si hay múltiples invocaciones el mecanismo facilitado por el ORB dará ordenación FIFO, pero si sólo hay un checkpoint por cada IFO nuestro protocolo funciona correctamente aún en esa situación. En ese caso se transmite en la única invocación de checkpoint toda la información que debía incluirse en la primera y en la última de esas invocaciones.

- *Serializador*. Este dominio mantiene un objeto que realiza las tareas de control de concurrencia, suspendiendo la invocación hasta que ya no esté en conflicto con ninguna otra iniciada previamente. El protocolo de control de concurrencia se describirá en detalle en el capítulo 5 que empieza en la página 107.

Cada uno de estos dominios crea o utiliza una serie de objetos auxiliares que son incluidos también en algunos de los mensajes utilizados en este mecanismo de invocación fiable. Estos objetos auxiliares se describen a continuación.

4.3.2 Objetos auxiliares

Para realizar algunas de las detecciones que exigimos al protocolo de invocaciones fiables resulta necesario utilizar una serie de objetos auxiliares que son creados en los niveles internos de nuestro ORB con soporte a replicación. El programador no necesita utilizar directamente estos objetos.

Para poder llevar a cabo estas detecciones se utiliza la cuenta de referencias que era llevada a cabo en nuestro ORB tanto para objetos simples como para replicados. La *notificación de no referencia* es utilizada con este objetivo en este protocolo.

Veamos para qué sirve cada uno de estos objetos auxiliares:

- *RoiID*. Nuestro soporte crea un objeto de este tipo cada vez que una invocación fiable es iniciada dentro del dominio cliente.

  Cuando el código del programa cliente inicia la invocación y pide los servicios del ORB para ello, nuestro ORB crea una instancia de un objeto de este tipo y mantiene tal instancia en el dominio cliente. El objetivo de este objeto auxiliar es identificar a la invocación fiable que se está iniciando. Para ello, el ORB inserta una referencia a este RoiID en el mensaje de petición que transmitirá el ORB para hacer llegar esta invocación al dominio coordinador.

  Si se utilizan clientes replicados, el RoiID también pasará a ser un objeto replicado, pues convendrá tener una réplica de este objeto en cada dominio cliente. De esta forma, si fallara la réplica cliente que inició la invocación, otras réplicas clientes podrán reintentar dicha invocación y podrán utilizar el mismo identificador para que el objeto invocado pueda detectar fácilmente que se trata de un reintento. Si el intento anterior ha terminado satisfactoriamente, que es lo más probable, sus resultados se habrán conservado y serán devueltos de inmediato, sin necesidad de reejecutar la operación.

- *TObj*. Este objeto resulta necesario para detectar cuándo una invocación fiable ha terminado en todas las réplicas de un objeto que actúen como servidoras para una determinada petición. Es también un objeto replicado. Su primera réplica es creada por nuestro ORB en el dominio coordinador cuando el mensaje de petición con una referencia a un determinado RoiID llega a tal dominio. Otras réplicas son creadas en el dominio serializador, cuando se efectúa la llamada para realizar el control de concurrencia y en las réplicas cohortes cuando se realiza el primer envío para actualización de estado. Posteriormente, una vez creadas todas sus réplicas, sus referencias clientes son eliminadas cuando se recibe en los cohortes la última actualización de estado (puede darse el caso de que únicamente exista un envío de actualización de estado, que a estos efectos será considerado tanto el primero como el último) y cuando en el coordinador se devuelve el resultado hacia el cliente. Con ello, todas

sus réplicas reciben la notificación de no referencia y el serializador da paso a otras invocaciones fiables que estuvieran en conflicto con la invocación actual y todavía estuviesen suspendidas.

- *CObj*. Al igual que el TObj, este objeto es creado en el dominio coordinador cuando llega el primer mensaje relacionado con una determinada invocación fiable (con una referencia a un nuevo RoiID). También es un objeto replicado. En este caso, su utilidad reside en detectar cuándo el cliente ha podido recibir el resultado de la invocación que ha realizado sobre el objeto servidor replicado.

  Las réplicas del CObj se crean únicamente en los dominios cohortes cuando reciben las actualizaciones de estado y los resultados que deberán retenerse. Una primera referencia cliente se envía al dominio cliente tras haber creado la primera réplica del CObj en el dominio coordinador. Cuando la invocación ha finalizado en los *dominios servidores* (esto es, en el *dominio coordinador* y en los *cohortes*), se liberan las referencias internas. Finalmente, cuando el cliente ha recibido el resultado, libera la referencia cliente que recibió al ser creada la primera réplica y con ello se genera una notificación de no referencia en cada una de sus réplicas que dará pie a que se descarten los resultados retenidos.

4.3.3 Descripción del protocolo

El *protocolo básico IFO* se muestra en las figuras 4.1 a 4.5. En estas figuras los rectángulos representan referencias a objeto mientras que las cajas con bordes redondeados representan instancias de un determinado objeto.

Los pasos seguidos en esta variante básica son los siguientes (en cada paso se indica entre paréntesis el agente que lo lleva a cabo):

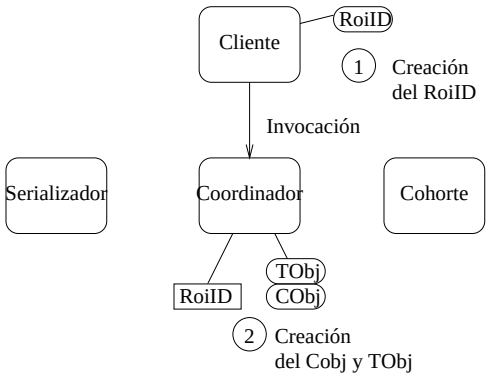

Figura 4.1: Pasos 1 y 2 en una invocación fiable.

1. *Creación del RoiID* (Cliente). Una vez el programa cliente ha iniciado una invocación sobre un objeto replicado que siga el modelo coordinador-cohorte, nuestro ORB detecta tal invocación en el soporte interno de su núcleo y crea en dicho *dominio cliente* un objeto RoiID. Una vez creado este objeto, genera una referencia cliente para él y la inserta en el mensaje que se ha construido para hacer llegar esta petición al dominio coordinador.

   Además, si el cliente no es un objeto replicado, como en el caso que estamos suponiendo,

se pone a cierto un *flag SIMPLE* en la cabecera de tal mensaje para indicar al dominio coordinador que el cliente no estaba replicado.

2. *Recepción de la invocación* (Coordinador). El núcleo del ORB ubicado en el dominio coordinador recibe los mensajes con la petición ligada a esta invocación. Una vez recibidos, y antes de hacer llegar la invocación al objeto invocado, el ORB busca en el mensaje la referencia cliente al RoiID.

   Una vez obtenida la referencia cliente, se comprueba si su RoiID aparece en la lista de invocaciones fiables para las que siguen manteniéndose resultados retenidos en esa réplica. Si es así, se cogen los resultados de la lista, se construye con ellos el mensaje de respuesta y se envía de inmediato al cliente, sin necesidad de reprocesar la petición. En este caso, el protocolo continúa a partir del paso 8 que se explica más adelante.

   Esta situación puede darse debido al fallo previo de la anterior réplica coordinadora elegida por ese mismo cliente, siempre y cuando tal réplica hubiese podido dar ya las actualizaciones de estado a sus cohortes. Uno de esos cohortes sería el coordinador que se ha elegido en este nuevo reintento, el cual podrá encontrar los resultados proporcionados por el anterior coordinador antes de fallar.

   En caso de que el RoiID no pueda encontrarse en la lista de resultados retenidos, el dominio coordinador genera las primeras réplicas de los objetos CObj y TObj y las asocia a la referencia al RoiID que acaba de recibir. Nótese que en ningún caso se ha llegado a entregar la invocación al código de la réplica coordinadora. Todas estas labores son realizadas dentro del núcleo del ORB existente en el dominio coordinador.

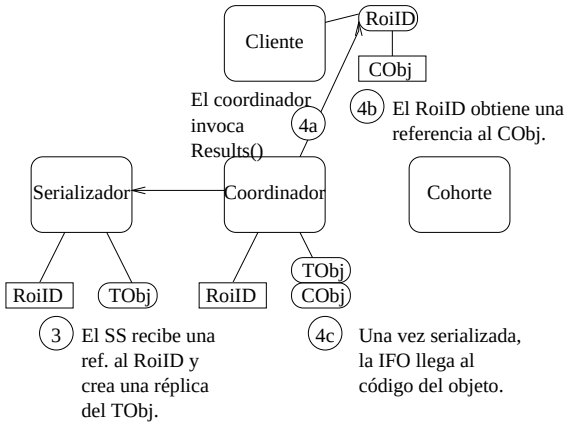

Figura 4.2: Pasos 3 y 4 en una invocación fiable.

3. *Petición de serialización* (Coordinador). El núcleo del ORB en el dominio coordinador realiza una petición sobre el método `Serialize()` del objeto serializador. El serializador recibe entre otros argumentos una referencia al RoiID para identificar la invocación fiable y una referencia al TObj que le servirá para construir una réplica de este último objeto. Con esta información junto a una descripción de la operación invocada, el serializador averigua si la invocación que se intenta realizar está en conflicto con alguna otra ya en curso o encolada previamente. De ser así, suspende esta invocación hasta que todas sus invocaciones en conflicto que la precedan hayan terminado. Cuando esto ocurra, o si no había ninguna

invocación que precediera a ésta, el control es devuelto al dominio coordinador, para que continúe con el servicio de la invocación.

La referencia cliente del TObj es descartada una vez ya se ha creado la réplica de este objeto. De esta manera, cuando todas las réplicas servidoras den por terminada la invocación en curso y liberen las referencias mantenidas en sus propias réplicas del TObj, este objeto recibirá la notificación de no referencia que el serializador interpretará como la terminación de esta invocación en los dominios servidores (ver paso 8, más adelante).

4. *Transferencia del CObj al dominio cliente* (Coordinador). Antes de que la invocación llegue al código de la réplica, el núcleo del ORB en el dominio coordinador realiza una invocación al método `Results()` del RoiID para transferirle una referencia al objeto CObj. Esto es necesario porque de esta manera, cuando los resultados sean devueltos al dominio cliente, éste podrá liberar la referencia que ahora se le acaba de entregar y con ello los CObjs que haya en los dominios servidores, si todo ha ido bien, recibirán una notificación de no referencia que les indicará que podrán descartar los resultados retenidos.

   Cuando la invocación sobre el RoiID ha terminado, el ORB del dominio coordinador entrega finalmente la invocación al código de la réplica y empieza la ejecución de la operación invocada.

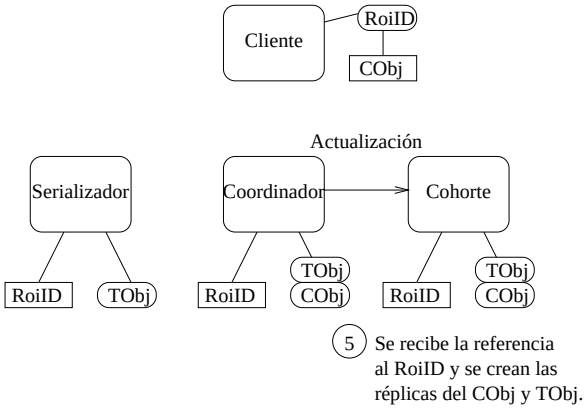

Figura 4.3: Paso 5 en una invocación fiable.

5. *Se realiza la primera actualización de estado* (Coordinador). En la primera llamada a un método de las réplicas cohortes para realizar una actualización de estado, el soporte incluido en nuestro ORB añade al contenido del mensaje referencias sobre los objetos RoiID, CObj y TObj. Cuando el ORB en los dominios cohortes recibe estos mensajes, extrae esas referencias y crea las réplicas de los objetos CObj y TObj a partir de las referencias obtenidas.

   Puede que una invocación sólo necesite realizar una actualización de estado. En ese caso, las tareas que comentamos para este paso 5 y para el siguiente, se realizarían a la vez. Sin embargo, si necesita más de una invocación de actualización de estado, debido a que la operación es muy costosa o requiere la utilización de múltiples objetos externos y vale la pena guardar los resultados parciales obtenidos en cada una de las invocaciones realizadas sobre tales objetos, nuestro soporte únicamente requiere un tratamiento especial en la primera y en la última actualización de estado realizadas dentro de una misma invocación fiable.

Si se da el caso de que la invocación necesita más de un envío de actualización de estado, en el primero de ellos debe adjuntarse una copia de los argumentos de entrada que tenía la invocación original, para permitir que la invocación pueda ser reiniciada por otra réplica en caso de caída simultánea del cliente y del coordinador.

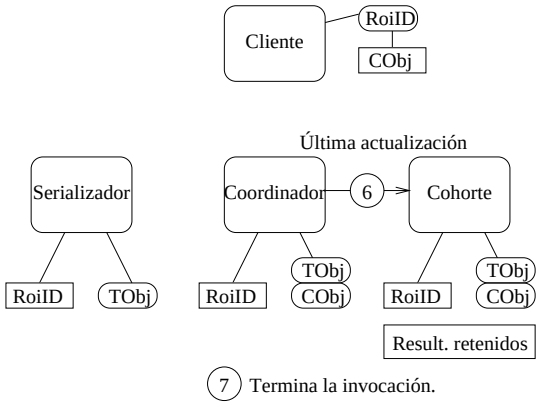

Figura 4.4: Pasos 6 y 7 en una invocación fiable.

6. *Se realiza la última actualización de estado* (Coordinador). En el mensaje que transporte la última actualización de estado iniciada por la réplica coordinadora debe añadirse en un lugar especial una copia de los argumentos de salida y resultados que se devolverán al cliente. Estos argumentos de salida y resultados son copiados por el ORB de los dominios cohortes y retenidos hasta que se detecte que han podido ser obtenidos por el programa cliente. Para ello, nuestro ORB los asocia con el RoiID y el CObj que obtuvo en la primera actualización de estado (de hecho, el RoiID ha tenido que incluirse en todas las invocaciones de actualización de estado para que el ORB pueda saber a qué invocación fiable corresponden).

   Una vez las réplicas cohortes han procesado este último mensaje de actualización de estado, nuestro ORB libera las referencias clientes mantenidas en las réplicas de los objetos TObj y CObj en todos los dominios cohortes.

7. *Termina la invocación en la réplica coordinadora* (Coordinador). Al igual que hicieron los ORBs de los dominios cohortes, cuando termina la invocación en el dominio coordinador hay que liberar las referencias internas de las réplicas del CObj y TObj. Aquí no hay que guardar una copia de los resultados y argumentos de salida si el cliente no está replicado, ya que tales resultados podrían requerirse de nuevo si fallase el cliente y otra réplica del cliente reiniciara la misma operación con este mismo coordinador. Si, por el contrario, sólo existe una réplica cliente y ésta falla, no podrá reintentar nadie la operación con lo que no tiene excesivo sentido mantener los resultados en la réplica coordinadora para este caso en particular.

   Nótese que al liberar la referencia cliente mantenida en esta réplica del TObj ya no queda ninguna referencia sobre este objeto en todo el sistema (las que había en las réplicas ubicadas en los dominios cohortes fueron liberadas en el paso anterior). Con ello, las réplicas del TObj recibirán todas ellas la notificación de no referencia (ver paso 9). Esto es particularmente interesante en el objeto serializador, que en base a ello sabrá que la operación

ha concluido en los dominios servidores y podrá dar paso a otras invocaciones fiables que estuvieran en conflicto con la que acaba de terminar.

Por otra parte, nuestro ORB en el dominio coordinador emitirá finalmente el mensaje con la respuesta para el dominio cliente.

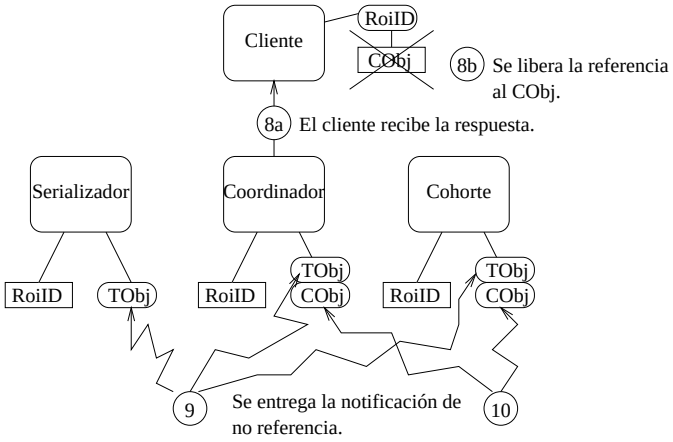

Figura 4.5: Pasos 8 a 10 en una invocación fiable.

8. *El cliente recibe la respuesta* (Cliente). El núcleo del ORB que gestiona las invocaciones en el dominio cliente, antes de entregar la respuesta comprueba a qué RoiID está asociada ésta y entonces libera la referencia cliente sobre el objeto CObj que éste mantenía. Nótese que ésta era la última referencia que quedaba en el sistema sobre ese objeto CObj. Con ello, se habilita la entrega de la notificación de no referencia sobre todas las réplicas de ese objeto (ver paso 10).

   El objeto RoiID también descarta su propia referencia cliente en este momento. Una vez hecho esto, la respuesta es entregada al programa cliente.

9. *Las réplicas del TObj reciben la notificación de no referencia* (Servidores y serializador). Con ello el serializador ya puede dar paso a al menos una de aquellas invocaciones fiables que estuvieran en conflicto con la invocación que acaba de terminar y que hubiesen llegado tras ella. Además, todas las réplicas del TObj son eliminadas y en el serializador también se descarta la referencia cliente del RoiID.

   Nótese que la notificación de no referencia tarda cierto tiempo en entregarse (generalmente la espera es muy breve, pero hay que propagar algunos mensajes del protocolo de cuenta de referencias para que la cuenta baje a cero). Por ello, no hay ninguna garantía del orden en que se darán los pasos 8, 9 y 10, aunque el protocolo ya está pensado para no depender de esto.

10. *Las réplicas del CObj reciben la notificación de no referencia* (Servidores). Con ello, en los dominios cohortes se pasa a eliminar los resultados retenidos y en todos los dominios servidores se eliminan las réplicas del CObj y las referencias clientes del RoiID.

    Con ello, más pronto o más tarde llegará una notificación de no referencia también al RoiID del dominio cliente que entonces será eliminado con lo que desaparecerá por completo el contexto que se ha necesitado para esta invocación fiable.

4.3.4 Variante para clientes replicados

Si se emplean *clientes replicados*, habrá unos cuantos cambios en el protocolo que hemos descrito en la sección anterior. Estos cambios afectan a algunos de los pasos anteriores, no a todos ellos. Pero el primer problema que hay que resolver aquí es poder averiguar cuándo el cliente está replicado y cuándo no.

Para ello debería saberse desde qué objeto se inicia la invocación hacia el objeto replicado. Un ORB está pensado para establecer un canal de comunicación entre los clientes y los objetos a los que éstos invocan, pero para construir este canal únicamente se necesita información sobre el objeto destino de la invocación, no sobre el origen. Por tanto, resulta complejo determinar qué tipo de cliente es el origen de una invocación.

En nuestro sistema, la solución que se ha implantado trabaja gestionando los *hilos de ejecución* que utiliza el ORB. Cuando el hilo del cliente ha sido generado previamente para dar servicio a una invocación recibida por dicho cliente, no habrá ningún problema porque el ORB ya ha comprobado que ese cliente es un objeto replicado. El código del núcleo del ORB que proporciona servicio a objetos simples es diferente al que proporciona servicio a objetos replicados, por lo que cuando se generó el hilo ya se anotó el ORB que ese hilo estaba asociado a un objeto replicado. Además, sabrá qué objeto replicado es y dónde están todas sus réplicas.

Sin embargo, si el hilo de ejecución que está utilizando el cliente es nuevo, es decir, no ha sido utilizado para servir una invocación recibida por tal cliente, el ORB no sabrá si ese programa cliente corresponde a un objeto simple o a un objeto replicado. En ese caso, asumirá que era un objeto simple. Para corregir esta situación, la interfaz de nuestro ORB se ha ampliado con una operación no estándar en la que se asocia el hilo de ejecución que se está utilizando ahora con el objeto donde se está ejecutando. Esta operación deberá usarse antes de realizar la invocación fiable sobre un objeto replicado externo si el hilo que se está utilizando es un hilo de nueva creación, es decir, no pertenece a una operación invocada desde un dominio externo (cosa bastante rara, pero que podría llegar a ocurrir en algunos programas).

Veamos a partir de este punto qué cambios se darán en algunos de los pasos del protocolo de invocaciones fiables para generar el *protocolo IFO para clientes replicados*. Los pasos que no sufren ninguna variación no aparecerán en este listado:

1. *Creación del RoiID* (Cliente). Una vez está creado el RoiID y antes de incluir su referencia en el mensaje que transportará la petición al dominio coordinador, nuestro ORB transfiere la referencia al RoiID a los núcleos del ORB ubicados en los dominios cohortes del objeto cliente. Con ello, en dichos núcleos se crearán réplicas de ese RoiID.

   Una vez hecho esto, se transfiere la referencia al RoiID hacia el dominio coordinador, al igual que ocurría en el protocolo básico. Otro cambio que se hará aquí consistirá en poner a falso el *flag SIMPLE* para indicar que el cliente está replicado.

4. *Entrega de la invocación a la réplica coordinadora* (Coordinador). Si el cliente estaba replicado no es necesario transferir a su dominio una referencia sobre el CObj. Por tanto, en esta variante del protocolo el paso 4 se reduce a entregar la invocación a la réplica coordinadora. No hay que hacer nada más.

7. *Termina la invocación en la réplica coordinadora* (Coordinador). Ahora sí se tendrá que

retener una copia de los argumentos de salida y resultado de la invocación, por si fallara la réplica coordinadora cliente y otra de sus réplicas reintentara posteriormente la operación eligiendo al mismo coordinador.

Otro cambio consistirá en la adición al mensaje de respuesta de una referencia cliente del CObj. Esta referencia debe insertarse en dicho mensaje antes de haber descartado las referencias internas en las réplicas del CObj y TObj. Esa referencia del CObj será transmitida ahora hacia el dominio cliente.

8. *El cliente recibe la respuesta* (Cliente). Cuando el ORB ubicado en el dominio cliente recibe el mensaje con la respuesta, realiza un checkpoint interno y síncrono transfiriendo a los dominios cohortes del cliente la referencia al CObj y la copia del resultado y argumentos de salida. Con ello, si el cliente fallara una vez se ha iniciado este checkpoint interno, una de sus réplicas sería capaz de obtener el resultado y procesarlo de manera adecuada. Una vez realizado este checkpoint, el ORB invoca la operación `DiscardResults()` del CObj. El CObj sigue un modelo de replicación similar al activo: sus invocaciones van a todas las réplicas, aunque no nos importa en que orden se entreguen por lo que no necesitamos protocolos de difusión atómica ordenada para realizar tales invocaciones. Esta operación del CObj tiene como misión indicar a tales objetos que ya se pueden descartar los resultados retenidos y se pueden eliminar los objetos CObj. Por tanto, gracias a este paso se eliminará el paso 10 en esta variante para clientes replicados.

   En todos los dominios clientes, cuando se recibe este checkpoint interno síncrono se liberan las referencias internas existentes en los objetos RoiID. Con ello, se prepara a este objeto para recibir la notificación de no referencia en cuanto el paso 9 concluya en los dominios servidores.

9. *Las réplicas del TObj reciben la notificación de no referencia* (Servidores y serializador). En este caso añadiremos al trabajo efectuado en el protocolo básico (eliminar los objetos TObj y que el serializador deje pasar alguna invocación en conflicto con la que termina y elimine su referencia cliente al RoiID) la eliminación de todas las referencias cliente al RoiID en todos los dominios servidores, no sólo en el serializador.

10. Este paso no se necesita en esta variante del protocolo. Los objetos CObj ya fueron eliminados en el paso 8 y las referencias cliente sobre el RoiID mantenidas en los dominios servidores ya han sido eliminadas en el paso 9.

4.3.5 Variante para operaciones unidireccionales

La variante del *protocolo IFO para operaciones unidireccionales* no necesitará ningún objeto CObj ya que la operación no genera ni resultados ni argumentos de salida con lo que no tiene sentido utilizar resultados retenidos en este caso. Con ello el protocolo se simplifica un poco.

Vamos a describir tanto el protocolo básico para esta situación como los pequeños cambios que habría que introducir si el cliente estuviera replicado. Todo ello aparecerá en las figuras 4.6 a 4.10 y se describe seguidamente:

1. *Creación del RoiID* (Cliente). Una vez el programa cliente ha iniciado una invocación nuestro ORB detecta tal invocación y crea en este dominio cliente un objeto RoiID. Una vez

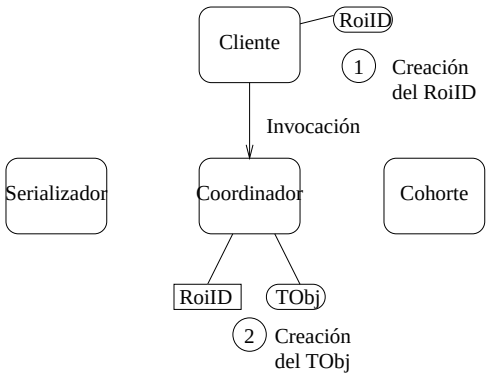

Figura 4.6: Pasos 1 y 2 en una invocación fiable unidireccional.

creado este objeto, genera una referencia cliente para él y la inserta en el mensaje que se ha construido para hacer llegar esta petición al dominio coordinador.

En este caso el *flag SIMPLE* debe ponerse siempre a falso para que en el paso 8 se liberen todas las referencias clientes del RoiID en todos los dominios, al igual que ocurría en la variante del protocolo IFO para clientes replicados.

Si el cliente está replicado habrá que realizar el checkpoint interno síncrono al igual que en la variante anterior del protocolo. Sin embargo, aquí el objetivo es distinto. Esas referencias transferidas a los dominios cohortes del cliente se mantendrán siempre ahí para evitar que en caso de fallo se reintente la invocación unidireccional. Parece extraño, pero esto es así debido a que según el estándar CORBA, las operaciones unidireccionales deben tener *semántica "como máximo una vez"*, por lo que la operación no debe reintentarse si se ha detectado un fallo.

Una vez se ha transferido la referencia al RoiID hacia el dominio coordinador, la referencia interna almacenada en el propio objeto RoiID es descartada. Con ello cuando en el paso 7 se liberen todas las referencias mantenidas en los dominios servidores y en el serializador, si el cliente era simple el objeto RoiID recibirá la notificación de no referencia y podrá ser descartado. Si el cliente está replicado, las referencias internas mantenidas en los dominios cohorte del cliente nunca se descartan, por lo que el RoiID no será eliminado en ese caso.

