TESIS DOCTORAL

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSIDAD POLITÉCNICA DE VALENCIA

Hidra: Una Arquitectura para Alta Disponibilidad en Sistemas Distribuidos. Soporte a Objetos.

Presentada por: Pablo Galdámez Saiz
Dirigida por: José Manuel Bernabéu Aubán

Resumen

Debido al abaratamiento de los ordenadores, al continuo aumento de la velocidad y fiabilidad de los sistemas de comunicaciones, y a la progresiva demanda del mercado de sistemas más robustos y eficientes, los sistemas distribuidos altamente disponibles están recibiendo un interés creciente tanto desde el punto de vista comercial como de investigación.

Hidra ha sido el resultado fundamental de 2 proyectos de investigación financiados por la CICYT[1] y ha dado lugar a varias publicaciones y ponencias en congresos nacionales e internacionales. Se trata de una arquitectura diseñada para servir de soporte al desarrollo de sistemas y aplicaciones altamente disponibles. Está compuesta de un Gestor de Invocaciones a Objetos (ORB) altamente disponible con soporte a objetos replicados y diversos servicios relacionados con transacciones distribuidas, control de concurrencia y recolección de residuos. El diseño se ha realizado para permitir la inclusión de Hidra como parte del núcleo de un sistema operativo, permitiendo construir, no sólo aplicaciones altamente disponibles, sino también componentes del sistema operativo que permitan el desarrollo futuro de un sistema operativo distribuido que ofrezca la imagen de sistema único.

Hidra proporciona el concepto de objeto como la pieza básica con la que se construirán las aplicaciones. Los objetos podrán estar replicados o ser objetos normales de implementación única. La ubicación de los objetos y su condición de replicados es transparente a los usuarios de dichos objetos, garantizando en el caso de los objetos replicados una mayor disponibilidad. El sistema incluye recolección de residuos para todos los objetos, permitiendo el desarrollo de sistemas que no consuman más recursos que los estrictamente necesarios, sin la sobrecarga que le supondría al programador realizar tal tarea sin soporte.

Hidra está compuesta por dos partes fundamentales: el soporte a objetos y el soporte a un modelo específico de replicación, el modelo coordinador-cohorte. Esta tesis se centra en el soporte a objetos de la arquitectura Hidra, detallando la estructura de las referencias a objeto, el recolector de residuos y los protocolos necesarios para su gestión, describiendo cómo se toleran los fallos y cómo se garantiza alta disponibilidad. Se demuestra la correción de uno de los algoritmos más representativos y se propone un mecanismo para facilitar la depuración del sistema.

[1] Comisión Interministerial de Ciencia y Tecnología.

Resum

Degut a l'abaratiment dels ordenadors, al continu augment de la velocitat i fiabilitat dels sistemes de comunicacions i a la progressiva demanda del mercat de sistemes més robusts i eficients, els sistemes distribuïts altament disponibles estan rebent un interés creixent, tant des del punt de vista comercial com d'investigació.

Hidra ha sigut el resultat fonamental de dos projectes d'investigació finançats per la CICYT[2] i ha generat diverses publicacions i ponències en congresos nacionals i internacionals. Es tracta d'una arquitectura dissenyada per a servir de suport al desenvolupament de sistemes i aplicacions altament disponibles. Està composta d'un Gestor d'Invocacions a Objectes (ORB) altament disponible amb suport a objectes replicats i diversos serveis relacionats amb transaccions distribuïdes, control de concurrència i recollida de residus. El disseny s'ha realitzat per a permetre la inclusió d'Hidra com a part del nucli d'un sistema operatiu, permetent construir, no només aplicacions altament disponibles, sinó també components del sistema operatiu que puguen permetre el desenvolupament futur d'un sistema operatiu distribuït que oferisca la imatge de sistema únic.

Hidra proporciona el concepte d'objecte com la peça bàsica amb la qual es construiran les aplicacions. Els objectes podran estar replicats o ser objectes normals d'implementació única. La localització dels objectes i llur condició de replicats és transparent als usuaris d'eixos objectes, garantint per al cas dels objectes replicats una major disponibilitat. El sistema inclou recollida de residus per a tots els objectes, permetent el desenvolupament de sistemes que no consumisquen més recursos dels estrictament necessaris, sense la sobrecàrrega que li suposaria al programador fer aquesta tasca sense cap suport.

Hidra està composta per dues parts fonamentals: el suport a objectes i el suport a un model específic de replicació, el model coordinador-cohort. Aquesta tesi es centra en el suport a objectes de l'arquitectura Hidra, detallant l'estructura de les referències a objecte, el recollidor de residus i els protocols necessaris per a la seua gestió, descrivint com es toleren les fallades i com es garanteix alta disponibilitat. Es demostra la correcció d'un dels algorismes més representatius i es proposa un mecanisme per a facilitar la depuració del sistema.

[2]Comisión Interministerial de Ciencia y Tecnología.

Abstract

In recent years, distributed systems are receiving increasing attention from both researchers and industry. The main reasons that make this field more attractive than in previous years can be found in the increasingly low cost of computers, the continuous increase in the speed and reliability of network communication devices, and the progressive demand for more robust and efficient systems.

Hidra has been the main result of two research projects funded by the CICYT[3] and several papers have been published focusing on it. Hidra is an architecture designed to support the development of highly available applications and systems. It is composed by a highly available object request broker (ORB) with replicated objects support, with several services related with distributed transactions, concurrency control and garbage collection. Its design allows the inclusion of Hidra into the operating systems kernel. This inclusion would permit the development of several components which reside into the kernel making use of the Hidra services. The final goal of this approach is to enable the future development of a distributed operating system offering a single system image.

Hidra provides the object concept as the basic building block to develop applications. Hidra objects will be standard non replicated objects or they will be replicated objects. Their location and whether they are replicated or not will be transparent to the object users, providing higher degrees of availability when objects are replicated. The system includes garbage collection for every kind of object. Garbage collection will relieve programmers of the task of controlling the disposal of unused resources, making it easier to develop components that do not use more resources than those strictly needed.

Hidra is composed by two fundamental parts: the object support and the support for an specific replication model, the coordinator-cohort replication model. This thesis focuses into the object support provided by Hidra, detailing the structure of its object references, the garbage collection and the protocols that manage references and objects. Special emphasis will be given on how the system copes with failures and how it increases availability. One the most representative algorithms is formally proven and finally a mechanism to debug the systems is presented.

[3]Comisión Interministerial de Ciencia y Tecnología.

A Raquel.

Índice

Índice de Figuras

Capítulo 1

Introducción

Un sistema distribuido es un sistema que impide realizar tu trabajo cuando se estropea un ordenador del que nunca habías oído hablar antes.

Leslie Lamport.

Contenidos del capítulo

Las necesidades de comunicación entre organizaciones, entre particulares y dentro de una misma organización han venido haciendo de los sistemas de comunicación entre ordenadores un elemento de amplia y creciente implantación. Desde las redes locales que interconectan los ordenadores de una entidad, las redes de área extensa que conectan diversas entidades, hasta Internet como medio de interconexión global, tenemos que recientemente se asocia el uso de ordenadores al uso de algún medio de comunicación.

1.1 Sistemas distribuidos

Este amplio y creciente uso de varios ordenadores interconectados por redes de comunicaciones dio lugar a la aparición hace varias décadas del concepto de sistema distribuido. Los sistemas distribuidos no son más que un conjunto de ordenadores que trabajan de forma conjunta para realizar alguna tarea. Esta definición, similar a las que podemos encontrar en multitud de fuentes [127, 55] queda más completa si enumeramos las tres componentes [127] que forman los sistemas distribuidos:

- *Ordenadores:* Todo sistema distribuido está formado por más de un ordenador físico, al que denominaremos nodo del sistema distribuido o simplemente nodo. Los nodos al menos tendrán CPU, memoria y dispositivos de entrada/salida que les permitan comunicarse con el entorno.

- *Red:* Alguno de los dispositivos de entrada/salida de los nodos serán periféricos de comunicaciones que permitirán interconectar a los nodos.

- *Estado compartido:* Los nodos de un sistema distribuido mantienen un estado compartido. Para mantener este estado de forma correcta, deberán coordinarse entre ellos.

La tercera componente de los sistemas distribuidos, el estado compartido es la que establece la diferencia fundamental que existe entre los sistemas distribuidos y otros sistemas de ordenadores conectados con redes. Estos últimos son conocidos como sistemas centralizados, por el hecho de que mantienen el estado del sistema en un ordenador centralizado: el servidor.

En los sistemas centralizados, el servidor es un cuello de botella para todo el sistema, ya que es el único que almacena la información relevante. Adicionalmente, los fallos del servidor, las reparaciones que se le deban realizar, o su posible sustitución por equipos más potentes hacen que tales sistemas dejen de estar disponibles mientras el servidor no esté completamente operativo.

Los sistemas distribuidos intentan paliar esta deficiencia, incrementando por contra notablemente su complejidad. Al estar formados por nodos independientes interconectados por algún tipo de red, fuerzan al diseñador a contemplar las siguientes cuestiones:

- *Fallos en los nodos:* Conforme más nodos formen parte de un sistema distribuido, más alta es la probabilidad de que alguno de ellos falle. Por este motivo, los sistemas distribuidos, que se construyen en gran medida para evitar la dependencia de los posibles fallos del servidor, deben diseñarse para tolerar los fallos que aparecerán en los distintos nodos.

- *Comunicaciones no fiables:* Por regla general, los sistemas de comunicaciones que interconectan los nodos de un sistema distribuido no son perfectos, esto es, a veces los mensajes enviados de un nodo a otro no llegan, otras veces se pierden o se duplican, y otras veces los mensajes llegan desordenados respecto al orden en que fueron enviados.

- *Comunicaciones costosas:* Los canales de comunicación entre los diferentes nodos del sistema tienen habitualmente menor ancho de banda, mayor latencia y mayor coste, que los canales disponibles para interconectar distintos procesos dentro de un único ordenador.

1.1.1 Sistemas cluster

Otros autores van más allá y asimilan el concepto de sistema distribuido al de sistema operativo distribuido. Así, Tanenbaum y Van Renesse [130] sugirieron hace más de 15 años esta definición:

> *Un sistema [operativo] distribuido es aquel que ofrece a los usuarios la imagen de un único sistema [operativo] centralizado, pero que se ejecuta en múltiples e independientes CPUs. La clave principal es la transparencia. En otras palabras, el uso de múltiples ordenadores debe ser invisible (transparente) al usuario. [...] el usuario ve al sistema como un "uniprocesador virtual".*

Esta definición, fue completada poco tiempo después por Sape Mullender [99] añadiendo lo siguiente:

> *La definición de Tanenbaum y van Renesse proporciona una condición necesaria para los sistemas [operativos] distribuidos, pero no suficiente en mi opinión. Un sistema [operativo] distribuido, debe además no tener ningún único punto de fallo – que ningún fallo en un sólo punto pueda provocar que todo el sistema deje de estar disponible.*

En este caso, tenemos lo que comúnmente se refiere en la literatura a sistemas cluster[1]. En este trabajo utilizaremos el término *sistema distribuido* para el concepto más general y *cluster* para el concepto de sistema operativo distribuido presentado por Tanenbaum y van Renesse. Tal y como menciona Pfister en su libro dedicado específicamente a los clusters [116], encontramos varias características diferenciadoras entre ambos conceptos:

- *Nominación:* Los nodos de los sistemas distribuidos suelen estar identificados por algún nombre, lo que hace que tanto usuarios como aplicaciones puedan acceder a los nodos de forma independiente. Por su parte en los clusters, las aplicaciones o usuarios no accederán a los nodos de forma independiente, sino al cluster como un todo. Deberá por tanto haber una capa de software que se encargue de proporcionar la característica de *imagen de sistema único*.

[1] En esta tesis preferimos el término anglosajón *cluster* a posibles traducciones como *cúmulo*, por ser la primera la que aparece con más frecuencia en textos en castellano.

- *Jerarquía:* En los sistemas distribuidos suele existir alguna relación de jerarquía entre los diferentes nodos que lo forman. Podremos encontrar servidores de cierto tipo de servicio o meros clientes. Por contra, en los clusters veremos que los nodos son homogéneos en sus roles, encargándose cada uno de forma potencial de las mismas tareas.

- *Interfaz:* Los sistemas en cluster ofrecen la imagen de sistema [operativo] único, mientras que los sistemas distribuidos pueden estar formados por ordenadores que ejecuten diferentes sistemas operativos y la interfaz que éstos ofrecen a sus usuarios puede ser heterogénea.

- *Comunicación:* Los sistemas distribuidos al estar formados por máquinas heterogéneas suelen adoptar algún estándar para comunicar a los diversos nodos, mientras que en los clusters, se suele adoptar algún protocolo nativo que maximice su eficiencia al estar adaptado a las características propias del sistema.

- *Envergadura:* Sistemas distribuidos pueden ser desde un conjunto de pocos ordenadores, hasta miles de ellos (p.e: Internet), mientras que los clusters suelen estar formados como mucho por pocas decenas de máquinas.

Todas estas características hacen de los clusters un subconjunto de los sistemas distribuidos, donde primará la eficiencia y la homogeneidad al coste de perder posiblemente estandarización y escalabilidad, teniendo en ambos casos especial importancia el diseño del sistema para hacer frente a los fallos y el diseño de un adecuado sistema para su administración.

1.2 Alta disponibilidad

Tal y como hemos comentado en los apartados precedentes, los sistemas distribuidos y en particular los clusters, se construyen en gran medida para eliminar la dependencia de un único cuello de botella tanto en lo referente al rendimiento, como a la disponibilidad general del sistema. Este objetivo se logra diseñando un sistema que no dependa de la disponibilidad de un único nodo, que permita introducir nuevos nodos durante su ejecución y que enmascare la aparición de los fallos que puedan aparecer tanto en los nodos como en el propio sistema de comunicaciones. De esta forma, se ofrecerá la visión de disponer de una plataforma más potente que la ofrecida por un único ordenador y que estará "siempre" funcionando.

El término *disponibilidad* se refiere a la cantidad de tiempo que un sistema está proporcionando servicio, en relación al tiempo en que los usuarios desean utilizarlo. La disponibilidad [109] suele cuantificarse con la siguiente ecuación:

$$D = MTTF/(MTTF + MTTR) * 100$$

Donde *MTTF*[2] es el tiempo medio durante el cual el sistema está proporcionando servicio y *MTTR*[3] es el tiempo medio que tarda el sistema en ser reparado o acondicionado para su funcionamiento. La disponibilidad suele expresarse en tanto por ciento, y permite clasificar a los

[2]Del inglés *Mean Time To Failure*, o tiempo medio hasta fallo.
[3]Del inglés *Mean Time to Repair*, o tiempo medio para reparar.

sistemas en función del grado de disponibilidad que proporcionan. Así, a los sistemas disponibles menos del 99.99% del tiempo se los suele clasificar como sistemas de disponibilidad baja o media, mientras que se consideran *altamente disponibles* a aquellos que superen el 99.99% de disponibilidad (una indisponibilidad de menos de 1 hora al año aproximadamente). En este rango se suelen situar las pretensiones de los clusters, que demandando disponibilidad $365\text{x}24$, asumen como tolerable ciertas indisponibilidades temporales de reducida duración (del orden de segundos las más prolongadas). En el escalafón más exigente se encuentran las aplicaciones que demandan ultra-alta disponibilidad o disponibilidad total, como por ejemplo aplicaciones de control de centrales nucleares, de navegación automatizada o de mantenimiento de constantes vitales de seres humanos.

1.2.1 Fallos

Decimos que un servicio funciona correctamente, si como respuesta a sus entradas, proporciona sus salidas[4] de forma consistente a su especificación. Un *fallo* en el servicio ocurre cuando éste no se comporta de acuerdo a su especificación. En este trabajo utilizaremos por tanto el término fallo en sentido amplio, y por *tolerancia a fallos*[5] entenderemos las técnicas que intentan recuperar o enmascarar fallos en sentido amplio.

Tipos de fallos

Los fallos que pueden hacer que un servicio no funcione correctamente son muy diversos y pueden agruparse en diversas categorías. Una clasificación generalmente aceptada la proporciona Cristian [27], en la que se jerarquizan los fallos de acuerdo a su severidad:

Un *fallo de respuesta* ocurre cuando el servicio responde de manera incorrecta. Puede incluso que el servicio responda de manera incorrecta, pero que esta respuesta sea la misma que habría proporcionado el servicio frente a otras entradas. Las causas que pueden producir fallos de respuesta son múltiples y van desde la aparición de defectos físicos en el hardware, a errores en el diseño de los servicios e incluso pueden estar provocados de forma malintencionada. A estos últimos fallos de respuesta, de compleja solución, se los conoce como *fallos bizantinos*. Un *fallo de temporización* ocurre cuando la respuesta del servicio es funcionalmente correcta, pero se realiza fuera del rango temporal en el que se debería proporcionar. Un *fallo de omisión* ocurre cuando el servicio no responde a una entrada. Un *fallo de caída* (crash failure) ocurre cuando después de que ocurra un fallo de omisión, el servicio ya no responde a ninguna entrada más hasta que sea reiniciado.

Cabe destacar que la clasificación presentada es jerárquica, donde los fallos de caída están incluidos en los fallos de omisión, que estos son un tipo de fallos de temporización y estos a su vez son una subclase de los fallos de respuesta.

[4] Tanto a otros servicios, como a sus clientes, como salidas que sean meros cambios del estado interno del servicio.

[5] Muchos trabajos prefieren utilizar *tolerancia a fallos* para referirse exclusivamente a técnicas para tolerar defectos físicos del hardware y alta disponibilidad para las técnicas que proporcionan a mayor nivel de abstracción disponibilidad de servicios. Sin embargo también existen multitud de trabajos que utilizan el término *tolerante a fallos* aplicado a protocolos, algoritmos, servicios o sistemas.

Dentro de los fallos de caída y dependiendo del estado del servicio cuando éste sea reiniciado, se distinguen cuatro tipos: *fallo de caída con amnesia, fallo de caída con amnesia parcial*, *fallo de pausa* y *fallo de parada*. Un *fallo de caída con amnesia*, ocurre si el servicio al arrancar de nuevo comienza con un estado inicial que no depende de las entradas que recibiera el servicio antes del fallo de caída. *Fallo de caída con amnesia parcial* ocurre si parte del estado del servicio es el mismo que tenía el servicio antes del fallo y la otra parte se inicializa a cierto estado por defecto. Un *fallo de pausa* (también llamado fallo con semántica de transacciones atómicas) ocurre si al reiniciarse el servicio, éste parte con el estado que tuviera antes del fallo y por último, un *fallo de parada* ocurre si el servicio nunca vuelve a activarse una vez le ocurra un *fallo de caída*.

Modelos de fallos

Los sistemas tolerantes a fallos deben diseñarse para hacer frente a los tipos de fallos que se consideren relevantes para el sistema. Todos los tipos de fallos pueden ocurrir, pero la adecuada elección de qué fallos son los que se pretenden tolerar permitirá elegir un punto de compromiso entre el coste en que se incurrirá para tolerarlos, el rendimiento que tendrá el sistema, y la fiabilidad y disponibilidad que exhibirá el sistema. Esta elección define el modelo de fallos del sistema. El modelo, recogerá los fallos que son previsibles que ocurran en el servicio y que no son despreciables para lograr determinados objetivos de disponibilidad y fiabilidad.

Los sistemas pueden tener más de un modelo de fallos según el nivel de abstracción al que estemos definiéndolos. Por ejemplo, si un servicio depende de un servicio de nivel inferior para funcionar correctamente, entonces un fallo de cierto tipo a ese nivel inferior, puede ser o puede convertirse en un fallo de otro tipo en el nivel superior. Esto es lo que se conoce como *enmascaramiento de fallos jerárquico* [27]. En estos casos, es necesario diseñar el sistema detallando el modelo de fallos de cada nivel y cómo cada nivel enmascara determinados tipos de fallos o cómo los convierte en otro tipo de fallos de menor complejidad.

1.2.2 Mantenimiento

Como ya hemos comentado, para aumentar la disponibilidad hay que reducir al mínimo el tiempo durante el cual el sistema no esté operativo. Las causas tradicionalmente reconocidas de indisponibilidad en los sistemas son los fallos y el mantenimiento del sistema.

Supóngase por ejemplo un sistema que tolere gran cantidad de fallos, pero que deba ser detenido para realizar tareas como ampliar su memoria, mejorar los sistemas de comunicaciones, sustituir cierto servidor por otro más potente, sustituir el software por una versión más actualizada, etc. Esto sin duda resultaría en un sistema que pese a tolerar fallos, tendría un nivel de disponibilidad mejorable. De hecho las caídas planificadas de los servicios debidas a tareas de administración, son una causa habitualmente mayor de indisponibilidad en los sistemas no específicamente diseñados para evitarlas, que las caídas no planificadas debidas a cualquier tipo de fallo.

Por tanto, al diseñar un sistema altamente disponible será necesario además de tolerar los fallos que puedan aparecer, proporcionar mecanismos que faciliten el mantenimiento del sistema sin que sea necesario detener sus servicios, o al menos reduciendo al mínimo el tiempo

durante el cual no se proporcione servicio.

1.3 Alta disponibilidad en sistemas distribuidos

Para construir un sistema distribuido altamente disponible es necesario conocer primero qué tipos de fallos pueden aparecer, después se debe seleccionar un modelo de fallos que agrupe los fallos que se pretendan considerar como relevantes, y por último, se deben diseñar las diferentes componentes de forma que funcionen correctamente en presencia de fallos. Para ello se deberán diseñar algoritmos tolerantes a fallos o se deberán implantar técnicas capaces de enmascararlos.

1.3.1 Fallos en los sistemas distribuidos

Los sistemas distribuidos están formados por múltiples nodos y enlaces de comunicaciones que pueden fallar. Lo habitual es que cada componente falle de forma independiente a las demás, con lo que el diseño de la tolerancia a fallos deberá realizarse para tolerar los fallos de cada componente por separado. Sin embargo, y aunque con una probabilidad mucho menor, la aparición de varios fallos al mismo tiempo pueden provocar situaciones que deben ser consideradas de forma adicional.

Fallos en las comunicaciones

Las comunicaciones entre los distintos nodos de un sistema distribuido pueden verse afectadas por: pérdidas de mensajes, duplicidad en los mensajes, desorden en los mensajes, retrasos y corrupción del contenido.

Para tolerar estos fallos, si se trata de fallos transitorios, se suelen construir protocolos de transporte que utilicen determinadas técnicas para enmascarar los fallos, proporcionando canales de comunicación libres de la mayoría de los fallos. Así por ejemplo, para enmascarar pérdidas sobre determinados mensajes, duplicidades y desorden, se suelen implantar, como por ejemplo en TCP/IP [121], protocolos que utilizan números de secuencia, que exigen confirmaciones de la entrega de mensajes y que realizan reenvíos de mensajes no confirmados. Por su parte, para enmascarar la corrupción del contenido se suelen utilizar códigos de redundancia. Si el fallo de pérdidas de mensajes ocurre en todos los mensajes enviados a través de un canal (fallo de caída), la solución suele consistir en utilizar canales de comunicación alternativos, que permitan el reenvío de los mensajes que fueron enviados por el canal caído a través de otro canal.

Por último, el retraso en los mensajes divide a los sistemas distribuidos entre sistemas distribuidos *síncronos* y *asíncronos*. Los síncronos son los que asumen que los mensajes siempre se envían sin fallos de temporización, por lo que de no recibirse un mensaje en el tiempo esperado se dice que ha ocurrido un fallo de temporización. Los sistemas asíncronos, por contra, se construyen sin hacer asunciones del tiempo que puede tardar un mensaje en llegar a su destino, por lo que no tiene sentido hablar de fallos de temporización.