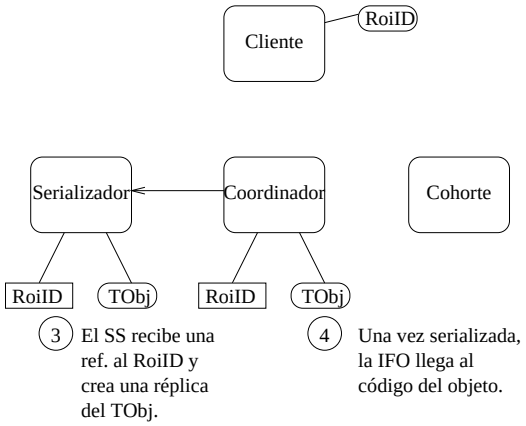

Figura 4.7: Pasos 3 y 4 en una invocación fiable unidireccional.

2. *Recepción de la invocación* (Coordinador). El núcleo del ORB ubicado en el dominio coordinador recibe los mensajes con la petición ligada a esta invocación. Una vez recibidos, y antes de hacer llegar la invocación al objeto invocado, el ORB busca en el mensaje la referencia cliente al RoiID. Este RoiID será necesario para los pasos siguientes, ahora como esta operación no puede tener resultados no es necesario buscar ningún resultado retenido.

   El dominio coordinador genera la primera réplica del objeto TObj y la asocia a la referencia al RoiID que acaba de recibir.

3. *Petición de serialización* (Coordinador). El núcleo del ORB en el dominio coordinador realiza una petición sobre el método `Serialize()` del objeto serializador. El serializador recibe entre otros argumentos una referencia al RoiID y una referencia al TObj que le servirá para construir una réplica de este último objeto. Con esta información junto a una descripción de la operación invocada, el serializador averigua si la invocación que se intenta realizar está en conflicto con alguna otra ya en curso o encolada previamente. De ser así, suspende esta invocación hasta que todas sus invocaciones en conflicto que la precedan hayan terminado. Cuando esto ocurra, o si no había ninguna invocación que precediera a ésta, el control es devuelto al dominio coordinador, para que continúe con el servicio de la invocación.

   La referencia cliente del TObj es descartada una vez ya se ha creado la réplica de este objeto. De esta manera, cuando todas las réplicas servidoras den por terminada la invocación en curso y liberen las referencias mantenidas en sus propias réplicas del TObj, este objeto recibirá la notificación de no referencia que el serializador interpretará como la terminación de esta invocación en los dominios servidores (ver paso 8, más adelante).

4. *Entrega de la invocación a la réplica coordinadora* (Coordinador). El ORB del dominio coordinador entrega la invocación al código de la réplica y empieza la ejecución de la operación invocada.

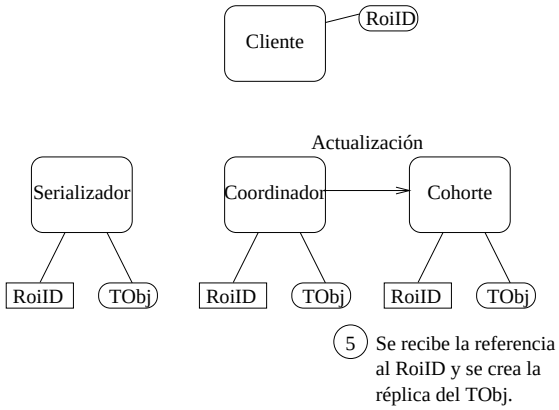

Figura 4.8: Paso 5 en una invocación fiable unidireccional.

5. *Se realiza la primera actualización de estado* (Coordinador). En la primera llamada a un método de las réplicas cohortes para realizar una actualización de estado, el soporte incluido en nuestro ORB añade al contenido del mensaje referencias sobre los objetos RoiID y TObj.

Cuando el ORB en los dominios cohortes recibe estos mensajes, extrae esas referencias y crea las réplicas de los objetos TObj a partir de las referencias obtenidas.

Además, en este mensaje se debe añadir en su cabecera un *flag UNIDIRECCIONAL* para indicar al ORB de los dominios cohortes que no se tendrá que buscar una referencia al CObj, ni generar su réplica, ni en la última actualización de estado habrá que buscar los argumentos de salida ni el resultado para retenerlos.

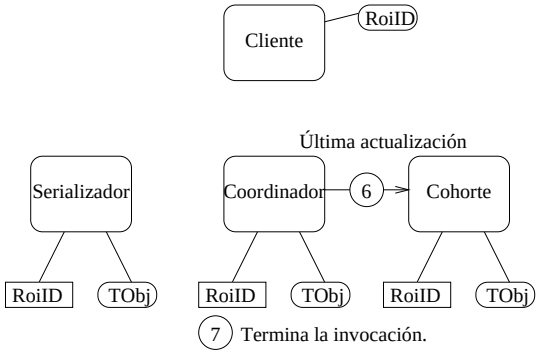

Figura 4.9: Pasos 6 y 7 en una invocación fiable unidireccional.

6. *Se realiza la última actualización de estado* (Coordinador). Una vez las réplicas cohortes han procesado este último mensaje de actualización de estado, nuestro ORB libera las referencias clientes mantenidas en las réplicas de los objetos TObj en todos los dominios cohortes.

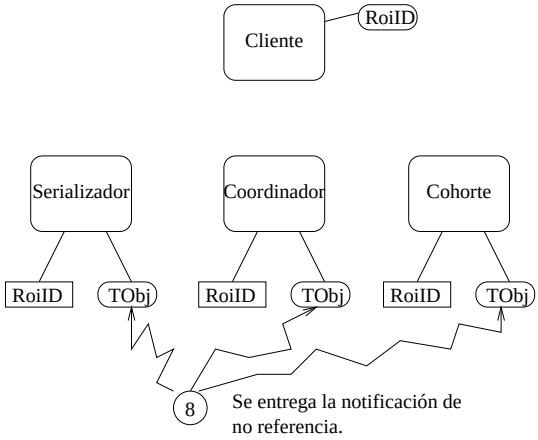

Figura 4.10: Paso 8 en una invocación fiable unidireccional.

7. *Termina la invocación en la réplica coordinadora* (Coordinador). Al igual que hicieron los ORBs de los dominios cohortes, cuando termina la invocación en el dominio coordinador hay que liberar las referencias internas de las réplicas del TObj, con lo que ya no quedarán referencias para este objeto en todo el sistema. Por otro lado, en estas invocaciones no hay que devolver respuesta al cliente, por lo que ya se puede dar por terminada la invocación.

8. *Las réplicas del TObj reciben la notificación de no referencia* (Servidores y serializador). Con ello el serializador ya puede dar paso a al menos una de aquellas invocaciones fiables

que estuvieran en conflicto con la invocación que acaba de terminar y que hubiesen llegado tras ella. Además, todas las réplicas del TObj son eliminadas y se descartan las referencias cliente del RoiID en todos los dominios. Con ello, el RoiID recibirá la notificación de no referencia si el cliente no estaba replicado y podrá eliminarse, desapareciendo todo el contexto asociado a esta invocación fiable unidireccional.

Tal como se ha explicado este protocolo da la sensación de que si el cliente estaba replicado nunca se eliminará el objeto RoiID que se ha generado para identificar la invocación fiable unidireccional. Esto no es exactamente así, pero la forma de liberar ese objeto no tiene relación directa con el protocolo descrito. Aquí nos apoyamos de nuevo en la gestión de hilos de ejecución que esbozamos en la variante anterior del protocolo.

Como ya se dijo en la sección anterior, nuestro ORB utiliza un componente interno que relaciona cada hilo de ejecución con las invocaciones fiables que ha ido sirviendo. Por tanto, en la información mantenida en este componente se mantiene qué cadenas de invocaciones fiables existen en el sistema.

Así, cuando se realiza el checkpoint síncrono interno del paso 1 de esta variante del protocolo. Aparte de la información citada, se adjunta también la identidad de la invocación "madre" de la que ahora se está generando. De esta forma, en los dominios cohortes también se sabrá dentro de qué invocación anterior se está desarrollando esta invocación unidireccional. Con ello, cuando esa invocación madre termine, en cada uno de los dominios donde se detecte su finalización se informa al componente que lleva la gestión de hilos de ejecución y que también mantiene las cadenas de invocaciones realizadas. Con ello, si hay alguna invocación unidireccional descendiente de la que ahora está terminando, todo su contexto es eliminado del sistema, liberando así sus objetos RoiID.

4.3.6 Variante para operaciones de sólo lectura

En caso de tener una invocación fiable sobre una *operación de sólo lectura* no es necesario realizar ningún envío de actualizaciones de estado hacia los dominios cohortes. Tampoco habrá necesidad de retener los resultados en ningún caso, ni de realizar un checkpoint síncrono para replicar el RoiID en caso de tener un cliente replicado. Con las operaciones de solo lectura, por ser *idempotentes*, no hay ningún problema si son reintentadas y ejecutadas más de una vez, incluso con distinto RoiID.

Por todo lo dicho anteriormente, ésta será la variante más sencilla del protocolo de invocaciones fiables. Sus pasos serán los siguientes (ver figuras 4.11 a 4.15):

1. *Creación del RoiID* (Cliente). Al igual que en los casos anteriores, nuestro ORB detecta que se está iniciando una invocación sobre un objeto replicado y crea en el dominio cliente un objeto RoiID. También genera una referencia cliente para él y la inserta en el mensaje que se ha construido para hacer llegar esta petición al dominio coordinador. El *flag SIMPLE* se fija a falso para indicar que no es necesario transmitir hacia el dominio cliente una referencia al CObj que se creará temporalmente en el dominio coordinador.

2. *Recepción de la invocación* (Coordinador). El núcleo del ORB ubicado en el dominio coordinador recibe los mensajes con la petición ligada a esta invocación. Al igual que en el protocolo básico, se realizará una búsqueda en el área de resultados retenidos, pero no tendrá

éxito pues para este tipo de operaciones los resultados no se mantienen. Tras esto, el dominio coordinador genera las primeras réplicas de los objetos CObj y TObj y las asocia a la referencia al RoiID que acaba de recibir.

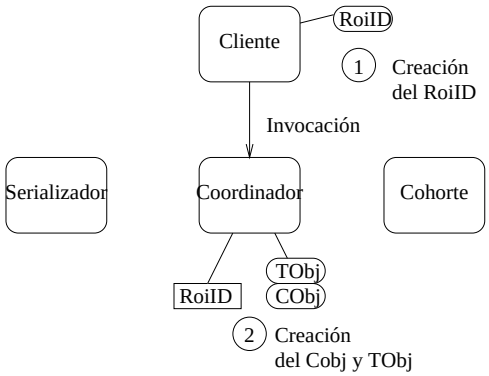

Figura 4.11: Pasos 1 y 2 en una invocación fiable de lectura.

3. *Petición de serialización* (Coordinador). El núcleo del ORB en el dominio coordinador realiza una petición sobre el método `Serialize()` del objeto serializador.

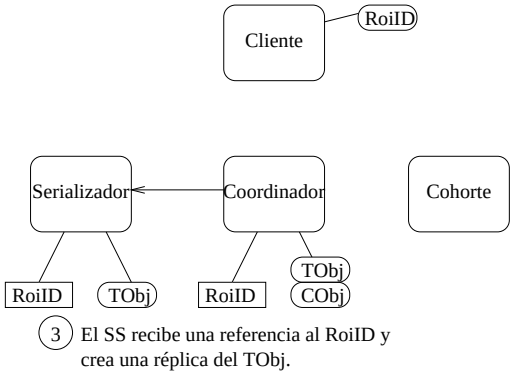

Figura 4.12: Paso 3 en una invocación fiable de lectura.

La referencia cliente del TObj es descartada una vez ya se ha creado la réplica de este objeto. De esta manera, cuando todas las réplicas servidoras den por terminada la invocación en curso y liberen las referencias mantenidas en sus propias réplicas del TObj, este objeto recibirá la notificación de no referencia que el serializador interpretará como la terminación de esta invocación en los dominios servidores (ver paso 6, más adelante).

4. *Retorno de la petición de serialización* (Coordinador). El serializador es el único objeto del sistema que conoce los conflictos existentes entre las operaciones de la interfaz del objeto y si hay alguna operación que es de sólo lectura. Cuando comprueba que la operación en curso es de este tipo y verifica que ya no tiene operaciones precedentes en conflicto devuelve el control al dominio coordinador y le adjunta un *flag LECTURA*. Al recibir la respuesta y comprobar dicho flag, el ORB en el dominio coordinador sabrá que esta operación no necesita un objeto CObj ni resultados retenidos. Por tanto, pasa ahora a descartar el objeto CObj, eliminándolo del contexto de esta IFO.

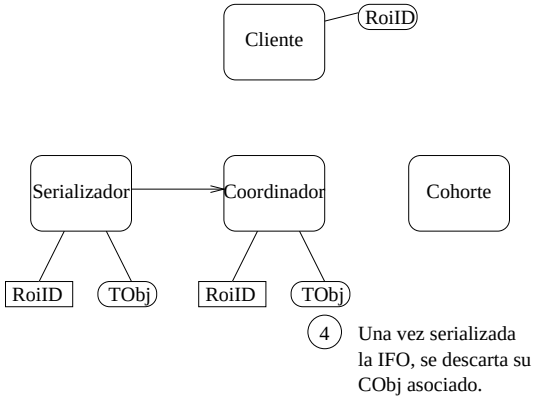

Figura 4.13: Paso 4 en una invocación fiable de lectura.

5. *Entrega de la invocación a la réplica coordinadora* (Coordinador). El ORB del dominio coordinador entrega la invocación al código de la réplica y empieza la ejecución de la operación invocada.

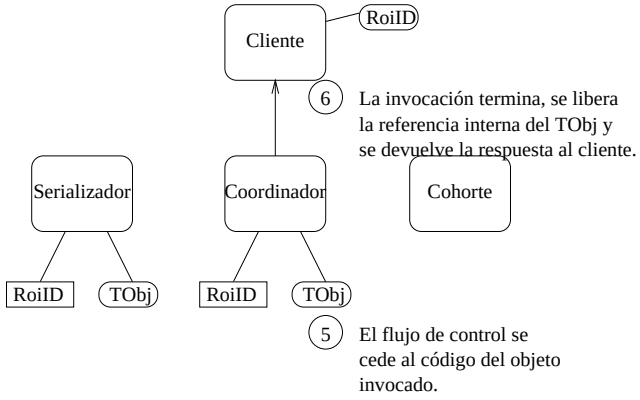

Figura 4.14: Pasos 5 y 6 en una invocación fiable de lectura.

6. *Termina la invocación en la réplica coordinadora* (Coordinador). Cuando termina la invocación en el dominio coordinador hay que liberar la referencia interna de la réplica local del TObj. Aquí no hay que guardar una copia de los resultados y argumentos de salida. Si el cliente o el coordinador fallan, no pasa nada si la operación se repite por completo desde su inicio. Una operación de sólo lectura es idempotente.

   Tras esto, nuestro ORB en el dominio coordinador emitirá el mensaje con la respuesta para el dominio cliente.

7. *Las réplicas del TObj reciben la notificación de no referencia* (Coordinador y serializador). Con ello el serializador ya puede dar paso a al menos una de aquellas invocaciones fiables que estuvieran en conflicto con la invocación que acaba de terminar y que hubiesen llegado tras ella. Además, todas las réplicas del TObj son eliminadas y tanto en el serializador como en el coordinador también se descarta la referencia cliente del RoiID. En este último, esta acción está asociada a tener el flag SIMPLE a falso.

   Eventualmente, la liberación de las referencias clientes del RoiID provocará que éste reciba

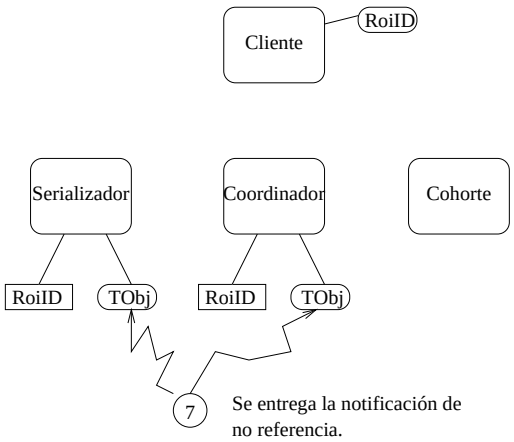

Figura 4.15: Paso 7 en una invocación fiable de lectura.

la notificación de no referencia y que sea eliminado por completo del sistema.

4.4 Comportamiento del protocolo en caso de fallos

El protocolo IFO resultaría completamente inútil si no fuera capaz de tolerar los fallos de los agentes que participan en una invocación fiable. Para ello, los objetos auxiliares utilizados cuando ningún agente falla también resultarán útiles para detectar cualquier situación de fallo y dirigir su recuperación.

En las próximas secciones se irán estudiando diferentes situaciones de fallo. En cada uno de los casos se presenta en primer lugar cómo se puede detectar el fallo y posteriormente se describen las medidas que se adoptan para que el protocolo concluya satisfactoriamente.

4.4.1 Caída de un cliente sin réplicas

Este caso corresponde a la caída de un cliente no replicado que haya sucedido tras haber enviado éste el mensaje de petición hacia el dominio coordinador. Se asume que no hay problemas de intercomunicación y que dicho mensaje de petición puede entregarse al dominio coordinador.

Detección

Si se trata de una IFO “normal” que utilice el protocolo básico explicado en la sección 4.3.3, este tipo de fallo se puede detectar en diversos momentos, dependiendo de cuándo haya ocurrido. Así, tenemos las siguientes alternativas:

A1. El fallo se ha producido antes de o durante el paso 4 del protocolo básico. Podrá detectarse cuando se realice la invocación del método `Results()` del objeto RoiID. En ese intento de invocación el ORB devolvería una excepción indicando que el objeto destino ha dejado de existir debido a la caída del nodo donde residía.

A2. El fallo se ha producido tras el paso 4 y antes del paso 8 del protocolo básico. Si ocurre esto, cuando el protocolo básico llega al final del paso 7, como se habrá perdido la referencia

cliente existente en el dominio cliente, se generará una notificación de no referencia para el objeto CObj.

Esto no implica necesariamente una caída del dominio cliente. Su tratamiento estándar es eliminar el objeto CObj de todos los dominios servidores y descartar los resultados retenidos. Ese tratamiento es el adecuado para este caso.

Además de este tipo de detección, el ORB del dominio coordinador también recibirá una excepción del protocolo de transporte al final del paso 7 indicando que el nodo donde se ubicaba el cliente ha dejado de ser accesible. Sin embargo, no es necesario añadir ningún tratamiento especial para dicha excepción, que por otra parte, es interna al núcleo del ORB.

A3. El fallo se ha producido durante o tras el paso 8. Tiene el mismo tipo de detección y tratamiento que el caso anterior, salvo que no se da la excepción interna en el ORB del dominio coordinador provocada por el protocolo de transporte.

Cuando se está realizando una invocación sobre una operación unidireccional, resulta imposible detectar el fallo del cliente. Tampoco importa en exceso pues para este tipo de operaciones resulta irrelevante si el cliente sigue activo o no, ya que no hay que retornar ningún resultado ni argumento de salida.

Por último, para las invocaciones sobre operaciones de sólo lectura, cuyo protocolo se ha descrito en la sección 4.3.6, la detección podría llevarse a cabo en el sexto paso de dicho protocolo. En ese paso se retorna el resultado hacia el dominio cliente y si éste ya había fallado en ese instante, el protocolo de transporte se lo notificará al ORB del dominio coordinador. No es necesario hacer nada para tratar esta situación, pues no habrá ningún otro cliente que podrá reintentar la operación.

Tratamiento

Únicamente hay que tomar medidas especiales en dos de las situaciones descritas anteriormente:

- Cuando se da el caso de detección A1 y el ORB devuelve una excepción indicando que el dominio cliente ha dejado de existir, por lo que la invocación sobre el RoiID es imposible que tenga éxito. El código que realiza esta invocación sobre el RoiID en el dominio coordinador no debe tomar ninguna acción especial ante esta situación. Es decir, el protocolo debe continuar exactamente igual que si hubiese podido completar dicha llamada.

- Cuando el mecanismo de transporte genera una excepción interna indicando que es imposible entregar el mensaje de respuesta al dominio cliente. Al igual que antes, el protocolo debe proseguir y no hacer caso de esta excepción interna.

Como puede comprobarse, el protocolo no toma ninguna acción especial en caso de caída de la única instancia de las que consta el cliente. Se siguen los pasos normales del protocolo y éste termina normalmente. Esto es debido a que poco importa si el cliente es capaz de recibir los resultados de la invocación o no. Nuestro objetivo será terminar con la invocación iniciada por él y para ello poco importa si el cliente ha caído o no.

4.4.2 Caída de un cliente con otras réplicas

En esta situación se asume que se está utilizando la variante del protocolo IFO descrita en la sección 4.3.4 y que cae la réplica cliente coordinadora que ha iniciado la invocación fiable sobre otro objeto replicado externo.

Detección

Debemos distinguir las siguientes situaciones de fallo, cuyo tratamiento se discutirá posteriormente:

S1. Fallo durante el paso 1. Si se da antes de realizar el checkpoint para dar a conocer el RoiID al resto de réplicas del cliente, la invocación ni siquiera se habrá dado por iniciada en el resto de dominios (tanto clientes como servidores) por lo que este caso no reviste interés.

S2. Fallo tras el paso 1, pero anterior al paso 8. En este caso, el fallo de la réplica coordinadora del cliente no se puede detectar hasta que el coordinador del objeto invocado trata de devolverle el resultado al cliente. Esto ocurre en el paso 7 y se detecta porque el protocolo de transporte le notifica al dominio coordinador que va a resultar imposible transmitir el resultado al nodo donde residía el cliente porque tal nodo ha fallado.

S3. Un fallo posterior, incluso si se da en el paso 8 antes de realizar el checkpoint hacia las demás réplicas clientes para transferirles la referencia al CObj, no sería detectado por los dominios servidores.

Tratamiento

El tratamiento para cada una de las situaciones anteriores es el siguiente:

S1. En este caso no se ha conseguido dar a conocer a ninguna réplica el RoiID creado por el coordinador cliente anterior. Incluso si ese RoiID hubiese llegado a alguna réplica cliente pero no a todas, no habría ningún problema: si se eligiera como nuevo coordinador alguno de los que había recibido ese RoiID, se reutilizaría, en caso contrario, se crearía otro nuevo. Debe tenerse en cuenta que ninguna réplica servidora ha sabido nada sobre el anterior RoiID, por lo que serán incapaces de distinguir un caso del otro.

   El tratamiento final consistiría en que la nueva réplica coordinadora elegida para el dominio cliente debería reiniciar la invocación con el nuevo RoiID generado o con el recuperado del intento fallido anterior.

S2. En esta situación el dominio coordinador servidor ha conseguido completar la invocación y se ha dado cuenta del fallo del dominio cliente cuando ha intentado enviarle el mensaje de respuesta. Los dominios cohortes servidores también han completado la invocación y tanto ellos como el coordinador mantienen los *resultados retenidos*. El dominio coordinador no debe hacer nada especial tras darse cuenta de que el cliente ha fallado. Es decir, los resultados retenidos seguirán manteniéndose, al igual que el resto del contexto ligado a la invocación fiable.

Eventualmente el protocolo de pertenencia también notificará sobre el fallo al resto de las réplicas clientes. Utilizando cierto criterio, se elegirá un nuevo coordinador para los clientes y éste reintentará la IFO. Pero en el dominio de este nuevo coordinador cliente ya existe suficiente información para que éste advierta que esa IFO ya fue iniciada previamente y cuál fue su RoiID en el intento anterior. El nuevo coordinador cliente reutilizará el RoiID anterior, siguiendo punto por punto el mismo protocolo descrito en la sección 4.3.4. Cuando se llegue al paso 2, el coordinador servidor que se haya elegido en esta ocasión (da igual si no coincide con el coordinador que tuvo el anterior intento) se dará cuenta de que para ese RoiID ya existen resultados retenidos y pasará a devolverlos de inmediato, sin necesidad de serializar la invocación de nuevo, ni entregarla al código servidor, ni realizar ninguna actualización de estado sobre los dominios cohortes. Con ello, la IFO pasa directamente del paso 2 al paso 7 y se evita así reprocesarla.

S3. Si el ORB ubicado en el dominio del cliente anterior consiguió realizar el checkpoint donde se daban a conocer a los cohortes clientes los resultados devueltos por el objeto invocado, no habrá ningún trabajo que hacer para tratar este caso de fallo.

Si el cliente falló inmediatamente antes de realizar tal checkpoint, el nuevo coordinador cliente que se haya elegido reprocesará la IFO tal como hemos descrito en el punto anterior, correspondiente a la situación S2.

4.4.3 Caída de uno o más cohortes

Los cohortes intervienen en todas las variantes del protocolo IFO, excepto en la *variante para operaciones de sólo lectura*, por lo que ese caso no va a tratarse aquí.

Detección

Ya que los dominios cohortes son agentes que participan activamente en el protocolo IFO, la detección de su fallo puede llevarse a cabo de dos maneras diferentes. La primera, en caso de haber fallado precisamente en ese instante, se daría cuando el dominio coordinador intenta realizar un envío de un mensaje de actualización de estado, recibiendo una excepción enviada por el ORB indicando que el objeto destino de tal envío ya no existe. La segunda forma, la más normal, se apoya en el soporte que da el *protocolo de pertenencia a grupo*. En cuanto este protocolo encuentra el fallo, procede a notificarlo a todos los componentes interesados y entre estos componentes está el ORB que realizará una reconfiguración de todas las referencias donde intervenga algún objeto ubicado en el nodo que acaba de fallar. De esta manera, tales referencias serían corregidas y cuando el protocolo IFO necesitara, en uno de los pasos posteriores, realizar una actualización de estado, el cohorte ya habría sido eliminado del grupo con lo que el protocolo IFO no se vería afectado para nada.

Tratamiento

El tratamiento es muy sencillo. Basta con no hacer llegar los mensajes de actualización de estado al cohorte que acaba de fallar. Si debido a la eliminación de dicho cohorte, ya no quedan suficientes réplicas en ese objeto, será conveniente generar una nueva réplica en algún otro nodo.

4.4.4 Caída del coordinador

El coordinador es el principal agente que participa en una invocación fiable. Normalmente, su caída implicará rehacer tal invocación. Ese reinicio puede ser emprendido por uno de los dominios cohortes, a iniciativa propia o, en el caso más normal, a petición del cliente que reintentará de nuevo la invocación fiable al observar el fallo del coordinador.

Detección

Normalmente, cuando cae un objeto o dominio, un ORB únicamente notifica ese fallo a aquellos clientes que intenten invocar a dicho objeto tras haberse producido el fallo, devolviéndoles una excepción que indique que el dominio destino ya no existe. Sin embargo, las invocaciones en curso no tienen por qué enterarse del fallo, con lo que se quedarían "colgadas".