Fallos en los nodos

Los nodos pueden verse igualmente afectados por fallos de respuesta, de omisión, de temporización y de caída. Si se pretenden enmascarar todos los fallos, incluyendo los fallos de respuesta y omisión, generalmente se utilizan técnicas de replicación y voto, también llamadas técnicas de versionado [3]. La idea fundamental en la que se apoyan estas técnicas, consiste en ejecutar diversas versiones del aplicativo en nodos distintos, de forma que la respuesta correcta será la respuesta que sea alcanzada por la mayoría de las versiones. Con esta técnica se pueden solucionar fallos causados por una defectuosa implementación del software e incluso tolerar fallos bizantinos. Para lograr estos ambiciosos objetivos, es necesario que cada versión a ejecutar haya sido diseñada e implementada por diferentes programadores, pudiéndose reducir al mínimo la probabilidad de que la mayoría de versiones falle. Tolerar fallos de omisión y respuesta de los nodos suele ser el objetivo de los sistemas de disponibilidad ultra-alta o total.

Los fallos de temporización en los nodos, tienen relevancia en los sistemas distribuidos síncronos y en particular en los sistemas distribuidos de tiempo real. En este tipo de sistemas, que el aplicativo no responda dentro de unos límites temporales previamente fijados es considerado como un fallo. Por contra, en los sistemas asíncronos, al no considerarse como fallo el retraso en los mensajes, tampoco se considera como fallo el retraso en la ejecución del software, pues ambos casos son indistinguibles para un nodo remoto. Para solucionar este tipo de fallos en sistemas síncronos también se suelen utilizar técnicas de replicación como por ejemplo los bloques de recuperación distribuida [74]. La idea consiste en ejecutar diferentes réplicas que hayan sido diseñadas para responder dentro de un tiempo prefijado, y admitir la respuesta que se genere en el menor plazo posible, dando a todo el servicio como fallido en caso de que ninguna réplica conteste a tiempo.

Fallos de caída

Los fallos de caída son relevantes en todos los sistemas distribuidos, y forman el mínimo subconjunto de fallos que debería tolerar cualquier sistema que desee ofrecer garantías de alta disponibilidad.

Desafortunadamente, la mayoría de problemas que aparecen en los sistemas distribuidos no se pueden solucionar en sistemas asíncronos si ocurren fallos de caída. Varios trabajos demuestran que es imposible alcanzar consenso de forma determinista en sistemas distribuidos asíncronos donde al menos uno de los nodos falle [41]. Esta imposibilidad incluye a los protocolos de compromiso distribuido, a los protocolos de comunicación a grupos [65] totalmente ordenados, e incluso a los protocolos de pertenencia a grupo [26, 102]. Estos últimos protocolos fueron considerados durante tiempo como solucionables en sistemas asíncronos por presentar ciertas condiciones aparentemente más débiles que el caso general del consenso, sin embargo fue posteriormente demostrado que tampoco son solucionables ni siquiera utilizando su especificación más conservadora [22]. La idea común a estos resultados de imposibilidad consiste en apreciar que es imposible que un nodo dado decida que otro ha fallado, si no puede hacer asunciones en cuanto a la velocidad con la que progresa. Es decir, es imposible distinguir entre un nodo caído y un nodo que progresa lentamente o cuyos mensajes tardan un tiempo no acotado en llegar a su destino.

Estos resultados de imposibilidad hacen que se deba desechar la idea de trabajar con sistemas distribuidos asíncronos si se pretenden tolerar fallos de caída, que como ya comentamos son los fallos más débiles que se deben tratar. Para construir sistemas que toleren fallos de caída y que admitan la posibilidad que los mensajes puedan retrasarse, se modelizan los sistemas como *parcialmente síncronos*. Este modelo puede definirse de varias formas alternativas, como por ejemplo las dos siguientes planteadas por Dwork, Lynch y Stockmeyer [34].

1. Existe un tiempo máximo asociado a la transmisión de los mensajes y una cota a la velocidad relativa entre los procesadores, pero ambos límites no se conocen con precisión.

2. Existe un tiempo máximo asociado a la transmisión de los mensajes y una cota a la velocidad relativa entre los procesadores, conociéndose con exactitud su cuantía, sin embargo sólo se garantiza que ambas cotas serán respetadas a partir de cierto instante *t* desconocido a priori.

La diferencia fundamental entre este modelo y el modelo puramente síncrono, radica en que en el modelo parcialmente síncrono, los retrasos en los mensajes pueden ocurrir, en cuyo caso se admitirá la posibilidad de determinar que un nodo ha caído aunque realmente no lo haya hecho. Para detectar estas caídas (con posibles errores) se asume que en general los mensajes tardarán un tiempo acotado en llegar a su destino.

Para detectar fallos de caída en este modelo, se emplean detectores de fallos que sospechan que un nodo ha caído, cuando éste tarda cierto tiempo en responder a los mensajes que se le envíen, o cuando tarda cierto tiempo en emitir mensajes que debería emitir con determinada cadencia. Como consecuencia, podrá ocurrir que un nodo correcto sea considerado como nodo caído, mientras que un nodo caído siempre será detectado. A estos detectores de fallos que pueden cometer errores en la detección de nodos caídos se les llama *detectores no fiables de fallos* [21].

Particiones

En un sistema distribuido, ocurre una partición cuando el conjunto de nodos que lo forma se descompone en varios subconjuntos disjuntos de nodos, y donde los nodos de cada subconjunto no pueden acceder a los nodos de los demás subconjuntos. Este problema puede aparecer por fallos de caída en la red de interconexión, o por la aparición de varios fallos de caída en los nodos del sistema.

El tratamiento de las particiones es problemático debido fundamentalmente a que un conjunto de nodos *M*, que no pueda acceder a los nodos de un conjunto *N*, nunca podrá saber si el sistema se ha dividido en 2 particiones, en más particiones, o si todos los nodos del conjunto *N* han caído. Debido a este problema, es imposible ejecutar con éxito ningún algoritmo que requiera consenso en un sistema donde puedan ocurrir particiones. Este problema, conocido como la *paradoja de los generales* [60], fue inicialmente descrito en el contexto de las bases de datos, pero es aplicable a cualquier sistema distribuido que requiera consenso. Cabe destacar que estos resultados de imposibilidad son distintos de la imposibilidad de resolver consenso de forma determinista en sistemas asíncronos donde al menos un nodo falle. En caso de particiones, no se puede alcanzar consenso ni siquiera en sistemas síncronos.

Para tratar las particiones se suelen construir protocolos de pertenencia a grupos que sólo permitan seguir funcionando a las particiones que contengan a más de la mitad de los nodos. Lo que se logra con esta técnica es enmascarar la aparición de las particiones, convirtiendo los sistemas particionables en sistemas donde se garantiza la existencia de una única partición, conocida como la *partición primaria*. Los nodos de las particiones minoritarias serán obligados a detenerse, para evitar que realicen acciones que pudieran generar inconsistencias respecto a la partición primaria.

Sin embargo, una de las cuestiones más importantes a analizar cuando se está estudiando la problemática de las particiones, consiste en razonar si tiene sentido que distintas particiones de nodos funcionen de forma independiente unas de otras, proporcionando servicio a sus clientes sin poder interactuar con el resto de las particiones para ello. Sabiendo que es imposible alcanzar consenso en un sistema particionado, sólo tendrá sentido admitir la existencia simultánea de más de una partición, en aquellas aplicaciones para las que no se exija consistencia completa de la información que proporcionan, o donde se permitan períodos de inconsistencia, y sea posible integrar la información de las distintas particiones una vez que las particiones desaparezcan.

1.3.2 Modelos de fallos

Al tratar de definir un modelo de fallos se debe tener precaución a la hora de atribuir a un determinado componente los fallos que puedan ocurrir. Por ejemplo, si consideramos un fallo consistente en la pérdida de un mensaje enviado por un canal de comunicaciones, podríamos atribuirlo tanto al canal, como a alguno de los nodos que intervienen en la comunicación. Si el fallo se le atribuye al canal, para enmascararlo podríamos intentar utilizar más de un canal, de tal forma que si un canal pierde un mensaje, otro canal lo pueda retransmitir. Sin embargo, también podríamos atribuir el fallo a alguno de los nodos intervinientes en la comunicación, que pueden haber sido incapaces o bien de generar el mensaje, o bien de recibirlo.

Existen muchos modelos de fallos, que consideran de forma combinada los fallos de los nodos y de los canales de comunicaciones. Schneider [126] presenta una clasificación de los modelos de fallos más comúnmente utilizados al diseñar sistemas, lo que no implica que no existan otros modelos. Cada modelo presenta los fallos que deberán considerarse como relevantes.

- *Parada:* Los nodos fallan parando. Una vez que un nodo se para, permanece en ese estado. El hecho de que un nodo haya fallado, lo pueden detectar los demás nodos.

- *Caída:* Los nodos fallan parando. Una vez que un nodo se para, permanece en ese estado. El hecho de que un nodo haya fallado, puede no ser detectable por los demás nodos.

- *Caída y enlace:* Los nodos fallan parando. Una vez que un nodo se para, permanece en ese estado. Un canal falla perdiendo algunos mensajes, pero ni los retrasa, ni los duplica, ni corrompe su contenido.

- *Omisión de recepciones:* Un nodo falla recibiendo sólo un subconjunto de los mensajes que se le envían, o parando y permaneciendo parado.

- *Omisión de envíos:* Un nodo falla transmitiendo tan sólo un subconjunto de los mensajes que realmente intenta enviar, o parándose y permaneciendo parado.

- *Omisión general:* Un nodo falla recibiendo un subconjunto de los mensajes que se le envían o transmitiendo un subconjunto de los mensajes que intenta enviar, o parando y permaneciendo parado.

- *Fallos bizantinos:* Un nodo falla mostrando un comportamiento arbitrario.

Los modelos de fallos expuestos forman una jerarquía desde los que consideran los fallos menos problemáticos, los fallos de parada, hasta los que consideran los más complejos de solucionar: los fallos bizantinos. Entre medio de estos dos modelos de fallos, existen gran variedad de modelos, entre los que Schneider tan sólo considera cinco. Los cuatro modelos de fallos previos a los fallos bizantinos, incluyen fallos en los canales de comunicación que llevan a la pérdida de mensajes, atribuyendo cada uno de ellos la causa a un componente distinto, mientras que el modelo de fallos de caída, como ya hemos comentado, es imposible de tratar en sistemas asíncronos, mientras que es equivalente al modelo de fallos de parada si se consideran sistemas síncronos.

1.3.3 Técnicas para tolerar fallos

En los apartados precedentes hemos tratado varias técnicas para enmascarar fallos, como los empleados por los protocolos de transporte fiable para los fallos transitorios en los canales de comunicación, las técnicas de versionado para enmascarar fallos de respuesta, omisión y temporización de los nodos o las técnicas implantadas por los protocolos de pertenencia a grupos para enmascarar las particiones. Respecto a los fallos de caída, hemos resaltado la imposibilidad de detectarlos en sistemas asíncronos, y su viabilidad en sistemas parcialmente síncronos.

Sin embargo, una vez detectada la caída de un nodo, todos los servicios que estuvieran ubicados en él, dejan de estar disponibles. Para solucionar esta indisponibilidad, la única técnica adecuada en general es la replicación. Por su especial relevancia dedicaremos el siguiente apartado a su estudio más detallado.

Además de las técnicas expuestas para enmascarar fallos, otra alternativa a evaluar a la hora de construir sistemas altamente disponibles, consiste en la construcción de sistemas y algoritmos que toleren fallos de forma intrínseca. Un caso típico de esta alternativa agrupa a los protocolos *autoestabilizantes*, descritos hace décadas por Dijkstra [31] y que han llevado al desarrollo de versiones autoestabilizantes de múltiples algoritmos distribuidos. Estos protocolos se diseñan de tal forma, que tras ocurrir un fallo, en un número finito k de pasos del algoritmo en el que no ocurran fallos, el algoritmo se vuelve a comportar como si el fallo nunca hubiera ocurrido. Siendo deseable la construcción de sistemas autoestabilizantes, el mayor problema que existe es la falta de algoritmos que se estabilicen en un número reducido de pasos y a bajo coste, quedando en general su aplicación a un campo habitualmente teórico.

1.4 Replicación

Colocando réplicas de un determinado servicio en distintos nodos, se puede conseguir que el servicio esté disponible aún ocurriendo fallos de caída. Esto será posible siempre que exista al menos un nodo con una réplica del servicio que sea capaz de atender las peticiones que se le hagan.

El hecho de que una réplica falle, plantea el problema de cómo garantizar que las demás réplicas tengan actualizado su estado, para ser capaces de responder a las peticiones de servicio como si el fallo no hubiera ocurrido. Para solucionar este problema, se han venido desarrollando protocolos de replicación que podemos clasificar en tres grandes modelos de replicación: *replicación activa*, *replicación pasiva* y *replicación coordinador-cohorte*. La diferencia fundamental entre ellos, radica en qué tareas realiza cada réplica y cómo se garantiza la consistencia en el estado de las distintas réplicas.

1.4.1 Replicación activa

En el modelo de replicación activa [126], las peticiones enviadas al servicio replicado, van dirigidas a todas y cada una de las réplicas. Cada réplica, al recibir la petición, realizará la misma tarea, cada una actualizará su estado de forma conveniente, y en la mayoría de casos contestará al que realizó la petición. Para asegurar que todas las réplicas reaccionan de igual forma a cada invocación realizada al servicio replicado, este modelo necesita asumir que el código ejecutado por las réplicas es determinista, y que las peticiones les llegan a todas en cierto orden. El orden impuesto debe impedir que a dos réplicas dadas les lleguen peticiones de actualización desordenadas, de tal forma que el estado quede inconsistente, o que les lleguen peticiones de consulta a las que retornen valores distintos.

Variantes La replicación por versiones y los bloques de recuperación distribuida que describimos en el apartado 1.3.1, son variantes de la replicación activa que no requieren de código determinista. Estas variantes, más que basarse en que cada réplica ejecute exactamente las mismas instrucciones, se basa en que cada réplica debe diseñarse para producir las mismas salidas frente a las mismas entradas, con lo cual requieren de orden en la entrega de mensajes, pero cada réplica habrá sido diseñada de forma independiente.

Ventajas Las principales ventajas de este modelo de replicación son dos: primero, proporcionan un modelo de programación muy sencillo al programador de servicios replicados y segundo, son muy eficientes en la recuperación del servicio frente a fallos. Programar servicios replicados con replicación activa es casi tan sencillo como programar servicios no replicados. Tan solo debe recordarse el hecho de que se debe programar código determinista, y prácticamente el resto del trabajo lo realizará la infraestructura de comunicación ordenada.

Cuando una réplica falle, no hará falta ejecutar ningún protocolo para asegurar que las demás réplicas tienen su estado actualizado, pues esto es algo que mantienen en todo momento; un *protocolo de comunicación a grupos* asegurará la entrega ordenada de los mensajes a todas las réplicas, de forma que todas ellas, en base a ejecutar el mismo código en el mismo orden, mantendrán en todo momento el mismo estado.

Desventajas Las principales desventajas del modelo activo son las siguientes: el elevado coste de los protocolos de comunicación a grupos necesarios para su implantación, el desaprovechamiento de recursos resultado de que todas las réplicas ejecuten el mismo código, el determinismo que se exige al código de los servicios, los problemas de anidamiento que aparecen cuando un servicio replicado accede a otro servicio replicado y por último, la falta de generalidad del modelo.

El coste de los protocolos de comunicación a grupos, de los que existen multitud de variantes, [72, 132, 95, 33], aparece tanto por los mensajes que deben enviarse entre los clientes del servicio y las réplicas para acordar un orden común en la entrega de mensajes, como en los retardos que deben forzarse en los mensajes enviados a las réplicas, que deberán retrasarse hasta que al menos la mayoría de las réplicas hayan acordado el orden en la entrega de los mensajes.

Por otra parte, el hecho de que todas las réplicas ejecutan su código cada vez que reciben una petición, resulta en que todo el coste que se derive de atender un servicio: coste computacional, espacial y los mensajes que se generen, será multiplicado por el número de réplicas que tenga el servicio, resultando sistemas donde debe tenerse precaución al decidir el número de réplicas del servicio, no sólo en base al número de fallos que se deseen tolerar, sino en base al coste en que se incurrirá para atender al servicio.

La restricción de código determinista, tiene un reflejo importante en varios aspectos relacionados con el diseño de los servicios. Primero, impide a los servicios programados para replicación activa hacer uso, en general, de dispositivos que proporcionen salidas no previsibles de antemano, como relojes, sensores, etc. y en general, fuerzan a tener especial precaución al invocar a los servicios del sistema operativo, que en su mayoría no son previsibles ni reproducibles en su totalidad. Y segundo, y no menos importante, fuerzan en general a evitar la multitarea en los servicios. Si se permitiera a dos peticiones proceder en paralelo dentro de los servicios, se estaría generando una fuente de indeterminismo, resultado de necesitar el uso de primitivas de sincronización entre las tareas, que posiblemente llevaría a que distintas réplicas evolucionaran de forma distinta.

El modelo activo exige que todas las réplicas reciban las peticiones efectuadas al servicio. Si se utiliza replicación activa para programar servicios que acceden a otros servicios replicados, se genera un anidamiento de las llamadas a los servicios replicados que resulta en un coste exponencial en cuanto a los mensajes que se deben generar. Si para solucionar este problema, se intentan implantar protocolos de elección de líder para que sólo una réplica realice las peticiones a otros servicios, realmente se estará adoptando el modelo de replicación coordinador-cohorte, que describiremos en esta misma sección.

El hecho de que cada réplica ejecute la acción asociada a la petición, hace que no se pueda utilizar este modelo para dotar de alta disponibilidad a servicios donde el trabajo a realizar deba realizarse exactamente una vez. Considérese por ejemplo el movimiento de cierto robot, el envío de cierto mensaje al exterior, la impresión de algún documento, etc. En estos casos, tenemos un servicio replicado para asegurar la realización de cierta acción sobre un dispositivo no replicado (al que posiblemente puedan acceder todas las réplicas por diferentes caminos) y cuya tolerancia a fallos esté garantizada por otros medios. Para soportar este tipo de servicios, deberá elegirse a una de las réplicas como la responsable final en realizar la acción, llevando a la adopción del modelo de replicación pasivo o del modelo coordinador-cohorte.

1.4.2 Replicación pasiva

El modelo de replicación pasiva [19] es un modelo asimétrico. En este modelo una de las réplicas detenta el rol de réplica primaria, mientras que las demás llevan el rol de réplicas secundarias. Las peticiones son procesadas únicamente por la réplica primaria, la cual enviará mensajes de actualización (*checkpoint*) a sus secundarias para mantener el estado común. Si falla una réplica secundaria debe hacerse poco trabajo, como mucho, actualizar la lista de réplicas en cada una de las réplicas que permanezcan activas. Sin embargo, en caso de fallos de la réplica primaria, debe ejecutarse un protocolo de recuperación que puede ser bastante costoso. Una de las réplicas secundarias será elegida como réplica primaria, y todos los agentes (los clientes y las réplicas), deberán actualizar la identidad de esta réplica como la primaria. Por último, deberá garantizarse que todas están de acuerdo en el estado común antes de admitir nuevas peticiones de servicio.

Variantes Hay fundamentalmente dos variaciones del modelo pasivo: *replicación pasiva en frío* (cold replication) y *replicación pasiva en caliente* (warm replication). La replicación en frío consiste en hacer que la réplica primaria almacene en un fichero de registro las actualizaciones que se efectúen sobre su estado, de forma que en caso de caídas, la réplica que sea elegida como nueva primaria, sea capaz de restaurar su estado utilizando la información del registro. La replicación en caliente por su parte, consiste en mantener a las réplicas secundarias permanentemente actualizadas, haciendo que la primaria les envíe mensajes periódicos de actualización o haciendo que la primaria envíe las actualizaciones a las secundarias justo antes de contestar a las peticiones que le realicen. Como puede apreciarse hay multitud de variantes que diferirán entre ellas por el grado de consistencia que sea necesario mantener entre las réplicas, el coste que se considere asumible debido a la gestión de la replicación en ausencia de fallos y el coste que sea tolerable asumir para recuperarse una vez se detecten los fallos.

Ventajas Sus principales ventajas respecto al modelo activo son un menor consumo de recursos y una mayor genericidad. Se trata de un modelo adecuado para cualquier tipo de servicio, tanto para los servicios replicables con el modelo activo como para servicios que realicen acciones de ejecución única o cuyo código no sea determinista. En lo referente al coste computacional, éste será en general menor ya que tan sólo una de las réplicas realizará la acción asociada al servicio. Este coste será menor lógicamente si el coste de procesar los mensajes de actualización consume menos recursos que servir las peticiones.

Desventajas Comparando este modelo de replicación con el modelo activo, el modelo pasivo incurre en mayores costes de recuperación ante fallos y ofrece un modelo de programación más complejo al programador de servicios replicados. Para desarrollar servicios replicados con el modelo de replicación pasivo, al menos en la variante en caliente, el programador debe desarrollar la versión del servicio primaria y la versión secundaria. La versión primaria recibirá peticiones y enviará mensajes de actualización al registro o a las réplicas secundarias y las versiones secundarias deberán encargarse de actualizar su estado cada vez que reciban un mensaje de actualización o de cargar su estado desde el registro.

1.4.3 Replicación coordinador-cohorte

El modelo coordinador-cohorte [14] es un modelo generalmente poco documentado, y que se encuentra a medio camino entre el modelo activo y el modelo pasivo. En este modelo aparecen primarios y secundarios, pero en lugar de tener una réplica primaria fija para el servicio, el rol de réplica primaria está asociado a la atención de cada petición. Cada petición será procesada por una de las réplicas que será conocida como la réplica coordinadora de la petición. Para esta misma petición, las demás réplicas actuarán como réplicas secundarias, recibiendo el nombre de réplicas cohortes de la petición.

Ventajas Ya que cada petición puede ser servida por una réplica distinta, en este modelo se podrán aplicar técnicas de balanceo de carga para mejorar el rendimiento global del sistema. La recuperación frente a fallos es más natural que la recuperación en el modelo pasivo ya que en el modelo coordinador-cohorte todas las réplicas están continuamente preparadas para atender peticiones. Es un modelo tan general como el modelo pasivo y usualmente los protocolos de comunicación necesarios para implementarlo son menos costosos que los protocolos utilizados para implantar el modelo activo.

Desventajas Sin embargo este modelo presenta algunas desventajas importantes. Las desventajas principales son: primero, respecto al modelo activo, el complicado modelo programación proporcionado a los programadores de servicios y segundo, respecto a los demás modelos, la necesidad de utilizar un mecanismo distribuido de control de concurrencia. El modelo de programación es más complejo debido a que todas las réplicas pueden estar sirviendo peticiones diferentes al mismo tiempo y pueden por tanto aparecer condiciones de carrera. También es necesario realizar un diseño adecuado, no sólo del código necesario para atender peticiones, sino de qué información enviar a las réplicas cohorte y en qué momentos hacerlo, para que estas puedan recuperar el servicio en caso de fallos. Adicionalmente hace falta un mecanismo de control de concurrencia que evite que peticiones diferentes procesadas por réplicas distintas alteren el estado compartido sin control. Sin este control, cada réplica mantendría un estado diferente con lo que no se tendría un estado único del servicio sino múltiples estados que serían posiblemente inconsistentes entre ellos.

1.4.4 Técnicas para implantar replicación

Existen varias técnicas que suelen utilizarse para implantar replicación. Algunas de ellas pueden servir para más de un modelo, mientras que otras son necesarias o exclusivas para poder implantar algún modelo específico. A continuación describimos las tres más representativas: *multienvíos*, *compromiso atómico* y *redundancia de discos*.

Multienvíos En prácticamente todos los modelos de replicación debe utilizarse alguna forma de multienvío de mensajes. Tanto en replicación activa, que necesita multienvíos ordenados como requisito imprescindible, como en replicación pasiva en caliente y replicación coordinador-cohorte, es necesario enviar un mensaje a más de un destinatario.

Existen multitud de posibilidades para efectuar multienvíos. Desde un multienvío donde no exista ninguna garantía de fiabilidad, ni orden en la entrega de los mensajes, hasta multienvíos que garanticen atomicidad, orden total y orden causal en las entregas. Las diferentes variantes las podemos agrupar en tres categorías: no-fiables, fiables y fiables-ordenados. Los fiables garantizan al menos las tres propiedades siguientes:

- *Validez:* Si un nodo que no falla envía un mensaje, todos los destinatarios que no fallen, lo entregarán.
- *Acuerdo:* Si un nodo que no falla entrega un mensaje, todos los demás destinatarios del mensaje eventualmente lo entregarán.
- *Integridad:* Para todo mensaje *m*, los destinatarios que no fallen lo entregarán como máximo una vez, y sólo lo entregarán si el mensaje fue enviado previamente por el emisor de *m*.

Los no-fiables, no garantizan alguna de ellas, y los fiables-ordenados, además de garantizar las tres propiedades, garantizan que la entrega se realizará en cada destinatario respetando cierto orden. Los tipos de ordenación más frecuentes son FIFO, Causal, Total, FIFO+Total y Causal+Total. Tal y como describen Hadzilacos y Toueg [65], cada una de estas ordenaciones impone un coste adicional respecto a la ordenación anterior y a partir de la ordenación total, se requiere de una red no asíncrona para poder tolerar fallos.

Para implantar replicación activa, se utilizan protocolos de multienvío que garanticen al menos orden total, y en la mayoría de casos orden causal y total. Por su parte, para implantar replicación pasiva en caliente y replicación coordinador cohorte, suele bastar con multienvíos fiables y a lo sumo multienvíos con orden FIFO.

Compromiso atómico En todos los modelos de replicación, las acciones que debe ejecutar el servicio pueden pertenecer a una transacción a la que se le exijan las características ACID[6] [61] de las transacciones. En particular, para lograr que un servicio sea recuperable frente a caídas de todas las réplicas, es necesario utilizar transacciones que garanticen persistencia en el estado. Para ello se deben utilizar protocolos de compromiso atómico tolerantes a fallos, habitualmente conocidos como protocolos de compromiso en tres fases o 3PC [4].

Aun sin requerir persistencia, los modelos pasivo y coordinador-cohorte necesitan que se garanticen las características ACID de las peticiones que efectúen los clientes del servicio en presencia de fallos. La *durabilidad* vendrá determinada por el número de réplicas del servicio, ya que en caso de no utilizar persistencia, será violada al caer todas las réplicas. El *aislamiento* lo debe garantizar el control de concurrencia, local en el modelo pasivo y distribuido en el modelo coordinador cohorte. Para garantizar *atomicidad* se deberá utilizar algún protocolo de compromiso atómico que ejecutarán las réplicas y quizás el cliente. Respecto a la *consistencia*, podrá optarse por consistencia total o alguna forma de consistencia más relajada, como por ejemplo las sugeridas por Ladin *et al.* [76] para servicios que no demanden consistencia fuerte. Adicionalmente, si se requiere anidamiento en las llamadas, deberán implantarse protocolo similares a las transacciones anidadas detalladas por Moss [98], más o menos relajadas en función de las necesidades de aislamiento y consistencia.

[6] Atomicidad, consistencia, aislamiento y durabilidad (**A***tomicity,* **C***onsistency,* **I***solation,* **D***urability*).

Redundancia de discos Siempre que se requiera persistencia en los datos para poder recuperar el servicio frente a fallos de todas las réplicas, o para implantar el modelo pasivo en frío, es necesario disponer de almacenamiento estable de la información que tolere fallos. Para ello, suele utilizarse también replicación, pero en este caso con técnicas especializadas para la construcción de discos redundantes, que pueden basarse tanto en técnicas software como en aspectos hardware.

Posibles soluciones incluyen a los discos situados en máquinas distintas, donde las operaciones implicarán el uso transacciones, o en discos multipuerto. Estos últimos permiten conectar diversos nodos a un mismo disco a través de interfaces distintas. Con ello se permite el acceso al disco a distintos nodos que pueden fallar de forma independiente. Para aumentar la disponibilidad del propio disco, se suelen utilizar múltiples discos o redundancia de pistas, que en este caso serán escritos simultáneamente ante cada petición de escritura, utilizando para ello, fundamentalmente mecanismos hardware.

1.4.5 Elección de un modelo

Las razones para elegir un modelo frente a los demás ha generado controversias desde hace años [23, 16]. Las controversias vienen causadas por el coste que implica utilizar protocolos de comunicación a grupos con orden total y causal, que han derivado generalmente en cuestionar la eficiencia del modelo de replicación activa.

Todos los modelos son útiles en sus propios escenarios, y tal y como sugiere Birman [16], uno de los máximos defensores de los protocolos de comunicación ordenados y por extensión de la replicación activa, en respuesta a las críticas de Cheriton y Skeen a tales protocolos [23], la replicación activa es útil en multitud de escenarios, pero sobre todo para dotar de alta disponibilidad a aplicaciones que ya estén construidas y que sean autocontenidas, o donde se demande simplicidad y elegancia en el modelo de programación, o donde las réplicas tengan interfaz con los usuarios. Sin embargo, quizás no sean tan adecuadas para proporcionar tolerancia a fallos si hay necesidad de persistencia de datos, en entornos basados en llamadas a procedimiento remoto (RPC) o si es cuestionable el determinismo de las réplicas.

Respecto al modelo coordinador-cohorte, pocas críticas o comentarios ha recibido desde su aparición [13]. Sin embargo muchas de las soluciones basadas en replicación activa, acaban por implementar el modelo coordinador-cohorte para solucionar el problema de las invocaciones anidadas [63], con lo que de forma prácticamente desapercibida se está adoptando en gran medida el modelo coordinador-cohorte en sustitución del modelo activo.

Entre el modelo pasivo y el modelo coordinador-cohorte hay menos diferencias, y la elección de uno frente a otro suele deberse a pequeños factores más que a motivos de trascendencia. Ambos son igual de genéricos, y ambos requieren un protocolo para reconfigurarse frente a fallos. El modelo pasivo es más sencillo de implantar que el coordinador-cohorte, tanto por parte de los programadores de servicios replicados, como por parte del soporte necesario para su gestión. Por contra, el modelo coordinador-cohorte presenta menos variabilidad en el tiempo de reconfiguración frente a caídas, y pueden aplicarse técnicas de balanceo de carga considerando cada invocación por separado. Por otra parte, el modelo pasivo puede verse como una particularización del modelo coordinador-cohorte donde no puede variar la réplica primaria en cada invocación y donde no es necesario por tanto control de concurrencia distribuido. Por tanto,

sin más que eliminando esta variabilidad del primario y eliminando el control de concurrencia distribuido al modelo coordinador-cohorte, se obtiene el modelo pasivo sin coste adicional. Por ello, si es posible asumir la complejidad del modelo de programación que impone el modelo coordinador-cohorte y si se dispone de un mecanismo adecuado de control de concurrencia, puede resultar más ventajoso el modelo coordinador-cohorte que el pasivo, mientras que el pasivo será más adecuado si no se dispone de control de concurrencia distribuido, si el balanceo de carga no es esencial, o si la complejidad del modelo de programación no es asumible.

1.5 Sistemas de objetos distribuidos

Para construir un sistema distribuido hay que elegir el modelo de programación que se pretende proporcionar a los desarrolladores del sistema. En principio lo deseable al construir una aplicación distribuida sería abstraerse lo máximo posible de la problemática específica de la distribución, para centrarse lo máximo posible en la implantación de la funcionalidad que se desee. Por tanto sería deseable poder desarrollar los sistemas distribuidos de la misma forma y con las mismas herramientas de diseño que las que se emplean para sistemas centralizados:

> *"Los resultados*[7] *continúan sugiriendo que tanto los desarrolladores que utilizan técnicas orientas a objeto como los que no las utilizan, aprecian que el desarrollo orientado a objetos es superior. Que los desarrolladores que utilizan técnicas orientadas a objeto apoyan esta percepción con más intensidad que los que no las utilizan y que todos ellos consideran que las desventajas que se les suelen atribuir a las técnicas orientadas a objeto son virtualmente inexistentes." [71]*

Desde hace unos años, y fundamentalmente desde la aparición del estándar CORBA [111, 113], se está adoptando la programación orientada a objetos como la más aceptada para el desarrollo de sistemas distribuidos. La mayoría de los sistemas distribuidos en la actualidad y fundamentalmente la mayoría de las infraestructuras de comunicaciones para sistemas distribuidos se están desarrollando para permitir el uso de objetos como elemento básico en la construcción de sistemas distribuidos.

Los ejemplos de estos sistemas son innumerables, yendo desde implementaciones del estándar CORBA como Visibroker [134], Orbix [70] o Java IDL, a infraestructuras de comunicaciones para sistemas operativos de uso general como DCOM [35] para Windows, o infraestructuras de comunicaciones propias de determinados entornos, como RMI [68] para Java.

1.5.1 Alta disponibilidad en sistemas de objetos distribuidos

Al estructurar el software de un sistema distribuido altamente disponible hay que elegir la granularidad con la que se pretende replicar los servicios. En los sistemas podemos encontrar básicamente dos tendencias: replicar procesos y replicar objetos.

Para replicar un servicio mediante replicación de procesos, se implementa el servicio como un proceso. El proceso es capaz de recibir peticiones y de responder mediante mensajes. Como

[7] Se refiere al estudio realizado en el que se analiza la opinión de multitud de profesionales del diseño y programación de sistemas informáticos.

proceso, dispone de su propia memoria y de uno o más hilos de ejecución. Cada proceso se coloca en un nodo distinto y un nivel de transporte orientado a procesos, envía las peticiones que efectúan los clientes del servicio a un proceso o a más de uno en función del modelo de replicación. Hay varios sistemas que optan por ofrecer replicación de procesos, tanto con replicación pasiva [18], como con replicación activa [132, 11] o replicación coordinador-cohorte [14].

La desventaja fundamental de esta aproximación es su elevada granularidad, que conduce a diseños poco flexibles y difíciles de mantener, siendo su principal ventaja, la mayor simplicidad en el desarrollo del soporte, que en la mayoría de casos se ha limitado a un protocolo de comunicación a grupos que proporcione orden total o total y causal en los mensajes.

Por su parte, al replicar un servicio mediante objetos, se implementa la funcionalidad del servicio como un objeto y se colocan las réplicas del objeto en diferentes nodos. Un cliente del servicio realizará una invocación al objeto replicado utilizando en este caso un *gestor de invocaciones a objetos* (ORB) que le independice de la ubicación de los objetos e incluso de la condición del objeto de ser replicado. En este caso, los objetos estarán formando parte de un dominio de protección (un proceso o el propio núcleo del sistema operativo) y en general serán entidades pasivas que no dispondrán de hilos de ejecución propios. Será el propio ORB de su nodo el que los invocará cuando algún ORB remoto inicie una invocación hacia el objeto por cuenta de algún cliente.

La principal desventaja de la replicación de objetos es la complejidad del soporte a los objetos replicados, que puede constatarse al comprobar que no existen demasiados sistemas que lo proporcionen, o que el propio estándar CORBA creado en 1990, no haya adoptado hasta el momento de redactar este trabajo el servicio de replicación (ahora llamado servicio de tolerancia a fallos [114]) o que este propio futuro estándar en su redacción actual no incluya soporte al modelo de replicación coordinador-cohorte.

Como nexo de unión entre ambos tipos de sistemas cabe citar los esfuerzos reflejados en [62], donde se estudian las diferencias entre ambos modelos, que dificultan transvasar con naturalidad los protocolos de replicación ya existentes en el modelo de procesos a sistemas distribuidos orientados a objeto. La dificultad radica en que los soportes de replicación orientados a procesos rara vez han considerado en toda su extensión la problemática que surge al replicar procesos que a su vez necesiten utilizar otros procesos replicados para poder realizar su trabajo. Esta problemática aparece con claridad si se estructura el servicio con objetos y se pretende dotar al sistema con transparencia de ubicación y transparencia de replicación[8], donde cada objeto puede a priori invocar a cualquier otro objeto del sistema esté o no replicado. En este escenario se pueden llegar a formar cadenas de invocación arbitrariamente largas que incluso contengan ciclos. Los fallos que pueden ocurrir en estas situaciones son complejos de solucionar y pocos sistemas los afrontan de forma completa.

En todo caso, la tendencia actual en sistemas distribuidos altamente disponibles va encaminada hacia la construcción de sistemas distribuidos con soporte a objetos. Además de los sistemas diseñados para soportar objetos desde un principio como Arjuna [115], Solaris-MC [9, 73], Spring [93, 66], o el propio estándar CORBA con su reciente especificación del servicio de tolerancia a fallos [114], la mayoría de sistemas originalmente construidos orientados a

[8]Los usuarios del objeto replicado no aprecian diferencias en cuanto a su uso, respecto al uso que harían de objetos no replicados.

procesos, como Isis [11], Horus [132], Transis [33], Amoeba [72] o Totem [96], han ido evolucionando para soportar objetos. Así han aparecido entornos como Electra [86], que construye un ORB sobre Horus o Isis, Garf [64] y Orbix+Isis que proporcionan objetos utilizando Isis como biblioteca de comunicación a grupos, o Eternal [97], que se basa en Totem para proporcionar objetos replicados, entre otros.

1.6 Recolección de residuos

La recolección de residuos agrupa a un conjunto de técnicas dedicadas a liberar los recursos consumidos por ciertas componentes que forman parte de las aplicaciones una vez que se detecta que dichas componentes ya no serán utilizadas por la aplicación.

Podemos encontrar recolección de residuos en aplicaciones de uso general, sistemas operativos, bases de datos, en soportes en tiempo de ejecución de determinados lenguajes de programación como Smalltalk o Java [2], o incluso en sistemas distribuidos; casos de Emerald, RMI [68] o Solaris-MC [73] por citar algunos.

Disponer de un sistema de recolección de residuos libera al programador de la tarea de llevar control sobre qué entidades liberar y en qué momento hacerlo, y le permite centrarse en el desarrollo de la funcionalidad de su aplicación. La principal desventaja que se ha atribuido a los sistemas de recolección de residuos es el elevado coste espacial y computacional en que incurren. Por otra parte, la mayoría de sistemas que potencialmente podrían incluirlo suelen descartarlo con frecuencia por su inherente complejidad. Por ejemplo podemos comprobar como CORBA pidió en 1997 [111] propuestas para incluir alguna forma de recolección de residuos en su estándar, y sin embargo no se ha llegado a ninguna conclusión definitiva hasta la fecha.

Si está fuera de toda duda su utilidad y la aceptación de los programadores cuando disponen de tal soporte para el desarrollo de aplicaciones centralizadas, más relevancia tiene si cabe cuando se trata de aplicaciones distribuidas. En el caso de un sistema distribuido que no tenga integrada la recolección de residuos, se podrán dar dos opciones, o cada componente de software que se cree es cuidadosamente analizada y liberada en su justo momento por su programador, o irán apareciendo entidades en los diferentes nodos, que una vez que no sean accesibles desde los nodos remotos, se limitarán a consumir recursos, siendo su existencia simplemente inútil. El hecho de que las componentes que se crean puedan ser utilizadas de forma remota hace que sea prácticamente imposible su liberación por parte del programador si el sistema no le proporciona soporte, con lo que en general un sistema distribuido sin recolección de residuos irá poco a poco consumiendo cada vez más recursos.

Si hablamos de sistemas distribuidos altamente disponibles, el caso se agrava aun más, pues estos sistemas se diseñan para estar funcionando sin interrupción alguna durante largos períodos de tiempo y en el mejor de los casos para no detenerse nunca. Si no se dispone entonces de un sistema de recolección de residuos, puede incluso llegarse al caso extremo en que todo el sistema se detenga por no disponer de los recursos necesarios para seguir funcionando.

1.6.1 Técnicas de recolección de residuos

En todo sistema de recolección de residuos encontramos dos fases: *detección* y *reclamación*. En la fase de detección se dividen los objetos[9] entre objetos útiles y residuos, y en la fase de reclamación, se libera el espacio consumido por los objetos inútiles. En la práctica, estas dos fases pueden funcionar de forma entrelazada, siendo la fase de detección la que suele determinar la calidad del sistema. La fase de reclamación es dependiente de detalles propios de cada arquitectura íntimamente relacionados con la gestión de memoria del sistema operativo o del soporte en tiempo de ejecución que se utilice, y su estudio queda fuera del ámbito de este trabajo. Puede consultarse [137] para una visión más detallada de las técnicas habituales de reclamación de residuos.

La corrección en la detección de residuos se basa en dos criterios: *seguridad* y *viveza*. La viveza del sistema debe garantizar que todo residuo será eventualmente detectado y la seguridad que sólo serán considerados como residuos los objetos que lo sean. Que un objeto sea residuo o no, se suele determinar partiendo de un conjunto de *raíces* de objetos y de una *relación de alcanzabilidad*. Un objeto será residuo si y solo si no es alcanzable desde ningún objeto raíz.

Las principales técnicas de recolección de residuos se pueden agrupar en dos categorías: *cuenta de referencias* y *trazas de referencias* (reference tracing).

Cuenta de referencias

Este técnica fue propuesta en los años 60 por Collins [25] como una técnica para liberar recursos en programas en Lisp y poco a poco fueron apareciendo variaciones de esta técnica para otros lenguajes y para otros entornos. La cuenta de referencias se basa en llevar la cuenta en cada objeto de cuántas referencias le apuntan, es decir, de cuántos usuarios potenciales del objeto existen en cada instante. Cuando la cuenta de referencias llega a cero, el objeto es un residuo.

La principal desventaja de los sistemas de cuenta de referencias, consiste en que no son capaces de detectar ciclos de residuos. Un ciclo de residuos aparece por ejemplo cuando un objeto O apunta a un objeto P, el objeto P apunta al objeto O, y no hay ninguna referencia más que apunte a P ni a O. En estos casos O y P son residuos pues nadie los puede utilizar pero sus contadores no serán cero, por lo que no serán considerados como residuos.

Una de las principales ventajas de estos algoritmos, consiste en que su coste computacional se distribuye de forma constante entre las operaciones que utilizan las referencias. Existe un coste fijo asociado a la cuenta de referencias que aparecerá al crear, al duplicar y al eliminar cada referencia y no existe ningún otro coste computacional adicional que sea necesario para la detección de los residuos. Esta característica hace que estos algoritmos sean idóneos para sistemas donde sea conveniente evitar períodos en los que se aprecie una degradación del rendimiento del sistema por efecto de la recolección de residuos. Este es el caso por ejemplo de los entornos de tiempo real y de los sistemas operativos, donde se han venido utilizando desde sus orígenes, técnicas de cuenta de referencias para gestionar el uso que los procesos hacen de los recursos del sistema.

[9] En este caso utilizamos el término objeto de forma general para referirnos a una entidad software identificable y no necesariamente a un objeto en el sentido estricto utilizado en diseño y programación orientada a objetos.

Trazas de referencias

Para detectar ciclos de residuos se utilizan técnicas basadas en trazas de referencias, entre las que destacan las técnicas de marcado y barrido (mark and sweep) [91, 32]. En estas técnicas se aprecian de forma muy clara las dos fases que se realiza en toda recolección de residuos: la detección o marcado y la recolección o barrido.

Los algoritmos de marcado, marcan inicialmente a todos los objetos como potencialmente residuales. Después seleccionan un objeto raíz o un conjunto de objetos raíces y estos objetos iniciales son marcados como no residuales. Desde estos objetos se siguen las referencias contenidas en ellos y se marca a dichos objetos referenciados como no residuales. Este proceso de marcado se sigue hasta que no se puedan alcanzar nuevos objetos utilizando para ello algoritmos de recorrido de árboles en profundidad o amplitud. Una vez finaliza el recorrido, se marcan los objetos que aún permanezcan como potencialmente residuales, como residuos. La siguiente ejecución del marcado ya no considerará a los objetos marcados como residuos para el marcado inicial.

De forma posiblemente desligada al algoritmo de marcado se ejecuta un algoritmo de barrido que recolecta todos los objetos marcados como residuos. Esta desligazón entre marcado y barrido permite generalmente a los algoritmos diseñados para sistemas centralizados, optimizar la liberación de recursos. Liberando los recursos en un algoritmo que se dedica en exclusiva a ello, se puedan agrupar estos recursos para optimizar su posterior reutilización, empleando por ejemplo técnicas de compactación.

Las ventajas de los sistemas de trazas de referencias son varias: primero, detectan residuos cíclicos. Segundo, es fácil ajustar la periodicidad de sus ejecuciones para detectar y recolectar los residuos con la rapidez que se desee. Se podrán detectar y recolectar de forma prácticamente inmediata a su generación, incrementando con ello el coste de la detección, o se podrá retrasar su ejecución, hasta que por ejemplo se detecte una notoria carencia de recursos en el sistema. Y finalmente, tal y como comentaremos en el apartado siguiente, se trata de algoritmos inherentemente tolerantes a fallos.

La desventaja fundamental de estos algoritmos es su elevado coste computacional, que está en relación directa con el número de objetos del sistema. Adicionalmente, durante la ejecución tanto de la fase de marcado como de la fase de barrido puede que llegue a apreciarse de forma significativa una disminución en el rendimiento del sistema. Esta característica provoca que estos algoritmos sean inadecuados para entornos de tiempo real o para entornos donde no sean deseable una acusada variabilidad del rendimiento. Una tercera desventaja, que aunque sutil, tienen estos algoritmos se desprende del hecho de que asumen que los objetos contienen entre otras cosas, referencias a objeto. Esta apreciación puede no ser cierta en determinados entornos orientados a objeto, como el caso de CORBA, donde los objetos residen en procesos y los procesos contienen tanto objetos como referencias a objeto, pero donde no existe una asignación directa de qué referencias a objeto pertenecen estrictamente hablando a cada objeto.

1.6.2 Recolección de residuos en sistemas distribuidos

Para detectar residuos en sistemas distribuidos se utilizan las mismas técnicas que para detectarlos en sistemas centralizados, pero en este caso con importantes adaptaciones y variaciones

para tener en cuenta las características propias de la distribución: grado de sincronía, fallos y coste derivado del envío de mensajes.

En los sistemas distribuidos podemos tener un objeto en cierto nodo, mientras que las referencias pueden estar en otros nodos. Cualquier operación que partiendo de una referencia deba alcanzar a su objeto o viceversa, se transformará en uno o más mensajes que podrán sufrir retrasos, que podrán perderse, duplicarse o corromperse y de igual forma, los nodos involucrados podrán fallar en cualquier momento.

La adaptación de la cuenta de referencias a sistemas distribuidos ha dado lugar a la aparición de numerosas variaciones que en su mayoría intentan evitar la aparición de condiciones de carrera debidas al desorden en que pueden entregarse los mensajes. Ejemplos de estas variaciones son la cuenta ponderada de referencias (weighted reference counting) [10, 135] y la cuenta de referencias indirecta [117], sin embargo ninguna de ellas logra tolerar fallos de caída en los nodos. Otra variación ha dado lugar a la aparición de un grupo distinto de algoritmos: los basados en listas de referencias [17, 128]. Estos algoritmos en lugar de mantener en cada objeto un simple contador, mantienen la lista completa de ubicaciones donde se encuentran las referencias. Las listas de referencias se utilizan para tolerar fallos más severos en los mensajes, tales como pérdidas y duplicidades y para en cierta medida tolerar caídas de los nodos. La contrapartida la encontramos en su mayor coste tanto espacial como de utilización de mensajes.

Por su parte las adaptaciones de los algoritmos de trazas de referencias a sistemas distribuidos [69, 75, 78], son más naturales en concepción. Todos ellos parten de un conjunto de objetos raíces, y van marcando objetos alcanzables hasta que no se encuentren más objetos. En este caso el coste ya de por sí elevado del marcado es más notorio, pues cada seguimiento de una referencia puede involucrar el acceso a un nodo remoto y los algoritmos de marcado deben alcanzar a todos los objetos del sistema distribuido. Cuando ocurren fallos de caída ejecutando estos algoritmos, la aproximación suele ser volver a comenzar la fase de marcado, pues se trata de algoritmos idempotentes: cada fase de marcado puede realizarse sin utilizar el estado resultante de marcados previos.