En nuestro ORB, gracias al soporte facilitado por el algoritmo de pertenencia, esta situación no se dará. Cuando se haya producido un fallo y durante el protocolo de reconfiguración que se ejecuta tras haber sido notificado este fallo al ORB, se comprueban aquellas invocaciones en curso existentes en cada ORB (para ello basta con examinar la información asociada a los hilos de ejecución suspendidos tras haber iniciado invocaciones externas). Si alguna de estas invocaciones en curso debía haber llegado al nodo que ha fallado, nuestro ORB reanudará su hilo de ejecución y éste devolverá una excepción a los niveles superiores, donde será tratada de manera adecuada, reintentándose la invocación fiable pero esta vez sobre otra réplica servidora.

Con esto ya se tiene suficiente soporte para reiniciar la invocación de nuevo. Sin embargo, hay que distinguir algunos casos que hacen que el comportamiento de los demás agentes participantes en la invocación fiable sea uno u otro. Estos casos son los siguientes:

C1. Se trataba de una *invocación unidireccional*. Aquí el cliente no tiene por qué mostrar ningún interés sobre el fallo ocurrido en el dominio coordinador. No importará en exceso lo que haya sucedido.

C2. El coordinador había llegado a realizar la petición de serialización, pero o bien ésta todavía no había sido contestada o bien el coordinador había recibido la respuesta y había entregado la IFO a la réplica coordinadora pero ésta todavía no había iniciado ningún envío de actualización de estado hacia las réplicas cohortes.

    En este caso sólo ha llegado a haber dos réplicas del TObj, de manera que cuando cayó el coordinador también se perdió una de estas réplicas. Con ello, cuando el serializador conteste la petición de serialización y elimine la referencia interna de su réplica del TObj, se deja a este objeto en una situación en la que recibirá de inmediato una notificación de no referencia en cuanto falle el coordinador.

C3. El fallo se ha producido en este caso tras haber realizado ya alguna actualización de estado sobre las réplicas cohortes, pero todavía no la última de ellas, con lo que en las réplicas cohortes no ha terminado esta invocación fiable. Están pendientes de recibir alguna actualización de estado adicional.

C4. El fallo se había producido tras haber realizado todas las actualizaciones de estado sobre las réplicas cohortes, pero antes de devolver el resultado al cliente.

Tratamiento

Veamos como se tratan cada uno de los casos distinguidos en el apartado de detección de fallos:

C1. Esta situación será ignorada por el cliente. Según se recoge en la especificación del estándar CORBA [OMG99a], las invocaciones unidireccionales deben tener una *semántica "como máximo una vez"* por lo que en caso de duda será preferible no volver a intentar la invocación. Como no disponemos de ningún mecanismo que permita garantizar con seguridad si dicha IFO pudo completarse o no, no vale la pena dedicar ningún esfuerzo para realizar un reintento.

C2. Dentro de este caso tienen cabida varias situaciones. Un primer ejemplo sería una *operación de sólo lectura* donde sí es admisible una *semántica* diferente: la de *"al menos una vez"* porque en este caso las repeticiones que se puedan dar no plantearán ningún problema en la consistencia del objeto. Este tipo de operaciones se adaptan perfectamente a la situación descrita en este caso, pues en su variante del protocolo no es necesario realizar ninguna actualización de estado sobre las réplicas cohortes. Con ello, el intento inicial que había fallado debido a la caída del coordinador, será considerado como terminado por el serializador cuando el TObj reciba la notificación de no referencia tras haber fallado tal coordinador. No importa, el nuevo reintento iniciado por el cliente llevará el mismo RoiID, será dirigida a otra réplica coordinadora y será serializada de nuevo, utilizando un TObj distinto. El serializador no precisa recordar el intento anterior. Como precaución siempre se encarga de comprobar si el RoiID que acaba de recibir todavía está activo, pero al no encontrarlo en este reintento simplemente considerará que es una invocación fiable nueva y la procesará como tal.

    El resto de casos se procesan exactamente de la misma forma. Si el coordinador que falló fue incapaz de realizar ninguna actualización de estado, poco importará que el nuevo intento por completarla sea considerado una invocación nueva y distinta al intento anterior.

C3. En la situación descrita en este tercer caso, el nuevo coordinador será elegido entre los cohortes del intento anterior que fueron capaces de recibir algún envío de actualización de estado, pero ninguna réplica recibió todos ellos.

    Al igual que en el caso anterior, el nuevo serializador pasará a procesar de nuevo la petición, pero en este caso no será necesario generar nuevos objetos CObj ni TObj. Respecto al TObj la diferencia residirá en que ahora no será necesaria ninguna petición de serialización asociada a este reintento. Visto el contexto del intento anterior, ya sabemos que la operación fue serializada y que todavía no ha terminado. Por tanto, se iniciará de nuevo, se irán propagando sus actualizaciones de estado, las cuales sólo serán entregadas en las réplicas cohortes si puede comprobarse que no fueron recibidas en el intento anterior. Finalmente se difundirá la última actualización de estado y la invocación terminará, eliminándose los objetos TObj y CObj cuando sea necesario.

C4. En este último caso, si alguna réplica había recibido todas las actualizaciones de estado, se propagarán estas a todas las demás réplicas cohortes durante el protocolo de reconfiguración que se ejecuta cuando se ha detectado el fallo del antiguo coordinador. Una vez hecho esto,

todas las antiguas réplicas cohortes ya disponen de resultados retenidos (excepto si se trataba de una invocación de sólo lectura, que por ser un caso más sencillo no ha sido considerado aquí) y los objetos TObj habrán sido eliminados y la invocación dada por terminada en el serializador.

Cuando el cliente la reintenta de nuevo, el coordinador elegido en este caso busca el RoiID recibido en el área de resultados retenidos y lo encuentra, con lo que devuelve de inmediato los resultados y la invocación prosigue a partir de ese punto.

4.4.5 Caída del serializador

La caída del serializador tiene como consecuencias principales la pérdida de la información referente a la compleción de las invocaciones fiables en curso, así como las relaciones de precedencia entre aquellas invocaciones fiables suspendidas que todavía no han iniciado su ejecución. Por tanto, es vital garantizar que el serializador no falle, para lo que también se procederá a replicarlo de cierta manera. Los detalles sobre la detección de sus caídas y el tratamiento de éstas aparecen en el capítulo 5, por lo que no van a ser repetidos aquí.

Cabe resaltar que el objeto serializador mantiene réplicas en aquellos nodos donde exista alguna réplica del objeto al cual está sirviendo. Como el serializador es común para todos los objetos replicados, en la práctica existirá una réplica del serializador en cada nodo del cluster. A estas réplicas se las llama *agentes de serialización*.

4.4.6 Caída de coordinador y cliente

El comportamiento que podrá seguir una invocación fiable en caso de caída simultánea de la réplica coordinadora servidora y del cliente (que podrá estar replicado o no), depende del tipo de invocación realizado y del grado de replicación que presente el cliente. Generalmente se requerirá iniciar de nuevo la invocación fiable o, si al menos se ha podido realizar una actualización de estado sobre los dominios cohortes servidores, completar la invocación en curso en tales dominios.

Detección

Para detectar estas situaciones habrá que distinguir algunos subcasos:

C1. Si el cliente no estaba replicado y el coordinador no ha llegado a realizar una petición de serialización, resultará imposible detectar tal situación, pues los únicos agentes que han participado hasta ese momento en la invocación han fallado y los demás no saben nada acerca de ella.

C2. Si el cliente estaba replicado y el coordinador no ha llegado a realizar una petición de serialización, ocurrirá inicialmente lo mismo que en el caso anterior. Sin embargo, más pronto o más tarde el algoritmo de pertenencia notificará el fallo del cliente al ORB y se reconfigurará el estado del objeto replicado al cual pertenecía el cliente. Con esto, su papel coordinador será adoptado por otra réplica del cliente y la invocación será repetida, utilizando el mismo RoiID que fue difundido en el envío anterior antes de emitir el mensaje de petición hacia el dominio coordinador servidor.

C3. El coordinador ha conseguido realizar la petición de serialización pero no ha realizado ninguna actualización de estado sobre las réplicas cohortes, bien porque era una invocación fiable de sólo lectura o bien porque no ha tenido tiempo para iniciarlas.

En este caso la detección se realiza de manera idéntica a como ocurre en los casos C1 y C2, dependiendo del grado de replicación del cliente. Sin embargo, hay otra situación adicional que conviene describir y que guarda relación con el serializador. Como éste ha recibido la petición de serialización y ha debido ordenarla para sincronizar su ejecución, se ha generado una réplica del TObj en su dominio. Al fallar el coordinador, esta réplica del TObj recibirá la notificación de no referencia, con lo que el serializador dará a esta IFO por terminada y la eliminará de su dominio.

Este tratamiento será el que recibirán todas aquellas peticiones de sólo lectura que hayan sido ya serializadas. No hay ningún problema a este respecto. Los posteriores reintentos de ejecución de estas invocaciones en caso de tener al cliente replicado serán serializados de nuevo.

C4. Como en el caso anterior, pero ahora el coordinador logró realizar al menos una actualización de estado, sin haber terminado todas las actualizaciones de estado necesarias para concluir la invocación fiable. La forma de detectar esta caída en caso de clientes replicados será la misma que hemos expuesto en el caso C2. Si no existen otras réplicas del cliente, la operación no podrá ser reintentada.

Existen algunos detalles más que afectan a la manera en que puede detectarse la ocurrencia de esta situación. Ya que ha podido realizarse al menos la primera actualización de estado, los dominios cohortes también tendrán réplicas del objeto TObj, con lo que el serializador no podrá dar por terminada la invocación, ya que este objeto no recibirá la notificación de no referencia.

Cuando el protocolo de reconfiguración iniciado por el protocolo de pertenencia a grupo, notifique que ha fallado alguna máquina, los dominios cohortes comprobarán la integridad del cliente. Para ello, invocan el método `CORBA::Object::non_existent()` de sus referencias al RoiID. Si devuelve un valor cierto, eso querrá decir que el cliente ha desaparecido y que algún cohorte debe retomar la invocación y terminarla. Si por el contrario devuelve un valor falso, los cohortes esperarán a que otra réplica del cliente reintente la invocación.

Además, estos cohortes pueden comprobar también fácilmente si el coordinador ha caído o no. Para ello basta con que pregunten al protocolo de pertenencia sobre la disponibilidad del nodo donde se encuentra la *réplica principal* (se entiende por tal, la réplica inicial de un objeto y cuyo identificador de nodo aparece en sus referencias clientes) de los objetos TObj y CObj.

C5. El coordinador logró realizar todas las actualizaciones de estado sobre los cohortes antes de fallar. En este caso, los objetos TObj recibirán en algún momento la notificación de no referencia y la invocación será dada por terminada en el serializador.

Si el cliente no estaba replicado, ocurrirá lo mismo con el objeto CObj. Pero si el cliente

estaba replicado, los CObj no reciben la notificación de no referencia y los resultados retenidos se mantienen hasta que otra réplica del cliente reintente la operación y los encuentre.

Tratamiento

Veamos cómo se trata cada una de las situaciones distinguidas en el apartado anterior correspondiente a la detección del fallo:

C1. En esta situación no puede darse ningún tratamiento porque la invocación fiable ha desaparecido del sistema sin dejar ningún rastro. De todas formas, como ninguna de las réplicas que se han mantenido vivas ha sabido nada sobre tal invocación, no se da ningún problema.

C2. Alguna de las réplicas del cliente tomará de nuevo el papel de coordinadora para la invocación que falló en el intento previo. Como tendrá el RoiID antiguo en el soporte que mantiene el ORB, reutilizará dicho RoiID para realizar el nuevo intento, ahorrándose la realización del checkpoint interno síncrono que precedía el envío del mensaje de petición hacia el dominio coordinador servidor.

Por lo demás, esa invocación seguirá los mismos pasos sucesivos que en el protocolo estándar que corresponda a su variante de invocación.

C3. El tratamiento que se realizará en este caso coincide con el presentado en los apartados C1, si el cliente no estaba replicado, o C2, si el cliente sí estaba replicado. Para esta última alternativa, el serializador ya había procedido en el intento anterior, a asignar un orden de serialización a la invocación fiable. La caída de su coordinador anterior hizo que el serializador, de manera equivocada, procediera a dar por terminada tal invocación. Ahora se requerirá que el serializador repita la asignación de orden. Sin embargo, esto no conlleva ningún problema ya que el serializador no recuerda en ningún momento si un determinado RoiID ya ha sido utilizado por una invocación o no. Por tanto, se repetirá el trabajo de serialización, pero eso es justamente lo que se espera que suceda.

C4. Como ya hemos comentado en el apartado de detección, si los cohortes saben que todas las réplicas del cliente y el coordinador servidor han caído, deberán proceder a ejecutar un pequeño protocolo de reconfiguración interna mediante el cuál una réplica cohorte pasa a convertirse en coordinadora para la invocación iniciada en el intento anterior. Esta nueva réplica coordinadora recogerá la copia de los argumentos de entrada que recibió en la primera actualización de estado y con ella procederá a reiniciar la invocación. No es necesario que repita la petición de serialización, pues ya se sabe que esta invocación está autorizada por el mecanismo de control de concurrencia. Tampoco deberá preocuparse por retornar ningún resultado al cliente, pues éste ya se sabe que ha fallado. Con ello repite el procolo con el mismo tratamiento visto en la sección 4.4.1 donde se estudió la caída de un cliente no replicado.

Si el cliente estaba replicado, debe esperarse a que éste reintente la invocación fiable, eligiendo otro coordinador. En este caso no resulta necesaria tampoco la petición de serialización y se procede directamente a iniciar la invocación en la nueva réplica coordinadora. Por lo demás, no habrá ninguna otra diferencia con el caso normal de invocación utilizando clientes replicados.

C5. En este último caso, si el cliente no estaba replicado la invocación ya ha sido dada por terminada, con lo que no resulta necesario hacer nada más.

Por el contrario, si el cliente estaba replicado, otra de sus réplicas reintentará la invocación fiable que había fallado previamente. Cuando la invocación llegue al nuevo coordinador elegido, éste encontrará el RoiID asociado a este nuevo intento en el área de resultados retenidos. Con ello, se recogerán tales resultados y se retornarán de inmediato al nuevo cliente, junto con una referencia al CObj. De esta forma continuará el protocolo para clientes replicados a partir del paso 7, como si hubiera sido una invocación normal.

4.4.7 Caída de coordinador y serializador

Como ya hemos comentado previamente, el serializador estará replicado de tal manera que resultará prácticamente imposible su fallo total. Con ello este caso de fallo se reduce al de la caída única del coordinador, con lo que debe consultarse la sección 4.4.4 para encontrar su descripción.

4.4.8 Caída de todas las réplicas servidoras

En este caso se asume que han caído tanto la réplica coordinadora escogida inicialmente como todas las demás réplicas existentes para el objeto invocado.

Detección

Esta situación será detectada por nuestro soporte incluido en el ORB del dominio cliente que tras realizar la oportuna reconfiguración tratará de reemplazar la réplica coordinadora por cualquiera de los demás, pero se dará cuenta de que ninguna de tales réplicas podrá ser invocada pues todas ellas han dejado de existir.

Tratamiento

El ORB del dominio cliente generará una excepción que llegará al programa cliente comunicándole la caída de todas las réplicas del objeto que se había invocado previamente. En este caso no se podrá hacer nada más. El programa cliente tampoco podrá efectuar ningún reintento adicional. Como máximo podría reiniciar el programa servidor que regenere al objeto cuyas réplicas acaban de caer.

4.4.9 Caída de todas las réplicas clientes

Esta situación es algo más problemática que la anterior, pues deberá desaparecer eventualmente todo el contexto asociado a las invocaciones fiables iniciadas por el cliente que acaba de fallar. Esto será posible en nuestro protocolo gracias al uso de los objetos RoiID.

Detección

Si se tenía un cliente replicado, la detección de la caída de todas sus réplicas podrá lograrse examinando durante las fases de reconfiguración tras el fallo cuál es el estado de las referencias cliente

para el RoiID. Si se invoca durante esta reconfiguración el método `CORBA::Object::non_existent()` de la referencia cliente del RoiID en los dominios coordinador o cohortes y tal invocación devuelve el valor cierto, entonces todas las réplicas del programa cliente habrán fallado, pues en cada dominio cliente ha debido generarse una réplica del objeto RoiID en el paso 1 de la variante para clientes replicados del protocolo IFO.

Tratamiento

En caso de detectarse tal situación, tanto los dominios cohortes como el coordinador se anotarán en un variable booleana que el cliente ha fallado y en base a esto el coordinador ya no intentará devolver el resultado hacia el dominio cliente y los cohortes descartarán los resultados retenidos y los CObjs en cuanto reciban el último mensaje de actualización de estado.

4.5 Trabajo relacionado

En diversos trabajos se han dado otros intentos de garantizar la fiabilidad (entendiendo por tal la atomicidad, consistencia y recuperación en caso de fallo) de las invocaciones realizadas sobre objetos o procesos replicados. En las próximas secciones se da un pequeño estudio de estas soluciones, clasificándolas según el modelo de replicación asumido o las técnicas empleadas como base para implantarlas.

4.5.1 Modelo pasivo

Los primeros trabajos relacionados con invocaciones fiables se dieron en el modelo de replicación pasivo, también llamado entonces *modelo primario-copia* [AD76]. En estos primeros sistemas la petición llegaba en primer lugar a la réplica primaria, la cual la servía localmente para después enviar las actualizaciones de estado hacia las réplicas secundarias (puede que sólo haya una réplica de este segundo tipo). La respuesta era enviada hacia el cliente bien por la única réplica secundaria tras haber actualizado su estado, o bien por la réplica primaria tras haber efectuado el envío del mensaje de actualización si se asume que existe un protocolo de transporte fiable o no pueden haber fallos de interconexión. Para completar el mecanismo de invocación, se utilizaban mensajes periódicos de comprobación de estado o *mensajes de heart-beat*.

Como puede observarse, el principio de estos protocolos es similar al empleado en HIDRA. Por una parte, existe un mecanismo de detección directa de fallos entre las réplicas, gracias al envío de mensajes periódicos. Por otra parte, el cliente también tiene medios suficientes como para detectar el fallo de alguno de los agentes participantes en la invocación y reintentar ésta. Esto último no resulta difícil aquí, pues basta con esperar el mensaje de respuesta durante un tiempo prefijado y reintentar en caso de que no llegara ésta.

Estos primeros protocolos plantearon algunos problemas al no utilizar todavía un algoritmo de pertenencia como su base para la detección de fallos. El envío de mensajes periódicos permitía que o bien el primario o el secundario detectaran el fallo de la otra réplica, pero los clientes no podían enterarse al mismo tiempo sobre lo que estaba sucediendo y debían diseñarse soluciones específicas para reconfigurar las referencias a la réplica primaria mantenidas por estos clientes. Además, también existía el peligro de que se perdiera la petición en curso si caían simultáneamente

las réplicas primarias tanto del cliente como del servidor, si éstas no habían realizado ninguna actualización de estado.

En [BBG+89] se describe una técnica que resuelve este último problema y que perfecciona así el mecanismo de invocación empleado en el modelo pasivo. Su solución consiste en emplear un mecanismo de difusión atómica ordenada. Para ello, cuando un proceso replicado debe invocar a otro, no solamente hace llegar tal petición a su réplica primaria, sino que también entrega ésta a las secundarias, tanto del cliente como del servidor. Las réplicas secundarias no procesan directamente los mensajes, pero los tienen disponibles en caso de fallo para poderlos procesar entonces. Además, estas réplicas secundarias guardan un contador de actualizaciones con el que pueden saber cuántos mensajes ya pueden ser descartados por haber recibido las actualizaciones emitidas por el primario tras haber procesado por completo tales mensajes.

Esta técnica permite garantizar progreso al igual que el mecanismo de invocaciones fiables que hemos presentado en este capítulo. Sin embargo, a diferencia de nuestra solución, requiere un mecanismo adicional de ordenación de difusiones. El protocolo empleado para ello en [BBG+89] se basa en una reserva de bus, por lo que no es caro en cuanto al tiempo que requiere ni en cuanto al número de mensajes que necesita, pero no es fácilmente migrable a otros equipos diferentes al propuesto en dicho trabajo.

Otra solución para las invocaciones fiables en el modelo pasivo aparece en [Ghe90], integrada en el lenguaje de programación *Argus*. En su variante del modelo pasivo (llamada "*viewstamped replication*") se realizan las actualizaciones de estado de manera asíncrona, con lo que no se puede garantizar que cuando el cliente reciba la respuesta todas las réplicas del objeto invocado mantengan un estado consistente. Para lograr que esta consistencia se dé utilizando actualizaciones asíncronas, se usa como complemento las transacciones anidadas, asociando una subtransacción con cada una de las invocaciones realizadas. Para que la transacción raíz pueda ser confirmada, todas las subtransacciones participantes deben comprobar previamente que los resultados de dicha transacción han sido recibidos por una mayoría de las réplicas de cada uno de los objetos invocados.

En Argus, el uso de las transacciones anidadas resulta transparente para el programador, pues está integrado en el soporte que proporciona el lenguaje. Así, no resulta necesario que el programador utilice ninguna sentencia para iniciar una transacción, sino que es el lenguaje quien las genera al observar una invocación sobre un objeto replicado. El protocolo de *confirmación* de estas transacciones se inicia cuando la operación o método que dio inicio a la transacción ha terminado y devuelve el resultado a su cliente. Como puede verse, el soporte es similar al empleado en nuestro protocolo IFO. Además, ya que en Argus se utilizan transacciones anidadas, el aborto de una subtransacción no significa que la invocación completa (asociada a la transacción raíz) deba abortarse, sino que la subtransacción podrá reintentarse.

El modelo de replicación utilizado en este lenguaje maneja identificadores de "vista" o configuración, que son incrementados cada vez que se produce un cambio en el objeto replicado (se añade una réplica o falla alguna). Las transacciones también manejan estos identificadores para conocer el estado en que se encontraba cada réplica de todos los objetos participantes en una transacción con descendientes y utilizan esta información para determinar si una transacción raíz podrá ser confirmada o no.

4.5.2 Modelo activo

Una de las primeras propuestas de replicación siguiendo el modelo activo fue la iniciada por Cooper en su sistema *Circus* [Coo84, Coo85], quien propuso un primer mecanismo de invocación de objetos replicados. Para poder realizar las invocaciones de *módulos* (nombre que recibían los objetos replicados en Circus), en este trabajo se diseñaron varios protocolos a los que en su conjunto se les dio el nombre de *llamada replicada a procedimiento*. Así, se necesitó un protocolo para realizar la invocación desde un cliente simple a un servidor replicado, que constituye el caso más básico. Además, se diseñó un protocolo para realizar invocaciones desde clientes replicados a servidores replicados o simples, donde había que efectuar un filtrado de peticiones y una replicación de respuestas.

La solución de Cooper no estaba basada en los protocolos de difusión atómica ordenada que llegaron a proponerse posteriormente, sino que protegía y ordenaba sus invocaciones mediante lo que en sus trabajos se llamó *transacciones ligeras replicadas*. Estas transacciones utilizaban marcas temporales para llevar a cabo el control de concurrencia. La gestión propia de las transacciones garantizaba su progreso siempre y cuando quedase alguna réplica libre de fallo.

Por otra parte, la falta de un protocolo de difusión atómica ordenada evitaba que se pudiera dar ninguna garantía en el orden en que las diferentes peticiones llegaban a cada una de las réplicas. Con ello resultaba posible que se dieran interbloqueos a la hora de iniciar el servicio de las invocaciones si el orden de entrega había resultado distinto para dos invocaciones en conflicto (ambas conseguirían reservar parte de las réplicas pero no la totalidad, con lo que no podrían proseguir si ambas necesitaban modificar el estado del objeto). Por ello, en el soporte que se dio en este sistema se introdujo también un mecanismo de detección y resolución de interbloqueos.

Posteriormente, en el sistema *Isis* [BvR94] desarrollado por la *Univ. de Cornell* también se adoptó el modelo de replicación activo y la *sincronía virtual* [BJ87]. Para ello se desarrollaron varios protocolos de difusión atómica ordenada, con lo que se garantizaba que todas las réplicas de un mismo objeto recibieran todas sus peticiones en un mismo orden (al menos si se utilizaba el *protocolo ABCAST*, que respeta un *orden total causal* [HT93]). Si todas las réplicas reciben las peticiones que puedan tener algún conflicto en un mismo orden y además se utiliza una implementación sin múltiples *hilos de ejecución* desaparece la posibilidad de interbloqueos e inconsistencias que se planteaba en el sistema Circus. Todo esto tuvo como consecuencia una simplificación en el soporte necesario para garantizar la fiabilidad de las invocaciones en este modelo de replicación. Con ello se eliminó el soporte transaccional utilizado en Circus tanto para ofrecer fiabilidad como para implantar el control de concurrencia. Ahora el control de concurrencia es establecido por el propio protocolo de difusiones atómicas ordenadas, pues cada réplica sólo podrá atender una petición a la vez y este protocolo dicta el orden en que deben ser entregadas tales peticiones.

La sincronía virtual, por su parte, complementa los servicios de estos protocolos de difusión atómica ordenada proporcionando garantías adicionales en la *entrega* de los mensajes. En estos protocolos se distingue entre la *recepción* de un mensaje por su nodo destino y la *entrega* de éste. La entrega al proceso destino debe retrasarse hasta que haya suficiente información acerca de la recepción de dicho mensaje por todos los demás destinos a los que iba dirigido.

Estas garantías adicionales en la entrega se reducen a exigir que se siga un cierto orden en la entrega, que como mínimo debe ser causal, y que las entregas se realicen a ser posible en una configuración del grupo idéntica a la existente durante su emisión o en una configuración

transitoria que se utilice antes de instalar la siguiente configuración. El objetivo es proporcionar una imagen de sincronismo lógico, de manera que todos los procesos que figuren como destinos de una difusión reciban tal mensaje en un mismo instante lógico. Esto facilitará bastante las tareas de programación en un sistema que respete este modelo, pues el programador observará pocas diferencias entre este sistema y un sistema centralizado.

Sin embargo, aunque esto facilita un mecanismo aceptable para garantizar la fiabilidad de las invocaciones, pues únicamente hay que preocuparse por el protocolo de difusión atómica y algunos protocolos complementarios para tratar dichas difusiones cuando falla algún proceso destino, el propio modelo activo ofrece algunas características que resultan difíciles de gestionar. Estas dificultades ya se presentaron cuando se describió el sistema Circus al empezar esta sección: necesidad de filtrado de peticiones y repetición de respuestas cuando deben intercomunicarse dos objetos replicados o cuando un objeto replicado requiere los servicios de uno simple. Esto complica bastante los protocolos finales que deben utilizarse para implantar las invocaciones en este modelo, necesitando al final varias rondas de intercambio de mensajes, con el correspondiente coste en número de mensajes y retardo necesario.

De todas formas, el soporte desarrollado en el sistema Isis fue posteriormente adoptado por todos los demás sistemas que desarrollaron soluciones empleando el modelo de replicación activo. Como ejemplos cabe citar los sistemas *Relacs* [BBD96], *Totem* [AMMS$^+$93, AMMS$^+$95] y *Transis* [DM96], que extendieron los protocolos de difusión y las garantías de la sincronía virtual para que pudieran ser implantados en *sistemas particionables*. Con ellos surgió la *sincronía virtual extendida* [MAMSA94] para este tipo de entornos.