Existen multitud de variaciones y combinaciones de los algoritmos que hemos comentado, otras técnicas de propósito menos general y algunas adaptaciones a situaciones particulares. En el capítulo 4, donde presentamos nuestro algoritmo de recolección de residuos, ampliaremos su estudio dando una visión mas detallada de cada algoritmo comparándolo con nuestro trabajo.

1.7 Objetivos, estructura y contribuciones de esta tesis

Una vez comentados los aspectos más relevantes de los sistemas distribuidos altamente disponibles y en particular de los sistemas de soporte a objetos distribuidos con recolección de residuos, estamos en disposición de describir nuestro trabajo. Primero, comentamos los objetivos que nos marcamos, después resumimos la estructura del resto de la memoria y para finalizar este capítulo introductorio resumimos las contribuciones más importantes que realizamos.

1.7.1 Objetivos

El objetivo de esta tesis, compartido con la tesis *Hidra: invocaciones fiables y control de concurrencia* realizada por F.D. Muñoz Escoí [108], es diseñar Hidra: una arquitectura de soporte a objetos distribuidos que sirva de núcleo para el desarrollo de sistemas y aplicaciones altamente disponibles.

En esta tesis describiremos la arquitectura completa y nos centraremos en la capa central de Hidra: el gestor de invocaciones a objeto. La tesis de F.D. Muñoz [108], complementa nuestro trabajo describiendo el protocolo de pertenencia a grupos que utiliza Hidra y el diseño del soporte al modelo de replicación coordinador-cohorte.

Los objetivos que nos planteamos con Hidra los comentamos a continuación:

- Diseñar una arquitectura que sirva para el desarrollo de aplicaciones y sistemas altamente disponibles.
- Lograr que la arquitectura pueda utilizarse como soporte al desarrollo de clusters, para lo que deberá diseñarse teniendo en cuenta la diferencia entre soporte a nivel del sistema operativo y soporte a nivel de procesos de usuario.
- Primar la eficiencia.
- Proporcionar un modelo de programación sencillo y potente.
- Integrar la recolección de residuos como parte fundamental de la propia arquitectura.

Para ello decidimos que el modelo de programación a ofrecer fuera el de objetos distribuidos. Esto nos llevó a integrar un gestor a invocaciones a objeto en la arquitectura, cuyos objetivos fueran los siguientes:

- Lograr el objetivo común a todo ORB: permitir la programación orientada a objetos con transparencia de ubicación.
- Dar soporte a objetos replicados. Preferiblemente con el modelo de replicación coordinador-cohorte, pero permitiendo por el propio diseño del ORB la inclusión de la forma más sencilla posible del soporte a otros modelos de replicación.
- Proporcionar recolección de residuos de forma poco intrusiva y al menor coste posible para todos los objetos, tanto replicados como no replicados. El estado interno del ORB que sea necesario mantener para cada objeto y para cada referencia a objeto que se encuentre en cada nodo deberá desaparecer cuando el objeto o la referencia en cuestión ya no existan. Y por otra parte, los objetos serán avisados cuando sean inalcanzables, es decir, cuando no hayan referencias a ese objeto en todo el sistema distribuido.
- Tolerar fallos de caída de los nodos, retrasos y desorden en los mensajes. En ORB no deberá hacer asunciones en cuanto al orden en que se transmiten los mensajes, ni en cuanto al tiempo que tardan en entregarse. Por su parte, ante cualquier fallo de caída, el estado del ORB en cada nodo deberá adaptarse para reflejar la ausencia del nodo caído. Esto incluye a la recolección de residuos y al soporte a los objetos replicados que deberán reconfigurarse en caso de que la caída haya afectado a alguna réplica.

1.7.2 Estructura

En el capítulo 2 se describe la arquitectura Hidra. Para ello se describe cada nivel que forma parte de la arquitectura, los servicios que ofrece a los niveles superiores y los servicios que utiliza de los niveles inferiores, y en su caso, el tipo de fallos que se enmascara en dicho nivel y en qué tipo de fallos se convierten los fallos enmascarados. El capítulo incluye una comparativa de nuestra arquitectura con otros sistemas de alta disponibilidad con soporte a objetos.

El capítulo 3 está dedicado al gestor de invocaciones. Describimos sus componentes, la función de cada componente y las comparamos en la medida de lo posible con CORBA y con otros mecanismos distribuidos de soporte a objetos. Seguidamente, nos centramos en el soporte a objetos replicados. Se describe el diseño del soporte a objetos replicados como una extensión del soporte a objetos de implementación única, y detallamos los protocolos que se ejecutan para su gestión. Los más destacados son los protocolos para añadir y eliminar réplicas y los protocolos de recuperación de los objetos replicados frente a fallos de caída.

En el capítulo 4 se describe el sistema de recolección de residuos tanto de objetos no replicados, como de objetos replicados. Demostramos su corrección y lo comparamos con varios sistemas de recolección de residuos, en términos de coste, tolerancia a fallos y adecuación para objetos móviles.

El penúltimo capítulo lo dedicamos a la depuración de Hidra. Después que de hayamos descrito varios algoritmos de Hidra y hayamos demostrado uno de ellos, describimos un mecanismo de depuración que nos facilitará la tarea de encontrar posibles errores de implementación. Con este sistema, cerramos el ciclo de desarrollo que se afronta en cualquier entorno complejo: diseño, demostración, implementación y depuración. El mecanismo de depuración está integrado en la propia arquitectura Hidra y nos servirá tanto para inyectar y simular fallos, como para encontrar errores de implementación, como para estudiar el rendimiento del sistema.

Finalmente, en el capítulo 6 se resumen las conclusiones que hemos obtenido durante el desarrollo de este trabajo, comentamos la implementación que hemos realizado de la arquitectura y discutimos las posibles líneas de trabajo futuras que quedan abiertas a raíz de nuestro trabajo.

1.7.3 Contribuciones

Las aportación fundamental de esta tesis consiste en un diseño novedoso del soporte a objetos replicados para un gestor de invocaciones a objeto. El soporte incluye protocolos de gestión de los objetos replicados y un sistema de recolección de residuos. El sistema lo hemos implementado en su gran mayoría y hemos diseñado un sistema de depuración integrado en la propia arquitectura. El mecanismo de depuración nos permite ejercitar el soporte a objetos, inyectar fallos y detectar posibles errores de implementación.

Los objetos replicados podrán seguir diferentes modelos de replicación, el estado de las réplicas podrá mantenerse consistente según sea necesario y los objetos replicados que sean inútiles serán detectados como residuos. Todo ello tolerando fallos de caída en los nodos, asumiendo un modelo de sistema asíncrono con notificación fiable de fallos e incurriendo en costes reducidos.

Esta contribución global, la podemos dividir en cuatro aspectos relevantes:

Referencias a objeto: La estructura de las referencias a objeto de Hidra y de toda su maqui-

naria interna tienen un diseño distinto al que podemos encontrar en la mayoría de los sistemas de objetos distribuidos. Casi ninguna arquitecturas de soporte a objetos está descrita con el nivel de detalle y con la separación de funciones como la arquitectura que presentaremos en el capítulo 3.

Recolección de residuos: En el capítulo 4 describimos un protocolo distribuido de cuenta de referencias nuevo. La aportación del protocolo respecto a otros existentes es su mayor asincronía sin incurrir en mayor coste. Para dotar al protocolo de cuenta de referencias de tolerancia a fallos describimos un protocolo de marcado y barrido que sin ser novedoso en sí mismo, sí sale beneficiado de la estructuración de las referencias que planteamos.

Recolección de residuos para objetos replicados: Hasta la fecha, no sabemos de ningún protocolo de recolección de residuos diseñado específicamente para objetos replicados. En el capítulo 4 describimos nuestro protocolo distribuido de cuenta de referencias adaptado a objetos replicados. Es sencillo y de coste comparable al protocolo para objetos no replicados. La aproximación que toman la mayoría de sistemas que contemplan replicación de objetos, de construir objetos replicados como agregaciones de objetos no replicados, implica que una referencia a un objeto replicado contiene tantas referencias a objeto como réplicas tenga el objeto. Los sistemas que utilicen esta aproximación, si integraran recolección de residuos basada en cuenta de referencias, comprobarían que el coste asociado a las operaciones que se realizan sobre las referencias: creación, duplicación y destrucción, sería proporcional al número de réplicas del objeto. En nuestra aproximación el coste es constante y no depende del número de réplicas del objeto.

Modelo de ejecución de todo ORB: En el capítulo 5 describimos un mecanismo que hemos diseñado e implementado para depurar una arquitectura como Hidra. El aspecto más novedoso de nuestro mecanismo consiste en la descripción del modelo de ejecución que encontramos en Hidra y que entendemos aparece en todo sistema de soporte a objetos distribuidos.

Capítulo 2

La arquitectura Hidra

Está demostrado que un sistema complejo que funciona, se ha creado siempre a partir de un sistema simple que funciona.

Primera ley de la Sistemática.

Un sistema complejo diseñado partiendo de cero, no funciona nunca y no se puede modificar para que funcione. Hay que volver a empezar, partiendo de un sistema sencillo que funcione.

Segunda ley de la Sistemática.

Contenidos del capítulo

Hidra [49, 48, 46, 47] es una arquitectura diseñada para servir de soporte al desarrollo de servicios altamente disponibles en un cluster.

Un cluster Hidra (ver figura 2.1) estará formado por un conjunto de nodos interconectados con una red. Cada nodo ejecutará su propio núcleo de sistema operativo que ofrecerá unos servicios básicos a Hidra, e Hidra proporcionará a su vez servicios de alta disponibilidad a las aplicaciones que se construyan sobre ella. Las aplicaciones podrán residir en procesos de usuario o en el propio sistema operativo. Esta última posibilidad es la que permitirá construir servicios propios del sistema operativo de forma altamente disponible, con el objetivo final de construir un sistema operativo distribuido con imagen de sistema único.

Los servicios que necesita Hidra del núcleo del sistema operativo son los típicos ofrecidos por la mayoría de núcleos actuales [100]. NanOS [101] es un ejemplo válido de núcleo sobre el que implantar Hidra y sobre el que ya se han realizado algunos trabajos [105] de integración con Hidra. También se podrían utilizar sistemas operativos convencionales como Linux, Solaris, etc. En cualquier caso, los servicios mínimos que debe ofrecer el núcleo a Hidra son: el concepto de tarea, dominio de protección (o proceso), mecanismos de llamada al sistema (downcall) y de invocación desde el sistema operativo hacia los procesos (upcall) [24], mecanismo de sincronización entre tareas, gestión de memoria, y primitivas de envío y recepción de mensajes por la red (transporte no fiable). Además es conveniente que se disponga de un mecanismo para incluir módulos en el propio núcleo, ya que Hidra en sí misma formará parte del núcleo para mejorar en eficiencia y aumentar la seguridad y estabilidad del sistema.

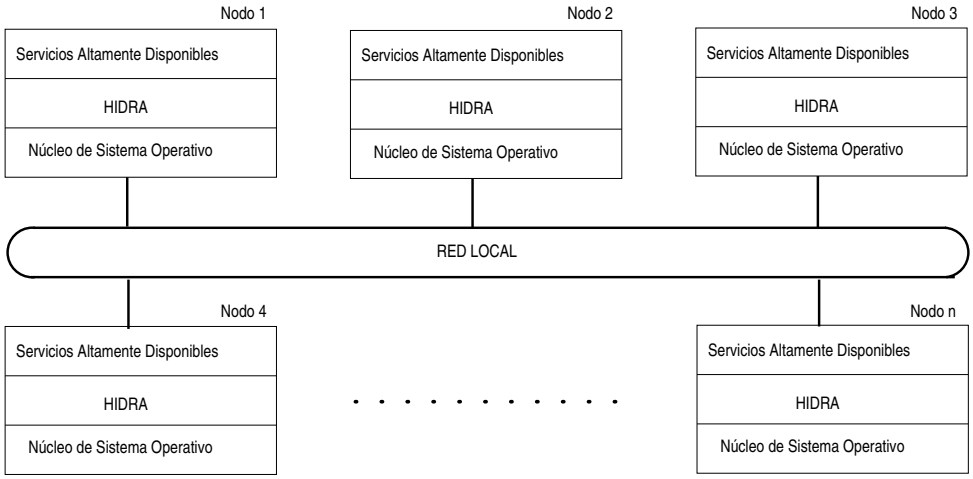

Figura 2.1: Un cluster Hidra

2.1 Principios de diseño

Para abordar la complejidad del diseño de Hidra, hacemos uso de la descomposición del sistema en niveles y componentes, de forma que cada uno de ellos proporcione cierta abstracción a los niveles superiores, y que estos se construyan en base a la funcionalidad ofrecida por los niveles inferiores.

Hacemos uso del principio de diseño de expuesto por Saltzer, Reed y Clark en los argumentos *extremo a extremo* (End-to-End arguments) [124], donde defienden que diseñar sistemas donde las capas más bajas de la arquitectura proporcionen excesiva funcionalidad, conduce a sistemas donde finalmente se ve penalizada la eficiencia. Según este principio, se debe tener especial cuidado en no sobrecargar aquellos niveles que sean utilizados por niveles superiores, si alguno de los niveles superiores realmente no necesita tanta funcionalidad. Los niveles inferiores tampoco debes proporcionar excesiva funcionalidad a los niveles superiores si éstos últimos están obligados a volver a implementarla aunque sea en parte, o si son capaces de implementarla con menor coste.

El hecho de pretender que Hidra sirva de soporte para el desarrollo de clusters, hace que ante cualquier decisión de diseño en la que aparezca un compromiso entre eficiencia y otros aspectos tales como simplicidad, estandarización, interoperabilidad, etc., tendamos a adoptar aquella aproximación, que siendo razonable en los demás aspectos, prime la eficiencia.

Por último, en la medida de lo posible, siempre que no se comprometa la eficiencia, utilizaremos patrones de diseño conocidos [56] para disminuir la complejidad, clarificar el diseño y dotar al sistema de formas previamente estudiadas de interacción entre componentes.

Nos decantamos por el modelo orientado a objetos, no sólo como modelo a ofrecer a las aplicaciones que se construyan sobre Hidra, sino también como metodología de diseño e implementación de toda la arquitectura. Las interfaces relevantes de las componentes las mostraremos en Java por conveniencia, ya que éste es el lenguaje con el que hemos implementado el primer prototipo.

2.2 El modelo Hidra

El modelo Hidra engloba al modelo de sistema distribuido subyacente a la arquitectura, el modelo de replicación que se adopta en Hidra y el modelo de objetos que se proporciona a las aplicaciones que se construyan sobre Hidra.

2.2.1 Modelo del sistema

Hidra se construye sobre un conjunto de ordenadores corrientes interconectados con una red local no fiable que modelizamos como un sistema distribuido parcialmente síncrono, donde los mensajes pueden perderse, duplicarse, sufrir retrasos y llegar desordenados, donde los nodos pueden caer, fallar en el envío o recepción de mensajes y ralentizarse de forma arbitraria y donde adicionalmente la red de interconexión puede sufrir particiones. La única asunción que hacemos sobre la máquina extendida subyacente es que no existen errores bizantinos, ni aparecen mensajes no enviados de forma arbitraria. Por tanto el modelo de fallos del que partimos es el modelo de *omisión general* enunciado en el apartado 1.3.2, extendido con la aparición de particiones.

Sobre este modelo de fallos, la arquitectura Hidra, en base a cada uno de sus niveles y componentes, irá proporcionando a los niveles superiores mayores garantías, para concluir ofreciendo garantías de alta disponibilidad a los servicios que se construyan sobre Hidra.

La asunción de disponer de un sistema parcialmente síncrono es totalmente razonable y al mismo tiempo necesaria. Razonable, ya que las redes de área local tienen en la práctica una cota temporal en la entrega de mensajes y los ordenadores disponibles hoy en día tienen relojes bastante precisos. Puede que la cota para la entrega de mensajes no se conozca con precisión, pero haremos asunciones pesimistas en cuanto a su valor, la obtendremos por experimentación y permitiremos que sea configurable. No asumiremos que los relojes de los distintos nodos estén sincronizados entre ellos, pero sí asumiremos que tienen cadencias similares. Cuando un nodo falle en el cumplimiento de alguna de estas dos hipótesis, permitiremos que el nodo sea considerado como caído. En estos casos, el propio nodo será forzado a detenerse.

La segunda asunción que hacemos, de no existir comportamientos arbitrarios ni errores bizantinos también resulta razonable en un cluster. En los clusters, tanto la ubicación del hardware que suele estar físicamente instalado en lugares relativamente seguros, como el contexto habitualmente cerrado en el que se suelen utilizar, hacen que la frecuencia de aparición de este tipo de errores sea prácticamente despreciable. En todo caso, esta hipótesis nos centra en el desarrollo de sistemas altamente disponibles, y no en el desarrollo de sistemas ultra-altamente disponibles ni de disponibilidad total.

2.2.2 Modelo de replicación

Habiendo evaluado los distintos modelos de replicación, en Hidra nos decantamos por el modelo de replicación coordinador-cohorte. También podíamos haber considerado el modelo pasivo, por presentar características similares al coordinador-cohorte. Preferimos este último por ser una generalización del modelo pasivo, con lo cual si soportamos el modelo coordinador-cohorte, logramos disponer del modelo pasivo si así lo deseáramos sin incrementar por ello el coste.

El modelo activo lo descartamos como modelo de uso general, pero hemos diseñado la arquitectura para facilitar su inclusión en el futuro si así lo consideráramos conveniente. En todo caso, sí proporcionamos soporte para un tipo muy simplificado de objetos replicados de forma activa: objetos replicados sin consistencia en su estado, lo que no nos forzará a utilizar protocolos de comunicación ordenada. Como ya comentaremos en el apartado 2.3.6, este tipo de objetos serán utilizados para simplificar la terminación de determinados protocolos necesarios para implantar el modelo coordinador-cohorte. La idea es similar a la que podemos encontrar en [4], donde para implantar un protocolo de compromiso atómico no bloqueante, se implanta el protocolo de compromiso atómico en dos fases y posteriormente se hace uso de un multi-envío fiable para informar a todos los participantes del protocolo acerca de su terminación. Nosotros realizaremos algo similar utilizando para ello un objeto replicado de forma activa, al que invocaremos en determinadas circunstancias, informando con ello acerca de la terminación de alguno de nuestros protocolos. Como puede observarse este tipo de objetos no requerirán estado, tan sólo la habilidad de ser invocados.

Consideramos que el tipo de programador que se enfrenta al desarrollo de aplicaciones altamente disponibles en un cluster, es personal cualificado, consciente de la necesidad de primar la eficiencia, por lo que no consideramos la simplicidad del modelo de programación ofrecida por el modelo activo, como condición suficiente para su adopción, si por ello la eficiencia o la genericidad se ven comprometidas.

Uno de los objetivos de Hidra es permitir el desarrollo de componentes del sistema operati-

vo. La mayoría de acciones a ejecutar dentro del sistema operativo no son deterministas, y gran parte de las acciones dependerán de las condiciones en las que se encuentre el hardware o el núcleo del sistema operativo subyacente en cada instante, por lo que no podemos asumir como realista el determinismo que implica el modelo activo. Multitud de componentes que podrán ser desarrolladas sobre la arquitectura exigirán soporte a altos grados de concurrencia, utilizando para ello múltiples tareas y primitivas de sincronización entre tareas, por lo que de nuevo podemos ver comprometido el determinismo de los servicios. Sin embargo, otros servicios de Hidra, como por ejemplo el servicio de nombres, si que se podrían implementar con facilidad y sin notables incrementos del coste mediante replicación activa. Por ello, pese a descartar el modelo activo como modelo general, sí que nos planteamos incluir soporte opcional en el futuro.

Hidra no ofrece soporte para persistencia de los objetos. Esto implica que los servicios altamente disponibles que deseen prevenir la caída de todas las réplicas, deberán utilizar mecanismos externos a Hidra para salvar su estado en almacenamiento estable y restaurarlo cuando sea apropiado. Sin embargo, el hecho de adoptar un modelo no activo, permitirá a los desarrolladores de este tipo de servicios elegir la aproximación que consideren conveniente como almacenamiento estable. Haber elegido el modelo activo de forma estricta, hubiera obligado a disponer de tantos discos independientes como réplicas tengan los servicios, y en general hubiera forzado a utilizar el disco de cada nodo para salvar el estado de las réplicas ubicadas en él. Utilizando un modelo no-activo, permitimos tanto el uso de esa aproximación, como el uso de discos multipuerto, en los que tan sólo se realizan las operaciones de lectura/escritura una vez.

2.2.3 Modelo de objetos

El modelo de objetos que ofrece Hidra es el modelo de objetos CORBA [113] ampliado para soportar objetos replicados y recolección de residuos. Es decir, del modelo CORBA adoptamos la posibilidad de que los objetos se implementen en cualquier lenguaje de programación, las interfaces de los objetos se deberán definir en IDL, y un compilador generará los esqueletos y los proxies adecuados. Los objetos residen en dominios de protección o simplemente dominios[1] y el ORB no tiene control acerca de cómo implementa cada dominio sus objetos ni de cómo ni dónde almacena sus referencias a objeto. En particular los dominios contienen objetos y referencias a objeto, pero no se puede asumir que los objetos contengan referencias a objeto ni que las referencias a objeto estén contenidas en objetos. Podrían incluso programarse los objetos con lenguajes no orientados a objeto.

Los objetos pueden ser replicados o de implementación única. La decisión de si un objeto es replicado o no, debe tomarse en tiempo de ejecución al registrar a los objetos en Hidra. Los objetos replicados pueden ser objetos replicados con replicación coordinador-cohorte o con replicación activa. Los objetos replicados mediante el modelo coordinador-cohorte pueden crearse sin control de consistencia en el estado de las réplicas (sin control de concurrencia) o con control de concurrencia distribuido. Los objetos replicados mediante replicación activa no tienen garantías de consistencia en su estado, ni se garantiza que todas las réplicas reciban las invocaciones efectuadas sobre el objeto replicado en caso de caídas del invocador. Tan sólo

[1] En este trabajo utilizamos el término dominio de protección o dominio para referirnos de forma genérica tanto a procesos de usuario como al propio sistema operativo. Por contra, utilizaremos el término proceso cuando queramos explícitamente excluir al dominio de protección formado por el sistema operativo.

garantizan que las invocaciones efectuadas sobre el objeto replicado serán entregadas en orden FIFO a las réplicas.

Los objetos de implementación única y los objetos replicados, independientemente del modelo de replicación y de las garantías de consistencia que se demanden pueden tener soporte a recolección de residuos. La decisión de si un objeto requiere este soporte o no, se toma al igual que en el caso de la replicación, al registrar el objeto en Hidra. Si un objeto se registra en Hidra demandando soporte a recolección de residuos, Hidra invocará al objeto (upcall) cuando el objeto sea un residuo. Es decir, cuando no queden referencias en todo el cluster que apunten al objeto. Esta invocación, a la que denominaremos *notificación de no referenciado* llegará a la implementación del objeto si éste es de implementación única, y llegará a todas las implementaciones de las réplicas si el objeto es replicado.

Las notificaciones son totalmente asíncronas, lo que significa que puede transcurrir un tiempo arbitrariamente largo desde que el objeto sea un residuo hasta que Hidra notifique de este hecho al objeto. De igual forma, para el caso de objetos replicados, puede transcurrir un tiempo no acotado entre el instante en que una réplica reciba la notificación y el instante en que lo reciban las demás.

Al recibir la notificación de no referenciado, el objeto podrá ejecutar código para realizar la acción que desee. Por ejemplo, podría liberar los recursos que estuviera consumiendo y destruirse a sí mismo, podría previamente salvar su estado en almacenamiento estable, podría volver a registrarse en Hidra y enviar una referencia de sí mismo a algún otro dominio, etc. Haga la acción que sea, tendrá la seguridad de que la notificación le ha llegado porque no existen referencias válidas que le apunten y que por tanto es un residuo. Evidentemente será un residuo a nivel distribuido, lo que no implica que Hidra pueda eliminarlo por sí mismo. Según el modelo de objetos de Hidra, la decisión de si debe eliminarse el objeto o de cómo hacerlo la debe tomar el dominio de protección al recibir la notificación, pues los dominios son libres en cuanto a la forma de implementar los objetos. Por ejemplo, sería perfectamente posible que un dominio de protección recibiera la notificación de no referenciado para alguno de sus objetos y que decidiera no eliminarlo, por estar él mismo haciendo uso localmente del objeto mediante punteros privados externos al control de Hidra.

2.3 Niveles y componentes de la arquitectura

Tal y como puede observarse en la figura 2.2, la arquitectura Hidra está formada por varios niveles y componentes. Cada nivel utiliza los servicios de los niveles inferiores y ofrece servicios a los niveles superiores, enmascarando en varios de ellos la aparición de determinados tipos de fallos. A continuación describimos para cada uno de los niveles, su diseño, funcionalidad y en su caso, el modelo de sistema que asumen y el que proporcionan a los demás componentes.

2.3.1 Transporte no fiable

El nivel de transporte no fiable, permite enviar mensajes a cualquier nodo del cluster Hidra, utilizando para ello su identificador de nodo. En este nivel, el modelo del sistema corresponde al de un sistema parcialmente síncrono, particionable y con modelo de fallos de omisión general.

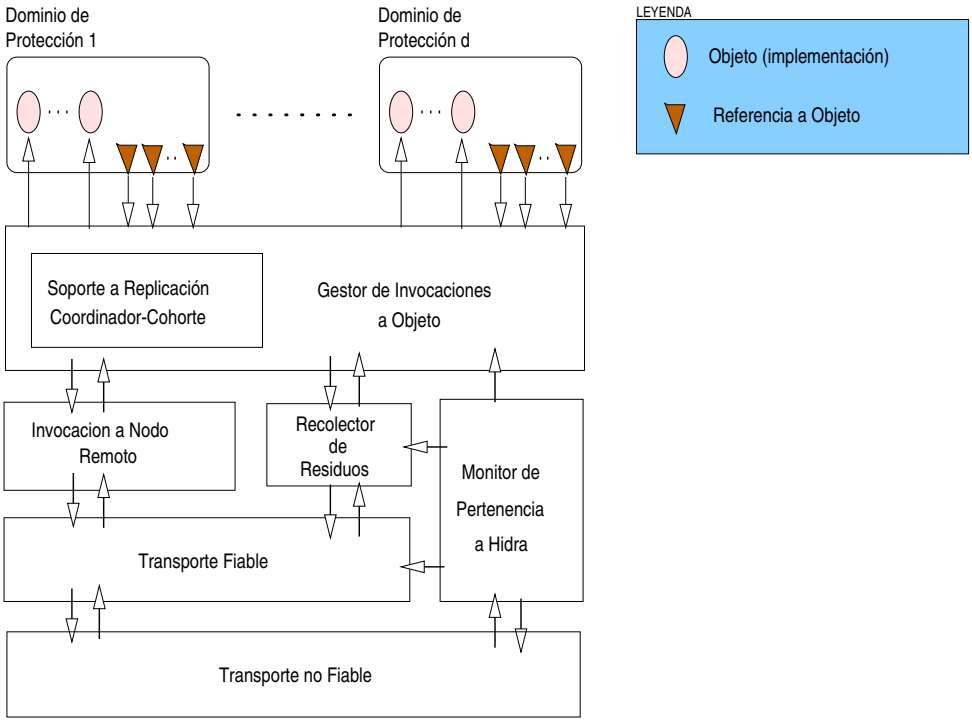

Figura 2.2: La arquitectura Hidra

Estrictamente hablando este nivel no forma parte de Hidra, sino que es, junto con el soporte del núcleo del sistema operativo, una componente necesaria para poder construir la arquitectura.

La interfaz ofrecida por este nivel, que encapsulará por tanto al servicio de transporte de que disponga el sistema operativo la podemos observar en la figura 2.3.

```
public interface UnreliableTransport
{
    public void initialize (Properties transportProperties);
    public void start();
    public void send (Node destinationNode, Message message);
    ...
}
```

Figura 2.3: Interfaz del Transporte no Fiable

El método `initialize()` configura el transporte con información dependiente del protocolo de comunicaciones empleado. El propósito de esta configuración es asociar los identificadores de nodo que emplearán los niveles superiores con las direcciones de los nodos que debe utilizar el protocolo de transporte. Por ejemplo en el prototipo actual de Hidra, implementado haciendo uso de *UDP*, se asocia cada identificador de nodo con el par <nombre de máquina, puerto UDP>.

Con el método `send()` se inicia el envío de un mensaje sin garantías de fiabilidad hacia el nodo `destinationNode`. Por su parte el método `start()` inicia la tarea de recepción de mensajes. Cada vez que se recibe un mensaje, se examina el tipo del mensaje que se encuentra codificado en su cabecera y se entrega el mensaje a la componente que se haya registrado como manejador de este tipo de mensajes. Las dos componentes que se registrarán como manejadores de mensajes no fiables, serán el nivel de transporte fiable y el monitor de pertenencia a Hidra.

2.3.2 Monitor de pertenencia a Hidra

Sobre el nivel de transporte no fiable se apoya un protocolo de pertenencia a grupos [26] ampliado con servicios adicionales. Esta componente constituye el monitor de pertenencia a Hidra (HMM) [107]. Este monitor mantiene qué nodos forman parte del cluster Hidra, permitiendo que simultáneamente se produzcan nuevas incorporaciones al cluster y detectando qué nodos han caído. Ante cada reconfiguración del cluster (caídas o adiciones de nodos) el HMM genera un número de reconfiguración del cluster que será utilizado por los niveles superiores para enmascarar ciertos tipos de fallos. De igual forma, para proporcionar el modelo de fallo de parada [125], cada vez que un nodo se une al cluster, se le proporciona un número de encarnación nuevo, de forma que aunque un mismo nodo se una al cluster y caiga de forma repetida, será considerado cada vez que lo haga como un nodo distinto.

La detección de caídas se realiza asumiendo que la red de comunicaciones es parcialmente síncrona. Es decir, el protocolo de pertenencia está continuamente enviando mensajes de latido (heartbeat) y escuchándolos, sospechando de la caída de un nodo cuando éste deje de enviar sus mensajes de latido durante cierto período de tiempo. Para evitar retrasos en estos mensajes de latido y para ajustar lo máximo posible el tiempo de espera máximo a dichos mensajes, una configuración óptima de Hidra utilizaría una interfaz de red específica para los mensajes del HMM. El uso de esta interfaz dedicada, que no es obligatorio en Hidra, minimizaría los retrasos que en la práctica pueden sufrir los mensajes en un entorno tan cerrado como un cluster. Con el mismo objetivo, la tarea que se encarga de ejecutar el HMM tiene la máxima prioridad, evitando de esta forma la mayoría de los vencimientos del tiempo de espera debidos a la propia carga del nodo.

En relación con las particiones, el HMM adopta el modelo de partición primaria. Para ello, se proporciona un peso a cada nodo del cluster y sólo se permite progresar a la partición cuyos nodos sumen más de la mitad del peso total del cluster. Cuando el HMM detecta una o varias caídas y comprueba que el nodo no forma parte de la partición mayoritaria, él mismo fuerza la parada del nodo, que podrá reintegrarse de nuevo cuando sea reiniciado. De esta forma, decimos que el HMM enmascara la aparición de particiones y proporciona un modelo de sistema en el que no ocurren particiones.

El HMM además de mantener de forma distribuida en todo momento el conjunto de nodos que pertenecen al cluster, permite ejecutar de forma virtualmente síncrona una serie de pasos a todos los nodos en cada reconfiguración del cluster. Muchos sistemas utilizan el concepto de sincronía virtual [15, 12] en el contexto de tolerancia a fallos. Su utilización más frecuente aparece en sistemas donde se construyen protocolos de entrega de mensajes con orden total o causal. En este tipo de entornos, la notificación de caída de un nodo o la adhesión de un nodo, se integra con el resto de mensajes, de forma que también sean recibidas por todos los nodos

en el mismo orden. Gracias al concepto de sincronía virtual, sus aplicaciones pueden construirse como si el sistema distribuido [asíncrono] subyacente fuera un sistema síncrono. Nuestra aproximación, a diferencia de estas, únicamente utiliza cierta forma de sincronía en cada reconfiguración del cluster (caídas de nodos y/o adiciones de nodos), no para entregar mensajes, sino para permitirle a cada nodo ejecutar pasos de reconfiguración de forma sincronizada al resto de nodos. Durante la ejecución normal del cluster, la sobrecarga inherente a este tipo de protocolos es evitada.

```
public interface MembershipMonitor
{
    public Membership joinCluster ();
    public void leaveCluster ();
    public Membership getMembership();
    public void registerObserver (MembershipObserver observer,
                                  int stepNumber,
                                  int millisTimeout);
    ...
}
```

Figura 2.4: Interfaz del Monitor de Pertenencia a Hidra

En la figura 2.4 mostramos la interfaz del monitor de pertenencia a Hidra. Una vez que se haya inicializado la componente, y todavía en la inicialización de Hidra, se invoca al método `joinCluster()`. Al invocarse esta operación, el monitor comienza a ejecutar un protocolo para unirse al cluster que estuviera previamente formado. Si el monitor no encuentra a ningún nodo, se constituirá un cluster formado únicamente por el nodo local. El método retorna los nodos que forman parte del cluster junto con sus números de encarnación y el número de reconfiguración actual del cluster. Por su parte, el método `leaveCluster()` puede ser invocado para abandonar de forma voluntaria el cluster, permitiendo de esta forma al resto del cluster reconfigurarse de forma más rápida a como se reconfiguraría en caso de una caída del nodo. El método `getMembership()` lo utiliza el resto de Hidra para averiguar la identidad de los nodos que forman el cluster en cualquier momento y finalmente `registerObserver()` se emplea para registrar observadores interesados en actuar en las reconfiguraciones del cluster.

Cada componente de Hidra que desee realizar alguna acción para reconfigurar su estado de forma sincronizada con el resto de nodos, registra un objeto de tipo `MembershipObserver` en el HMM. En el registro se indica qué número de paso de reconfiguración es el que se está registrando y el tiempo máximo que dedicará a ejecutar su código. Por su parte, el HMM de cada nodo ejecutará los pasos de reconfiguración respetando el orden impuesto por el número de reconfiguración, de forma que todos los nodos completarán las acciones registradas para el paso i antes de que alguno de ellos comience a ejecutar las acciones del paso $i + 1$. Si algún paso de reconfiguración tarda más tiempo en completarse del que fuera especificado con el argumento `millisTimeout` en su registro, el HMM considerará que el nodo ha caído y reiniciará una nueva reconfiguración. Esta estrategia permite asegurar que todo el cluster estará reconfigurado y dispuesto a volver a ejecutarse con normalidad consumiendo un tiempo acotado a priori. En la práctica rara vez se encontrará el caso de un nodo Hidra que no complete su trabajo en el tiempo

indicado, puesto que el código a ejecutar en los pasos de reconfiguración se diseña y se prueba para ello. Sin embargo es una medida adicional que permite eliminar del cluster a los nodos que por el motivo que sea estén funcionado de forma incorrecta. Una configuración típica de un cluster Hidra no deberá requerir mucho más de un segundo en reconfigurarse completamente con lo que las garantías de alta disponibilidad del cluster quedarían satisfechas.

La mayoría de componentes de Hidra son clientes del HMM y para reaccionar ante reconfiguraciones, registran objetos de tipo `MembershipObserver`. Para simplificar la exposición, diremos que un componente ejecuta cierta acción ante caídas de nodos o ante reconfiguraciones, para indicar que esa misma componente u otra que esté coordinada con la componente que estemos describiendo, registra un objeto de tipo `MembershipObserver` en el HMM, que al ser invocado, ejecuta la acción deseada.

Por último, resaltar que gracias al HMM, el modelo de sistema que se ofrece a los niveles superiores tiene ya mayores garantías. En particular, los fallos de temporización de los nodos y de los canales de comunicación que impidan la entrega de mensajes dentro del límite impuesto por el HMM serán convertidos en fallos de caída. Lo mismo ocurrirá con los fallos de omisión de envío y de recepción de mensajes. Ante la aparición de estos fallos, alguno de los nodos involucrados en la comunicación fallida será considerado como caído y por tanto excluido del cluster.

Lógicamente, el detector de fallos del HMM es no fiable [21], pero en cambio, gracias a él se dispone en los niveles superiores de un modelo de sistema parcialmente síncrono o asíncrono, sin particiones y con detector de fallos perfecto. De hecho, el resto de componentes verán que un nodo cae cuando este hecho sea decidido por el HMM, sin importarles si esta decisión es perfecta o no. En el resto de esta memoria y salvo que estemos interesados en algún tipo de decisión específica del HMM acerca de los nodos, diremos que un nodo ha *caído* o que ha *fallado* cuando el HMM lo excluya del cluster, y en caso contrario diremos que el nodo *no ha caído* o que está *vivo* o que se trata de un *nodo del cluster*.

2.3.3 Transporte fiable

Este nivel, que se construye sobre el nivel de transporte no fiable y sobre el monitor de pertenencia a Hidra, elimina duplicidades en los mensajes y evita las pérdidas de mensajes de forma similar a como lo haría cualquier protocolo de transporte orientado a conexión (p.e. TCP [121]). Con ello, los errores transitorios más frecuentes de los canales de comunicación son eliminados.

Adicionalmente, apoyándose en el HMM, este transporte proporciona un nivel de fiabilidad adicional: Los mensajes que un nodo *N* envía a un nodo *M*, son reintentados hasta que la entrega sea confirmada por el destinatario o hasta que el destinatario o el origen fallen. Conviene destacar que no se impone ningún tiempo de espera máximo, con lo que teóricamente los mensajes pueden reintentarse durante tiempo ilimitado, convirtiéndose con ello el sistema hacia niveles superiores en asíncrono.

Estas garantías de entrega fuertes, permiten al resto de niveles de Hidra confiar en que los mensajes siempre llegarán, y de no llegar, los niveles interesados podrán ejecutar acciones de recuperación de forma sincronizada al resto del cluster.

El nivel de transporte fiable de cada nodo incluye en todos los mensajes que envía el identificador del nodo local origen del mensaje, el número de encarnación del nodo y el número

de reconfiguración del cluster, cuyos valores habrá obtenido de su HMM. El número de encarnación del nodo, permite al nivel de transporte fiable filtrar y descartar cualquier mensaje que provenga de un nodo con un número de encarnación distinto al que el propio nodo conozca para dicho nodo. Esta técnica nos permite enmascarar los fallos de caída convirtiéndolos en fallos de parada que generalmente son más sencillos de tratar.

Considérese por ejemplo el caso de un nodo que emite un mensaje y posteriormente falla. Si el mensaje llega al destino antes de que sea detectada la caída del nodo no aparece ningún efecto indeseable pues se estaría reflejando la realidad. Sin embargo, si se detecta la caída antes de que llegue el mensaje, el mensaje se debe descartar: al detectar la caída del nodo, los nodos vivos del cluster habrán tenido la oportunidad de reconfigurarse para reflejar este hecho o estarán progresando en la reconfiguración y de admitirse la entrega de este mensaje con posterioridad a la reconfiguración o durante ella, se podría comprometer la consistencia de algún protocolo del cluster que ya puede estar funcionando asumiendo que el nodo caído no participa. También sería si cabe más dañino permitir la entrega de este mensaje si el nodo caído se vuelve a unir al cluster rápidamente. En este caso es necesario descartar los mensajes originados por el nodo caído durante la reconfiguración en la que cayó pero admitir los mensajes que genere con una nueva encarnación.

En principio los demás mensajes originados por aquellos nodos que no hayan caído y que se intenten entregar después de iniciada una reconfiguración, pero que no hayan sido emitidos por ninguno de los pasos de reconfiguración, deberán entregarse con normalidad a sus destinatarios. Sin embargo, hay diversos protocolos en Hidra que necesitan conocer este hecho porque asumen que después de cada reconfiguración del cluster se parte de un estado inicial que no debe verse alterado por mensajes originados en reconfiguraciones previas. Por ello se incluye el número de reconfiguración del cluster en los mensajes, que no será utilizado por el nivel de transporte para filtrar mensajes, sino por los niveles superiores para habilitar o impedir las acciones asociadas con la entrega de los mensajes.

Con el tratamiento que hemos expuesto, el nivel de transporte fiable de Hidra proporciona un modelo de sistema asíncrono donde los nodos pueden sufrir fallos de parada, y donde los mensajes pueden desordenarse, pero cuya entrega tendrá garantías fuertes. Adicionalmente, gracias a la incorporación de los números de reconfiguración del cluster, habilita la construcción de protocolos que sólo utilicen mensajes de cada reconfiguración del cluster, es decir, que asuman que durante su ejecución no aparecerán fallos de caída ni incorporaciones de nodos. Estos protocolos son los que necesitarán la ejecución de los pasos de reconfiguración que ofrece el HMM para inicializar su estado global al estado inicial del protocolo, sabiendo que los mensajes originados en reconfiguraciones previas podrán ser descartados.

De la interfaz de este nivel (ver figura 2.5), destacan los métodos `send()` y `multicast()`. Ambos permiten enviar mensajes con garantías fuertes de entrega: El método para realizar multienvíos, a diferencia de lo que podemos encontrar en la mayoría de arquitecturas para alta disponibilidad no implementa ningún tipo de orden en la entrega de los mensajes, ni tan siquiera entrega atómica[2]. Tan sólo garantiza la entrega de los mensajes con la misma semántica que resultaría de invocar a `send()` tantas veces como nodos haya en el destino del

[2]Nótese que utilizamos el término *atómico* como sinónimo de fiable, y *fiable* con el sentido expuesto en el apartado 1.4.4 de la página 15 y no con el sentido que se emplea en varios de trabajos donde el término *atómico* se utiliza como sinónimo de totalmente ordenado.

```
public interface ReliableTransport
{
    public void send (Node destination, Message msg);
    public void multicast (Node [] destination, Message msg);
    ...
}
```

Figura 2.5: Interfaz del Transporte Fiable

multienvío. Es decir, en caso de que falle el emisor de un mensaje dado, puede que algunos de los destinatarios entreguen el mensaje y que otros no lo hagan. Este tipo de multienvíos es útil en Hidra en dos contextos: para enviar mensajes de tipo *checkpoint* a un conjunto de réplicas para actualizar su estado y también para construir un tipo muy específico de objetos replicados con replicación activa pero sin consistencia en su estado. En ninguno de los dos casos se demanda entrega atómica, ya que en ambos casos, en caso de caída del emisor del mensaje se tomarán acciones correctoras en pasos síncronos de reconfiguración dirigidas por el HMM.

Por último, la tarea principal que realiza el transporte fiable para reaccionar ante cambios en el cluster, será precisamente actualizar sus datos internos para filtrar los mensajes provenientes de nodos que hubieran caído y poder incorporar la información actualizada en los mensajes salientes: tanto reconfiguración del cluster como encarnación del nodo local.

2.3.4 Gestor de invocaciones a nodo remoto

Por encima del nivel de transporte fiable, una fina capa implementa un nivel de llamada a nodo remoto sencillo. No llega a ser llamada a procedimiento remoto [136] pues este nivel no se encarga de empaquetar ni de desempaquetar argumentos. Tan sólo se encarga de enviar un mensaje a un destino, bloqueando a la tarea invocadora hasta que llegue la respuesta.

```
public interface RemoteNodeCallBroker
{
    public IncomingReply invoke (Invocation msg) throws DestinationDown;
    public void multiInvoke (Invocation msg) throws DestinationDown;
    ...
}
```

Figura 2.6: Interfaz del Gestor de Invocaciones a Nodo Remoto

La misión de invocar se realiza con las operaciones `invoke()` y `multiInvoke()`. El destino de la invocación está codificado en la propia invocación, y puede extraerse mediante la operación `getDestination()` del objeto `Invocation`. Este método retorna un objeto de tipo `Destination` que contiene el destino de la invocación. El destino puede ser múltiple

para el caso de objetos replicados y será simple para el caso de objetos de implementación única.

El método `invoke()` se utiliza tanto para invocaciones que se efectúen sobre objetos de implementación única, como sobre objetos replicados de forma pasiva o según el modelo coordinador-cohorte. Si el objeto destino no está replicado y éste cae antes de responder a la invocación, el método `invoke()` retornará una excepción de tipo `DestinationDown`. Por su parte, si el destino de la invocación es replicado, la invocación será reintentada de forma transparente al usuario de este nivel sobre otra réplica, retornando en este caso la excepción `DestinationDown` sólo si han caído todas las réplicas. El objeto `Destination` es el responsable de implantar la lógica de la elección del nuevo destino tanto para el modelo pasivo como para el modelo coordinador-cohorte permitiendo de esta forma los reintentos de las invocaciones.

Por su parte el método `multiInvoke()` es equivalente a realizar simultáneamente tantas invocaciones ordinarias sobre destinos no replicados como destinos estén codificados en la invocación. Este método se utiliza para implantar una forma de replicación activa bastante limitada, en la que no se garantiza orden causal, ni orden total, ni entrega atómica, ni se esperan datos de respuesta. La diferencia fundamental entre `multiInvoke()` y la operación `multicast()` del nivel de transporte fiable, es que los destinos de `multiInvoke()` están especificados como ubicaciones de objetos y no como números de nodo. Gracias a esta diferencia `multiInvoke()` proporciona orden FIFO en la entrega de los mensajes que vayan dirigidos a las réplicas de un mismo objeto. Esta funcionalidad es útil para el caso de los `checkpoints` que se envían a los objetos replicados para que las réplicas actualicen su estado de forma ordenada. Gracias al orden FIFO[3] , cada réplica recibirá las actualizaciones emitidas desde un nodo en el mismo orden.

Hidra, en su implementación actual del modelo coordinador-cohorte no permite invocaciones anidadas de objetos si estas invocaciones generan ciclos [108]. Si nos planteáramos el soporte de este tipo de invocaciones, sería en este nivel donde podríamos implementar aquel mecanismo de detección de interbloqueos que consideráramos adecuado. De hecho esta posibilidad es la que nos ha llevado, entre otros motivos, a separar este nivel de invocaciones, del gestor de invocaciones a objeto.

El nivel de invocaciones a nodo remoto también es cliente del HMM y por tanto será notificado ante reconfiguraciones del cluster. Ante una reconfiguración, este nivel anota las caídas, para iniciar los reintentos o retornar excepciones tan pronto como finalice la reconfiguración del cluster.

2.3.5 Transporte de lotes de mensajes

Hidra dispone de un protocolo distribuido de cuenta de referencias que permite detectar residuos. Este protocolo es totalmente asíncrono y de prioridad baja. Que sea totalmente asíncrono significa que funciona correctamente sea cual sea el retardo que sufran sus mensajes y sea cual sea el orden en el que se entreguen éstos en su destino. Por su parte, que sea de prioridad baja

[3]De acuerdo a los argumentos *extermo a extremo*, deberíamos construir el orden FIFO en las propias réplicas o en el soporte más cercano a las réplicas. Sin embargo, proporcionamos orden FIFO en este nivel porque es de implementación eficiente y no aumentamos el coste de ningún objeto que no sea de tipo checkpoint.

significa que se desea que sus mensajes causen el mínimo impacto posible en la actividad del sistema y si es posible que generen el mínimo número de mensajes dedicados en exclusiva a servir al protocolo.