4.5.3 Modelo coordinador-cohorte

El diseño original de un protocolo de invocación para el modelo coordinador-cohorte [BJRA85] en la primera versión del sistema Isis asociaba al igual que nuestro protocolo IFO un identificador a cada invocación y mantenía también *resultados retenidos* para evitar reejecuciones en caso de fallo. Pero esta primera solución no aportaba ninguna regla práctica sobre cómo (y cuándo) debían ser descartados estos resultados, principalmente porque no existía ningún subprotocolo que asegurase que todas las réplicas habían sido actualizadas cuando la invocación terminaba. Esta solución confiaba únicamente en la sincronía de todas las actualizaciones de estado efectuadas. Además, no se tenía en cuenta que pudiese fallar un objeto al completo y las consecuencias que esto podría tener en los objetos a los que había invocado previamente, los cuales mantendrían de forma indefinida los resultados retenidos.

Estos problemas fueron resueltos en prototipos posteriores del sistema Isis [Bir85] tomando algunas de las ideas propuestas en el sistema Circus que hemos descrito anteriormente [Coo85]. Estas mejoras se centraron en la adopción de *transacciones anidadas* [Mos81], asociando una transacción a cada invocación y un identificador a cada subtransacción generada, protocolos de difusión atómica ordenada para realizar las actualizaciones de estado (con lo que ya se introdujo un coste similar al de la replicación activa) y control de concurrencia basado en cerrojos y ligado a las transacciones anidadas. Se siguió manteniendo el soporte a los resultados retenidos, pero ahora sin ningún problema asociado, ya que la duración de estos resultados estaba asociada a la de la subtransacción a la que estaban ligados y si ésta abortaba o terminaba, los resultados eran eliminados. La detección de fallos y su tolerancia se integró en el soporte transaccional asociado

a las invocaciones, con lo que el sistema resultante ofrecía unas garantías similares a las nuestras, pero un mayor coste debido a la utilización de protocolos de difusión atómica para realizar las actualizaciones de estado y las reservas de cerrojos.

En ese mismo prototipo también se fijó el criterio para decidir en cada invocación qué réplica debía ser elegida como coordinadora. Para ello se asociaban a los nodos unos identificadores que daban una cierta idea de la proximidad física entre ellos y se elegía como coordinador a la réplica activa más cercana al cliente. Ese mismo criterio puede adoptarse en HIDRA si se utiliza un protocolo de transporte fiable no estándar. Por otra parte, algunos protocolos de red, como por ejemplo *IPv6* [Com99] facilitan ciertos tipos de direcciones de red, como es el caso de las *direcciones anycast* (también llamadas *direcciones de cluster*) que corresponden a un conjunto de nodos y donde los mensajes únicamente se hacen llegar a uno de los nodos del conjunto, en concreto al más cercano al emisor del mensaje. Este tipo de direcciones se pueden adoptar en HIDRA para dar soporte a este modelo de replicación para elegir a la réplica coordinadora, siempre y cuando el objeto tenga una réplica en cada nodo del cluster.

Posteriormente el sistema Isis adoptó el modelo de replicación activo y propuso la sincronía virtual [BJ87], pero no se abandonó por completo el modelo coordinador-cohorte que pasó a implantarse por encima del activo. Esta solución evita tener que utilizar el soporte transaccional junto a los protocolos de difusión atómica. Sin embargo, el coste de los protocolos de difusión atómica para poder realizar las invocaciones del modelo coordinador-cohorte sigue siendo alto y excesivamente complicado cuando se realizan invocaciones donde tanto el cliente como el servidor son objetos replicados.

4.5.4 Soporte transaccional

El diseño original de las *transacciones anidadas* para sistemas distribuidos [Mos81] sirvió para proteger mediante estas transacciones a una cadena de invocaciones entre procesos que podían estar ubicados en diferentes nodos. En la propuesta de Moss se utilizaba un *protocolo de confirmación de dos fases* para concluir cada una de las subtransacciones generadas. El control de concurrencia se realizaba utilizando *cerrojos* y siguiendo también un *protocolo de cierre de dos fases* [Gra79], donde los cerrojos de cada subtransacción que concluía eran heredados por su transacción antecesora y una subtransacción podía obtener un cerrojo si la única transacción que mantenía ese cerrojo en un modo conflictivo era una de sus antecesoras.

Debido a la posibilidad de interbloqueos, se admitían los abortos de subtransacciones y éstos no forzaban a que la transacción en la que estaba incluida la subtransacción abortada debiera abortar. Por tanto, resulta posible en este entorno reintentar subtransacciones que han fallado previamente.

En este modelo, se genera una *transacción raíz* cuando un cliente inicia una invocación sobre un determinado objeto. Dentro de una transacción determinada se pueden generar *subtransacciones* si su hilo de ejecución realiza invocaciones de otros objetos externos. Si se da soporte a múltiples hilos de ejecución en el código de un mismo objeto es posible que la cadena de invocaciones se ramifique y que surja un árbol transaccional que incluya múltiples transacciones que se ejecuten concurrentemente.

La extensión de este soporte de transacciones anidadas para objetos replicados es bastante sencilla. Basta con incluir múltiples agentes en cada subtransacción, cada uno de ellos asociado con

cada una de las réplicas del objeto invocado. Sin embargo, la semántica normalmente asociada con una subtransacción obligaría a su aborto en caso de fallo de alguna de las réplicas participantes en la invocación, cosa que no procede si tenemos en cuenta los objetivos fijados a la hora de diseñar nuestro protocolo IFO. Por tanto, a la hora de adaptar este soporte transaccional en entornos con objetos replicados se han dado múltiples soluciones, de entre las que presentaremos algunos ejemplos seguidamente.

Uno de los primeros sistemas distribuidos que adoptó soporte transaccional en su núcleo y que admitía transacciones anidadas fue *Clouds* [All83, Ken86], desarrollado en el *Georgia Institute of Technology*. El soporte facilitado por este sistema no era transparente para el programador. Es decir, se debían utilizar explícitamente las operaciones necesarias para generar una transacción y asociar ésta con las invocaciones a objeto que se desearan proteger. Este soporte no fue ideado para gestionar objetos replicados, por lo que no existen extensiones diseñadas para proteger las invocaciones sobre este tipo de objetos en el sistema Clouds.

El soporte para *replicación con marcas de vista* (o *viewstamped*) [Ghe90] en el lenguaje Argus ya fue analizado en la sección 4.5.1. En este sistema, las transacciones anidadas se necesitaban para garantizar la consistencia de las invocaciones y forzaban a que las actualizaciones asíncronas de estado sobre las réplicas secundarias fueran completadas antes de confirmar y terminar una invocación.

En [HT92] encontramos un soporte para transacciones en el sistema *Apertos* desarrollado por la *Universidad Keio de Yokohama* (un *sistema distribuido orientado a objetos*) que al igual que nuestro mecanismo IFO utiliza objetos auxiliares incluidos en tal soporte para modelar sus transacciones. Sin embargo, el modelo transaccional propuesto en este trabajo es bastante más simple que el presente en HIDRA. Por una parte no se soportan objetos replicados sino únicamente simples, aunque se pueden encadenar múltiples invocaciones anidadas en una misma transacción. Por otro lado, no se admite concurrencia en la invocación de métodos; es decir, en un momento determinado sólo podrá haber un método en ejecución en cada objeto, debiendo encolarse el resto de invocaciones recibidas, esperando su turno. Por último, el inicio y terminación de una invocación no resulta transparente para el programador, quien deberá dar órdenes explícitas para realizar tales tareas.

En Apertos se asocia un *objeto gestor de transacciones* o *TMO* a cada transacción. Con ello se elimina la necesidad de un componente global para todo el sistema distribuido, ya que la gestión se realiza en base a los TT.MM.OO., los cuales almacenan el contexto de su transacción asociada. De esta manera, cada transacción registra en su TMO el conjunto de objetos que ha ido accediendo, de manera que cuando se solicite la invocación de un nuevo objeto se pueda averiguar si la transacción invocante está en conflicto con alguna otra de las que están encoladas o con la que está en ejecución en dicho objeto, reordenándolas de manera que se garantice cierto *orden de serie* entre dichas transacciones. Con ello se evita que se den inconsistencias en el estado de varios objetos debido a una incorrecta mezcla en las ejecuciones de un conjunto de transacciones.

El problema resuelto en [HT92] no es exactamente el mismo para el que se ha diseñado el protocolo IFO, sino que aparece cuando se consideran las inconsistencias que pueden causar las cadenas de invocaciones. Esto también se puede resolver utilizando transacciones anidadas y en HIDRA pueden utilizarse gestores externos de transacciones (asociando una transacción anidada con cada cadena de invocaciones) para tratarlo.

Otro soporte transaccional para mejorar la fiabilidad de las invocaciones sobre objetos replicados se ha dado en el sistema *Arjuna* [LS93, LMS93] de la *Universidad de Newcastle*. En Arjuna se permite tanto el modelo de replicación pasivo como el activo y en este último el uso de protocolos de difusión atómica ordenada es opcional. Por otro lado, el estado de los objetos replicados es persistente, es decir, cuando una actualización ha terminado el estado del objeto se copia en *almacenamiento estable*. Esta última característica permite desactivar los objetos, eliminando temporalmente sus procesos servidores hasta que nuevas invocaciones los activen de nuevo, regenerando los procesos que les dan servicio.

Las transacciones de Arjuna en el modelo pasivo sirven para detectar en primer lugar si existe una réplica primaria en algún nodo del sistema. De no ser así, el sistema procederá a activar una y dirigir la invocación hacia ella. Tanto en el modelo activo como en el pasivo, una transacción puede englobar la invocación de múltiples objetos (y de las cadenas que generen hacia otros objetos externos). Cuando la transacción termina, en este modelo se procederá a guardar en almacenamiento estable una copia del estado del objeto y si no hay otras transacciones pendientes, se suspenderá su servidor.

En el modelo activo, si no se utiliza ningún protocolo de difusión atómica ordenada, el uso de transacciones servirá además de lo visto en el modelo anterior, para realizar la reserva de las réplicas (utilizando cerrojos). Si se dan conflictos entre dos invocaciones, de manera que ambas consigan parte de los cerrojos pero no la totalidad de las réplicas y deseen realizar una operación de escritura, el gestor transaccional de Arjuna procederá a dar precedencia a una de las dos transacciones, bloqueando temporalmente a la otra. Con ello se logran los mismos resultados que con los protocolos de difusión atómica si consideramos un objeto replicado en particular, pero se logra un mejor control si se tiene en cuenta que una transacción puede englobar un conjunto de operaciones sobre una colección de objetos replicados. Este segundo punto no está recogido en el soporte tradicional de los sistemas que emplean el modelo de replicación activo y suele necesitarse en algunas aplicaciones para las que el sistema Arjuna será especialmente recomendable.

Al igual que ocurría en el sistema Apertos, en Arjuna el uso de transacciones tampoco resulta transparente al programador. En este sistema se deberá declarar un objeto del tipo `AtomicAction` dentro del código que se está implantando y llamar explícitamente a sus métodos `begin()` y `end()` para definir el conjunto de operaciones que estarán englobadas en la transacción.

4.6 Conclusiones

Como hemos visto en la sección anterior, las garantías proporcionadas por el mecanismo IFO han sido utilizadas (todas o parte de ellas) en otros entornos altamente disponibles. Del soporte de invocación para el modelo pasivo se puede extraer como aportación al diseño de nuestro mecanismo que sin un protocolo de pertenencia será difícil implantar tolerancia a faltas en los componentes utilizados para el desarrollo de nuestra solución. Los primeros mecanismos de invocación para este modelo no utilizaron protocolos de pertenencia y no pueden catalogarse como verdaderamente tolerantes a situaciones en que se den fallos. Observando los trabajos realizados en Argus, también se concluye que para garantizar la consistencia del objeto replicado será necesario algún soporte similar al transaccional. Sin embargo, las soluciones implantadas en algunos de estos sistemas nunca han tenido especial cuidado en garantizar la propiedad de progreso, que en nuestro

mecanismo ha sido uno de los principales objetivos.

De la comparación con las soluciones para el modelo pasivo se desprende que en nuestro caso el soporte transaccional está fuertemente integrado en el mecanismo IFO. De hecho, la mayor parte de los objetos auxiliares que utilizamos internamente en el ORB tienen como objetivo proporcionar algunas de las garantías que tradicionalmente ofrecen las transacciones. Sin embargo, utilizar transacciones junto a invocaciones "normales" no puede resultar transparente al programador de aplicaciones clientes de los objetos replicados. Aunque su uso y gestión no es muy complicado, el mecanismo IFO elimina la necesidad de que el programador utilice estos servicios.

Por otra parte, la detección de fallos realizada en los niveles inferiores de HIDRA mediante un monitor de pertenencia permiten que los protocolos de cuenta de referencias trabajen adecuadamente. Esta cuenta de referencias es empleada por los objetos auxiliares de nuestro protocolo y permite su funcionamiento normal aunque se produzcan fallos. Es decir, de esta forma evitamos reconfigurar el estado de una IFO aunque se presenten fallos durante su realización. Esto ya supone una mejora importante respecto a gran parte de los mecanismos de invocación estudiados en el modelo pasivo. Los empleados en el modelo activo también suelen emplear estos servicios de monitorización.

Respecto al modelo activo poco se puede decir. Son modelos diferentes y cada uno podrá ser más apropiado que el otro según de qué aplicación estemos hablando. El coste final necesario para garantizar la consistencia en caso de invocaciones que modifiquen el estado de un objeto replicado no será muy diferente en ambos casos. Mientras que el modelo activo necesitará protocolos de difusión atómica y filtrados de peticiones y respuestas (esto último sólo en algunos casos), nuestro mecanismo IFO también requerirá los mensajes adicionales de los protocolos de cuenta de referencias, por lo que al final el coste será similar.

Por lo que respecta a las soluciones inspiradas en la descripción original del modelo coordinador-cohorte y el soporte de invocación utilizado entonces, nuestra solución es similar a la descrita en [BJRA85] pero evita los problemas presentados en aquél sistema. Su problema principal era el mantenimiento indefinido de los resultados retenidos en algunas situaciones, debido a que su solución confiaba en la sincronía de las invocaciones realizadas y no utilizaba ningún mecanismo auxiliar en caso de fallo de algún componente. Nuestro protocolo es ligeramente más complicado, pero tolera muchas más situaciones de fallo que aquel diseño original. Las soluciones posteriores que combinaron el soporte inicial con transacciones anidadas y difusiones atómicas para las actualizaciones de estado trataron de corregir estos defectos, pero dieron como resultado un sistema cuyo coste era mucho más alto de lo que convenía aceptar. Su tercer rediseño dio como resultado el modelo de replicación activo, que proporciona un modelo de aplicación bastante diferente.

Una última ventaja de este mecanismo es que no mantiene fuertes dependencias respecto al resto de componentes de nuestra arquitectura HIDRA. En concreto, este soporte para garantizar invocaciones fiables se ha podido prototipar [MGB99b] sobre OO.RR.BB. completamente estándares y que ofrecían soporte para interceptores. En esos prototipos se tuvo que simular el soporte a objetos replicados manteniendo en cada uno de ellos las referencias a los demás, exigiendo una comprobación periódica de la validez de dichas referencias. Teniendo dicho soporte, es factible utilizar este mecanismo de invocación, aunque con algunas limitaciones (Obviamente, el tiempo de reconfiguración de una IFO en caso de fallo sería mayor).

Por todo esto se puede decir que nuestro mecanismo de invocación fiable no resulta fácilmente

comparable con ningún otro servicio de este tipo. Básicamente, porque no ha habido soluciones desarrolladas para este mismo entorno.

En cualquier caso, las aportaciones principales de este mecanismo se centran en su transparencia para el programador de objetos replicados y en las garantías de evitación de reproceso de peticiones gracias a los resultados retenidos y al progreso de las invocaciones.

Capítulo 5

Control de concurrencia

5.1 Introducción

El modelo de replicación coordinador-cohorte, al permitir que cada invocación tenga una réplica activa diferente y que haya múltiples invocaciones iniciadas simultáneamente introduce más dificultades que los demás modelos de replicación a la hora de mantener un control de concurrencia. Por otra parte, como las réplicas de los objetos a gestionar estarán normalmente ubicadas en nodos diferentes, el mecanismo de control de concurrencia que se adopte deberá ser forzosamente distribuido.

Por el contrario, en el modelo de replicación pasivo el control de concurrencia puede ser local ya que únicamente existe una réplica primaria que está sirviendo todas las peticiones de los clientes. En este modelo, si no se soportan múltiples hilos de ejecución en la codificación del programa servidor, el control de concurrencia resulta innecesario. En caso de dar soporte multihilo, el control se reducirá a la adopción de mecanismos de sincronización propios del lenguaje de implementación utilizado o facilitados por el sistema operativo que gestione el nodo sobre el que se está ejecutando la réplica primaria. Estos mecanismos de sincronización podrán ser *semáforos* [Dij65], *mútex* (semáforos binarios o cualquier otro mecanismo que sirva para garantizar exclusión mutua en el acceso a cierta región de código), *variables de condición*, etc.

En el modelo de replicación activo la situación resulta similar a la presentada para el modelo pasivo. Las implementaciones actuales del modelo activo suelen utilizar protocolos de difusión atómica ordenada mediante los cuales se asegura que todas las réplicas de cada objeto reciban las peticiones en el mismo orden. Este orden resultante tras la entrega de las peticiones es el que debe utilizarse para procesar tales peticiones. Con ello suele ser difícil implantar el código de estos objetos dando soporte para múltiples hilos de ejecución puesto que si se dispusiera de un hilo para atender cada petición, el orden de servicio dependería del planificador de procesos utilizado por el sistema operativo. Por tanto, cada réplica podrá utilizar únicamente un hilo de ejecución con el que sólo podrá atender una petición a la vez. El protocolo de difusión dictará cuál será la petición a servir, con lo que desaparece el problema del control de concurrencia aunque esto tenga como resultado una ejecución estrictamente secuencial de todas las peticiones.

Como resultado de la situación que se acaba de describir, en HIDRA se necesitará un mecanismo de control de concurrencia distribuido que permita mejorar el rendimiento del modelo de replicación coordinador-cohorte, al menos cuando se compare con los otros dos modelos posibles.

Una de las primeras decisiones que deberá adoptarse se centra en la granularidad del control a efectuar. Existen dos alternativas principales. La primera consiste en mantener un control sobre las operaciones, sin entrar en detalles sobre su implementación interna ni sobre los recursos a los que se accede en cada una de ellas. Simplemente se debe conocer qué pares de operaciones no pueden conmutar (*operaciones no conmutables*), es decir, aquellos pares para los que el cambio de orden en su ejecución produciría un cambio en el estado del objeto o en los resultados proporcionados por tales operaciones. También diremos que las operaciones que cumplan dicha característica son *operaciones en conflicto*. Una vez conocidos todos los conflictos existentes entre las operaciones de una misma interfaz, bastaría con controlar qué invocaciones iniciadas existen en cada momento, suspendiendo temporalmente a las nuevas que estén en conflicto con alguna invocación en curso.

La otra aproximación posible consiste en utilizar alguno de los mecanismos de sincronización locales y extenderlo mediante una implementación distribuida, utilizándolo de manera similar a como se hacía en un sistema centralizado. Esto permitiría un control más fino de la concurrencia, pues las herramientas de sincronización podrían proteger fragmentos de código bastante más pequeños que en el caso anterior. Sin embargo, esto plantea el problema de que el control de concurrencia requeriría la participación del programador de los objetos replicados, mientras que en el caso anterior el control se puede automatizar e incluir en el soporte proporcionado por el propio sistema. Un ejemplo de algoritmo para extender los mútex para un sistema distribuido puede encontrarse en [BA89].

En nuestro caso se ha optado por la primera alternativa, esto es, mantener un control de concurrencia con granularidad de operación e incluyendo el soporte necesario para llevarlo a cabo dentro de nuestra arquitectura HIDRA, haciéndolo transparente para el programador de objetos replicados.

El resto del capítulo se ha estructurado como sigue. En la sección 5.2 se presentan los objetivos que deben ser satisfechos por el mecanismo de control de concurrencia de HIDRA. Antes de estudiar el diseño e implementación de ese mecanismo se describen en la sección 5.3 los principales mecanismos de control de concurrencia que se han llegado a diseñar en distintos ámbitos que presenten alguna relación con HIDRA. En la sección 5.4 se presenta el concepto de *potencia expresiva* y se describe de qué forma se puede mejorar esta característica en un mecanismo de control de concurrencia siguiendo las indicaciones citadas en [Blo79]. Visto esto, la sección 5.5 describe el mecanismo de control de concurrencia diseñado para HIDRA y en la sección final se compara nuestro trabajo con otros similares.

5.2 Objetivos

El mecanismo de control de concurrencia que se implante en HIDRA deberá satisfacer los siguientes objetivos:

O1. Debe serializar el acceso entre múltiples invocaciones sobre operaciones en conflicto para un mismo objeto replicado siguiendo el modelo coordinador-cohorte. Debe permitir la ejecución concurrente de todas las operaciones que no presenten conflictos.

O2. Debe tolerar fallos. El mecanismo debe seguir funcionando correctamente aunque caigan varios nodos del sistema distribuido y siempre que quede alguna réplica del objeto al que ha intentado invocar una determinada IFO.

O3. Debe ser transparente. El programador de un objeto replicado no debe tener en cuenta las tareas de control de concurrencia a la hora de escribir el código de sus objetos replicados. Como máximo deberá dar una especificación sobre las operaciones de las interfaces que estén en conflicto.

04. Debe ser *pesimista*. Es decir, las decisiones que se adopten para restringir la concurrencia deben ser lo más estrictas posible, de manera que posteriormente no necesiten ser corregidas. Si esta corrección tuviera que darse, implicaría el aborto de la invocación en curso, cosa que incumpliría los objetivos fijados para el mecanismo de invocaciones fiables.

Además de esto, se exige que la solución proporcionada sea lo más eficiente posible ya que la arquitectura HIDRA también ofrecerá su soporte para implantar algunos componentes del sistema operativo distribuido sobre el que esté funcionando. Para que el sistema ofrezca unos servicios eficientes, cosa que exigirán la mayor parte de sus aplicaciones, se requiere que todos aquellos servicios utilizados por sus componentes también sean eficientes.

El primero y tercero de los objetivos presentados aconseja utilizar una solución con granularidad de operación, puesto que una solución más fina exigiría la participación directa del programador de objetos replicados, eliminando la transparencia que se ha demandado en el objetivo O3. Ese mismo requerimiento se traducirá en la necesidad de utilizar algún lenguaje de especificación que permita indicar qué operaciones de una misma interfaz (o de cualquier interfaz básica de la que se herede directa o indirectamente) podrán ser concurrentes y qué otras presentarán conflictos.

El segundo objetivo se traducirá en que los agentes que participen en el control de concurrencia deberán estar también replicados, de manera que pueda reconstruirse su estado en caso de fallo. No obstante, el modelo de replicación que se utilizará para resolver esta cuestión será ciertamente especial y no concuerda con ninguno de los modelos presentados en capítulos anteriores.

Por último, tal como sugiere el cuarto objetivo, el control de concurrencia debe realizarse a priori, utilizando técnicas pesimistas. Si se utilizara una técnica de *control de concurrencia optimista* en la que la comprobación de conflictos se diera a posteriori se correría el peligro de detectar demasiado tarde los conflictos entre las invocaciones finalizadas. Esto conllevaría dejar el objeto replicado en un estado inconsistente, debiendo abortar alguna de las invocaciones realizadas y restaurar el estado previo del objeto.

5.3 Mecanismos de control de concurrencia

Una de las primeras áreas de aplicación de los mecanismos de control de concurrencia fue la gestión de bases de datos donde se implantaron los cerrojos (o, en inglés, *locks*) para controlar el acceso sobre los registros y también otras soluciones basadas en *marcas temporales* (o *timestamps*). En estas primeras soluciones no se contemplaba la replicación de los objetos protegidos. Posteriormente, dentro de esa misma área han aparecido otras soluciones, como las *votaciones*, que contemplan la replicación de las bases de datos.

Seguidamente se describirán estos mecanismos de control de concurrencia con mayor detalle y se analizarán también otros que se han utilizado en lenguajes de programación con soporte a múltiples hilos de ejecución o en sistemas operativos.

5.3.1 Cerrojos

Los cerrojos [Gra79] se han utilizado tradicionalmente para proteger cada una de las tablas o rangos de registros en una determinada *base de datos relacional*, aunque en general podemos suponer que sirven para controlar el acceso sobre cualquier recurso que pueda utilizar una transacción. Para ello, existen dos modos de utilización de los cerrojos: *exclusivo* o *de escritura*, cuando la transacción debe modificar el contenido de la *relación*, o *compartido* o *de lectura*, cuando la transacción únicamente quiere consultar el valor de los registros existentes.

El objetivo del control de concurrencia basado en cerrojos es gestionar los recursos que las transacciones podrán utilizar de manera que cuando éstas finalicen el resultado de sus acciones sobre dichos recursos sea el mismo que el de cualquiera de las *ejecuciones en serie* de tales transacciones. Para que la ejecución de un conjunto de transacciones pueda ser considerada equivalente a una ejecución en serie (es decir, a una ejecución en la que cada transacción se hubiese iniciado después de haber terminado la anterior) se requiere que el *grafo de serialización* asociado a tales transacciones no presente ningún ciclo [PBR77]. Este grafo se construye considerando todas las transacciones a estudiar como sus nodos y añadiendo aristas dirigidas desde una transacción T1 a una transacción T2 si T1 mantuvo un cerrojo sobre cierto recurso antes que T2 y los cerrojos presentaban conflictos (es decir, no eran ambos de lectura).

El protocolo necesario para utilizar los cerrojos ha sido tradicionalmente el de cierre en dos fases o *protocolo 2PL*. En él se exige que una transacción realice en primer lugar una *fase de adquisición* o reserva de los cerrojos hasta que resulte necesario liberar alguno de ellos. En ese instante empieza la *fase de liberación*, en la cual ya no se pueden solicitar más cerrojos y se deben liberar todos los que se hayan conseguido hasta ese momento. Una variante de este protocolo, conocida como *protocolo estricto de cierre de dos fases* [GR93], se caracteriza porque la liberación de los cerrojos en la segunda fase se realiza de una sola vez cuando ha concluido la transacción (bien al confirmar o al abortar); es decir, en este caso no se pueden liberar los cerrojos paulatinamente.

El protocolo 2PL presenta como principales problemas la aparación de *interbloqueos* y de *abortos en cascada*. Un interbloqueo se produce cuando un conjunto de transacciones mantiene cerrados un conjunto de recursos y todas ellas requieren nuevos recursos que han sido adquiridos por otras transacciones de ese mismo conjunto y que, por tanto, ya están bloqueadas.