Para este protocolo de cuenta de referencias y para otros protocolos que podamos construir en el futuro, que también sean totalmente asíncronos y de prioridad baja, Hidra incluye un nivel de transporte de lotes de mensajes. El nivel permite agrupar mensajes en lotes para su envío posterior (batching) y permite incorporar los lotes de mensajes en mensajes corrientes que se deban enviar de forma forzosa al mismo destino (piggybacking).

```
public interface BatchTransport
{
    public void addBatch (Node destination, BatchMessage msg);
    public PiggyMessage getPiggyMessage (Node destination);
    public void setPiggyMessage (PiggyMessage msg);
    ...
}
```

Figura 2.7: Interfaz del Transporte de Lotes

La interfaz de este nivel lo podemos observar en la figura 2.7. El método `addBatch()` lo emplean los clientes de este servicio para enviar un mensaje al nodo `destination`, con baja prioridad. Llamaremos a estos mensajes, mensajes en *segundo plano* y cuando hagamos referencia a que un nodo envía un mensaje en segundo plano, nos estaremos refiriendo a que utiliza esta operación para enviarlos. El transporte fiable, cada vez que vaya a enviar un mensaje a un nodo, invocará al método `getPiggyMessage()` para obtener los mensajes en segundo plano, que yendo dirigidos al mismo nodo, aún no hubieran sido enviados. Este método empaquetará algunos o todos los mensajes dirigidos a dicho nodo en un mensaje de tipo `PiggyMessage()` que será concatenado en el mensaje a enviar. El nivel de transporte fiable receptor, al recibir los mensajes, en caso de que estos contengan mensajes concatenados, se los cederá al transporte de lotes mediante el método `setPiggyMessage()`. Esta operación, extraerá los mensajes en segundo plano que estén incluidos, invocando para cada una de ellos a la operación `handle()` del mensaje, permitiendo de esta forma que cada mensaje sea manejado de forma dependiente al protocolo al que sirven.

Cada mensaje en segundo plano incluye el número de reconfiguración del cluster en el que ha sido generado, permitiendo de esta forma al receptor del mensaje procesarlo o ignorarlo en función de si el protocolo que se construye con mensajes en segundo plano admite mensajes que sobrevivan a distintas reconfiguraciones.

Ante fallos de caída, este nivel descarta aquellos mensajes que tenga almacenados cuyo destino sea algún nodo que haya caído y almacena el número de reconfiguración del cluster para incluirlo en los nuevos mensajes en segundo plano que los clientes del nivel intenten enviar.

2.3.6 Gestor de invocaciones a objeto

El nivel más elevado de la arquitectura Hidra lo constituye un gestor de invocaciones a objeto (ORB) con soporte para objetos replicados y recolección de residuos. El ORB es multitarea y multidominio, no utiliza protocolos de comunicación a grupos con orden causal ni total, ni tan siquiera multienvíos atómicos. Tan sólo utiliza mensajes punto a punto fiables y multienvíos no atómicos con orden FIFO.

En caso de fallos de caída y de uniones de nodos al cluster, el ORB se reconfigura, para reflejar la nueva situación del cluster. El objetivo de esta reconfiguración es que no quede ningún residuo en ningún nodo que haga referencia a los nodos caídos, entendiendo en este caso el concepto de residuo en sentido amplio, incluyendo el estado que mantengan los protocolos que esté relacionado con los nodos caídos.

```
public interface Hidra {
    public BOA getBOA();
    public ROA getROA();
    public FOA getFOA();

    public void release (HidraObject obj);
    public HidraObject duplicate (HidraObject obj);
    ...
}
```

Figura 2.8: Interfaz de Hidra

La interfaz ofrecida por el ORB a las aplicaciones aparece en la figura 2.8 y constituye la interfaz Hidra. Como puede observarse las operaciones más significativas están relacionadas con el tratamiento de objetos y referencias a objeto. Las tres primeras operaciones retornan un adaptador específico de Hidra que los dominios utilizarán para registrar sus objetos. Cada adaptador se utilizará según la funcionalidad que los dominios deseen para sus objetos. El BOA permite registrar objetos de implementación única que recibirán la notificación de no referenciado cuando el objeto sea detectado como residuo. El FOA permite registrar objetos de implementación única sin soporte a recolección de residuos y por último el ROA permite registrar réplicas y objetos replicados.

Por su parte los métodos `release()` y `duplicate()` permiten descartar y duplicar respectivamente referencias a objeto. En particular el método `release()` tiene especial relevancia pues puede desencadenar la liberación tanto de las estructuras de datos internas del ORB que soportan al objeto, como desencadenar la ejecución de parte del protocolo distribuido de cuenta de referencias.

Las interfaces de los adaptadores y el diseño interno del ORB para ofrecer esta funcionalidad los describiremos con detalle en el capítulo 3.

Objetos Hidra

Todos los objetos construidos sobre Hidra deben heredar de la clase `HidraObject`. Esta clase define un conjunto de operaciones básicas que serán utilizadas por el ORB para manipular el objeto. De todas las operaciones que define esta clase, solo puede sobreescribirse el método `unreferecenced()`, cuya implementación en la clase `HidraObject` es una simple rutina vacía. Esta operación será invocada por el ORB cuando el objeto sea detectado como residuo. Los objetos Hidra podrán implementar en esta operación las acciones que deseen para tratar este evento.

Soporte al modelo coordinador-cohorte

Para implantar el modelo de replicación coordinador-cohorte es necesario el diseño de un sistema que garantice la atomicidad de las invocaciones y la consistencia del estado de las réplicas incluso en presencia de fallos. Para ello en Hidra se ha diseñado un mecanismo fiable de invocaciones a objeto [103] y un mecanismo distribuido de control de concurrencia [104]. La descripción más actualizada y más detallada de ambos protocolos puede encontrarse en la tesis de Muñoz [108]. Seguidamente pincelamos el funcionamiento básico de ambos mecanismos, qué objetos necesitan y la forma en la que se integran en el ORB.

Las invocaciones fiables se construyen haciendo que las invocaciones que partan desde un cliente hasta un objeto replicado incluyan un pequeño objeto, llamado `RoiID` que identifica inequívocamente la invocación. Las invocaciones se dirigen únicamente hacia una réplica, la coordinadora de la invocación y ésta, conforme procesa la invocación, va emitiendo mensajes de actualización a sus cohortes. En las actualizaciones se incluye el `RoiID`, de forma que ante caídas del coordinador, las réplicas cohorte serán capaces de reanudar la invocación. Con este propósito, el ORB del cliente que ha iniciado la invocación, la repetirá sobre otra réplica cuando sea informado de la caída del nodo que contenía la antigua réplica coordinadora. Los `RoiID` no son invocados y su propósito es doble: por una parte identificar la invocación y por otra, permitir a los dominios cliente liberar todo el estado interno relacionado con la invocación cuando los `RoiID` reciban la notificación de no referenciado. Los `RoiID` son objetos ordinarios de implementación única si el cliente que inicia la invocación no está replicado. Si lo está, el `RoiID` será un objeto replicado. Cabe destacar que en este contexto, el soporte al modelo coordinador-cohorte requiere que exista un tipo de objeto replicado, que no será invocado, pero que debe recibir la notificación de no referenciado.

El control de concurrencia distribuido de Hidra sigue una aproximación similar a los monitores, tipos protegidos de ADA, o métodos sincronizados de Java, pero en nuestro caso extendida para tratar objetos distribuidos y reglas de sincronización para múltiples objetos. Para ello se proporciona una extensión de IDL que permite contemplar la especificación de conflictos entre las operaciones de distintas interfaces. La compilación de estas especificaciones de conflictos genera un objeto que será utilizado para serializar las invocaciones que se efectúen sobre los objetos replicados.

Ambos protocolos, invocaciones fiables y control de concurrencia, requieren de dos objetos replicados adicionales, los `TObj` y los `CObj`. Los `TObj` los utilizan las réplicas del objeto que se invoca y el mecanismo de control de concurrencia para determinar cuándo se ha finalizado la invocación. Estos objetos no son invocados, sino que de igual forma a como ocurría con los

`RoiID`, utilizan la notificación de no referenciado para detectar esta condición de terminación. Por su parte, los `CObj` se utilizan para que el cliente avise a las réplicas del objeto de que ha obtenido la respuesta a la invocación. Esta notificación permitirá a las réplicas del objeto que se invoca liberar los resultados retenidos de la invocación efectuada por el cliente.

Los `CObj`, no son invocados tampoco si el cliente que inicia la invocación no es replicado. Por tanto, se utilizará de nuevo el mecanismo de objeto no referenciado. Sin embargo, si el cliente que inicia la invocación es replicado, para recuperarse ante casos especiales de caídas múltiples, la notificación de no referenciado no es suficiente. En estos casos, los clientes notifican a las réplicas para que liberen los resultados retenidos por medio de una invocación. Esta invocación debe llegar a todas las réplicas de igual forma a como ocurría con la notificación de no referenciado y para lograrlo, construimos los `CObj` como objetos replicados mediante una forma de replicación activa bastante restringida. No es necesario que la invocación al grupo de réplicas sea atómica, pues en casos de caídas del invocador, la invocación será reiniciada por otra réplica del cliente. Las réplicas del `CObj` ignorarán toda invocación que reciban después de la primera.

Por último, es necesario un objeto adicional por cada objeto replicado. Se trata de un objeto para implantar los mensajes de actualización. Las réplicas de un objeto (lo hará el programador) crearán un objeto replicado de tipo *checkpoint* que estará replicado de forma activa. De igual forma a como ocurre con los `CObj` no son necesarios multienvíos atómicos, ni ordenados de forma distribuida. Que no sean multienvíos atómicos implica que ante caídas de réplicas de objetos, es necesario ejecutar un protocolo de reconstrucción del estado de las invocaciones efectuadas sobre las réplicas de cada objeto. Por su parte, no es necesario orden total ni causal en el envío de actualizaciones, pues de hecho, si varias réplicas están sirviendo al mismo tiempo distintas invocaciones, significa que ambas pueden proceder concurrentemente. Esta concurrencia significa que no hay conflictos entre estas operaciones y por tanto las actualizaciones que sean emitidas por los coordinadores de estas invocaciones podrán llegar con órdenes diferentes a las distintas réplicas. Para simplificar el soporte a las actualizaciones, y tal y como describimos en el apartado dedicado al nivel de invocaciones a nodo remoto, las invocaciones a múltiples destinos sí proporcionan orden FIFO, que como es sabido son de implementación sencilla y eficiente.

Por tanto, para cada invocación fiable con control de concurrencia, el soporte crea tres objetos. Dos de ellos replicados y el tercero, replicado en función de si está replicado el cliente. Para todos ellos es necesario que las réplicas reciban la notificación de no referenciado y para uno de ellos, es necesario un mecanismo no fiable de multienvíos sin garantías de orden. Además de los objetos que crea el soporte, para implantar los mensajes de actualización es necesario que el programador cree un objeto replicado de forma activa, pero este objeto no exige un transporte distribuido de mensajes con orden total ni causal, ni entrega atómica.

Para encapsular la lógica de los protocolos que utilizamos para implantar la replicación coordinador-cohorte en el soporte a objetos replicados, en Hidra proporcionamos un mecanismo similar a los interceptores CORBA [113]. El soporte a los objetos, que veremos en el capítulo 3 se ha diseñado para permitir la adopción de diferentes modelos de replicación sin que sea necesario modificar de forma significativa la estructura interna del ORB. Las invocaciones que se efectúen sobre todos los objetos podrán ser interceptadas, tanto en el origen de la invocación como en su destino. De esta forma, tanto el protocolo de invocaciones fiables, como

el control de concurrencia, podrán actuar sobre las invocaciones, insertando en ellas los objetos que necesiten, e implantando de forma externa al soporte de los objetos toda su lógica.

Reconfiguración del ORB

Ante reconfiguraciones que incluyan caídas de nodos, el ORB ejecutará pasos síncronos para reconfigurar su estado. Las acciones a ejecutar para este propósito son las siguientes:

1. Eliminar todas las referencias a objeto que sean inválidas debido a que haya caído el objeto. Garantizar que todas las referencias apunten a la ubicación correcta de los objetos.

2. Ejecutar un protocolo de marcado y barrido de las referencias, entregando las notificaciones de no referenciado a todos los objetos que ya no estén referenciados como consecuencia de las caídas. A este protocolo lo llamaremos *protocolo de reconstrucción de la cuenta de referencias* y su objetivo fundamental consistirá en restablecer los invariantes del protocolo de cuenta de referencias.

3. Alcanzar consenso entre las réplicas de los objetos replicados por el modelo coordinador cohorte en cuanto al estado más actualizado de las invocaciones que estuvieran en proceso.

Los dos primeros aspectos de la reconfiguración del ORB se encuentran descritos en el capítulo 4. Por su parte, los protocolos relacionados con el soporte al modelo coordinador-cohorte, incluida la reconstrucción de su estado se encuentran en la tesis de Muñoz [108].

2.4 Trabajo relacionado

Existen multitud de arquitecturas software que se han diseñado para dar soporte a la construcción de sistemas altamente disponibles. En este apartado vamos a referirnos a aquellas que proporcionen el concepto de objeto replicado como la pieza básica con la que construir las aplicaciones. De entre ellas, nos vamos a centrar en exclusiva en las distintas aproximaciones que han tomado diversos trabajos para dar soporte a objetos replicados de forma independiente al lenguaje.

Así pues, el estudio de lenguajes de programación distribuida con soporte a objetos replicados (como por ejemplo Argus [81]) queda fuera del alcance de este trabajo. Otras arquitecturas orientadas a procesos, tales como ISIS, Horus, Totem, Transis, Psync, Amoeba, Delta-4 o Phoenix no serán comentadas en profundidad, excepto en la medida en que hayan sido utilizadas como mecanismo de comunicación a grupos para proporcionar objetos replicados sobre ellas. Tampoco comentaremos clusters completos ni arquitecturas comerciales sobre las que no existan detalles en cuanto a la propia arquitectura, no soporten objetos replicados, o no estén públicamente disponibles. Casos relevantes de estos sistemas los podemos encontrar en *Sun Cluster 3*, *HP Service Guard*, o *Microsoft Wolfpack*, entre otros.

2.4.1 CORBA

CORBA [113] es una arquitectura para la construcción de gestores de invocaciones a objeto. Se trata de un estándar promovido por la OMG (Object Management Group) para facilitar el desarrollo de ORBs que sean interoperables entre ellos, de forma que se puedan construir aplicaciones distribuidas orientadas a objeto sin necesidad de que la implementación sea dependiente de la empresa que comercialice el ORB, ni del ORB en sí mismo. La especificación CORBA no da detalles de cómo deben construirse los ORBs, limitándose a especificar la funcionalidad que deben ofrecer en base al cumplimiento de interfaces especificadas en IDL.

Además del propio ORB, CORBA especifica un conjunto de servicios [112] que pueden implantarse sobre el ORB para dotarlo de funcionalidades más específicas. Ejemplos relevantes son el servicio de nombres, el servicio de transacciones, el servicio de eventos, control de concurrencia o el servicio de persistencia entre otros.

CORBA no es una arquitectura para la construcción de aplicaciones altamente disponibles, sin embargo, desde su aparición han ido apareciendo multitud de trabajos que han intentado dotar de ciertas características de alta disponibilidad a las aplicaciones CORBA utilizando servicios del estándar para ello. Por ejemplo, en [36] se ha utilizado el servicio de eventos como mecanismo para lograr cierto soporte a replicación. En [106], Muñoz, Galdámez y Bernabéu han incluido un limitado soporte a replicación coordinador-cohorte en los interceptores que ofrece CORBA. También puede usarse la combinación de los servicios de transacciones sobre objetos, control de concurrencia y persistencia para dotar de cierto grado de tolerancia a fallos a las aplicaciones.

El principal defecto de estas aproximaciones es que todas ellas parten de un ORB que no tolera fallos. No se dispone de detector de fallos, y por tanto las acciones que ejecuten cada uno de los nodos para reaccionar ante fallos no estarán sincronizadas entre ellas. Distintos ORBs situados en nodos distintos podrán tener una visión distinta de los fallos que ocurran en el sistema, y por tanto, tanto su estado interno, como el estado de las aplicaciones que se construyan por encima no reflejará de forma consistente el estado del sistema.

En lo que respecta a la recolección de residuos, hubo un intento por parte de CORBA de captar propuestas para la especificación de un servicio o componente del sistema que se encargara de la recolección de residuos [111]. Más precisamente, el enfoque que se daba a la propuesta era permitir desactivar objetos no referenciados, más que recolectar residuos. De cualquier forma, hasta la fecha esta petición de propuestas no ha tenido continuidad.

2.4.2 Servicio de tolerancia a fallos de CORBA

El servicio de tolerancia a fallos de CORBA [114] ha surgido recientemente por la creciente importancia que está recibiendo la construcción de aplicaciones orientadas a objeto altamente disponibles. Ante la consolidación de esta necesidad y ante la insuficiente adecuación del resto de servicios para dotar de alta disponibilidad a las aplicaciones, este servicio pretende estandarizar el soporte necesario para lograr alta disponibilidad mediante objetos distribuidos.

Este servicio no es en sí mismo una arquitectura de alta disponibilidad, pues de igual forma al propio estándar CORBA, se trata de una especificación de funcionalidad y de interfaces donde no se describen detalles internos de cómo proporcionar dicha funcionalidad. Es relevan-

te su estudio, pues todos aquellos ORBs que cumplan con el estándar y deseen proporcionar alta disponibilidad, probablemente lo harán siguiendo la especificación de este servicio. Por tanto, comparar Hidra con este servicio puede servir como un principio de comparación con estas futuras arquitecturas. Sin embargo, cabe destacar que en el instante de la redacción de esta memoria, este servicio de CORBA no está todavía integrado en el estándar, no existen implementaciones de él, y todavía puede sufrir importantes modificaciones antes de su completa adopción.

Este servicio CORBA permite crear objetos replicados y contempla la necesidad de disponer de un detector de fallos. Para crear objetos replicados, las aplicaciones crean objetos y los registran como integrantes de un grupo de objetos. Cada grupo de objetos podrá ser manipulado como si se tratara de un único objeto y el ORB, mediante un detector de fallos proporcionará una visión homogénea de qué replicas forman parte del grupo. En este sentido, la aproximación es la misma que seguimos en Hidra. La diferencia radica en el modelo de replicación que se puede utilizar.

El servicio CORBA ha intentado ser exhaustivo en la definición del soporte a replicación según distintos modelos. Permite tanto el modelo activo como el pasivo. El activo con votación o sin votación y el pasivo tanto en frío como o en caliente. Para el caso de la replicación pasiva, permite optar por consistencia garantizada por el propio soporte, como por control de consistencia implementado por las propias réplicas. También permite la construcción de objetos replicados donde no sea necesaria la consistencia en el estado de las réplicas. Todo ello lleva a que el servicio de tolerancia a fallos de CORBA contempla el desarrollo de aplicaciones altamente disponibles mediante 8 variantes de objetos replicados. Los ORBs que deseen cumplir con el estándar deberán proporcionar al menos uno de los modelos de replicación con garantías de consistencia. Sin embargo, no se hace mención alguna al modelo coordinador-cohorte ni a recolección de residuos para objetos replicados.

En lo que respecta a la consistencia en el estado de las réplicas, según la terminología que emplea CORBA, Hidra estaría optando por consistencia proporcionada por las réplicas, pues los mensajes de actualización deben ser emitidos y procesados por las réplicas de forma explícita sin más ayuda que el uso de invocaciones a objeto. El programador será el único responsable de reaccionar adecuadamente a este tipo de invocaciones de actualización para mantener consistente el estado de las distintas réplicas. Para el soporte a replicación activa, el estándar indica la necesidad de utilizar algún protocolo de comunicación a grupos ordenada, sin embargo, para el caso de objetos replicados de forma activa sin garantías de consistencia, tal y como ocurre en Hidra, este tipo de algoritmos no son necesarios.

2.4.3 Solaris MC

Solaris MC [8, 9, 73] es un prototipo de cluster desarrollado en los laboratorios de investigación de Sun Microsystems (Sunlabs). La arquitectura de este sistema es similar a la Hidra, de hecho varios aspectos de Hidra están basados en el diseño de Solaris-MC. Solaris-MC dispone de un protocolo de pertenencia a grupos y de un nivel de transporte fiable punto a punto con características similares a las de Hidra, dispone de un ORB que se construye sobre estos mecanismos, el ORB tolera fallos y los objetos disponen de un mecanismo de recolección de residuos.

Por otra parte, en Solaris-MC no se describe ningún mecanismo de replicación de objetos

integrado en el ORB, ni por tanto recolección de residuos para objetos replicados. El nivel de transporte sólo permite comunicación fiable punto a punto y no incorpora ni el nivel de transporte de lotes de mensajes ni el de invocación a nodo remoto. De cualquier forma, la funcionalidad de estos niveles está integrada en el ORB de Solaris-MC, por lo que a nivel arquitectónico ambos sistemas son bastante similares.

2.4.4 Electra

Electra [86], resultado de la tesis de Silvano Maffeis, fue el primer ORB, que pretendiendo cumplir con el estándar CORBA ofreció el concepto de objetos replicados. La idea fundamental de este sistema consiste en utilizar una biblioteca de comunicación a grupos que ofrezca al menos ordenación total y construir sobre ella un soporte para el desarrollo de objetos replicados. En particular, la implementación de Electra se realizó sobre ISIS [122] y el modelo de replicación que adoptó principalmente fue el modelo activo. Las operaciones para gestionar grupos de objetos, añadir y elminir réplicas a grupos, etc. se encuentran en el BOA (Adaptador Básico de Objetos) de CORBA, que fue ampliado por Electra.

A parte de las características propias del modelo de replicación activo que ya hemos descrito en el apartado 1.4.1, la principal deficiencia de Electra consiste en que se asocia el concepto de réplica de objeto con el de réplica de proceso ofrecida por ISIS. Esto lleva a que tanto los protocolos de comunicación ordenada, como los de pertenencia a grupo se realicen por cada objeto replicado, en lugar de optimizar su coste al apreciar que cada ORB se ejecuta en una máquina distinta. Por ejemplo si se dispone de 3 nodos A, B y C y de tres objetos replicados O_1, O_2 y O_3 donde cada objeto replicado dispone de réplicas en cada uno de los nodos, se ejecutarán 3 protocolos de pertenencia a grupos entre los nodos A, B y C en lugar de uno sólo. Esta deficiencia se debe a que ISIS proporcionan de forma integrada la comunicación ordenada a grupos y la detección de fallos de los miembros del grupo. Esta aproximación que puede ser acertada para el caso de un único proceso por nodo, deja de ser razonable para el caso de ORBs que puedan contener simultáneamente multitud de objetos.

Otros sistemas han adoptado aproximaciones similares a las de Electra, como por ejemplo Orbix+ISIS, donde el ORB comercial Orbix fue modificado para utilizar ISIS como medio de comunicación para objetos replicados. Previamente a Electra, otros sistemas adoptaron una aproximación similar, como por ejemplo Amoeba [131] y Psync [92]. En Amoeba, el protocolo de comunicación a grupos fue ampliado [138] para soportar invocaciones a procedimiento remoto sobre objetos replicados de forma activa, algo que fue también realizado en Psync.

2.4.5 Eternal

El objetivo de Eternal, de igual forma a como lo era en Electra y Orbix+ISIS, es proporcionar replicación de objetos por medio de un ORB y de un mecanismo de comunicación a grupos ordenado. Sin embargo, Eternal utiliza una aproximación distinta. En Electra se construye un ORB sobre una biblioteca de comunicación a grupos existente, mientras que en Eternal, se utiliza un ORB existente y un protocolo de comunicación a grupos (Totem [96]) existente y sin apenas modificaciones, se amplia su mecanismo de comunicación para proporcionar replicación.

Para ello, se interceptan los mensajes IIOP que genera el ORB antes de que estos mensajes se conviertan en mensajes TCP/IP. La intercepción se realiza de forma altamente dependiente del ORB y de la plataforma. Una vez interceptados, en lugar de enviarse por TCP/IP como lo haría el ORB que Eternal utiliza, se ceden a un gestor de replicación, que analiza si el destino del mensaje está o no replicado, y en caso de tratarse de objetos replicados, averigua el modelo de replicación que adoptan. Si el objeto destino o el origen no son replicados, se permite que el mensaje llegue a su destino utilizando TCP/IP, mientras que si los objetos intervinientes son replicados, se cede el mensaje a Totem para que éste procese y envíe el mensaje a sus destinatarios.

Tanto si los objetos siguen el modelo de replicación activa, como el modelo pasivo, las invocaciones se envían de forma ordenada a todas las réplicas, pero en caso de replicación pasiva tan sólo una de ellas procesará la invocación.

2.4.6 OGS

El Servicio de Grupos de Objetos (OGS) fue diseñado recientemente por Pascal Felber en su tesis [37, 38], como una posible propuesta a la petición inicial que efectuó CORBA para contemplar objetos replicados en el estándar y en todo caso como un servicio CORBA independiente. Como ya hemos visto en el apartado 2.4.2 esta petición inicial de propuestas ha dado lugar a la especificación de un servicio de tolerancia a fallos. Sin embargo el OGS fue planteado previamente, y proporciona una aproximación que ha sido seguida con posterioridad y hasta la actualidad por diferentes trabajos [45].

El OGS plantea proporcionar un servicio de replicación de objetos que sea totalmente externo al ORB y que por tanto pueda ser integrado y utilizado por todo ORB que cumpla con el estándar. Esta idea ha sido argumentada en el sentido de que los servicios CORBA deben entenderse como totalmente externos, portables e interoperables y que no requieran de cambios en la implementación de los ORB.

El diseño del OGS contempla varias componentes, entre las que se incluyen protocolos de consenso distribuido, multienvíos ordenados, detectores de fallos y pertenencia a grupos. Sobre estas componentes se construye el concepto de *referencia a grupos de objetos*, que serán utilizados por las aplicaciones que deseen acceder a objetos replicados. Debido a que el OGS pretende ser independiente al ORB, las referencias a grupos de objetos no son referencias a objetos tal y como las maneja el ORB, sino objetos CORBA en sí mismos.

Esta aproximación tiene la ventaja de ser independiente del ORB al precio de incluir un nivel de indirección adicional en las invocaciones y de eliminar la transparencia de replicación a los clientes de objetos replicados. El nivel de indirección aparece debido a que los clientes deben contactar con el OGS y el OGS contactar con las réplicas. La transparencia de replicación se pierde pues los clientes deben crear de forma explícita las referencias a grupos de objetos en lugar de ser el ORB quien las construya y las mantenga.

2.4.7 Resumen

De todas las arquitecturas analizadas, tan sólo Solaris-MC proporciona recolección de residuos y en este caso sin contemplar objetos replicados. Las demás arquitecturas requieren el uso de

protocolos de comunicación ordenada con orden total aún permitiendo la mayoría de ellas tanto replicación activa como pasiva. Hidra no necesita el uso de estos protocolos y permite replicar objetos de forma transparente a los clientes. La detección de fallos se utiliza en la mayoría de los sistemas analizados para implantar sincronía virtual [15, 12], mientras que en Hidra se utiliza para poder reconfigurar el ORB en presencia de fallos, adoptando con ello una visión más optimista de la ejecución ordinaria del sistema.

Solaris-MC proporciona objetos con recolección de residuos en un ORB que no incorpora soporte interno a los objetos replicados. En todo caso los objetos replicados se construirán sobre referencias ordinarias de objetos. Electra construye un ORB sobre una biblioteca de comunicación a grupos, Eternal proporciona replicación de objetos interceptando los mensajes que emite un ORB y convirtiendo aquellos dirigidos a objetos replicados, en mensajes tratados por una biblioteca de comunicación a grupos. El OGS proporciona objetos replicados de forma independiente del ORB con menor transparencia y menor eficiencia que el resto de aproximaciones y por su parte Hidra proporciona un ORB construido explícitamente para proporcionar objetos replicados sin necesidad de utilizar protocolos de comunicación ordenada e incluyendo recolección de residuos.

Hidra no cumple con el estándar CORBA, ni con el servicio de tolerancia a fallos de CORBA. Sin embargo tanto el modelo de objetos, como la aproximación de Hidra a la replicación de objetos está cercana al modelo de objetos CORBA y a la más reciente aproximación de CORBA a la replicación de objetos. En esta aproximación se sugiere que el propio ORB debe construirse de forma explícita para permitir replicación, proporcionar consistencia y detectar fallos. Por otra parte, CORBA no especifica ningún mecanismo de recolección de residuos ni contempla el modelo de replicación coordinador-cohorte.

Capítulo 3

El gestor de invocaciones

Se debe hacer todo tan sencillo como sea posible, pero no más sencillo.

Albert Einstein

Contenidos del capítulo

El ORB de Hidra es el nivel más elevado de la arquitectura y constituye la infraestructura de comunicación que utilizarán las aplicaciones que se desarrollen sobre Hidra. En este capítulo comentaremos los objetivos que nos planteamos con la construcción del ORB, describiremos su diseño interno y su funcionamiento tanto en situaciones normales como en presencia de fallos.

3.1 Planteamiento de objetivos

El principal objetivo del ORB de Hidra es compartido con la práctica totalidad de ORBs que podemos encontrar en la actualidad: permitir el desarrollo de aplicaciones orientadas a objeto en un entorno distribuido, donde el soporte dedique un especial énfasis a ocultar a los programadores las características propias de la distribución.

Para lograr este objetivo, en Hidra hemos detectado los siguientes subobjetivos que intentaremos satisfacer: el ORB debe facilitar el desarrollo de partes del sistema operativo con la idea de permitir la construcción de un cluster, debe ser multitarea, multidominio, con soporte a diferentes tipos de objetos, incluyendo soporte a objetos replicados según diferentes modelos de replicación, debe incorporar recolección de residuos y debe reaccionar convenientemente frente a fallos.

3.1.1 Soporte a clusters

El desarrollo de un cluster es una tarea ambiciosa, que requiere el desarrollo específico de varias componentes esenciales del sistema, que crearán la ilusión de disponer de un único ordenador. Aspectos tales como la gestión de memoria, sistemas de ficheros, la gestión de procesos e incluso el acceso a los distintos periféricos deben ser modificados respecto al soporte tradicional que podemos encontrar en los sistemas operativos convencionales. Las modificaciones que se introduzcan deben poder realizarse tanto dentro del propio núcleo del sistema operativo como fuera de él si así se estimara conveniente, de forma que pueda optarse por una solución eficiente, estable y segura cuando sea necesario. Por contra, será deseable poder realizar desarrollos de ciertas componentes del sistema como módulos que no residan dentro del propio núcleo, tanto en etapas de desarrollo y pruebas, como para unidades funcionales no esenciales o cuya estabilidad no sea determinante.

Por tanto, derivados del objetivo de habilitar el desarrollo de un cluster, nos aparecen una serie de objetivos que deberán perseguirse:

- Soportar el desarrollo de componentes del sistema operativo que hagan uso de la arquitectura. Es asimismo deseable que el soporte al desarrollo de componentes sea lo más genérico posible, sin importar el hecho de que vayan a estar dentro o fuera del núcleo, permitiendo su inclusión en el núcleo en caso de estimarse oportuno.

- Optar por soluciones eficientes siempre que sea posible, frente a soluciones que primen otros aspectos.

3.1.2 Multitarea

Gran parte de los ORBs que encontramos hoy en día no son multitarea o no soportan dominios multitarea. En nuestro caso, el propio ORB se construirá con diferentes tareas con objeto de incrementar su rendimiento y en particular para agilizar aspectos tales como el acceso a la red, la recolección de residuos o las reconfiguraciones del sistema en caso de fallos. Adicionalmente, el diseño del ORB no impondrá ninguna restricción en cuanto a número de tareas que puedan crearse dentro de cada dominio de protección, ni de qué tareas podrán hacer uso del ORB para comunicarse con el resto.

Derivado de la concepción de Hidra como mecanismo multitarea, aparece un condicionante que es importante tener en cuenta a la hora de diseñar aplicaciones sobre Hidra: en principio, un mismo dominio de protección podrá estar recibiendo simultáneamente n invocaciones a sus objetos servidores. Esta característica fuerza al programador a hacer uso de los mecanismos convencionales de control local de concurrencia que estime convenientes para evitar inconsistencias, no sólo en el estado de los objetos CORBA internos al dominio, sino también para evitar inconsistencias en cualquier otra parte de la memoria que utilice el dominio desde las operaciones invocadas.

Nótese que esta vocación multitarea, está en contraposición con aproximaciones para la construcción de infraestructuras de objetos que exigen determinismo en las acciones a realizar por los objetos. Asumir determinismo, implica forzar a que todo el código de los dominios de protección se estructure en objetos con código determinista, incluyendo en nuestro caso el código del propio núcleo del sistema operativo.

En Hidra, no asumimos que el código del sistema operativo sea determinista, por considerar este aspecto como no razonable, ni exigimos que el código de los dominios de protección se estructure en objetos de código determinista. En Hidra optamos por una aproximación más cercana al modelo de objetos CORBA [113], en la que no se hace ninguna asunción acerca de cuál es la opción que elegirán los dominios para implementar sus objetos, dejando total libertad al programador en cuanto a la estructuración de los objetos dentro de los dominios que implemente.

3.1.3 Multidominio

El ORB de Hidra debe permitir que se construyan aplicaciones distribuidas orientadas a objeto, tanto dentro del sistema operativo, como en procesos de usuario. Por tanto, el ORB debe manejar los objetos de los diferentes dominios de su propio nodo y permitir a los dominios locales el acceso a los objetos de otros nodos. De igual forma, debe permitir que los dominios situados en nodos remotos puedan acceder a los objetos locales, sea cual sea el dominio en el que residan.

El hecho de que el ORB sea un *ORB multidominio* y un *ORB del sistema* hace que éste tenga unas características diferentes que no podemos encontrar en los ORB basados en otros esquemas. El propio estándar CORBA [113] enumera las alternativas más comunes como son: un ORB como biblioteca, un ORB residente, un ORB como servidor o un ORB del sistema. Las tres primeras aproximaciones comparten la característica de proporcionar un acceso a los objetos de forma homogénea a todos los clientes. En cambio un ORB del sistema debe tratar de forma diferente los accesos que efectúen los clientes en función de dónde residan los objetos:

en el mismo dominio que el cliente, en un dominio distinto pero dentro del mismo nodo o en nodos distintos. Adicionalmente, el hecho de que un objeto resida en el núcleo del sistema operativo o que resida en un proceso de usuario también puede requerir un tratamiento distinto. Estas distinciones permiten aumentar la eficiencia del soporte dentro del núcleo y aumentar la seguridad tanto del propio sistema como entre las aplicaciones que se construyan en procesos de usuario.

3.1.4 Tipos de objetos

En toda aplicación pueden existir múltiples tipos de objetos en función de cómo los gestione el ORB, o de qué funcionalidad propia del sistema proporcionen a las aplicaciones que se construyan utilizándolos. De igual forma a como sucede con el estándar CORBA, donde se preve la posibilidad de que distintos ORBs proporcionen distintos tipos de objetos, en Hidra consideramos este aspecto como fundamental para habilitar el desarrollo de aplicaciones de forma flexible. En CORBA se define explícitamente el tipo de objeto *portable* como el tipo de objeto que, utilizando el adaptador POA, esté concebido para utilizar únicamente servicios que deben existir en todo ORB. Para el resto de tipos de objetos, CORBA tan sólo menciona la posibilidad de que cada ORB defina una semántica distinta para cada uno de ellos y sugieren que en principio no es necesario disponer de demasiados tipos distintos de objetos.

De forma consistente con la previsión de CORBA de proporcionar diferentes tipos de objetos con distintas semánticas, en Hidra hemos identificado al menos tres tipos de objetos, lo que no excluye la posibilidad de que necesitemos algún otro en el futuro. Por tanto, el diseño del ORB debe ser tal, que sea sencillo incorporar nuevos tipos de objetos en caso necesario, y que al menos proporcione soporte para los que hemos identificado. Cada tipo de objetos deberá proporcionar funcionalidad distinta, cada uno requerirá de un tratamiento distinto por parte del ORB, y su gestión incurrirá posiblemente en costes diferentes. Los tres tipos que hemos contemplado hasta la actualidad son los siguientes: los *objetos básicos*, los *objetos fijos* y los *objetos replicados*.

Objetos básicos

El tipo de objeto análogo al objeto *portable* de CORBA es el *objeto básico*[1] de Hidra. Este tipo de objeto tiene como característica adicional el hecho de que las implementaciones de los objetos recibirán la notificación de objeto no referenciado cuando el ORB detecte esta situación. Es decir, los objetos básicos son objetos CORBA con un mecanismo de recolección de residuos integrado.

Objetos fijos

Estos objetos tienen como característica principal el estar ubicados en lugares conocidos a priori. Es decir, un dominio que desee acceder a uno de estos objetos no tendrá que acceder a ningún directorio o servicio de nombres para obtener una referencia, sino que todo dominio será capaz

[1] Nótese que no hacemos uso de la misma nomenclatura para no inducir a confusión, ya que Hidra no se plantea como una implementación de CORBA.

de construir en tiempo de ejecución una referencia válida a todo objeto fijo. Debido a que todo dominio puede crear en tiempo de ejecución una referencia a un objeto fijo, las implementaciones de estos objetos se deberán crear y deberán registrarse al arrancar el sistema y no deberán variar de ubicación en ningún caso. Ejemplos de estos objetos pueden ser aquellos servicios que deban existir en toda máquina individual, como por ejemplo un servicio de nombres para los recursos de nodo, la componente local a cada nodo del servicio de pertenencia al cluster Hidra, etc.

Objetos replicados

Dado que el objetivo fundamental de Hidra es lograr alta disponibilidad y la vía elegida para lograrlo es la replicación de objetos, el ORB debe proporcionar un medio para replicar objetos. En Hidra consideramos a los objetos replicados como un tipo de objetos distinto a los objetos básicos y a los objetos fijos. Es decir, consideramos que el ORB no deberá tratar a cada objeto replicado como la agregación de un conjunto de objetos básicos o fijos, sino como un único objeto que tiene la particularidad de disponer de un número variable de implementaciones. Esta particularidad la enunciaremos diciendo que los objetos replicados en Hidra *son objetos de primer nivel*, pues el ORB tratará internamente a los objetos replicados y a las referencias a objetos replicados como objetos y no como colecciones de objetos.

Los objetos replicados también tendrán integrado un mecanismo de recolección de residuos similar al de los objetos básicos, con la particularidad de que se deberá garantizar que todas las implementaciones del objeto recibirán la notificación de objeto no referenciado cuando ya no existan dominios que puedan invocar al objeto. Nótese que hacemos uso del término implementaciones del objeto para referirnos a cada una de las réplicas del objeto, y objeto replicado para referirnos al concepto de grupo de objetos tal y como lo podemos encontrar en trabajos previos [86].

El modelo de replicación que seguirán los objetos replicados será con carácter preferente el modelo coordinador-cohorte o el modelo pasivo, pero el soporte a los objetos replicados deberá ser lo suficientemente flexible para permitir la incorporación de soporte adicional para contemplar objetos replicados mediante replicación activa o mediante cualquier otra esquema de replicación.

3.1.5 Recolección de residuos

Uno de los objetivos a satisfacer por el ORB es el disponer de un mecanismo integrado de recolección de residuos que garantice que los residuos serán eliminados. Contemplamos fundamentalmente dos tipos de residuos: objetos de aplicación y recursos asociados a la gestión de objetos. El tratamiento de los objetos de aplicación que se conviertan en residuos lo hemos comentado en los apartados precedentes al hacer mención a la notificación de objeto no referenciado que recibirán las implementaciones de los objetos. Con esta notificación, le cedemos a la aplicación la responsabilidad de liberar los recursos que la implementación estuviera consumiendo. El segundo tipo de residuo que pretendemos que el ORB gestione es el asociado a la gestión de los objetos y en particular a la gestión de las referencias a objeto. El ORB deberá liberar los recursos que se asignen a gestionar las referencias a objetos una vez que las

aplicaciones las descarten. Nótese que en este caso pretendemos que el ORB se encargue de la detección y recolección de los residuos, mientras que para el caso de los objetos de aplicación, el ORB deberá detectar y notificar a los residuos acerca de su condición.

3.1.6 Tolerancia a fallos

El último objetivo, pero no el menos importante que perseguimos con el ORB de Hidra es la tolerancia a fallos. El ORB deberá reaccionar ante los fallos que puedan ocurrir, reconfigurando su estado convenientemente para reflejar la nueva situación. Tal y como comentamos en el apartado 2.3, el modelo de fallos subyacente al ORB es el modelo de fallo de parada, con partición única. Por ello, el tipo de fallos a considerar dentro del ORB es el de caída de un nodo, ante el cual el ORB deberá reconfigurarse para eliminar toda la información que exista en los nodos vivos referente al nodo caído.

3.2 Diseño

El diseño del ORB de Hidra toma diversas ideas de diferentes tipos de sistemas, de entre las que destacan fundamentalmente dos: CORBA [113] y Solaris-MC [73]. Sin embargo Hidra presenta aspectos diferenciadores con ambos. No se trata de un ORB que cumpla con el estándar CORBA por tener unos objetivos distintos, fundamentalmente alta disponibilidad, recolección de residuos y eficiencia, y no forzosamente estandarización ni interoperabilidad. En cambio el modelo de objetos de Hidra es similar al modelo de CORBA, y a nivel arquitectónico los adaptadores de objeto de CORBA y los interceptores han sido tenidos en cuenta en el diseño de sus homólogos en Hidra. De Solaris-MC, el ORB de Hidra toma determinados aspectos funcionales de los xdoors y de los handlers, comparables con los endpoints y los adaptadores de Hidra, sin embargo difiere de él por su diferente estructuración de las referencias y la inclusión de soporte a objetos replicados.

3.2.1 Niveles del ORB

El ORB de Hidra está diseñado en cuatro niveles que aparecen representados en la figura 3.1.

El nivel inferior del ORB está formado por la tabla de *endpoints*. Este nivel básicamente contiene un endpoint por cada implementación que resida en el nodo y un endpoint por cada objeto que sea referenciado en el nodo y que no resida en él. A los endpoints del primer tipo los llamamos *endpoints servidores* y a los del segundo caso *endpoints clientes*. Los endpoints son similares a los xdoors de Solaris-MC en concepto, sin embargo en Hidra hemos diseñado versiones específicas para objetos replicados que no están presentes en Solaris-MC y su interacción en general con el resto del ORB varía sustancialmente. Cada endpoint servidor está asociado como mínimo a una implementación del objeto en el mismo nodo. Esta asociación se realiza a través de la tabla de descriptores de objetos de cada dominio.

El nivel de *descriptores de objeto* contiene una tabla de descriptores de objeto por cada dominio de protección (incluido el propio núcleo del sistema operativo). La funcionalidad de este nivel proporciona a cada dominio de protección acceso a los objetos para los que el dominio

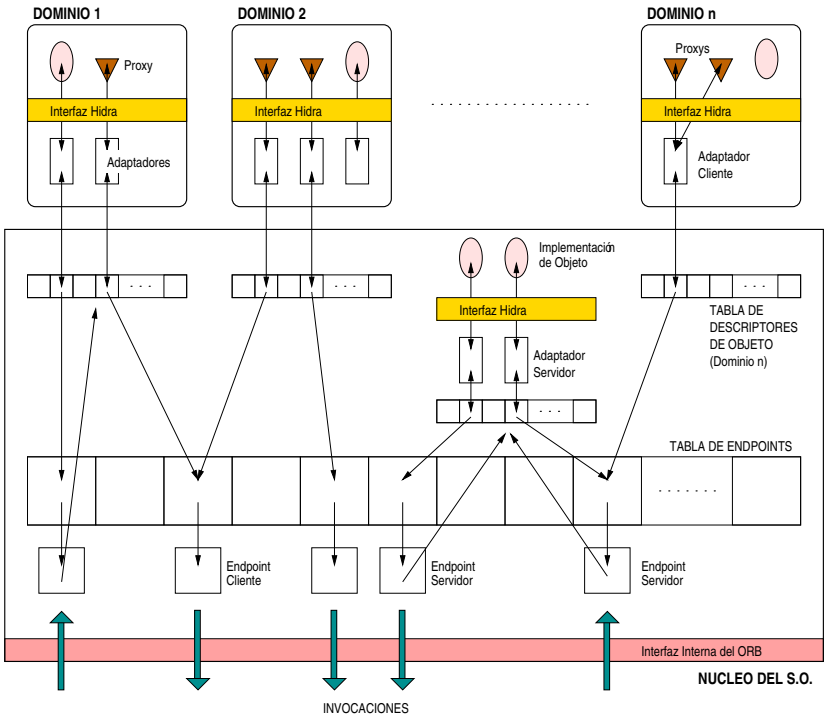

Figura 3.1: El Gestor de Invocaciones a Objeto de Hidra

posea referencias utilizando capacidades. De forma simétrica, proporciona al núcleo acceso a los objetos que residen en el dominio utilizando punteros a los adaptadores.

El tercer nivel, el *nivel de adaptadores*, ya reside en cada dominio de protección y consiste en una serie de adaptadores de objeto que están asociados, o bien a implementaciones de objeto (adaptadores servidores), o bien a referencias a objeto (adaptadores clientes). Los adaptadores contienen un descriptor de objeto para permanecer conectados al ORB del núcleo. La diferencia entre adaptadores servidores y adaptadores clientes consiste en que los primeros contienen un puntero para acceder a la implementación del objeto, mientras que los segundos contienen una tabla de punteros para permanecer conectados a los proxies del objeto.

Por último, el cuarto nivel es el *nivel de stubs*. Este nivel contiene los proxies de cada objeto y los esqueletos de cada implementación. Un adaptador cliente tiene asociado un proxy por cada interfaz que se utilice del mismo objeto remoto. Por su parte, los adaptadores servidores están conectados a un objeto intermedio, al que llamamos *stub servidor* que contiene un esqueleto por cada interfaz que haya exportado el objeto servidor y un puntero a la propia implementación del objeto.

3.2.2 Los adaptadores

Como ya comentamos en el capítulo anterior, el ORB de Hidra se ha diseñado para admitir diferentes tipos de objetos mediante adaptadores. Cuando un dominio crea un objeto, lo registra mediante el correspondiente adaptador de Hidra para asociarle su tipo. En la figura 2.8 de la pá-

gina 41 mostramos la interfaz externa de Hidra, mediante la cual podíamos obtener los distintos adaptadores de Hidra. En la figura 3.2 ahora mostramos las interfaces de los adaptadores FOA, BOA y ROA, que permiten registrar respectivamente a los objetos fijos, básicos y replicados.

```
public interface FOA {
    public void registerObject (HidraObject obj, int slotId);
    public HidraObject getReference (Node node, int slotId);
}
public interface BOA {
    public void registerObject (HidraObject obj);
}
public interface ROA {
    public HidraObject registerObject (HidraObject obj, Checkpoint chk,
                                       int replicationModel);
    public void joinObject (HidraObject obj, Checkpoint chk,
                            HidraObject localReplica);
    public void leaveObject (HidraObject obj);
}
```

Figura 3.2: Interfaces de los Adaptadores del ORB

Los objetos de tipo fijo son objetos cuya ubicación es conocida por todos los nodos a priori, de forma que para registrar este tipo de objetos, tal y como puede observarse en su interfaz, se debe proporcionar además de la implementación, qué entrada en la tabla de endpoints debe ocupar el objeto. Por su parte el método `getReference()` del FOA permite obtener una referencia a un objeto fijo sin más que especificando su ubicación. Por último comentar que el FOA sólo está disponible dentro del núcleo, por hacer uso explícito de los recursos internos del sistema.

Por su parte, para el tratamiento de los objetos básicos, el BOA simplemente proporciona la operación de registro de objetos.

Para los objetos replicados, el ROA proporciona además del método de registro de objetos, las operaciones `joinObject()` y `leaveObject()` para permitir añadir y eliminar réplicas a objetos replicados. Estas operaciones deberán ser ejecutadas por el dominio que posea la implementación que se desee añadir o eliminar del objeto replicado y su invocación no terminará hasta que todos los nodos hayan actualizado el estado de las réplicas en lo referente a su número y ubicación. Tanto la operación de registro como la operación de adición de una réplica admiten un argumento opcional[2] de tipo `Checkpoint`. Este tipo de objetos serán utilizados por las implementaciones de las réplicas para enviar los mensajes de actualización al resto de réplicas y para que el ORB sea capaz de extraer el estado de alguna de las réplicas mediante una invocación hacia arriba.