Para poder resolver un interbloqueo normalmente se requiere elegir una de las transacciones interbloqueadas, abortándola. Al abortar una transacción se necesita devolver todos los recursos que ésta ha modificado a su estado previo. Es posible que otras transacciones hayan efectuado cierres en modo lectura tras la modificación que ahora se está abortando, con lo que dichas transacciones también deberán abortar. Con ello se origina el problema de los abortos en cascada que hemos citado anteriormente.

El protocolo 2PL estricto reduce considerablemente la probabilidad de que se produzcan abortos en cascada [SS94], pues elimina el tiempo que transcurre entre la liberación de los cerrojos y la finalización de la transacción que los mantenía, con lo que resultará imposible que otras transacciones puedan consultar los resultados intermedios de una transacción que sea abortada.

El control de concurrencia mediante cerrojos presenta la característica de que la probabilidad de que una transacción quede suspendida depende del número de cerrojos que necesite adquirir (esto es, del número de recursos a los que deba acceder para completar su trabajo) y de la probabilidad de que otras transacciones en curso hayan adquirido previamente dichos cerrojos en un

modo no compatible con el que necesitaba la transacción analizada. En [Tho93] se da un estudio detallado del rendimiento que es posible obtener utilizando un protocolo de cierre de dos fases.

A las transacciones que siguen el protocolo 2PL se las conoce también con el nombre de *transacciones dinámicas* [Tho93]. Otro protocolo de cierre bastante más restrictivo es el *protocolo de cierre estático* [SS94] en el que se requiere que todas las transacciones efectúen el cierre de todos los recursos que necesiten antes de acceder a ninguno de ellos. Las transacciones que utilizan este protocolo reciben el nombre de *transacciones estáticas*. El uso de este protocolo reduce el grado de concurrencia que será posible obtener en el acceso a los recursos. Aparte, este protocolo es difícilmente aplicable en aquellos programas en los que el conjunto de recursos a acceder dependa de los resultados obtenidos en los accesos previos o en los iniciales dentro de una misma transacción.

5.3.2 Marcas temporales

Una *marca temporal* o *timestamp* es un valor numérico único que se asigna a una transacción o un recurso y que es elegido de entre una secuencia monotónicamente creciente. Una forma de generar estas marcas es utilizar los *relojes lógicos* de Lamport [Lam78], aunque puede utilizarse cualquier otra.

Si se sigue el esquema de Lamport, cada nodo N_i tendrá un reloj lógico R_i, que irá tomando valores crecientes. Cuando una transacción T llega a un nodo N_i, este nodo incrementa su reloj local R_i en una unidad y asigna la tupla (R_i, i) a la transacción T. El par (R_i, i) es la marca temporal de T. Cada mensaje relacionado con las transacciones recibirá la marca temporal asociada a la transacción que ha generado el mensaje. Cuando un nodo N_j recibe un mensaje marcado con el valor (t, i), este nodo fija su reloj al máximo de entre t y R_j. Utilizando estas marcas temporales se puede establecer un orden total de la siguiente manera: si tenemos dos marcas temporales $mt_1 = (t_1, i_1)$ y $mt_2 = (t_2, i_2)$, entonces $mt_1 < mt_2$ si $(t_1 < t_2)$ o bien si $(t_1 = t_2)$ e $(i_1 < i_2)$.

Siguiendo la clasificación dada por [SS94], utilizando marcas temporales se pueden dar dos tipos de control de concurrencia: algoritmos de cierre y algoritmos basados en marcas temporales. En el primer caso se trata de imitar el uso de cerrojos visto en la sección anterior, mientras que en el segundo no es necesario simular un cierre de los recursos. Veamos seguidamente cada alternativa.

Algoritmos de cierre

En estos algoritmos cuando una transacción se inicia recibe una marca temporal generada por el nodo donde ha empezado. Estas marcas temporales servirán para dar cierto tratamiento en caso de que dos transacciones presenten conflictos. Estos conflictos se dan cuando una transacción (a la que vamos a llamar *peticionaria*) intenta leer un recurso para el que actualmente otra transacción está realizando una escritura o bien cuando una transacción intenta escribir sobre un recurso en el que otra transacción está realizando una escritura o una lectura.

Para resolver los conflictos entre dos transacciones se podrán emplear las siguientes acciones:

- *Espera*. La transacción peticionaria debe esperar hasta que la transacción con la que está en conflicto termine o aborte.
- *Reinicio*. Bien la transacción peticionaria o bien la que está en conflicto con ella es abortada e iniciada de nuevo. Esta estrategia de reinicio se puede realizar de dos maneras distintas:

- *Muerte*. La transacción peticionaria es abortada e iniciada de nuevo.
- *Rebobinado*. La transacción en conflicto con la peticionaria es marcada con la etiqueta de "rebobinada" y un *mensaje de REBOBINADO* es enviado a todos los nodos que mantengan recursos utilizados por tal transacción. Si ese mensaje es recibido antes de que la transacción haya terminado en ese nodo entonces la transacción es abortada. En caso contrario, el mensaje no es tenido en cuenta. Si fuera abortada, la transacción será iniciada posteriormente. La transacción peticionaria conseguirá el recurso una vez la transacción en conflicto haya terminado o haya sido abortada.

Los dos algoritmos de cierre basados en marcas temporales son los siguientes:

1. *Algoritmo espera-muerte*. Es un *algoritmo no expulsivo* puesto que la transacción peticionaria nunca podrá forzar a la transacción con la que esté en conflicto a abortar.

   El algoritmo funciona de la siguiente manera. Si una transacción peticionaria T_1 está en conflicto con una transacción T_2 y T_1 es más antigua (es decir, tiene una marca temporal menor), entonces T_1 espera, sino T_1 muere.

2. *Algoritmo rebobinado-espera*. Este es un *algoritmo expulsivo* en el que la transacción peticionaria sí podrá abortar a la que esté en conflicto con ella.

   En este caso, si la transacción peticionaria T_1 y que está en conflicto con una transacción T_2 es más antigua que la T_2, entonces T_2 es rebobinada, pero en caso contrario T_1 deberá esperar.

Algoritmos basados en marcas temporales

Aunque existen múltiples algoritmos basados en marcas temporales para realizar la serialización de las transacciones [SS94], en esta sección únicamente se describirá el *algoritmo básico de ordenación con marcas temporales*. En este algoritmo, el *gestor de datos* de cada nodo mantiene para cada uno de sus objetos la marca temporal más reciente con la que se han realizado por una parte las operaciones de escritura y por otra las de lectura. Para un determinado objeto x, denotaremos mediante $mtL(x)$ su marca temporal más reciente de todas aquellas transacciones que han realizado operaciones de lectura sobre dicho objeto y mediante $mtE(x)$ su marca temporal más reciente de entre todas aquellas transacciones que han realizado operaciones de escritura sobre el objeto x.

El algoritmo se comporta de la siguiente manera. Si una transacción T_1 con marca temporal MT_1 intenta realizar una operación de lectura sobre un objeto x, entonces si $MT_1 < mtE(x)$ la transacción es abortada porque dicho objeto ha sido modificado por una transacción cuya marca temporal indica que ha sido generada posteriormente. Por el contrario, si la marca de las transacción peticionaria es posterior o igual a la marca de escritura del objeto, entonces la petición se sirve y $mtL(x)$ toma como valor el máximo entre su valor actual y el valor de MT_1.

En este algoritmo las marcas temporales asociadas a las transacciones se toman como una guía para realizar su serialización. Aquellas transacciones con marcas más antiguas deberán terminar antes que las que lleven marcas temporales más grandes. Además, se exige que una transacción no pueda consultar ni modificar las escrituras realizadas por una transacción con una marca temporal posterior.

En caso de que T_1 intente realizar una operación de escritura sobre el objeto x, se comprueba en primer lugar si $MT_1 < mtL(x)$ o si $MT_1 < mtE(x)$ y entonces la petición es rechazada y la transacción abortada. En caso contrario, la petición se sirve y $mtE(x)$ pasa a tomar el valor MT_1.

Cada vez que una transacción es abortada y reiniciada, cambiará su marca temporal. Con ello se mejora la probabilidad de que la transacción pueda terminar ahora con éxito pues al ser más nueva difícilmente encontrará conflictos con las transacciones que han ocasionado su aborto.

Como puede observarse, tanto esta solución como los dos algoritmos descritos en la sección anterior sobre algoritmos de cierre mediante marcas temporales resuelven los conflictos entre transacciones abortando en algunos casos a una de ellas. Esto tiene la ventaja de que se evitará la aparición de interbloqueos, pero presenta el inconveniente de que se debe dar soporte para recuperación del estado previo, cosa que en HIDRA no debe darse gracias a la propiedad de progreso garantizada por el protocolo de invocaciones fiables.

5.3.3 Votaciones y quórum

En las técnicas de control de concurrencia basadas en votaciones se asume que los objetos a proteger están replicados. Con ello, la situación descrita en este caso ya resulta más semejante a la que va a plantearse en HIDRA. Dentro de las técnicas de votación podemos distinguir las siguientes alternativas: votación estática, votación dinámica por mayoría y votación dinámica con reasignación de votos. Analizaremos cada una de estas técnicas a continuación, presentando las características básicas en el apartado correspondiente a las votaciones estáticas y tratando posteriormente las variaciones que aportan las otras dos técnicas.

Además de la descripción de estas técnicas, debe tenerse en cuenta que existen otros detalles relacionados con ellas que aunque a primera vista resultan sencillos pueden tener mucha importancia en el rendimiento y tolerancia a fallos de la solución final. Por ejemplo, decidir de qué manera se asignan los votos a cada réplica es uno de estos detalles. En [KS93] se describen varios algoritmos de asignación de votos cuando se utiliza votación estática y cómo afectan éstos al rendimiento del sistema y su tolerancia a fallos.

Votación estática

La técnica de *votación estática* [Gif79] asume que el objeto a proteger ha sido replicado y que para poder acceder a él debe obtenerse un cerrojo del modo apropiado (bien lectura o bien escritura). El *gestor de cerrojos* que se emplee forzará que las operaciones de escritura se realicen de forma exclusiva, es decir, sin ninguna otra operación en curso en ese momento, mientras que las operaciones de lectura podrán realizarse concurrentemente.

Para que una transacción pueda realizar una operación de lectura (o de escritura) deberá obtener previamente un *quórum de lectura* (respectivamente, *quórum de escritura*). Es decir, se debe obtener un cierto número de votos generado por las réplicas del objeto que permitan que la operación se lleve a cabo.

Para asignar valores a los quórums se utilizan los siguientes criterios. Asumiremos que el número total de votos que pueden generar todas las réplicas es n. El quórum de lectura, l, y el quórum de escritura, e, deben cumplir que:

$$l + e > n \qquad ; \qquad e > \frac{n}{2}$$

Es decir, la suma de ambos quórums debe superar el número total de votos y el quórum de escritura debe ser mayor que la mitad del número total de votos.

Con ello, cuando una transacción intenta realizar una lectura o escritura realiza una serie de pasos que pueden resumirse como sigue. En primer lugar solicita al gestor de cerrojos que cierre el objeto en el modo adecuado. Cuando se ha obtenido el cerrojo se pasa a solicitar los votos de las réplicas del objeto. Transcurrido cierto tiempo límite se comprueba si el número de votos recibidos es suficiente como para llegar al quórum de la operación solicitada. Para las operaciones de escritura debe tenerse en cuenta además que pueden existir múltiples versiones del estado del objeto y que deben descartarse todos aquellos votos generados por réplicas cuyo estado no coincida con el de la última versión existente. Si no ha podido obtenerse el quórum solicitado, se libera el cerrojo y se descarta la operación. Cuando el quórum ha podido obtenerse, se pasa a realizar la operación solicitada. En caso de ser una lectura, se elige a una de las réplicas que presenten la versión más nueva del estado del objeto para realizar sobre ella la lectura. Si la operación era una escritura se procede a escribir sobre todas aquellas réplicas que hayan otorgado su voto y presenten la versión más nueva del objeto. Posteriormente se actualiza el número de versión de estado de estas réplicas. Cuando la operación termina, se libera el cerrojo, con lo que otras operaciones en conflicto con la actual podrán proceder.

Como puede comprobarse, la técnica de votaciones se apoya en el uso de cerrojos para realizar el control de concurrencia, pero añade la gestión de múltiples réplicas con diferentes versiones de su estado y la posibilidad de llevar a cabo operaciones incluso cuando han fallado algunas de estas réplicas.

En el caso de la votación estática, la asignación de votos y el número de votos necesarios para completar un quórum no cambian aunque algunas de las réplicas del objeto hayan fallado.

Votación dinámica por mayoría

Los mecanismos de *votación dinámica* reajustan los quórums de lectura y escritura o bien el número de votos asignados a cada réplica en caso de fallo de algún nodo o de algún enlace de comunicación. De esta manera se incrementa la tolerancia a fallos de estas técnicas de votación.

Las *votaciones dinámicas por mayoría* fueron propuestas en [JM90] para permitir que una base de datos pueda funcionar en un sistema distribuido donde se puedan dar particiones. Así, cuando se produce una *partición* y el sistema se divide en dos o más subgrupos y en cada subgrupo existe alguna réplica de los objetos que debían acceder las transacciones, esta técnica permite elegir una *partición distinguida*. Esta partición corresponderá al subgrupo del sistema donde exista una mayoría de las réplicas del objeto o, en caso de empate entre varios subgrupos, allí donde esté ubicada la réplica con un identificador más bajo.

En la práctica, el algoritmo empleado en [JM90] puede verse como una variante de votación dinámica donde pueden reajustarse los quórums de lectura y escritura en caso de que se den particiones en el sistema o fusiones de subgrupos tras la reparación de una partición, de manera que en la partición distinguida dichos quórums se adapten a la nueva situación de dicho subgrupo (rebajándolos en caso de fallo y aumentándolos en caso de reparación) y en las demás se fijan a unos valores que resulta imposible alcanzar.

Votación dinámica con reasignación

En la *votación dinámica con reasignación* [BGMS86, Her87b] cuando se detecta un fallo en alguna de las réplicas del objeto o una partición de la red o una recuperación de cualquiera de estas dos situaciones, la distribución de votos efectuada entre las réplicas se modifica. El principal objetivo es conseguir que se siga llegando a los quórums de lectura y escritura en caso de fallo.

Para efectuar la reasignación de votos se puede emplear alguna de estas dos técnicas:

- *Cooperativa*. En este caso se puede utilizar un algoritmo distribuido para efectuar la reasignación o un algoritmo de elección de líder, quien impondrá posteriormente la asignación elegida. Como en la decisión sobre la nueva asignación van a participar todas las réplicas activas y se va a disponer de la información proporcionada por todas ellas, en estos algoritmos el reparto de votos efectuado podrá soportar mejor la ocurrencia de nuevos fallos.

- *Autónoma*. Cada réplica realiza una reasignación local de sus votos, que deberá ser aprobada posteriormente por el resto de réplicas, cada vez que detecta un cambio en la configuración del conjunto de réplicas que componen el objeto. Esta solución tiene la ventaja de lograr la reasignación más rápidamente que las soluciones cooperativas, pero presenta el inconveniente de que el reparto efectuado pueda resultar inadecuado para soportar futuros fallos.

5.3.4 Técnicas optimistas

En las técnicas de control de concurrencia optimista [Her90], las transacciones se ejecutan sin necesidad de utilizar ningún mecanismo de sincronización, pero antes de terminar deben pasar por una fase de *validación*, que en caso de ser superada permitirá que la transacción confirme sus cambios, pero que en caso de fallar forzará su aborto.

Durante esta validación se comprueba si la transacción que se está examinando ha tenido conflictos con otras transacciones que se han ejecutado concurrentemente, de manera que estos conflictos han ocasionado que el orden de ejecución de tales transacciones no pueda ser serializado. Como en cualquier situación de conflicto al menos una de las operaciones conflictivas ha de ser una escritura, las técnicas optimistas serán especialmente apropiadas para aquellos sistemas transaccionales donde se realicen principalmente lecturas.

Según con qué transacciones se compare la transacción que se está validando, existen dos tipos de validación. En la *validación hacia atrás*, se comprueba que los resultados de la transacción que se valida no hayan entrado en conflicto con los de otras transacciones ya validadas y confirmadas. En la *validación hacia delante* se examina si los resultados de la transacción comprobada invalidarán o no los de otras transacciones ya iniciadas.

En [Her90] se extiende el modelo tradicional de validación, basado en los conflictos que puedan darse entre las operaciones de escritura y lectura realizadas sobre los objetos de la base de datos para proponer otros modelos que permitan una mayor concurrencia. Así se presentan en primer lugar técnicas de *validación basada en conflictos* donde tales conflictos no se restringen únicamente a los tradicionales entre escrituras y entre escrituras y lecturas, sino a los que puedan especificarse entre cualquier par de operaciones definidas en la interfaz del objeto. De esta manera, algunas operaciones que impliquen la realización de una escritura podrán ejecutarse concurrentemente si no modifican la misma parte del estado del objeto. Para mejorar la concurrencia

se introducen posteriormente técnicas de *validación basada en estado* donde no sólo se utilizan los posibles conflictos entre operaciones sino que algunos de estos conflictos se eliminan según el estado que presente el objeto. Es decir, aparte de comprobar qué operaciones pueden estar en conflicto, se verifica si el estado del objeto durante la ejecución de ambas operaciones permite su ejecución dentro de transacciones concurrentes. Esto es debido a que en algunos casos una operación no modificará la misma parte del estado del objeto según el valor previo de dicho estado.

Otras técnicas de control de concurrencia optimistas se describen en [BK91, KB94, Rah93].

5.3.5 Objetos protegidos

En algunos lenguajes de programación se han utilizado ciertas construcciones que permiten controlar qué operaciones se pueden ejecutar concurrentemente y qué otras no. Una de las primeras construcciones de este tipo fueron los *monitores* [Hoa74] implantados en extensiones de los lenguajes *SIMULA67* y *Pascal concurrente*. En estos lenguajes un monitor encapsulaba una serie de variables (o atributos) y ofrecía una serie de operaciones, que eran el único medio de acceder y modificar tales variables. El monitor garantizaba que todas las operaciones que ofrecía en su interfaz se ejecutaran en *exclusión mutua*. Aparte, ofrecía el tipo *condición* mediante el cual se podía suspender un hilo de ejecución que ejecutase código interno del monitor, con lo que se podían construir mecanismos de sincronización más complejos que la simple exclusión mutua.

Aunque todavía no se había extendido la programación orientada a objetos en las fechas en que fueron propuestos los monitores, su estructura y propósito sí recuerdan a las construcciones que más tarde se han utilizado en este tipo de lenguajes. Además, estas construcciones cuando incluyen mecanismos de control de concurrencia son bastante más sencillas de utilizar por el programador que otros mecanismos, pues no requieren ningún esfuerzo adicional.

Otro ejemplo lo constituye el lenguaje *Ada* [US 83] donde se soportan *tipos protegidos* con los que es fácil resolver cualquier problema de sincronización que requiera exclusión mutua o una solución del mismo estilo que el problema de los lectores y escritores. Para acceder a un tipo protegido, Ada proporciona tres tipos de operaciones que ofrecen garantías diferentes. En primer lugar tenemos las *funciones*, mediante las cuales únicamente se podrá consultar el estado del objeto y que podrán ejecutarse concurrentemente con otras funciones. En la práctica siguen las mismas reglas que las operaciones de lectura.

También existen *procedimientos* y *entradas* mediante los que se podrá modificar el estado del objeto y para los que se garantizará exclusión mutua en su ejecución. Por tanto, siguen las mismas reglas de sincronización que las operaciones de escritura. La diferencia entre los procedimientos y entradas reside en que en el caso de estas últimas se debe superar una *barrera* antes de empezar la ejecución de su código. Dicha barrera es una condición que debe hacerse cierta para poderla superar y donde podrán consultarse las variables internas que formen parte del estado del objeto.

Un último ejemplo de lenguaje con cierto soporte para sincronización, aunque bastante menos potente que el de Ada es el proporcionado por *Java* [Sun99]. En este caso se ofrecen operaciones *synchronized*, con las mismas garantías que anteriormente hemos comentado para los procedimientos de un monitor.

5.4 Potencia expresiva

En [McH94], se define *potencia expresiva* como la habilidad por parte de un mecanismo de sincronización de implementar un conjunto de políticas de sincronización diferentes. El mecanismo de sincronización que se utilizará en HIDRA deberá dar soporte a diferentes políticas de sincronización, que podrán ser implantadas gracias a la flexibilidad que ofrezca este mecanismo. En concreto, resultará sencillo implantar políticas de exclusión mutua y de escritor-lectores.

Bloom [Blo79] da algunos criterios para identificar la potencia expresiva de los mecanismos de sincronización. En concreto, cita seis tipos de información necesarios para lograr una amplia potencia expresiva. Estos tipos de informaciones son: el nombre de la operación invocada, el orden de llegada relativo de las invocaciones, los argumentos de las invocaciones, el estado de sincronización del recurso, el estado local del recurso e información histórica sobre las invocaciones ya terminadas.

El nombre de la operación invocada o algún tipo de identificador asociado a ella resulta necesario para poder realizar el control de concurrencia con granularidad de operación, sirviendo como base tanto para los mecanismos de sincronización basados en conflictos como para los basados en estado como para los basados en el orden de llegada. Este orden de llegada también resulta necesario en aquellos mecanismos no basados exclusivamente en él para que puedan deshacer situaciones de empate según dicho orden.

Los argumentos de las invocaciones y el estado local del recurso servirán para construir mecanismos de sincronización basados en el estado del objeto. Como ya se ha comentado anteriormente, estos mecanismos permiten una mayor concurrencia que los basados exclusivamente en conflictos pero logran esto a costa de tener que manejar mucha más información para tomar las decisiones de sincronización, con lo que se compromete la eficiencia del mecanismo.

El estado de sincronización y la información histórica sobre las invocaciones finalizadas sirven como base para poder adoptar mejor las decisiones futuras del mecanismo de sincronización.

Nuestro mecanismo de sincronización puede manejar cuatro de estos seis tipos de información. Nuestro objeto serializador recibe el nombre de la operación invocada (en nuestro caso identificada por un número), el orden relativo de llegada lo mantiene el serializador en las listas de invocaciones activas, bloqueadas y las listas de operaciones predecesoras asociadas a las invocaciones bloqueadas. El estado de sincronización de las invocaciones se mantiene diferenciando invocaciones bloqueadas y activas en las listas del serializador y por último, también podría almacenar un histórico de invocaciones terminadas sobre cada objeto replicado, si fuera necesario para implantar alguna política de sincronización determinada, aunque actualmente esto no se hace.

5.5 HCC: Control de concurrencia en HIDRA

El mecanismo de control de concurrencia que se va a utilizar en HIDRA, al que vamos a llamar *HCC* [MGB98a, MGB98d], deberá integrarse fácilmente en el soporte para invocaciones fiables descrito en el capítulo anterior, ya que va a ser utilizado para controlar qué invocaciones podrán proceder y qué otras deberán suspenderse. De hecho, la información necesaria para mejorar su potencia expresiva será extraída del objeto RoiID que identifica a una invocación en curso y del contexto de la invocación que gestiona nuestro ORB con soporte para objetos replicados.

Como ya se comentó al describir las invocaciones fiables, existe un componente serializador en HIDRA que se encarga de controlar qué invocaciones deben ser suspendidas debido a que presentan conflictos con otras invocaciones ya iniciadas. Por tanto, nuestro control de concurrencia está basado en conflictos entre operaciones, pero no en la información de estado ni en el valor de los argumentos. Para que el componente serializador conozca qué pares de operaciones no deben ejecutarse concurrentemente resulta necesario dar en algún momento una especificación de las incompatibilidades existentes entre las operaciones de una interfaz. Esto se logra mediante una extensión del lenguaje IDL que se describirá seguidamente. Más tarde se describirán los agentes y objetos auxiliares que resultan necesarios para construir el soporte de este mecanismo de control de concurrencia. Una vez conocidos estos componentes básicos se describirá cómo se efectúa la serialización de invocaciones y cómo se han replicado los agentes de serialización para tolerar fallos. Esto constituirá el contenido de esta sección.

5.5.1 Especificación de conflictos

Para lograr el objetivo de "facilidad de uso" de nuestro mecanismo de control de concurrencia, se exige que el diseñador de la aplicación separe las tareas de control de concurrencia de las tareas de programación. Para ello se ha optado por realizar una extensión del lenguaje *IDL* que permita especificar qué operaciones de una interfaz determinada puden ejecutarse concurrentemente y qué otras no.

Una vez se tenga especificada la concurrencia posible dentro de un objeto, las invocaciones serán serializadas de acuerdo con esta especificación. Para cada invocación a un objeto replicado, el mecanismo comprobará qué invocaciones se están ejecutando en ese instante para decidir si la invocación entrante puede proceder concurrentemente con las invocaciones ya activas o si por el contrario debe ser bloqueada hasta que finalicen las que presenten conflictos con ella.

Las extensiones a IDL consisten en cláusulas opcionales que se incluyen en la declaración de las operaciones de una interfaz. Inicialmente se asume que dos operaciones sobre el mismo objeto no pueden ejecutarse concurrentemente. Como resultado de esto, tenemos que cualquier interfaz especificada en IDL sin utilizar las extensiones será interpretada como una clase de objetos cuyas instancias estarán protegidas por operaciones accesibles únicamente en exclusión mutua.

De igual forma, se asume que las invocaciones efectuadas sobre instancias distintas, satisfagan o no la misma interfaz, podrán ejecutarse concurrentemente. Esta decisión está basada en la práctica común del diseño y programación orientada a objetos donde lo habitual es encapsular estado en objetos, de forma que objetos diferentes no compartan estado, pero invocaciones sobre un mismo objeto sí accedan a un mismo estado interno. Siendo cierto esto, para flexibilizar el control de concurrencia y permitir la declaración de operaciones sobre una misma instancia como *compatibles*, es decir, como operaciones que puedan proceder concurrentemente se introducen nuevas cláusulas en la especificación de las operaciones de una interfaz.

La nueva sintaxis para la declaración de operaciones aparece en la figura 5.1 donde se muestra la *cláusula* `concurrent` junto al *calificador* `local` que servirá para indicar que la operación puede ser satisfecha examinando únicamente el estado de la réplica que ha recibido la invocación, sin necesidad de efectuar actualizaciones de estado sobre las réplicas cohortes. Esto se debe a que la operación únicamente consulta el estado del objeto, pero no lo modifica.

Mediante la cláusula `concurrent` se listan todas aquellas operaciones, de esa misma interfaz

```
<op_dcl>    ::=  [ <op_scope> ] [ <op_attribute> ] <op_type_spec> <identifier>
                 <param_dcls> [ <raises_expr> ] [ <context_expr> ]
                 [ <cnc_expr ]

<op_scope>  ::=  "local"

<cnc_expr>  ::=  "concurrent" "(" <scoped_name> { "," <scoped_name> }* ")"
```

Figura 5.1: Sintaxis de la declaración extendida de una operación.

o de cualquiera de las interfaces de las que herede directa o indirectamente ésta, que puedan proceder de manera concurrente con una invocación de la operación que se está declarando. Todo ello asumiendo que ambas invocaciones se han realizado sobre una misma instancia de esa clase, es decir, sobre el mismo objeto, ya que en caso contrario las invocaciones ya serían concurrentes. Para dos operaciones compatibles, sólo es necesario que aparezca esta cláusula en una de ellas, aunque puede especificarse en ambas operaciones si así se desea.

IDL permite herencia de interfaces. Por ello, HCC considera que todas las operaciones de todas las interfaces de un mismo objeto son incompatibles entre sí. Por tanto, si se desea alterar este tratamiento por omisión se debe especificar en la cláusula `concurrent` el nombre de las operaciones incluyendo su ámbito, es decir, el nombre de la interfaz a la que pertenece.

El ejemplo de la figura 5.2, muestra un caso simplificado de utilización de estas cláusulas para especificar la interfaz de un objeto. El objeto es un buffer de elementos con capacidad limitada con las típicas operaciones para añadir un elemento (`InsertItem()`), eliminarlo del buffer (`GetItem()`), escribir todo el contenido del buffer (`PrintBuffer()`), leer una componente determinada del buffer (`ListItem()`) y escribir una parte determinada del buffer (`PrintItems()`).

```
interface BoundedBuffer {
  void InsertItem(in Item TheItem);
  Item GetItem();
  local
  void PrintBuffer();

  local
  Item ListItem(in long Position);

  local
  void PrintItems(in long First,
                  in long Last);
};
```

(a)

```
interface BoundedBuffer {
  void InsertItem(in Item TheItem);
  Item GetItem();
  local
  void PrintBuffer()
       concurrent(BoundedBuffer::PrintBuffer);
  local
  Item ListItem(in long Position)
       concurrent(BoundedBuffer::ListItem,
                  BoundedBuffer::PrintBuffer);
  local
  void PrintItems(in long First, in long Last)
       concurrent(BoundedBuffer::PrintBuffer,
                  BoundedBuffer::PrintItems,
                  BoundedBuffer::ListItem);
};
```

(b)

Figura 5.2: Interfaz de ejemplo con política: (a) Exclusión mutua. (b) Lectores-escritor.

En este ejemplo, las operaciones `PrintBuffer()`, `ListItem()` y `PrintItems()` sólo accederán al estado del objeto para consultarlo por lo que se utiliza el calificador `local` para indicar que no se necesita realizar actualizaciones de estado sobre las réplicas cohortes, mientras

que las operaciones `InsertItem()` y `GetItem()` lo modificarán. Como resultado, tenemos la figura 5.2.b en la que se aprecia cómo se han declarado todas las operaciones de sólo lectura como compatibles con otras invocaciones a sí mismas y, además, compatibles entre ellas.

Nótese que las figuras 5.2.a y 5.2.b ya muestran cómo el mecanismo HCC permite especificar las políticas de sincronización de exclusión mutua (la política por omisión) y la de un escritor con varios lectores.

La interfaz extendida ha de ser procesada por el compilador de interfaces, quien se encargará de generar el código de un objeto que podrá ser utilizado por el serializador para saber si dos operaciones podrán ser ejecutadas concurrentemente o no. El objeto así generado se llama *CCS* y aparece en la figura 5.3. En esta figura se muestra en primer lugar una estructura *`InvoCtxt`* que es la que mantiene el contexto de una invocación y cuya información es facilitada por el soporte de nuestro ORB en el dominio del objeto invocado. Para identificar la operación invocada se facilita el número de operación dentro de la interfaz, un identificador de interfaz y un identificador de objeto. Con esto se tiene suficiente información como para saber si dos invocaciones se han realizado sobre un mismo objeto y a qué interfaz, de entre todas las que implemente el objeto ya que éste puede implantar alguna subclase, pertenece la operación invocada.

```
struct InvoCtxt {
    long          Operation,
    CORBA::TypeId Interface,
    ObjectId      ObjId
};

interface CCS { // pseudo IDL
    boolean CanBeConcurrent(
        in InvoCtxt         FirstInvocation,
        in InvoCtxt         SecondInvocation
    ) raises (UnknownInterface, BadOperationNumber);

    boolean IsLocal( in InvoCtxt TheInvo );
};
```

Figura 5.3: Interfaz del objeto CCS generado por el compilador de IDL extendido.

Por lo que respecta al objeto CCS, éste ofrece una operación mediante la cual se podrá saber si dos invocaciones pueden ser concurrentes o no. Para especificar cada invocación se utilizan las estructuras `InvoCtxt` que hemos descrito anteriormente.

Además de ésta, se ofrece también la operación `IsLocal()` que devuelve el valor cierto si la invocación se ha realizado sobre una operación local.

5.5.2 Agentes

Para llevar a cabo las labores de serialización, es decir, para determinar el orden en el que las invocaciones deberán ejecutarse, existe un objeto *serializador* que almacenará una lista con todas las invocaciones actualmente en curso y otra con aquellas invocaciones que de momento se mantengan suspendidas. Este objeto determina, según dictamine el CCS generado por el compilador

de interfaces, si una invocación que solicita su inicio podrá continuar o si está en conflicto con alguna invocación en curso o con alguna de las ya suspendidas.

En la figura 5.4 se muestra la interfaz del objeto serializador. En esta interfaz únicamente aparece una operación `Serialize()` que será utilizada por el ORB del dominio coordinador y en la que se pasan como argumentos de entrada el objeto RoiID que identifica a la invocación fiable, el contexto de la invocación utilizado para identificar la operación y el objeto que han sido invocados y el TObj que servirá para que el serializador advierta cuándo la invocación fiable ha terminado. Como resultado, la operación devuelve un argumento de salida booleano donde se indica si la operación invocada ha sido calificada como "local" (es decir, de sólo lectura).

```
interface ServiceSerializer { // pseudo IDL
  void Serialize( in RoiID        InvoID,
                  in InvoCtxt     Invocation,
                  in TObj         TerminationObject,
                  out boolean     IsLocal );
};
```

Figura 5.4: Interfaz del objeto serializador.

Uno de los objetivos citados al presentar este mecanismo de control de concurrencia era la tolerancia a fallos. En la descripción del mecanismo IFO realizada durante el capítulo 4 únicamente aparecía un objeto serializador. En la práctica basta con que haya un solo serializador en todo el cluster para que se pueda realizar el control de concurrencia, pero de esa manera no se podrían tolerar los fallos en este componente del sistema. Por ello, el serializador está replicado en todos los nodos del sistema y sus réplicas se conocen como *agentes del serializador*. El modelo de replicación utilizado es ciertamente especial, pues no se corresponde con ninguno de los presentados previamente. Aquí, cada agente mantiene los contextos de invocación y los objetos auxiliares asociados con aquellas invocaciones que hayan tenido su réplica coordinadora en dicho nodo. De esta forma, si cae el nodo donde está ubicado el serializador se perderían únicamente los contextos de las invocaciones que tienen su réplica coordinadora allí, pero esto no es grave ya que se necesitará regenerar dichas invocaciones con lo que deberán elegirse nuevas réplicas coordinadoras para los reintentos y entonces se recuperarán los contextos de tales invocaciones fiables. Esto se analizará con mayor detalle en la sección 5.5.7.

Ya que cada nodo tendrá su propio agente del serializador, el ORB de cada nodo no invocará directamente al serializador sino que lo hará a través de su agente local. Estos agentes presentan la interfaz que aparece en la figura 5.5. Como puede verse, el *método* `Serialize()` de estos agentes coincide con el presentado previamente para el serializador. Además, se ofrecen otras operaciones que serán descritas cuando se explique cómo se serializa una invocación.

Los agentes del serializador serán los encargados de liberar las invocaciones suspendidas debido a conflictos con otras invocaciones en curso. Por ello, el serializador presenta una interfaz para sus agentes que cambia un poco respecto a la presentada en la figura 5.4. En la práctica, su interfaz real es la que se muestra en la figura 5.6 y donde se devuelve como argumento de salida de la operación `Serialize()` una lista con los contextos de invocación y RoiIDs de todas las invocaciones fiables que deben concluir antes de que la invocación a serializar pueda empezar. Si esta lista está vacía, la invocación podrá proseguir de inmediato. En caso contrario deberá esperar

```
interface ServiceSerializerAgent {
    void Serialize( in RoiID            InvoID,
                    in InvoCtxt         theContext,
                    in TObj             TerminationObject,
                    in boolean          IsLocal );

    void Initiated( in RoiID            InvoID,
                    in TObj             TerminationObject );

    void LocallyCompleted(
                    in RoiID            InvoID );
};
```

Figura 5.5: Interfaz de los agentes del serializador.

a que todas las invocaciones fiables especificadas en esa lista hayan terminado. Para ello, el agente del serializador no responderá a la llamada de su método `Serialize()` hasta que todas esas invocaciones hayan recibido la notificación de no referencia en sus TObj asociados.

```
interface ServiceSerializer {
    struct InvoCtxt2 {
        RoiID           Identifier,
        CORBA::TypeId   Interface,
        ObjectId        ObjId,
        long            Operation
    };

    void Serialize( in RoiID            InvoID,
                    in InvoCtxt         Invocation,
                    in TObj             TerminationObject,
                    in long             NodeId,
                    out sequence<InvoCtxt2> PrecedentSet,
                    out boolean         IsLocal );
};
```

Figura 5.6: Interfaz del serializador con soporte para agentes.

Además, necesita un argumento adicional de entrada donde se especifica en qué nodo está ubicado el coordinador de esa invocación y que será necesario para tratar las invocaciones de sólo lectura, tal y como veremos en la sección 5.5.5.

5.5.3 Objetos auxiliares

Para poder realizar las secuenciación de invocaciones fiables, el serializador o sus agentes utilizan los siguientes objetos auxiliares:

CCS: Es el objeto generado por el compilador de interfaces IDL y que facilita una operación mediante la que se puede averiguar si dos invocaciones pueden proceder concurrentemente

o no.

Como ya se comentó en su momento, a la hora de realizar la especificación de compatibilidad no es obligatorio especificar en las dos operaciones que ambas se podrán ejecutar concurrentemente. Esto se dejó así para permitir que no fuera necesario revisar las interfaces de las clases básicas cuando al declarar una clase derivada se especificase que alguno de sus métodos podría ejecutarse concurrentemente con alguno de los de la clase básica. Sin embargo, esto fuerza a que el uso de los objetos CCS resulte algo más complicado dentro del serializador. Ahora, si las invocaciones a comparar se han realizado sobre el mismo objeto (en otro caso ya se sabe que serán concurrentes) y corresponden a interfaces diferentes, el serializador debe llamar el *método* `CanBeConcurrent()` de los objetos CCS ligados a ambas interfaces y admitir la ejecución concurrente si alguna de las dos llamadas ha devuelto un valor verdadero.

RoiID: Es el objeto que identifica a una invocación fiable, cuya utilización ya fue tratada en la descripción del protocolo IFO. Al igual que en aquel caso, este objeto se utilizará aquí para identificar a las invocaciones fiables y de esta manera detectar reintentos de invocación. También es utilizado por los agentes del serializador para recibir la notificación de la terminación de la invocación que representa ese RoiID en el nodo local.

TObj: Es el objeto utilizado en el protocolo IFO para detectar la finalización de una invocación fiable en los dominios servidores. En este protocolo de control de concurrencia es utilizado por el serializador para detectar la terminación de una invocación fiable y de esta manera dar paso a aquellas invocaciones que la deban seguir y que habían sido suspendidas por estar en conflicto con ella.

5.5.4 Serialización de peticiones

Para poder estudiar cómo se realiza la serialización de las invocaciones es necesario examinar qué tareas realiza tanto el serializador central como cada uno de sus agentes para tratar una determinada invocación. Para complementar esta descripción también se añadirá posteriormente un análisis de los pasos seguidos dentro del protocolo IFO para gestionar una invocación. Cada uno de estos puntos se tratará en los próximos apartados.

Pasos seguidos por el serializador

La misión del serializador principal consiste en recibir el contexto de la invocación a serializar y devolver a su agente de serialización ubicado en el dominio coordinador para dicha invocación fiable (que es quien ha solicitado sus servicios) la lista de *invocaciones precedentes* para dicha IFO, junto a un argumento booleano que indicará si la operación podrá ser exclusivamente local. Las operaciones locales son aquéllas que no requieren actualizaciones de estado sobre las réplicas cohortes, normalmente por no haber modificado el estado del objeto.

Para ello, este serializador principal mantiene dos listas de invocaciones. La primera es la lista de *invocaciones* actualmente *activas*. Es decir, aquellas invocaciones para las que se ha devuelto respuesta a su petición de serialización y para las que todavía no se ha recibido una notificación de terminación (a través de su TObj asociado, que recibiría una notificación de no referencia en

tal caso). La segunda corresponde a la lista de *invocaciones suspendidas* debido a que presentan conflictos con alguna invocación activa y posiblemente con algunas otras suspendidas.

Una vez recibida una petición de serialización, invocando al método `Serialize()` que aparecía en la figura 5.6, se siguen estos pasos para construir la lista de invocaciones precedentes:

1. Se examinan todas las componentes de las listas de invocaciones activas y de invocaciones suspendidas. Para cada una de las invocaciones de estas listas se utiliza su contexto para encontrar el identificador del objeto invocado y su objeto CCS asociado.

2. En primer lugar se comprueba si el identificador del objeto invocado en dicha componente coincide con el del objeto asociado a la invocación que se pretende serializar. Si es así, se sigue con los próximos pasos, pero de no ser así ya se sabe que estas dos invocaciones pueden ser concurrentes y se pasa a examinar la siguiente componente de las listas.

3. Sobre el CCS encontrado en el paso 1 y sobre el CCS de la interfaz a la que deseaba llamar la invocación que se está serializando se llama a sus métodos `CanBeConcurrent()` utilizando los contextos de ambas invocaciones (la que se está serializando y la que forma parte de la lista) como argumentos en tales llamadas.

4. Si al menos una de las llamadas devuelve un valor falso, la invocación que formaba parte de la lista se añade a la lista de invocaciones precedentes que se está generando.

Una vez hecho esto, se devuelve la lista al agente de serialización que ha efectuado la petición. Si la lista resulta estar vacía, la invocación serializada se añade a la lista de invocaciones activas. Si por el contrario, existía algún elemento en la lista de invocaciones precedentes, la invocación se añade a la lista de invocaciones suspendidas.

Además, el contexto asociado a la invocación que se está serializando también se utiliza para llamar al *método* `IsLocal()` del objeto CCS ligado a la interfaz invocada. El resultado de esta llamada es lo que se va a retornar al agente de serialización para indicarle si la invocación realizada podrá ser local o no.

Cuando alguna de las invocaciones activas recibe la notificación de no referencia en su objeto TObj asociado, el serializador principal realiza las siguientes tareas:

- Elimina a dicha invocación de la lista de invocaciones activas.

- Elimina a dicha invocación de la lista de invocaciones precedentes asociada a cada una de las invocaciones suspendidas. Si alguna de estas invocaciones suspendidas ha pasado a tener una lista vacía de invocaciones precedentes, entonces dicha invocación es pasada a la lista de invocaciones activas, donde esperará su terminación.

  Esto no tendrá ningún efecto adicional sobre la liberación de tal invocación que ha pasado a activarse, pues tal liberación ha de ser realizada por el agente de serialización ubicado en el nodo donde se encuentra la réplica coordinadora del objeto invocado.

Esta notificación de no referencia en el objeto TObj asociado a la invocación activa se puede recibir porque el serializador principal elimina de inmediato la referencia interna cliente que mantienen todos los objetos replicados de nuestro sistema. Esto lo hace tras haber construido

una réplica del TObj a partir de la referencia cliente transmitida en la llamada a `Serialize()`. El serializador principal es el único componente de los protocolos IFO y HCC que realiza esta liberación de referencia al generar su réplica del TObj.

Pasos seguidos por el agente coordinador

Las tareas realizadas por el agente de serialización ubicado en el nodo que mantiene a la réplica coordinadora de una invocación son las siguientes:

1. Recibir una llamada a su *método* `Serialize()` (ver figura 5.5), realizada por el soporte incluido en el ORB de su propio nodo al recibir la invocación desde el dominio cliente y antes de entregarla a la réplica coordinadora que era el destino de tal invocación.

2. Tras recibir esta llamada, se recoge el argumento `TerminationObject` que es una referencia cliente para un TObj que ha generado el ORB del dominio coordinador. Con esta referencia cliente se genera una réplica para este objeto TObj en el agente de serialización.

3. Cuando ya se tiene la réplica del TObj se realiza una invocación al método `Serialize()` del serializador principal. En la respuesta para esta llamada, el agente recibirá la lista de invocaciones precedentes y un argumento booleano que indica si la operación podrá ser local y que el protocolo IFO utiliza para generar su *flag LECTURA*.

4. Si la lista de invocaciones precedentes está vacía, el agente de serialización devuelve de inmediato una respuesta al componente del ORB que realizó la petición de serialización, con lo que la invocación podrá proseguir.

   En este caso, el contexto de la invocación y su RoiID se marcan como activos y se mantienen así hasta que se notifique la terminación de esa invocación.

5. Si la lista de invocaciones precedentes no estaba vacía, el agente de serialización no devuelve respuesta al componente del ORB que realizó la petición de serialización, con lo que esa invocación se mantiene bloqueada.

   El contexto de la invocación, su RoiID y la lista de invocaciones precedentes se guardan en una lista para las invocaciones suspendidas. La invocación se mantendrá en la lista hasta que el agente de serialización haya comprobado que todas las invocaciones existentes en la lista de precedentes hayan terminado. Entonces devolverá la respuesta y la invocación proseguirá.

   Para dar por terminada una invocación existente en la lista de precedentes se debe recibir una notificación de no referencia en su TObj asociado. Todas las invocaciones en curso mantenidas en los agentes de serialización tienen un objeto TObj asociado. Si es un agente ubicado en el mismo nodo que la réplica cohordinadora, el TObj se recibe al realizar la llamada de serialización. Si por el contrario es un agente ubicado en un nodo cohorte, el TObj se recibe al realizar la primera actualización de estado sobre esa réplica cohorte. Esto se describirá con mayor detalle en la próxima sección donde se analiza el comportamiento de un agente de serialización ubicado en el mismo nodo que una réplica cohorte.

6. Cuando la invocación ya ha sido activada y su respuesta enviada al componente que realizó la petición de serialización, la invocación estará marcada como activa y su TObj asociado seguirá manteniendo su referencia cliente interna.

   Para liberar esta referencia cliente interna debe esperarse a que el componente del ORB ubicado en ese dominio coordinador invoque al *método* `LocallyCompleted()` del agente de serialización. Eso indica que la réplica coordinadora ya ha iniciado el envío de la respuesta hacia el cliente y que, por tanto, la invocación ha terminado en ese dominio.

   Una vez se libera la referencia cliente, más pronto o más tarde la invocación será dada por terminada en todos los dominios servidores, con lo que desaparecerán todas las referencias al TObj y las réplicas de éste recibirán la notificación de no referencia.

7. Cuando la notificación de no referencia ha llegado al TObj asociado con esta invocación, su contexto y RoiID son eliminados. Además, se busca cualquier aparición de ese RoiID en las listas de invocaciones precedentes para aquellas invocaciones mantenidas en la lista de suspendidas y dichas componentes son eliminadas de las listas de precedentes.

De estos pasos que se acaban de analizar se desprende que un agente de serialización sólo puede mantener como invocaciones suspendidas a aquéllas que tengan su réplica coordinadora en su mismo nodo. Sin embargo, en las listas de invocaciones precedentes ligadas a las invocaciones suspendidas puede aparecer cualquier invocación, independientemente de en qué nodo tengan ubicada su réplica coordinadora.

Por tanto, al igual que en el caso del serializador principal, en los agentes de serialización se mantendrán tres tipos de listas:

- *Invocaciones activas*: Serán aquéllas para las que se tenga la réplica coordinadora en ese mismo nodo y se tenga una lista de invocaciones precedentes vacía; o bien, aquéllas en las que el agente de serialización esté junto a una réplica cohorte y ésta haya recibido al menos una actualización de estado. Este segundo caso se explicará en la próxima sección.

- *Invocaciones suspendidas*: Aquéllas que tengan su réplica coordinadora en el mismo nodo que el agente de serialización y que tengan una lista de predecesoras facilitada por el serializador principal que todavía no esté vacía. Este tipo de invocaciones no podrán encontrarse en aquellos agentes de serialización ubicados junto a una réplica cohorte.

- *Invocaciones precedentes*: Aquéllas que debido a los conflictos que presentan con la invocación serializada deben terminar antes de que esta última se inicie. A su vez pueden estar activas o suspendidas, por lo que hay que gestionarlas en la categoría que les corresponda.

Pasos seguidos por los agentes cohortes

En un agente de serialización ubicado en el mismo nodo que una réplica cohorte para una determinada invocación también se necesita conocer el estado de tal invocación, pues ésta ha podido aparecer en las listas precedentes de otras invocaciones. Por ello, estos agentes de serialización son los encargados de mantener las réplicas de los objetos TObj que se mencionaban en el protocolo IFO.

Esto se hace de la siguiente manera:

1. Cuando el ORB ubicado en el dominio cohorte recibe la primera (o la única) actualización de estado para esa invocación fiable realiza, antes de entregar la llamada a la réplica cohorte, una invocación al *método* `Initiated()` del agente de serialización, pasándole una referencia al RoiID y la referencia del TObj recibida en el mensaje de actualización. Con la referencia al TObj se crea una réplica para este objeto y se guarda tanto ésta como el RoiID en la lista de invocaciones activas.

2. Cuando el ORB de ese dominio cohorte ha entregado y ha recibido respuesta de la última (o la única) actualización de estado para esa invocación fiable realiza, antes de enviar el mensaje de respuesta hacia la réplica coordinadora (en caso de ser un mensaje de actualización síncrono), una invocación al *método* `LocallyCompleted()` del agente de serialización, pasándole una referencia al RoiID. Con esta referencia al RoiID, el agente de serialización podrá buscar el contexto de esta invocación en la lista de invocaciones activas. Una vez encontrado tal contexto se procede a liberar la referencia cliente interna de la réplica del TObj mantenida allí.

3. Eventualmente, el objeto TObj recibirá la notificación de no referencia. Entonces la invocación es eliminada de la lista de invocaciones activas de este agente de serialización y su RoiID se busca en las listas de precedentes para ser eliminado también en ellas. Si alguna de estas listas llega a vaciarse con esta eliminación, la invocación a la que estaba asociada pasará a marcarse como activa y se devolverá la respuesta que permitirá que prosiga su ejecución.

Relación con el protocolo IFO

Para entender mejor cómo funciona nuestro mecanismo de control de concurrencia se va a describir de nuevo la *variante básica del protocolo IFO*, teniendo en cuenta la existencia del serializador principal y de sus agentes. El tratamiento de las operaciones locales o de sólo lectura se deja para la sección 5.5.5 que empieza en la página 130.

Los pasos de este protocolo son ahora los siguientes:

1. *Creación del RoiID* (Cliente). Una vez el programa cliente ha iniciado una invocación sobre un objeto replicado, nuestro ORB detecta tal invocación en el soporte interno del núcleo del ORB y crea en dicho *dominio cliente* un objeto RoiID. Una vez creado este objeto, genera una referencia cliente para él y la inserta en el mensaje que se ha construido para hacer llegar esta petición al dominio coordinador.

2. *Recepción de la invocación* (Coordinador). El núcleo del ORB ubicado en el dominio coordinador recibe los mensajes con la petición ligada a esta invocación. Una vez recibidos, y antes de hacer llegar la invocación al objeto invocado, el ORB busca en el mensaje la referencia cliente al RoiID.

   Una vez obtenida la referencia cliente, se comprueba si su RoiID aparece en la lista de invocaciones fiables para las que siguen manteniéndose resultados retenidos en esa réplica. Si es así, se cogen los resultados de la lista, se construye con ellos el mensaje de respuesta y se envía de inmediato al cliente, sin necesidad de reprocesar la petición. En este caso, el protocolo continúa a partir del paso 8 que se explica más adelante.

En caso de que el RoiID no pueda encontrarse en la lista de resultados retenidos, el dominio coordinador genera las primeras réplicas de los objetos CObj y TObj y las asocia a la referencia al RoiID que acaba de recibir. Nótese que en ningún caso se ha llegado a entregar la invocación al código de la réplica coordinadora. Todas estas labores son realizadas dentro del núcleo del ORB existente en el dominio coordinador.

3. *Petición de serialización* (Coordinador). El núcleo del ORB en el dominio coordinador realiza una petición sobre el método `Serialize()` de su agente de serialización local. El agente de serialización recibe como argumentos una referencia al RoiID para identificar la invocación fiable y una referencia al TObj que le servirá para construir otra réplica de este último objeto. Con esta información junto al contexto de la operación invocada, el agente invoca al serializador principal quien le devolverá la lista de invocaciones precedentes para la invocación que se está serializando, junto al *flag LECTURA*. Por su parte, el serializador principal ha generado también una réplica del objeto TObj y ha liberado la referencia cliente que dicha réplica mantenía.

   Cuando hayan concluido todas las invocaciones en conflicto, es decir, aquellas que aparecían en la lista de precedentes, el agente de serialización local devuelve el control al ORB, con lo que éste puede proseguir la invocación. En ese momento, el ORB destruye la réplica original del TObj, pues el agente de serialización local ha construido otra que la sustituirá.

4. *Transferencia del CObj al dominio cliente* (Coordinador). Antes de que la invocación llegue al código de la réplica, el núcleo del ORB en el dominio coordinador realiza una invocación al método `Results()` del RoiID para transferirle una referencia al objeto CObj.

   Cuando la invocación sobre el RoiID ha terminado, el ORB del dominio coordinador entrega finalmente la invocación al código de la réplica y empieza la ejecución de la operación invocada.

5. *Se realiza la primera actualización de estado* (Coordinador). En la primera llamada a un método de las réplicas cohortes para realizar una actualización de estado, el soporte incluido en nuestro ORB añade al contenido del mensaje referencias sobre los objetos RoiID, CObj y TObj. Cuando el ORB en los dominios cohortes recibe estos mensajes, extrae esas referencias creando una réplica del objeto CObj y pasando el RoiID y el TObj al agente de serialización local mediante una llamada a su método `Initiated()`. Con ello, el agente de serialización local generará una réplica del TObj y marcará dicha invocación como activa.

6. *Se realiza la última actualización de estado* (Coordinador). En el mensaje que transporte la última actualización de estado iniciada por la réplica coordinadora debe añadirse en un lugar especial una copia de los argumentos de salida y resultados que se devolverán al cliente. Estos argumentos de salida y resultados son copiados por el ORB de los dominios cohortes y retenidos hasta que se detecte que han podido ser obtenidos por el programa cliente. Para ello, nuestro ORB los asocia con el RoiID y el CObj que obtuvo en la primera actualización de estado (de hecho, el RoiID ha tenido que incluirse en todas las invocaciones de actualización de estado para que el ORB pueda saber a qué invocación fiable corresponden).

Una vez las réplicas cohortes han procesado este último mensaje de actualización de estado, nuestro ORB libera las referencias clientes mantenidas en las réplicas del objeto CObj en todos los dominios cohortes. Tras esto, se invoca también el método `LocallyCompleted()` del agente de serialización local, con lo que éste componente procederá a liberar la referencia cliente del objeto TObj.

7. *Termina la invocación en la réplica coordinadora* (Coordinador). Al igual que hicieron los ORBs de los dominios cohortes, cuando termina la invocación en el dominio coordinador hay que liberar las referencias internas de las réplicas del CObj y TObj. Para liberar la referencia interna del TObj se debe invocar, al igual que en los dominios cohortes, el método `LocallyCompleted()` del agente de serialización local.

   Nótese que al liberar la referencia cliente mantenida en esa réplica del TObj ya no queda ninguna referencia sobre este objeto en todo el sistema (las que había en las réplicas ubicadas en los dominios cohortes fueron liberadas en el paso anterior). Con ello, las réplicas del TObj recibirán todas ellas la notificación de no referencia (ver paso 9). Esto es particularmente interesante en los objetos serializadores (tanto en el principal como en sus agentes), que en base a ello sabrán que la operación ha concluido en los dominios servidores y podrán dar paso a otras invocaciones fiables que estuvieran en conflicto con la que acaba de terminar.

   Por otra parte, nuestro ORB en el dominio coordinador emitirá finalmente el mensaje con la respuesta para el dominio cliente.

8. *El cliente recibe la respuesta* (Cliente). El núcleo del ORB que gestiona las invocaciones en el dominio cliente, antes de entregar la respuesta comprueba a qué RoiID está asociada ésta y entonces libera la referencia cliente sobre el objeto CObj que éste mantenía. Nótese que ésta era la última referencia que quedaba en el sistema sobre ese objeto CObj. Con ello, se habilita la entrega de la notificación de no referencia sobre todas las réplicas de ese objeto (ver paso 10).

   El objeto RoiID también descarta su propia referencia cliente en este momento. Una vez hecho esto, la respuesta es entregada al programa cliente.

9. *Las réplicas del TObj reciben la notificación de no referencia* (Serializador principal y agentes de serialización). Con ello los agentes de serialización ya pueden dar paso a aquellas invocaciones fiables que tuvieran como única invocación en su lista de precedentes a la que acaba de terminar. Además, todas las réplicas del TObj son eliminadas y en los serializadores también se descarta la referencia cliente del RoiID.

10. *Las réplicas del CObj reciben la notificación de no referencia* (Servidores). Con ello, en los dominios cohortes se pasa a eliminar los resultados retenidos y en todos los dominios servidores se eliminan las réplicas del CObj y las referencias clientes del RoiID.

    Con ello, más pronto o más tarde llegará una notificación de no referencia también al RoiID del dominio cliente que entonces será eliminado con lo que desaparecerá por completo el contexto que se ha necesitado para esta invocación fiable.

5.5.5 Tratamiento de operaciones de sólo lectura

Las *operaciones de sólo lectura* requieren un tratamiento especial, pues al no necesitar actualizaciones de estado no deben figurar como invocaciones precedentes de aquéllas otras que no tengan su réplica coordinadora en su mismo nodo.

Si diéramos el tratamiento general para estas invocaciones, todas las que las seguirían en aquéllos nodos no consultados por esta invocación (es decir, aquéllos en los que a priori debería haber habido cohortes para la invocación de sólo lectura) se quedarían suspendidas indefinidamente, pues en esos agentes de serialización nunca podrían ser eliminadas estas invocaciones de sólo lectura de las listas de precedentes.

La solución a este problema la encontramos en el argumento `NodeId` presente en el método `Serialize()` del serializador principal, tal como aparece en la figura 5.6 de la página 122. Con este argumento, el agente de serialización le indica al serializador principal desde qué nodo se está solicitando la serialización de la invocación actual. El serializador principal, al consultar el objeto *CCS* pertinente averiguará si la operación invocada está etiquetada como "local". En ese caso, en el contexto de invocación que mantendrá el serializador principal se marcará también la invocación como local y se añadirá al contexto el identificador del nodo donde la invocación está realizando la consulta.

De esta manera, la invocación de sólo lectura será incluida como precedente únicamente en aquellas otras invocaciones que la sigan, que estén en conflicto con ella y que tengan su réplica coordinadora en su mismo nodo. En todos los demás casos, esa invocación de sólo lectura no aparecerá como precedente.

5.5.6 Adición de réplicas

Otro caso particularmente interesante dentro de la gestión de los objetos replicados bajo el modelo coordinador-cohorte tiene relación con la forma de tratar las operaciones de adición de nuevas réplicas a un objeto. Para lograr que la adición de una réplica no plantee problemas deberíamos "congelar" momentáneamente el servicio de otras peticiones sobre el objeto replicado. Esto se debe a que por una parte debe añadirse la nueva réplica a las referencias que utilice el núcleo de nuestro ORB para poder utilizar el objeto resultante. Es decir, los clientes que intenten invocar al objeto ahora tendrán una nueva réplica que podrá adoptar el papel de coordinadora para las peticiones realizadas o que deberá comportarse como cohorte para aquellas invocaciones fiables no dirigidas directamente a ella.

Por otro lado, también hay que conseguir que como resultado de su inserción en el objeto replicado reciba una copia de su estado que coincida con lo que dicho objeto haya podido hacer hasta ese momento. Ni se puede perder nada respecto a las demás réplicas ni puede haber más información en la nueva que en las restantes.

Por tanto, como resultado de todo esto, nuestro soporte a este tipo de replicación debe evitar que una *adición de réplica* pueda ejecutarse concurrentemente con alguna invocación fiable. Si se permitiese la ejecución concurrente habría que decidir a partir de qué invocación se ha integrado la nueva réplica en el conjunto de cohortes y cómo ha de integrarse en las invocaciones ya iniciadas pero todavía no terminadas.

Como resultado de todo esto, habrá que distinguir a una adición de réplica como una operación especial (con un identificador numérico que no sea asignable a las operaciones presentes en

las interfaces IDL) y que será incompatible (es decir, presentará conflictos) con todas las demás operaciones de ese objeto. Con ello, cuando finalmente el serializador dé paso a la operación de adición, tendremos la garantía de que todas las demás operaciones habrán concluido o estarán suspendidas sin haber podido iniciarse en sus réplicas coordinadoras. Así, cuando la adición termine, la nueva réplica podrá funcionar como cohorte para todas aquellas invocaciones que hayan ido quedando suspendidas debido a que la tenían dentro de la lista de invocaciones precedentes.

5.5.7 Comportamiento en caso de fallos

Para asegurar que el mecanismo HCC tolera fallos, consideramos las posibles combinaciones de fallos que pueden producirse, mostrando en cada caso de qué forma se reconstruye el estado del serializador.

Hay dos grupos de fallos posibles. Uno en que sólo caen agentes y otro en el que cae el serializador y además también falla un conjunto de agentes.

Fallo de un agente de serialización

Cuando esto ocurre, todo el nodo donde se encuentra ubicado también falla ya que estos agentes se encuentran en el núcleo del sistema operativo. Por tanto también caerán las réplicas del servicio que pudieran encontrarse en ese nodo. Este tipo de fallos no plantea problemas ya que en cualquier caso el mecanismo IFO reintentará la invocación sobre una réplica diferente para que ésta actúe de coordinadora.

Si la invocación estaba activa, figurará como activa en el serializador, de forma que el reintento de la invocación, será detectado por el serializador principal como tal y retornará inmediatamente, pudiendo así volver a intentar su ejecución. O también puede ocurrir que si se da un reintento de ejecución de tal invocación su intento anterior ya haya sido dado por terminado, debido a que la réplica que cayó fue la coordinadora y todavía no había realizado ninguna actualización de estado. Este segundo caso no plantea ningún problema porque el nuevo reintento será serializado de nuevo. No importa si ahora pasa a suspenderse temporalmente.

Si la invocación estaba bloqueada, al contactar de nuevo con el serializador, éste también tendrá constancia de que la invocación está bloqueada, con lo que retornará la lista de operaciones predecesoras que aún no hayan terminado.

Fallo del serializador principal y de un conjunto de agentes

Como se ha mostrado anteriormente, recuperar el fallo de un agente tan sólo requiere que el serializador principal sea capaz de buscar para cada petición de serialización si la invocación a controlar ya había sido serializada.

En cambio, al fallar el serializador principal, se debe crear uno nuevo y se debe reconstruir el estado que éste mantenía. La creación de un nuevo serializador principal consiste en reconstruir, uniendo la información existente en todos los agentes que hayan quedado, las listas de invocaciones activas, de suspendidas y de precedentes que deba haber en una determinada situación del sistema.

Para reconstruir el estado, vemos que éste se encuentra distribuido entre los agentes que han sobrevivido al fallo. En cada agente encontramos toda la información relativa a las invocaciones

dirigidas a su nodo por estar en él la réplica coordinadora de la invocación.

Además, los agentes mantienen información acerca de las invocaciones para las que su réplica actúa como cohorte. Mantienen una referencia al objeto TObj y el identificador de la invocación. Con esto se puede averiguar que una invocación estaba activa incluso si falla al mismo tiempo el serializador principal y el agente que actuaba como coordinador. Incluso un agente puede saber si la invocación estaba a punto de terminar. Esto lo sabe si su réplica local invocó la operación `LocallyCompleted()`. En este último caso tenemos que la invocación ya había terminado a falta de contestar al cliente que inició la invocación y posiblemente a falta de actualizar alguna réplica.

Para reconstruir el estado del serializador principal a partir del estado de los agentes supervivientes al fallo se ejecutan dos pequeños protocolos que permiten reconstruir cada una de las listas mantenidas por este componente. Así, para rehacer la lista de invocaciones activas, se van a seguir estos pasos durante las fases de reconfiguración del sistema:

1. Se pide a cada agente que devuelva una lista con todos los RoiIDs que tengan asociada una réplica de su TObj (esto ocurre tanto en los agentes coordinadores como en los cohortes para una invocación). Téngase en cuenta que puede haber RoiIDs que todavía no tengan esta réplica del TObj (porque aún no la habían creado) o que ya la hayan destruido (porque habían recibido la notificación de no referencia). En este segundo caso, todos los agentes estarán de acuerdo en que la invocación ha concluido ya que durante los protocolos de reconfiguración se reajustan las cuentas de referencias y se entregan las notificaciones de no referencia en todos los nodos, con lo que no podrá haber desacuerdos sobre este punto. Ese protocolo de reajuste de las cuentas se ejecuta previamente al que ahora estamos describiendo.

   Para cada una de las invocaciones identificadas por esos RoiIDs, el agente devolverá además del propio RoiID, la referencia al TObj y la información de su InvoCtxt (en caso de tenerla). Debe tenerse en cuenta que esa copia del InvoCtxt únicamente existirá en el agente asociado al dominio que actuó como coordinador para esa invocación fiable.

2. Todos los RoiIDs devueltos en el paso anterior se insertan en la lista activa y se regenera también una réplica del objeto TObj a partir de la referencia recibida, siguiendo así el protocolo IFO descrito al final de la sección 5.5.4.

3. Si no se hubiera podido encontrar ninguna copia del InvoCtxt asociado a esta invocación, habrá que solicitar esta información al RoiID. Para ello, éste proporciona el *método* `GetInvoCtxt()`.

Por su parte, la reconstrucción de la lista de invocaciones suspendidas implica seguir estos pasos:

1. Se pide a cada SSA que devuelva la lista con todas sus invocaciones bloqueadas (con sus RoiIDs) y la lista de invocaciones precedentes para cada una de ellas.

2. Se unen las listas devueltas en el paso anterior. Nótese que para una invocación determinada no puede haber más de un agente que haya devuelto su lista de precedentes, pues esto únicamente puede hacerlo aquél agente ubicado en su nodo coordinador.

3. Finalmente, todas los conjuntos (o listas) de invocaciones precedentes se examinan para averiguar si alguno de ellos mantiene alguna invocación que no aparezca ni en la lista de activas, construida previamente, ni en la de suspendidas que se está construyendo actualmente. En caso de ser así, esa invocación debe ser eliminada de esa lista de precedentes porque corresponde a una invocación activa cuyo coordinador ha fallado y que todavía no había efectuado ninguna actualización de estado sobre sus réplicas cohortes. Esta IFO tendrá que ser reiniciada y, aunque va a conservar el mismo RoiID, será serializada de nuevo utilizando un TObj diferente.

Una vez se han ejecutado estos dos protocolos, el nuevo serializador va a mantener un estado dinámico que va a permitir que se sirvan todas las nuevas peticiones que se realicen a medida que vayan llegando nuevas invocaciones fiables.

Un último detalle que no se ha mencionado previamente es la recuperación de los objetos CCS. Para lograr esto, cada agente mantiene una copia de estos objetos, con lo que su adición al estado del nuevo serializador principal no conllevará ningún problema.

5.6 Trabajo relacionado

La estrategia utilizada en el mecanismo HCC para llevar a cabo el control de concurrencia puede considerarse idéntica a las empleadas en otros sistemas basados en *conmutatividad* entre operaciones [Wei88]. En general, estas estrategias pueden mejorar bastante la concurrencia de un objeto replicado si son comparadas con las técnicas tradicionales basadas en cerrojos, al menos si las comparamos en el ámbito de las bases de datos. Esta mejora se logra gracias a que a la hora de especificar la conmutatividad entre pares de operaciones, (o lo que es lo mismo, la ausencia de conflictos entre ellas) se puede tener en cuenta también la semántica de tales operaciones. Si esta declaración de conmutatividad la realiza el programador o el diseñador de la aplicación siempre podrán especificarse mejor los detalles de incompatibilidad que en el caso de catalogar a todas las operaciones cómo simplemente de lectura o de escritura.

Existen otros mecanismos de control de concurrencia que pueden lograr una potencia expresiva mayor, consiguiendo una concurrencia mayor que en el caso de la conmutatividad entre operaciones que acabamos de mencionar. Pero para lograr esa mayor potencia expresiva se necesita gestionar el estado interno del objeto y los argumentos recibidos en cada operación a controlar. Esto lleva asociado un mayor coste computacional, pues hay que examinar una gran cantidad de información por cada una de las invocaciones que vayan a controlarse, con lo que estos mecanismos difícilmente compensan con esa mejora en la concurrencia el alto coste que hay que pagar al tomar cada decisión de suspensión o liberación de invocaciones.

Un ejemplo de lo que acabamos de decir lo encontramos en uno de los pocos trabajos realizados sobre control de concurrencia específicamente diseñados para el modelo de replicación coordinador-cohorte, correspondiente a una de las primeras versiones del sistema *Isis* [BJR84]. En este mecanismo se identificaban ciertos tipos de dependencias entre las operaciones que se debían controlar:

- *Dependencias de datos*. Estas dependencias se darán entre instrucciones de una misma operación cuando una de las instrucciones requiere el resultado de la otra antes de iniciar su ejecución.

Las dependencias de este tipo pueden examinarse si en el sistema existe soporte para múltiples hilos de ejecución en una misma operación y cada hilo puede ejecutar suboperaciones diferentes dentro de una misma invocación. Nótese que esto exigirá llevar un control de los argumentos que vaya a recibir cada una de las operaciones a controlar.

- *Dependencias de precedencia*. Son las que se dan entre operaciones conflictivas. Es decir, entre aquellas operaciones que acceden a una misma parte de un objeto y donde al menos una de las dos operaciones modifica dicho estado.

Como puede verse, este mecanismo de control de concurrencia añade el estudio de las dependencias de datos a lo que hemos descrito que hace nuestro mecanismo HCC.

Sin embargo, para lograr la serialización entre operaciones no se utiliza una técnica similar a la nuestra (generar una estructura de datos que mantenga información sobre la conmutatividad que pueda darse entre pares de operaciones), sino que se optó por construir a priori un grafo de serialización donde apareciese para cada operación a controlar todas las dependencias de cualquier tipo que pudiera haber entre sus suboperaciones. Para ello se necesitó utilizar un lenguaje de programación especial y con su código se podían obtener las dependencias existentes entre todas las suboperaciones. No creemos que ésta sea la mejor solución para lograr un control de concurrencia efectivo, porque aunque se aumente la potencia expresiva del mecanismo resultante, también se limita la libertad del programador al forzarle a utilizar un determinado lenguaje. Aparte, también se rompe con el principio de encapsulación, porque ahora resulta importante el código utilizado para implantar cada operación. Nuestra solución es más flexible en este ámbito.

Una última dificultad presentada por el mecanismo de control de concurrencia que estamos discutiendo residía en la necesidad de utilizar un mecanismo de difusiones atómicas ordenadas para poder implantar la versión distribuida del algoritmo de control de concurrencia utilizado para efectuar la reducción del grafo de serialización. Esto eliminó gran parte de las ventajas que aportaba el diseño original del modelo de replicación coordinador-cohorte y en la práctica condujo a eliminar este mecanismo de control de concurrencia del sistema Isis, sustituyéndolo en un primer intento por el tradicional protocolo de cierre de dos fases [Bir85] y posteriormente a la eliminación del modelo de replicación coordinador-cohorte como modelo básico de Isis, siendo sustituido por el modelo activo [BJ87].

En nuestro caso se ha optado por un mecanismo de control de concurrencia más sencillo que el descrito en [BJR84] y que aunque conceptualmente utiliza un algoritmo centralizado para adoptar sus decisiones de suspensión o liberación de invocaciones, está replicado de tal manera que la caída de uno o más de sus componentes no comprometen su fiabilidad.

Capítulo 6

Conclusiones

Para terminar esta tesis se presentan las principales contribuciones aportadas por ella, así como el trabajo futuro que podrá desarrollarse en su entorno para mejorar el sistema tomado como base.

6.1 Contribuciones

Dentro de las *contribuciones* de este trabajo se puede realizar una clasificación según el ámbito que queramos tratar, pues los mecanismos que hemos citado en esta tesis son independientes y pueden ser comparados con otras soluciones dadas en otros sistemas, en cada una de las áreas en que son aplicables. Así, por un lado tenemos al algoritmo de pertenencia HMM y por otro está el soporte dado al modelo de replicación coordinador-cohorte, donde podremos distinguir además, entre el soporte genérico, el soporte a invocaciones fiables y el soporte a control de concurrencia. Veamos cada uno de estos puntos en los próximos apartados.

6.1.1 Algoritmos de pertenencia

El desarrollo de algoritmos de pertenencia a grupo es un área de investigación que aunque empezó bastante tarde (los primeros trabajos serios son de 1991 [Cri91a]) tuvo rápidamente un fuerte desarrollo y la mayor parte de las contribuciones en esta área se dieron antes de 1996. Mejorar las prestaciones o las características de un algoritmo de pertenencia es ciertamente difícil a estas alturas. Sin embargo, nuestro algoritmo sí presenta algunas diferencias destacables cuando es comparado con los algoritmos más importantes que ya hemos citado en este trabajo.

Una de las primeras diferencias que podemos observar se centra en la *ubicación* de nuestro algoritmo de pertenencia dentro de la arquitectura de nuestro sistema. En nuestro caso, HMM forma parte de los niveles inferiores de HIDRA y se encuentra en el núcleo del sistema operativo que se tome como base (Puede implementarse como un módulo cargable en Linux, por ejemplo). De hecho está en un nivel inferior al *transporte fiable* que debe implantarse en HIDRA. Hay otros sistemas que han adoptado una ubicación similar [KG94], pero la gran mayoría han utilizado los protocolos de pertenencia para dar soporte a mecanismos de *difusión atómica ordenada* que se implantaban fuera del sistema y que en algunos casos, en lugar de monitorizar el estado de las máquinas, monitorizan el estado de algunos procesos. Dejar el monitor de pertenencia dentro del núcleo ofrece la ventaja de poder utilizar sus servicios también dentro de los componentes que formen el propio sistema operativo. En un futuro, si en HIDRA se decide ofrecer la imagen

de sistema único, los componentes del núcleo del sistema operativo podrán utilizar los servicios del monitor de pertenencia como ayuda para lograr esa imagen. Con otro tipo de monitor de pertenencia estas ayudas serían imposibles.

La segunda diferencia importante se da en el propio algoritmo utilizado y tiene como consecuencia ciertas mejoras en su eficiencia. Para entender esta mejora conviene repasar cómo funciona un monitor de pertenencia. En general, los cambios que debe saber gestionar un monitor de pertenencia pueden ser de dos tipos: caídas o incorporaciones de miembros del grupo. Para adoptar una decisión en cada caso se suelen seguir una serie de pasos y para evitar que se intercambien muchos mensajes en cada paso se suele adoptar un algoritmo centralizado en la mayor parte de los monitores. Adoptar un algoritmo centralizado en algunos pasos tiene un riesgo: puede llegar a fallar el elemento que actúa como coordinador de esos pasos del algoritmo. Para ello, la mayor parte de los monitores que adoptan esta alternativa definen un elemento suplente que retoma el papel coordinador en caso de fallo del elegido inicialmente. Aun así, se corre el riesgo de que el suplente también falle. La mayor parte de los algoritmos ya no prestan atención a este segundo fallo y en caso de que se dé llegan a utilizar una tercera alternativa que es normalmente mucho más costosa: reiniciar el algoritmo desde el principio, partiendo de un conjunto de pertenencia vacío.

El protocolo HMM adopta un algoritmo centralizado durante la toma de decisiones que permite gestionar los cambios transfiriendo pocos mensajes. En nuestro caso, aparte de definir el rol de miembro coordinador (llamado *maestro* en nuestro algoritmo) también definimos el rol de miembro *iniciador*. Los miembros iniciadores dirigen el primer paso de reconfiguración en caso de fallo (no son necesarios en caso de que el cambio sea una incorporación) y así se puede afrontar con toda garantía cualquier caso de caída múltiple, incluso la caída del coordinador anterior. Con ello, en nuestro algoritmo se logra gestionar los casos de caídas de coordinadores sin ninguna variación importante. Todo ello se consigue con un número de mensajes intercambiados que, aun sin ser óptimo, tiene un coste muy bajo, comparable al de los mejores casos del resto de algoritmos y mucho mejor que en los peores casos de aquéllos.

6.1.2 Modelos de replicación

En esta tesis se han presentado algunos mecanismos que podrán utilizarse para proporcionar un soporte nativo dentro de un ORB para el modelo de replicación coordinador-cohorte. Para completar este soporte se necesitan algunos mecanismos adicionales, así como el modelo básico de referencias a objeto que deberá utilizar el ORB y que se describirá en detalle en [Gal01]. Hasta ahora no se había dado ningún soporte directo a este modelo de replicación en ningún otro sistema (o en caso de que se haya dado, el autor desconoce su existencia). Las únicas implementaciones de este modelo se han llegado a realizar encima del modelo activo con lo que comparte algunas de las limitaciones de dicho modelo.

Las ventajas que aporta el modelo de replicación coordinador-cohorte sobre los demás modelos pueden resumirse en los siguientes puntos:

- Tiempo de reconfiguración bajo en caso de fallo de réplicas. Similar al del modelo activo, pues al igual que en él no resulta necesario cambiar el rol de las réplicas que hayan sobrevivido al fallo. El tiempo necesario para la reconfiguración en nuestro modelo es el que se precise para rehacer las referencias que utilicen los clientes para acceder al objeto. En algunos casos, esto puede conseguirse cuando se intenten utilizar de nuevo tales referencias.

En el modelo activo, el tiempo de reconfiguración comprende lo necesario para lograr que el protocolo de difusión atómica ordenada utilice la nueva pertenencia del grupo formado por el objeto. Normalmente tampoco es alto.

Por contra en el modelo pasivo debe elegirse una nueva réplica primaria, cambiar su rol (porque hasta ese momento ha debido ser pasiva) y recuperar las invocaciones en tránsito hasta tener un estado idéntico al que tuvo la réplica primaria que falló. Esto será altamente variable y normalmente algo mayor que en cualquiera de los otros dos modelos.

- Carga mínima. Cada petición sólo es tratada directamente por una réplica. Sobre las demás se realizarán actualizaciones de estado cuando resulte necesario. Esta característica es compartida por el modelo pasivo. Por el contrario, el modelo activo requiere procesar todas las peticiones en todas las réplicas. Su carga es bastante mayor.

- Concurrencia elevada. El modelo coordinador-cohorte permite que múltiples peticiones que no estén en conflicto puedan ser atendidas en una o más réplicas al mismo tiempo. Lo mismo sucede en el modelo pasivo, aunque en este segundo caso, todas las peticiones concurrentes deben ser ejecutadas en el mismo nodo. El modelo activo no permite ejecución concurrente de múltiples peticiones. Sólo permite ejecución secuencial, según dicte el mecanismo de entrega de peticiones.

  Si implementásemos el modelo coordinador-cohorte sobre el activo sería difícil obtener un alto grado de concurrencia. En la práctica, el modelo activo utilizado por debajo evita que pueda haber múltiples peticiones servidas a la vez.

- Sin problemas de filtrado de peticiones o respuestas. Al igual que en el modelo pasivo, sólo una réplica sirve cada petición por lo que si necesita el servicio de algún otro objeto replicado externo sólo se enviará un mensaje de petición y sólo se recibirá una respuesta. En el modelo activo, cada réplica procesa todas las peticiones de los clientes y si tienen que pedir algo a un objeto externo, lo pedirán todas ellas. Para evitar que esas peticiones sean ejecutadas múltiples veces, hay que filtrarlas en el destino y sólo dejar pasar una de ellas. En el retorno debe hacerse lo mismo.

Como puede verse este modelo comparte las principales ventajas de los dos modelos de replicación básicos.

6.1.3 Invocaciones fiables

Pero no todo son ventajas en el modelo coordinador-cohorte que se ha adoptado. Para garantizar ese alto grado de concurrencia se necesita encapsular cada invocación en una especie de transacción. Para ello se ha diseñado el mecanismo IFO descrito en esta tesis.

Una alternativa para dar estas garantías es utilizar soporte transaccional para encerrar cada invocación. Comparado con las transacciones, el mecanismo IFO ofrece algunas ventajas. Por ejemplo, está implantado por el propio ORB utilizado como base por lo que el programador no tiene que preocuparse por la gestión de estas invocaciones. Su uso resulta transparente al programador.

Una segunda aportación reside en el hecho de que permite mantener los resultados de intentos anteriores en caso de fallo, por lo que si al final un reintento de invocación llega a una réplica

servidora no resulta necesario repetir su procesamiento. Esto no es posible obtenerlo con un soporte transaccional tradicional.

Por último, también se ofrece la garantía de progreso. Es decir, nunca se darán abortos de invocaciones. Cuando una invocación ha llegado a modificar una réplica y dicha réplica no falla, se garantiza que todas las demás réplicas que no fallen también actualizarán su estado. Las transacciones tampoco ofrecen esta propiedad.

Sin embargo, para facilitar la garantía de progreso se ha debido exigir que no haya dependencias circulares entre objetos replicados. Así se evita que aparezcan interbloqueos. Esto puede que impida desarrollar algunas aplicaciones que necesiten ese tipo de dependencias circulares.

6.1.4 Control de concurrencia

Por lo que respecta al control de concurrencia, el mecanismo HCC ofrece nuevamente como principal aportación su transparencia para el programador. Basta con que el programador especifique en la interfaz IDL del objeto replicado los conflictos que presentan sus operaciones y después ya no tiene que hacer nada más para gestionar la concurrencia de las invocaciones que estos objetos reciban. El soporte de nuestra arquitectura se encargará de gestionar todos los detalles necesarios.

Esta transparencia resulta ideal para el programador, quien podrá codificar al objeto replicado como si fuera un objeto sencillo. Ni siquiera resultará necesario emplear mecanismos de sincronización tradicionales para admitir soporte a múltiples hilos de ejecución en la codificación de las operaciones de un objeto. Esto facilita mucho el trabajo del programador.

Aparte, por el entorno para el que ha sido diseñado, el mecanismo HCC también debe ser tolerante a faltas. Con ello, aunque caigan algunos de los nodos que mantienen réplicas de objetos, el mecanismo de control de concurrencia puede reconfigurar rápidamente su estado y puede seguir dando servicio a las nuevas invocaciones que vayan llegando.

6.2 Trabajo futuro

En el prototipo actual de HIDRA se ha implantado ya el protocolo de pertenencia presentado en esta tesis, junto a algunos prototipos de los protocolos de invocaciones fiables y de control de concurrencia. Sobre estos últimos todavía no se dispone de cifras de rendimiento porque las aplicaciones implantadas para probarlos son excesivamente simples y porque el soporte definitivo para las referencias del modelo coordinador-cohorte todavía no está elaborado en la versión actual del ORB.

Cuando estos trabajos finalicen, se prevé ampliar la arquitectura HIDRA para que puedan agruparse múltiples clusters para formar *grupos de clusters*. En ese ámbito deberán desarrollarse nuevos protocolos de pertenencia, probablemente jerárquicos, para gestionar los elementos que pertenezcan a dicho tipo de grupo. En estos nuevos protocolos de pertenencia deberá admitirse soporte para particionado de la red, con lo que el algoritmo a emplear tendrá que variar ligeramente.

Otra tarea que puede llegar a incluirse en otro protocolo de pertenencia sería el mantenimiento del grupo de nodos clientes de un cluster. Esta información podría tener interés para algunos tipos de aplicaciones, por lo que podría también diseñarse un monitor de pertenencia de ese estilo. Nuevamente debería admitir particiones y el tipo de pasos a seguir para notificar los cambios también variaría respecto al utilizado en HMM.

También se prevé diseñar e implantar nuevas variantes del protocolo IFO donde desaparezca la restricción de la estructuración en niveles de objetos replicados. Con ello podrían aparecer interbloqueos y se debería introducir soporte para *versionado* de objetos, así como para poder efectuar abortos de invocaciones. Esta nueva variante del protocolo IFO sería similar a la extensión realizada en su momento con las *transacciones anidadas* respecto a las transacciones simples. Es decir, que habría que mantener toda la cadena de invocaciones iniciada desde cierto punto e identificar cada cadena de manera independiente. Por ello, los RoiIDs actuales deberían evolucionar para tener dos tipos. Por un lado los RoiIDs simples, asignados a cada invocación aislada. Por otro lado, los RoiIDs complejos, que englobarían a una lista de RoiIDs simples y que servirían para identificar una determinada cadena de invocaciones anidadas. Estos RoiIDs complejos serían utilizables para diseñar algoritmos de detección de interbloqueos (que es el problema principal introducido por la admisión de dependencias circulares entre objetos replicados). Mediante estos identificadores debería ser posible iniciar el aborto de alguna invocación aislada, de forma que se resuelva el interbloqueo y el sistema recupere su funcionamiento habitual.

Bibliografía

[AA92] Divyakant Agrawal y Amr El Abbadi. The generalized tree quorum protocol: An efficient approach for managing replicated data. *ACM Trans. on Database Sys.*, 17(4):689–717, diciembre 1992.

[ACDK98] Tal Anker, Gregory V. Chockler, Danny Dolev y Idit Keidar. Scalable group membership services for novel applications. En Marios Mavronicolas, Michael Merritt y Nir Shavit, editores, *Networks in Distributed Computing*, volumen 45 de *DIMACS*, págs. 23–42. American Mathematical Society, 1998. ISBN 0-8218-0992-X.

[AD76] Peter Allyn Alsberg y John D. Day. A principle for resilient sharing of distributed resources. En *Proc. of the 2nd International Conference on Software Engineering*, págs. 562–570, San Francisco, California, EE.UU., 1976.

[All83] James E. Allchin. *An Architecture for Reliable Decentralized Systems*. Tesis doctoral, Georgia Institute of Technology, Atlanta, EE.UU., septiembre 1983.

[AMMS+93] Yair Amir, Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal y P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. En *Proc. of the 13th International Conference on Distributed Computing Systems*, págs. 551–560, Pittsburgh, PA, EE.UU., mayo 1993. IEEE-CS Press.

[AMMS+95] Yair Amir, Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal y P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, noviembre 1995.

[BA89] José Manuel Bernabéu Aubán y Mustaque Ahamad. Applying a path-compression technique to obtain an efficient distributed mutual exclusion algorithm. En J. C. Bermond y M. Raynal, editores, *3rd International Workshop on Distributed Algorithms, Niza, Francia*, LNCS, págs. 33–44. Springer-Verlag, septiembre 1989.

[Bat98] Jordi Bataller Mascarell. *Aplicacions distribuïdes sobre memòria compartida: suport i anàlisi formal*. Tesis doctoral, Depto. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, junio 1998.

[BBD96] Özalp. Babaoğlu, Alberto Bartoli y Gianluca Dini. On programming with view synchrony. En *Proc. of the 16th International Conference on Distributed Computing Systems, Hong Kong*, págs. 3–10. IEEE-CS Press, mayo 1996.

[BBG+89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann y Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, febrero 1989.

[BDM95] Özalp Babaoğlu, Renzo Davoli y Albert Montresor. Failure detectors, group membership and view-synchronous communication in partitionable asynchronous systems. Informe técnico, UBLCS-95-18, Depto. de Ciencias de la Computación, Universidad de Bolonia, Bolonia, Italia, noviembre 1995.

[Ber96] Philip A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2), febrero 1996.

[BG93] Kenneth P. Birman y Bradford B. Glade. Consistent failure reporting in reliable communication systems. Informe técnico TR93-1349, Depto. de Ciencias de la Computación, Univ. de Cornell, EE.UU., mayo 1993.

[BGMS86] Daniel Barbará, Héctor García-Molina y Annemarie Spauster. Protocols for dynamic vote reassignment. En *Proc. of the ACM Conference on Principles of Distributed Computing*, págs. 195–205, New York, EE.UU., 1986. ACM.

[Bir85] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. En *Proc. of the 10th ACM Symp. on Operating System Principles, Orcas Island, Washington*, págs. 79–86, diciembre 1985.

[BJ87] Kenneth P. Birman y Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. En *Proc. of the 11th ACM Symposium on Operating Systems Principles*, págs. 123–138, Austin, Texas, EE.UU., noviembre 1987. ACM Press.

[BJR84] Kenneth P. Birman, Thomas Joseph y Thomas Raeuchle. Concurrency control in resilient objects. Informe técnico, TR 84-622, Depto. de Ciencias de la Computación, Universidad de Cornell, Ithaca, New York, EE.UU., julio 1984.

[BJRA85] Kenneth P. Birman, Thomas Joseph, Thomas Raeuchle y Amr El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Trans. on SW Eng.*, 11(6):502–508, junio 1985.

[BK91] Naser S. Barghouti y Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, septiembre 1991.

[Blo79] Toby Bloom. Evaluating synchronisation mechanisms. En *7th International ACM Symposium on Operating System Principles*, págs. 24–32, 1979.

[BMK96] José M. Bernabéu Aubán, Vlada Matena y Yousef A. Khalidi. Extending a traditional OS using object-oriented techniques. En *2nd Conf. on Object-Oriented Technologies & Systems, Toronto, Canada*, págs. 53–63. USENIX, junio 1996.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider y Sam Toueg. The primary-backup approach. En S. J. Mullender, editor, *Distributed Systems*, capítulo 8, págs. 199–216. ACM Press, Addison-Wesley, Wokingham, Reino Unido, 2^a edición, 1993. ISBN 0-201-62427-3.

[BvR94] Kenneth P. Birman y Robbert van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, California, EE.UU., 1994. 398 págs. ISBN 0-8186-5342-6.

[CDK94] George Coulouris, Jean Dollimore y Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishers Ltd, Harlow, Reino Unido, 2^a edición, 1994. 664 págs., ISBN 0-201-62433-8.

[Che86] David R. Cheriton. Request-response and multicast interprocess communication in the V kernel. En *Networking in Open Systems*, págs. 297–312. Springer-Verlag, LNCS no. 248, agosto 1986.

[Che88] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3), marzo 1988.

[Cho94] Vicente Cholvi Juan. *Formalización de modelos de memoria*. Tesis doctoral, Depto. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, diciembre 1994.

[CHTCB96] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg y Bernadette Charron-Bost. On the impossibility of group membership. En *Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, págs. 322–330, Nueva York, EE.UU., mayo 1996. ACM.

[Com99] Douglas E. Comer. *Computer Networks and Internets*. Prentice-Hall International, Inc., Upper Saddle River, New Jersey, EE.UU., 2^a edición, 1999. 608 págs., ISBN 0-13-084222-2.

[Coo84] Eric Charles Cooper. Replicated procedure call. En *Symposium on Principles of Distributed Systems (PODC'84)*, págs. 220–232, agosto 1984.

[Coo85] Eric Charles Cooper. *Replicated Distributed Programs*. Tesis doctoral, Universidad de California, Berkeley, CA, EE.UU., abril 1985.

[Cri91a] Flaviu Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 6(4):175–187, 1991.

[Cri91b] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):57–78, febrero 1991.

[Dij65] Edsger W. Dijkstra. Cooperating sequential processes. Informe técnico, EWD-123, Universidad Tecnológica, Eindhoven, Holanda, 1965.

[DM96] Danny Dolev y Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, abril 1996.

[DMS96] Danny Dolev, Dalia Malki y Ray Strong. A framework for partitionable membership service. En *Proc. of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, págs. 343–343, Nueva York, EE.UU., mayo 1996. ACM.

[Dol96] Dolphin Interconnect Solutions. The Dolphin SCI interconnect. White Paper, febrero 1996. 16 págs.

[FKM+95] Roy Friedman, Idit Keidar, Dalia Malki, Ken Birman y Danny Dolev. Deciding in partitionable networks. Informe técnico TR95-1554, Depto. de Ciencias de la Computación, Univ. de Cornell, Ithaca, NY, EE.UU., noviembre 1995.

[Gal00] Doreen L. Galli. *Distributed Operating Systems: Concepts and Practice*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, EE.UU., marzo 2000. 484 págs., ISBN 0-13-079843-6.

[Gal01] Pablo Galdámez Saiz. *HIDRA: Una arquitectura para alta disponibilidad en sistemas distribuidos. Soporte a objetos.* Tesis doctoral, Depto. de Sistemas Informáticos y Computación, Univ. Politécnica de Valencia, 2001. En preparación.

[Ghe90] Sanjay Ghemawat. Automatic replication for highly available services. Informe técnico, MIT-LCS-TR-473, Lab. of Computer Science, Massachusetts Institute of Technology, Cambridge, MA, EE.UU., enero 1990.

[Gif79] David K. Gifford. Weighted voting for replicated data. En *Proc. of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, págs. 150–159, Pacific Grove, California, EE.UU., diciembre 1979. ACM.

[GMB97a] Pablo Galdámez, Francesc D. Muñoz Escoí y José M. Bernabéu Aubán. HIDRA: Architecture and high availability support. Informe técnico, DSIC-II/14/97, Depto. de Sistemas Informáticos y Computación, Univ. Politécnica de Valencia, mayo 1997.

[GMB97b] Pablo Galdámez, Francesc D. Muñoz Escoí y José M. Bernabéu Aubán. High availability support in CORBA environments. En F. Plášil y K. G. Jeffery, editores, *24th Seminar on Current Trends in Theory and Practice of Informatics, Milovy, República Checa*, volumen 1338 de *LNCS*, págs. 407–414. Springer Verlag, noviembre 1997.

[GMB99a] Pablo Galdámez, Francesc D. Muñoz Escoí y José M. Bernabéu Aubán. Garbage collection for mobile and replicated objects. En J. Pavelka, G. Tel y M. Bartosek, editores, *26th Seminar on Current Trends in Theory and Practice of Informatics, Milovy, República Checa*, volumen 1725 de *LNCS*, págs. 379–386. Springer Verlag, noviembre 1999.

[GMB99b] Pablo Galdámez, Francesc D. Muñoz Escoí y José M. Bernabéu Aubán. Garbage collection in a kernel-based ORB. En Francesc D. Muñoz Escoí y José M. Bernabéu Aubán, editores, *Actas de las VII Jornadas de Concurrencia*, págs. 139–150, Gandía, junio 1999. Univ. Politécnica de Valencia.

[GR93] Jim Gray y Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, EE.UU., 1993.

[Gra79] Jim Gray. Notes on database operating systems. En R. Bayer, R. Graham y G. Seegmuller, editores, *Operating Systems: An Advanced Course*. Springer-Verlag, 1979.

[Her87a] Maurice Herlihy. Concurrency versus availability: Atomicity mechanisms for replicated data. *ACM Trans. on Computer Sys.*, 5(3):249–274, agosto 1987.

[Her87b] Maurice Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, junio 1987.

[Her90] Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Trans. on Database Sys.*, 15(1):96–124, marzo 1990.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, octubre 1974.

[HS95] Matti A. Hiltunen y Richard D. Schlichting. Understanding membership. Informe técnico 95-07, Depto. de Ciencias de la Computación, Univ. de Arizona, Tucson, AZ, EE.UU., julio 1995.

[HT92] Toshio Hirotsu y Mario Tokoro. Object-oriented transaction support for distributed persistent objects. En *Proc. of the 2nd International Workshop on Object-Orientation in Operating Systems*, septiembre 1992.

[HT93] Vassos Hadzilacos y Sam Toueg. Fault-tolerant broadcasts and related problems. En Sape J. Mullender, editor, *Distributed Systems*, capítulo 5, págs. 97–146. ACM Press, Addison-Wesley Publishing Company, Wokingham, Reino Unido, 2^a edición, 1993. ISBN 0-201-62427-3.

[JM90] Sushil Jajodia y David Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. on Database Sys.*, 15(2):230–280, junio 1990.

[JPA00] Ricardo Jiménez Peris, Marta Patiño Martínez y Sergio Arévalo. Deterministic scheduling for transactional multithreaded replicas. En Diego Cazorla, editor, *Actas de las VIII Jornadas de Concurrencia*, págs. 235–247, Cuenca, junio 2000. Univ. de Castilla-La Mancha. ISBN 84-8427-074-2.

[KB94] Narayanan Krishnakumar y Arthur J. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Trans. on Database Sys.*, 19(4):586–625, diciembre 1994.

[Ken86] Gregory Grant Kenley. *An Action Management Systema for a Decentralized Operating System*. Tesis doctoral, Georgia Institute of Technology, Atlanta, EE.UU., enero 1986.

[KG94] Hermann Kopetz y Günter Grünsteidl. TTP - A protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, enero 1994.

[KS93] Akhil Kumar y Arie Segev. Cost and availability tradeoffs in replicated data concurrency control. *ACM Trans. on Database Sys.*, 18(1):102–131, marzo 1993.

[KT91] M. Frans Kaashoek y Andrew S. Tanenbaum. Group communication in the Amoeba distributed operating system. En *Proc. of the 11th IEEE International Conference on Distributed Computing Systems*, págs. 222–230, julio 1991.

[Lam78] Leslie Lamport. Time, clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7), julio 1978.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira y Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. on Comp. Sys.*, 10(4):360–391, noviembre 1992.

[LMS93] Mark C. Little, D. L. McCue y Santosh K. Shrivastava. Maintaining information about persistent replicated objects in a distributed system. En *Proc. of the 13th International Conference on Distributed Computing Systems*, págs. 491–498, Pittsburgh, EE.UU., mayo 1993.

[LS93] Mark C. Little y Santosh K. Shrivastava. Object replication in Arjuna. Informe técnico, BROADCAST TR-50, Depto. de Ciencias de la Computación, Univ. de Newcastle, Newcastle upon Tyne, Gran Bretaña, 1993.

[MAAG96] F. J. Miranda, Ángel Álvarez, Sergio Arévalo y F. J. Guerra. Drago: An Ada extension to program fault-tolerant distributed applications. En A. Stohmeier, editor, *Proc. of the International Conference on Reliable Software Technologies*, volumen 1088 de *LNCS*, págs. 235–246. Springer Verlag, 1996.

[Maf95] Silvano Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. Tesis doctoral, Depto. de Ciencias de la Computación, Universidad de Zurich, febrero 1995.

[MAMSA94] Louise E. Moser, Yair Amir, Peter M. Melliar-Smith y Deborah A. Agarwal. Extended virtual synchrony. En *Proc. of the 14th IEEE International Conference on Distributed Computing Systems, Poznan, Polonia*, págs. 56–65, junio 1994.

[MB97] Francesc D. Muñoz Escoí y José M. Bernabéu Aubán. The NanOS microkernel: A basis for a multicomputer cluster operating system. En H. R. Arabnia, editor, *Proc. of the 3rd International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, EE.UU.*, págs. 127–135. CSREA, julio 1997.

[MBG97] Francesc D. Muñoz Escoí, José M. Bernabéu Aubán y Pablo Galdámez. Fault handling in distributed systems with group membership services. Informe técnico, DSIC-II/8/97, Depto. de Sistemas Informáticos y Computación, Univ. Politécnica de Valencia, mayo 1997. 29 págs.

[McH94] Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. Tesis doctoral, Depto. de Ciencias de la Computación, Trinity College, Dublín, Irlanda, octubre 1994.

[MFSW95] Cristophe P. Malloth, Pascal Felber, André Schiper y Uwe Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. En *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, EE.UU., octubre 1995.

[MGB98a] Francesc D. Muñoz Escoí, Pablo Galdámez y José M. Bernabéu Aubán. HCC: A concurrency control mechanism for replicated objects. En *Actas de las VI Jornadas de Concurrencia, Pamplona, España*, págs. 189–204, junio 1998.

[MGB98b] Francesc D. Muñoz Escoí, Pablo Galdámez y José M. Bernabéu Aubán. Reliable object invocation in HIDRA. Informe técnico, DSIC-II/9/98, Depto. de Sistemas Informáticos y Computación, Univ. Politécnica de Valencia, marzo 1998.

[MGB98c] Francesc D. Muñoz Escoí, Pablo Galdámez y José M. Bernabéu Aubán. ROI: An invocation mechanism for replicated objects. En *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems, Purdue Univ., West Lafayette, IN, EE.UU.*, págs. 29–35, octubre 1998.

[MGB98d] Francesc D. Muñoz Escoí, Pablo Galdámez y José M. Bernabéu Aubán. A synchronisation mechanism for replicated objects. En B. Rovan, editor, *Proc. of the 25th Conference on Current Trends in Theory and Practice of Informatics, Jasná, Eslovaquia*, volumen 1521 de *LNCS*, págs. 389–398. Springer Verlag, noviembre 1998.

[MGB99a] Francesc D. Muñoz Escoí, Pablo Galdámez y José M. Bernabéu Aubán. The NanOS cluster operating system. En R. Buyya, editor, *High Performance Cluster Computing*, volumen 1, capítulo 29, págs. 682–702. Prentice-Hall PTR, Upper Saddle River, NJ, EE.UU., 1999.

[MGB99b] Francesc D. Muñoz Escoí, Pablo Galdámez y José M. BernabéuAubán. Uso de interceptores CORBA para dar soporte a objetos replicados. En Francesc D. Muñoz Escoí y José M. Bernabéu Aubán, editores, *Actas de las VII Jornadas de Concurrencia*, págs. 209–222, Gandía, junio 1999. Univ. Politécnica de Valencia.

[MGB00] Francesc D. Muñoz Escoí, Pablo Galdámez y José M. Bernabéu-Aubán. HMM: A membership protocol for a multi-computer cluster. Informe técnico, ITI-DE-2000/02, Instituto Tecnológico de Informática, Univ. Politécnica de Valencia, febrero 2000.

[MGGB00] Francesc D. Muñoz Escoí, Óscar Gomis Hilario, Pablo Galdámez y José M. Bernabéu-Aubán. HMM: A membership protocol for a multi-computer cluster. En *Anexo de las actas de las VIII Jornadas de Concurrencia*, Cuenca, España, junio 2000.

[MMBG97] Francesc D. Muñoz Escoí, Vlada Matena, José M. Bernabéu-Aubán y Pablo Galdámez. A membership protocol for multi-computer clusters. Informe técnico, DSIC-II/20/97, Depto. de Sistemas Informáticos y Computación, Univ. Politécnica de Valencia, mayo 1997. 19 págs.

[MMSA+95] Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulis y T. P. Archambault. The Totem system. En *25th IEEE International Symposium on Fault-Tolerant Computing*, págs. 61–66. IEEE Computer Society Press, junio 1995.

[Mos81] John E. B. Moss. Nested transactions: An approach to reliable distributed computing. Informe técnico, MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[Nel90] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, 23(7):19–25, julio 1990.

[Nut97] Gary J. Nutt. *Operating Systems: A Modern Perspective*. Addison-Wesley Publishing Company, Reading, Massachusetts, EE.UU., 1ª edición, octubre 1997. 654 págs., ISBN 0-8053-1295-1.

[OMG98a] OMG. *CORBAservices: Common Object Services Specification*. Documento: formal/98-12-09, Object Management Group, diciembre 1998.

[OMG98b] OMG. *Fault Tolerant CORBA Using Entity Redundancy. Request for Proposals*. Documento: orbos/98-04-01, Object Management Group, abril 1998.

[OMG99a] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, julio 1999. Revisión 2.3.

[OMG99b] OMG. *Presentation on the status of the Fault Tolerance submission*. Documento: orbos/99-08-27, Object Management Group, agosto 1999.

[OMG00] OMG. *Fault Tolerant CORBA Specification, V1.0*. Documento: ptc/2000-04-04, Object Management Group, abril 2000.

[PBR77] C. H. Papadimitriou, Philip A. Bernstein y J. Rothanie. Some computational problems related to database concurrency control. En *Proc. of the Conference on Theoretical Computer Science*, 1977.

[Pfi98] Gregory F. Pfister. *In Search of Clusters. The Ongoing Battle in Lowly Parallel Computing*. Prentice Hall PTR, Upper Saddle River, New Jersey, EE.UU., 2ª edición, enero 1998. 606 págs., ISBN 0-13-899709-8.

[Pos80] Jonathan Bruce Postel. User Datagram Protocol. *RFC 768*, agosto 1980. Disponible en `ftp://ds.internic.net`.

[PSWL95] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater y Mark C. Little. The design and implementation of Arjuna. *USENIX Computing Systems*, 8(3):255–308, verano 1995.

[Rah93] Erhard Rahm. Empirical performance evaluation of concurrency and coherency control protocols for database sharing systems. *ACM Trans. on Database Sys.*, 18(2):333–377, junio 1993.

[RB94] Aleta Ricciardi y Kenneth P. Birman. Consistent process membership in asynchronous environments. En Kenneth P. Birman y Robert van Renesse, editores, *Reliable Distributed Computing with the Isis Toolkit*, capítulo 13, págs. 237–262. IEEE Computer Society Press, Los Alamitos, CA, EE.UU., 1994.

[RFJ93] Ragunathan Rajkumar, Sameh Fakhouri y Farnam Jahanian. Processor group membership protocols: Specification, design and implementation. En *Proc. of the 12th IEEE Symposium on Reliable Distributed Systems, Princeton, NJ, EE.UU.*, págs. 2–11, octubre 1993.

[RSB93] Aleta Ricciardi, André Schiper y Kenneth P. Birman. Understanding partitions and the "no partition" assumption. En *Proc. of the 4th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems*, págs. 354–360, Lisboa, Portugal, septiembre 1993.

[RVR93] Luís Rodrigues, Paulo Veríssimo y J. Rufino. A low-level processor group membership protocol for LANs. En *Proc. of the 13th International Conference on Distributed Computing Systems*, págs. 541–550, mayo 1993.

[Sch93a] Fred B. Schneider. Replication management using the state-machine approach. En S. J. Mullender, editor, *Distributed Systems*, capítulo 7, págs. 166–197. ACM Press, Addison-Wesley, Wokingham, Reino Unido, 2^a edición, 1993. ISBN 0-201-62427-3.

[Sch93b] Fred B. Schneider. What good are models and what models are good? En S. J. Mullender, editor, *Distributed Systems*, capítulo 2, págs. 17–26. ACM Press, Addison-Wesley, Wokingham, Reino Unido, 2^a edición, 1993. ISBN 0-201-62427-3.

[SG98] Abraham Silberschatz y Peter Baer Galvin. *Operating System Concepts*. Addison-Wesley Publishing Company, Reading, Massachusetts, EE.UU., 5^a edición, febrero 1998. 906 págs., ISBN 0-201-54262-5.

[SS83] Richard D. Schlichting y Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant systems. *ACM Trans. on Computer Sys.*, 1(3):222–238, agosto 1983.

[SS94] Mukesh Singhal y Niranjan G. Shivaratri. *Advanced Concepts in Operating Systems: Distributed, Database and Multiprocessor Operating Systems*. Mc Graw-Hill, Inc., New York, 1994. 544 págs., ISBN 0-07-057572-X.

[Sta98] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice-Hall, Inc., Upper Saddle River, New Jersey, EE.UU., 3^a edición, enero 1998. 800 págs., ISBN 0-13-887407-7.

[Sun99] Sun Microsystems. *The Source for Java Technology*. Sun Microsystems, Inc., Mountain View, California, EE.UU., 1999. Página web en `http://java.sun.com`.

[Tho93] Alexander Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Trans. on Database Sys.*, 18(4):579–625, diciembre 1993.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen y Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, diciembre 1990.

[US 83] US Dept. of Defense. Reference manual of the Ada programming language. Informe técnico, ANSI/MIL-STD-1815A, DoD, Washington, EE.UU., 1983.

[vRBM96] Robbert van Renesse, Kenneth P. Birman y Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, abril 1996.

[Wei88] William E. Weihl. Commutativity-based concurrency control for abstract data types. En *Proc. of the 21st Annual Hawaii International Conference on System Sciences*, págs. 205–214, enero 1988.

Índice de Materias

A

B

C

D

E

F

G

N

O

P

Q

R

S

T

U

V