Al registrar un objeto en el ORB utilizando alguno de los adaptadores de Hidra, se crea el adaptador de objeto[3] correspondiente y éste se asocia al objeto. Algo similar ocurrirá con los

[2] Su valor puede ser nulo.

[3] Nótese que hablamos de *adaptadores de Hidra* para referirnos a cada uno de los hasta ahora tres adaptadores disponibles para toda aplicación: FOA, BOA y ROA. Por su parte, utilizamos el término *adaptador de objetos* para referirnos a los adaptadores que se crean y se asocian a cada uno de los objetos al registrarlos, tal y como puede

endpoints que se creen para conectar a los adaptadores; se creará un tipo de endpoint distinto (fijo, básico o replicado) según sea el tipo del adaptador servidor que desencadene su creación, incluso será posible crear diferentes tipos de adaptadores y endpoints según el modelo de replicación que se elija para cada objeto.

Como caso especial, el registro de un objeto replicado que incluya soporte de "checkpoints", crea dos adaptadores. Un adaptador servidor que permanecerá unido al objeto replicado y un adaptador servidor que se mantendrá conectado a la implementación de la interfaz de checkpoint. Adicionalmente ambos adaptadores mantendrán referencias cruzadas. Cuando se exporte una referencia al objeto replicado, se asociará un mismo endpoint servidor a los dos adaptadores. Las referencias a los objetos de tipo Checkpoint no se pueden exportar para impedir que un cliente pueda invocar este tipo de objetos sin haberse registrado previamente como réplica. De forma análoga al registro, la operación de unión al objeto replicado también creará los dos adaptadores servidores que vendrán a sustituir al adaptador cliente que se hubiera recibido previamente.

El soporte a los objetos replicados fuerza una modificación adicional, en este caso a nivel de stubs. Si las referencias a los objetos no replicados pueden ser meros punteros a las implementaciones, en el caso de los objetos replicados, tenemos que las referencias son punteros a un proxy que accede al adaptador, el cual redirigirá la invocación hacia la implementación local, o la enviará a una o más réplicas en función del modelo de replicación o de si se trata de un objeto de tipo `Checkpoint`. Gracias a este mecanismo de lazo, los adaptadores de objetos replicados pueden interceptar toda invocación que se efectúe sobre el objeto, incluso cuando la invocación parta de su propio dominio, actuando entonces en función de como sea requerido por el modelo de replicación.

La función principal de los adaptadores será realizar acciones de soporte al objeto que serán diferentes en función del tipo de objeto que estemos considerando. Por ejemplo, la acción de pasar una referencia de un nodo a otro será realizada por el adaptador del objeto, que la realizará de forma diferente en caso de tratarse de un objeto replicado, básico o fijo. Para el fijo, bastará con que se incluya en el mensaje a transmitir la propia referencia del objeto, mientras que para los objetos básicos habrá que ejecutar la parte correspondiente del protocolo de cuenta de referencias.

3.2.3 Estructura de las referencias

Los endpoints son las referencias internas de los objetos en Hidra. Se almacenan a nivel del núcleo y mantienen fundamentalmente el OID y el localizador del objeto. El OID es un identificador único de objetos. El OID se crea cuando se crea el primer endpoint servidor correspondiente al objeto. En ese instante, se asocia el OID al endpoint y permanecerá asociado a él mientras el endpoint exista. Una vez que se haya asignado el OID a un objeto, el OID lo identificará inequívocamente durante toda su vida. Los OIDs en Hidra están formados por el número del nodo donde se creó el objeto, el número de encarnación que tenía el nodo al crearse el objeto, el número de puerto donde se ha instalado el endpoint servidor y un número de versión que indica cuántas veces se ha reutilizado el mismo puerto durante la encarnación actual del nodo.

observarse en la figura 3.1.

Para objetos replicados, además del OID, los endpoints mantienen el número de configuración del objeto. Este número lleva la cuenta del número de veces que el objeto replicado ha sido reconfigurado. Decimos que un objeto replicado ha sido reconfigurado cuando se le añade o se le elimina una réplica o cuando cae algún nodo que tuviera alguna réplica del objeto.

Los endpoints de cada objeto se crean en el lado servidor la primera vez que los objetos son exportados hacia otros nodos. Por tanto, los OIDs se asignan a los endpoints la primera vez en la que se exporte el objeto.

El uso de OIDs únicos es un aspecto importante de diseño que simplifica la mayoría de los protocolos necesarios para proporcionar un adecuado soporte a objetos. En el caso de Hidra, esta decisión es perfectamente asumible con coste razonable (unos 64 bits) ya que la plataforma de destino es un cluster con un número reducido de nodos.

Localizadores

Además de los OIDs, los endpoints mantienen el localizador del objeto. Los localizadores apuntan a las implementaciones del objeto y son utilizados durante las invocaciones que efectúan los clientes de los objetos y para permitir el envío de invocaciones de actualización desde una réplica a las demás. Los contenidos y la estructura de los localizadores de los objetos dependen del tipo de objeto que se considere. Para objetos no migrables de implementación única, los localizadores no mantienen información alguna, pues la ubicación de la implementación del objeto se encuentra totalmente identificada con la información contenida en el OID del objeto. Para estos objetos, decimos que el OID es el localizador del objeto. A los localizadores que presentan esta estructura tan sencilla (nodo, encarnación, puerto, versión) los llamamos *localizadores planos*. Los localizadores planos apuntan a la ubicación de una única implementación de objeto.

Para el caso de objetos replicados, los localizadores de los endpoints están compuestos de una lista de localizadores planos. Cada uno de estos localizadores planos apunta a la ubicación de una de las réplicas del objeto. Por este motivo, decimos que los localizadores de los objetos replicados son localizadores estructurados y hablaremos entonces de la estructura de estos localizadores para referirnos a cuántos localizadores planos contienen y dónde apuntan. Diferentes modelos de replicación, como el modelo pasivo, el activo, o el coordinador-cohorte pueden tener localizadores con estructuras distintas. Es incluso posible tener localizadores con estructuras diferentes en función de diferentes implementaciones de un mismo modelo de replicación

Finalmente, como caso particular de los objetos replicados, tenemos a los objetos migrables de implementación única. Podemos ver a estos objetos como objetos replicados con una única implementación. Para estos objetos, los localizadores apuntan a la ubicación de la implementación. Una vez que el objeto migre, el localizador no tendrá porqué coincidir con el OID. Será incluso posible que diferentes endpoints cliente tengan diferentes localizadores para el mismo objeto. Esta situación podrá aparecer cuando un objeto migre y existan clientes que apunten a la ubicación previa del objeto y que posteriormente aparezcan clientes que reciban la referencia nueva.

En Hidra prevemos la probable necesidad futura de disponer de diferentes tipos de localizadores para optimizar disponibilidad frente a coste, o para optimizar los distintos protocolos que pueden requerir el uso de los localizadores tanto en situaciones ordinarias como frente a fallos.

Sin embargo, hasta la fecha tenemos tan sólo implementado un tipo de localizador para objetos replicados con replicación coordinador-cohorte. Para este modelo utilizamos localizadores que almacenan el conjunto completo de localizadores planos en los endpoints de las réplicas y un subconjunto de los localizadores planos en los endpoints cliente. Este subconjunto que mantienen los clientes vendrá determinado por las referencias que reciban.

El localizador se crea al crearse el primer endpoint correspondiente al objeto. En principio este localizador apuntará al propio nodo para indicar que en él se encuentra la implementación. Cuando se envíe una referencia, se enviará el localizador ejecutando su correspondiente operación de `marshal()`. El localizar será extraído por el nodo receptor ejecutando la operación de desempaquetado. Nótese que podrán existir localizadores cuyas operaciones de empaquetado y desempaquetado simplemente transfieran uno o dos localizadores planos, mientras que existirán otros tipos de localizadores cuyas operaciones de empaquetado y desempaquetado transferirán la lista completa de localizadores planos.

El endpoint principal

Los objetos móviles (objetos migrables de implementación única y objetos replicados) pueden tener en un mismo instante varios endpoints servidores. Entre el conjunto de endpoints servidores, en Hidra elegiremos a uno de ellos para que detente un rol especial cuando sea necesario. De esta forma, podremos diseñar protocolos asimétricos para los que haga falta un líder, sin que para ello sea necesario que cada protocolo tenga que elegir a su líder de forma independiente. A este endpoint especial lo llamamos *endpoint principal*. El endpoint principal sirve para varios propósitos en Hidra, pero fundamentalmente como mecanismo de rotura de simetría entre los endpoints servidores de un mismo objeto.

El endpoint principal de un objeto migrable de implementación única es el endpoint que realmente mantiene la implementación del objeto. Sin embargo, puede que existan otros endpoints servidores mientras termina la migración del objeto, para permitir la reconexión de los clientes. Por otra parte, cada endpoint cliente apuntará a un único endpoint servidor.

Para los objetos replicados, el endpoint principal será uno de los endpoints servidores que en principio estará conectado a una de las réplicas del objeto. Este endpoint especial actuará de líder, rompiendo la simetría entre los diferentes endpoints servidores cuando sea necesario. El endpoint principal de un objeto será inicialmente el primer endpoint que se cree para el objeto. Es decir, tanto el OID del objeto, como el endpoint principal apuntarán inicialmente al mismo endpoint. La necesidad de distinguir entre ambos identificadores radica en que el endpoint principal puede variar, mientras que el OID no variará en ninguna situación. El hecho de que se utilice al endpoint principal como líder y que éste pueda migrar mientras se le está utilizando añade complejidad a los diferentes protocolos que podamos construir con él. Por su especial relevancia, presentaremos el protocolo de migración del endpoint principal en el apartado 3.3.2 donde describiremos el protocolo y sus interrelaciones con otros protocolos que puedan estar utilizando al líder al mismo tiempo en que se está cambiando.

Con la intención de proporcionar genericidad al concepto de endpoint principal, diremos que el endpoint servidor de cualquier objeto no migrable de implementación única es el endpoint principal. De esta forma podemos definir el término de endpoint principal diciendo que todo objeto tiene un endpoint principal y que este es uno de los endpoints servidores del objeto.

El localizador principal

Los localizadores planos pueden ser inexactos para el caso de objetos migrables o replicados. Esta posible inexactitud de los localizadores se debe a que los nodos pueden fallar y a que se pueden ejecutar tareas administrativas sobre los objetos para añadir o eliminar réplicas. El localizador plano más fiable que tiene todo endpoint es el *localizador principal*. Si el endpoint principal no migrara nunca, el localizador principal siempre apuntaría al endpoint principal. Sin embargo si el endpoint principal migra, el localizar principal de algunos endpoints puede quedar obsoleto. Para permitir la reconexión desde localizadores principales obsoletos al nuevo endpoint principal, utilizaremos *encaminadores* (forwarders) [43] que dejaremos en la ubicación previa del endpoint principal.

Para objetos no migrables de implementación única, el OID siempre coincide con el endpoint principal y como éste no puede migrar, el localizador principal de cualquier endpoint que haga referencia a estos objetos siempre apuntará al endpoint principal. Es decir, para este tipo de objetos los endpoints simplemente deben mantener el OID, que servirá como OID y como localizador principal exacto.

Para objetos migrables de implementación única, los endpoints mantienen como localizador únicamente el localizador principal, que podrá ser inexacto en caso que ocurra alguna migración. Por último, para los objetos replicados, sea cual sea el modelo de replicación que se considere, siempre existirá el localizador principal, que en ausencia de migraciones del endpoint principal, será exacto.

3.2.4 Operaciones básicas sobre los objetos

El soporte a los objetos de todo ORB se construye con la idea de permitir las invocaciones remotas sobre objetos. Para habilitar esta función principal, se proporcionan un conjunto básico de operaciones sobre los objetos, que permiten a los programadores tratar las referencias remotas a objetos de forma similar a como tratarían las referencias locales o punteros. Estas operaciones básicas son: la creación y registro de objetos en el ORB, la invocación de objetos, el paso y recepción de referencias a objetos como efecto lateral de las invocaciones, la duplicación y el descarte de referencias. En el resto de este apartado describiremos el funcionamiento de estas operaciones con el objetivo de ilustrar las relaciones existentes entre los distintos niveles del ORB.

Creación y registro de un objeto

Crear un objeto Hidra es una tarea muy sencilla; el único requisito que se debe satisfacer consiste en que los objetos hereden de la clase de objetos `HidraObject`. Esta clase incorpora ciertas operaciones que podrán ser reescritas por el programador, pero que ya proporcionan una mínima funcionalidad. Se trata de operaciones que invocará el propio ORB de forma transparente al programador. Como ejemplo típico de operación incluida en la clase `HidraObject` tenemos el método `unreferenced()` que será invocado por el ORB para aquellos objetos que sean residuos y que hayan solicitado ser notificados cuando ocurra esta situación.

La mera creación de un objeto no le permite al ORB gestionarlo. Para lograrlo, el programador debe registrar el objeto en el ORB. El registro, se efectuará mediante uno de los adaptadores

de Hidra: el FOA, el BOA o el ROA. La operación de registro crea un adaptador (o dos para el caso del ROA) de objetos del tipo correspondiente y se asocia el adaptador con la implementación que se está registrando. Una vez que se tiene asociado el adaptador con la implementación, el adaptador interceptará la mayoría de acciones que se efectúen sobre el objeto, realizando tareas diferentes en función del tipo de adaptador que se considere. Este modo de operar, en el que un adaptador intercepta las acciones que se realizan sobre los objetos, es similar al que podemos encontrar en sistemas tales como Spring [93] o Solaris-MC [8] donde los *Subcontracts* en el primer caso, o los *handlers* en el segundo juegan un papel similar al de los adaptadores de objeto de Hidra.

Invocación de objetos

Un dominio que disponga de una referencia a un objeto remoto puede invocarlo. Para ello, se garantiza que exista un camino entre la referencia y la implementación remota del objeto. En Hidra, la referencia tendrá asociado un adaptador de objetos cliente, el cual estará conectado a un endpoint cliente. En el endpoint cliente tenemos el localizador del objeto, que contendrá al menos el localizador principal y una función `getInvoLocation()` que retorna lo que conocemos como *localizador de invocación*. El localizador de invocación constituye el destino al que invocar. El destino será en general[4] un localizador plano que apunta a un endpoint en particular. El camino de la invocación alcanzará por tanto a cierto endpoint servidor que tendrá asociado un adaptador de objetos servidor, a través del cual se podrá acceder a la implementación del objeto.

Invocar implica enviar mensajes de invocación y esperar mensajes de respuesta. En el mensaje de invocación se habrán empaquetado previamente los argumentos de la operación a invocar. Cuando la invocación alcance el destino, se desempaquetarán los argumentos, se accederá a la implementación a través del adaptador servidor, y el adaptador esperará la respuesta. Cuando la invocación termine, se efectuará el camino de retorno. Se empaquetarán los argumentos de salida, y se enviará el mensaje de respuesta al iniciador de la invocación. El endpoint cliente que inició la invocación recibirá el mensaje de respuesta y se lo cederá al adaptador cliente, que procederá a desempaquetar los argumentos.

Como puede apreciarse, invocar un objeto remoto exige tener construido el camino entre referencia e implementación. En Hidra este camino se construye mediante una de dos formas:

- La primera utilizando objetos fijos. Estos objetos al ser registrados crean en el lado servidor, tanto el adaptador fijo de objetos, como el endpoint fijo servidor. Esta particularidad la podemos observar en la operación `registerObject()` del FOA (ver figura 3.2) en la que se explicita la ubicación donde se desea alojar al endpoint. Por su parte la operación `getReference()` permite obtener una referencia a un objeto fijo para la que se construirá la parte correspondiente del camino en el cliente, es decir el endpoint y el adaptador fijo cliente.

- La segunda opción que construye el camino entre referencias e implementaciones la tenemos en el paso de referencias.

[4]El hecho de que permitamos la posibilidad de retornar localizadores estructurados, deja abierta la opción de que implantemos el modelo de replicación activo.

Paso de referencias

El paso de referencias ocurre como efecto lateral de las invocaciones. En Hidra, de forma similar a como ocurre en todo ORB, cuando se invoca una operación que tiene algún argumento de tipo objeto, se desencadena el proceso de paso de referencias. Si se invoca una operación que recibe objetos como argumentos de entrada, se estarán enviando referencias, mientras que si invocamos operaciones que retornan referencias a objeto como argumentos de salida o como valores de retorno, se estarán recibiendo referencias. Las referencias a objeto que se envían se colocan en el mensaje de la invocación que se transmitirá al objeto remoto, análogamente, al recibirse el mensaje de respuesta, se podrán extraer de él las referencias a objeto que hubieran sido colocadas por el dominio remoto.

La forma de enviar referencias a objeto o de recibirlas se realiza con la colaboración del correspondiente adaptador del objeto. De hecho, el compilador de interfaces de Hidra habrá generado para cada operación que tenga argumentos de tipo objeto, llamadas a las operaciones `marshal()` o `unmarshal()` del adaptador. La operación `marshal()` se utilizará para indicarle al adaptador que debe transmitir la referencia del objeto al que representa, mientras que la operación `unmarshal()` se utilizará para realizar la acción opuesta. El delegar al adaptador esta tarea de empaquetado y desempaquetado de referencias, permite realizar acciones diferentes en función del tipo de adaptador de que se trate o de si se está actuando en el lado servidor o en el cliente.

El paso de referencias merece especial atención la primera vez que se realiza desde el lado servidor. Si un objeto (que no sea fijo) registrado en el ORB no ha salido de su dominio, tendrá tan sólo el correspondiente adaptador servidor asociado a la implementación. Cuando se envíe la referencia, se invocará la operación `marshal()` del adaptador. Esta operación, al comprobar que no existe endpoint servidor asociado, creará[5] uno ubicándolo en la tabla de endpoints. El tipo de endpoint que se cree dependerá del tipo de objeto que se esté enviando, así, para los objetos básicos se creará un endpoint servidor básico, mientras que para un objeto replicado se creará un endpoint servidor replicado.

Una vez que se ha creado el endpoint servidor, se procede de igual forma a como se procederá las siguientes ocasiones en las que se transmita una referencia al objeto. La operación `marshal()` del adaptador incluirá en el mensaje de la invocación el identificador del tipo del adaptador y el número de endpoint local, para posteriormente hacer lo que el adaptador estime conveniente para enviarse a un dominio remoto. Cuando todos los adaptadores de todos los objetos que se envían al dominio remoto hayan sido empaquetados en el mensaje de la invocación, se le cederá el control del mensaje al endpoint involucrado en la invocación en curso. Este endpoint recorrerá la lista de endpoints contenidos en el mensaje, invocando el método `marshal()` de cada uno de ellos. Esta operación a nivel de endpoint hará algo similar a la operación análoga del adaptador, es decir, incluirá el identificador del tipo del endpoint y el endpoint se introducirá a sí mismo en el mensaje. Este proceso de auto empaquetado incluirá al menos el OID del objeto.

Cuando el mensaje llegue al destino se extraerán todas las referencias a objeto. Para cada

[5] Más que crearse el endpoint en este instante, se anota una indicación en el mensaje que se está construyendo para reflejar que debe crearse el correspondiente endpoint. Cuando el mensaje llegue al nivel de los endpoints, se crearán todos los endpoints cuya creación figure pendiente en el mensaje.

una de ellas se extraerá el OID y el tipo del endpoint del mensaje. Con el OID se comprobará si la referencia ya existía con anterioridad. De no existir se creará un endpoint del tipo indicado y se invocará a su correspondiente método `unmarshal()`. En caso de que el endpoint ya existiera, simplemente se invocará a su método `unmarshal()`. Esta operación de desempaquetado realizará la acción opuesta a la que se realizara en la operación `marshal()` efectuada al enviar la referencia. Finalmente se invocará al método `unmarshal()` del adaptador que habrá sido previamente creado en función de su tipo de forma similar a como se creó el endpoint.

La flexibilidad que proporciona esta arquitectura, permite tratar de forma distinta a los distintos tipos de objetos. Por ejemplo, los objetos básicos incluirán cuenta de referencias en las operaciones `marshal()` y `unmarshal()`, que no figurará en las operaciones análogas de los objetos fijos. Por su parte, los objetos replicados podrán transmitir de unos nodos a otros toda la información relativa al modelo de replicación, los localizadores de la forma que estimen conveniente, el nodo o nodos que se desee invocar de forma preferente, etc.

Descarte de referencias

Utilizar referencias a objeto consume recursos. Si el ORB estuviera continuamente proporcionando referencias a los dominios y los dominios nunca las liberaran, poco a poco el sistema agotaría sus recursos. Para evitar esta situación se proporciona la operación de descarte de referencias. En Hidra el descarte de referencias se realiza mediante la operación `release()` de la interfaz Hidra. Esta operación accederá al adaptador del objeto e invocará a su propia operación `release()` con lo que nuevamente tendremos la posibilidad de ejecutar acciones distintas para los diferentes tipos de objetos. Por ejemplo, el descarte de una referencia a cierto objetos replicados o básicos, puede ocasionar que se ejecute parte del protocolo distribuido de cuenta de referencias, mientras que el descarte de referencias a objetos fijos sólo tendrá un efecto local.

En todo caso, si el descarte se realiza en el lado cliente, y este descarte provoca que no queden referencias asociadas al adaptador, se eliminará el adaptador y el endpoint, con la consiguiente liberación de recursos. Por su parte los descartes en el lado servidor, sólo afectarán a los objetos no fijos. En caso de descartarse la última referencia se habilitará la posibilidad de que la implementación reciba la notificación de objeto no referenciado. En caso contrario diremos que el dominio servidor mantiene referencias locales a su objeto, lo cual imposibilita que aparezca la situación de objeto no referenciado.

3.3 Operaciones sobre objetos replicados

Las operaciones básicas sobre los objetos y especialmente si los objetos no son replicados, resultan relativamente sencillas de implementar. Sin embargo la aparición de los objetos replicados plantea la necesidad de contemplar al menos dos operaciones adicionales: la adición de réplicas a un objeto replicado y la eliminación de réplicas. Además, se vuelve imprescindible construir mecanismos que reconstruyan el estado de los objetos replicados en caso de fallos y que reintenten las invocaciones sobre las réplicas vivas de forma transparente a los clientes. En esta sección describiremos el soporte que ofrece Hidra a estas operaciones centrándonos en todos aquellos aspectos que sean independientes del modelo de replicación.

3.3.1 Adición y eliminación de réplicas

La aproximación más elegante para implementar los protocolos administrativos de gestión de objetos replicados consiste en utilizar el propio protocolo de invocación a objetos replicados para hacerlo.

Un ejemplo de este tipo de sistemas lo tenemos en Arjuna [82, 83]. Este sistema proporciona acciones atómicas y persistencia como el mecanismo básico de replicación y utiliza este mismo mecanismo para implementar los protocolos de adición y eliminación de réplicas para objetos replicados. En este caso, se considera a la adición y eliminación de réplicas como otra acción que necesita ser ejecutada atómicamente por todas las réplicas activas del objeto.

Los sistemas que utilizan protocolos de comunicación ordenada a grupos como la infraestructura básica de comunicación, también tienden a utilizarlo para añadir y eliminar réplicas de los objetos replicados. Este es el caso de gran variedad de sistemas que utilizan *sincronía virtual* o *sincronía virtual extendida* [44, 95]. En estos sistemas, las peticiones para añadir o eliminar réplicas son enviadas al objeto replicado utilizando el mismo protocolo de comunicación a grupos que se utiliza para las operaciones ordinarias de los objetos. Esto asegura que todas las réplicas recibirán los mensajes de reconfiguración de los objetos en el mismo orden, permitiendo que se actualicen consistentemente sus estados.

En Hidra, los ROIs y el HCC pueden utilizarse para añadir y eliminar réplicas de los objetos replicados. Aunque este esquema no ha sido explícitamente probado, parece tratarse de una buena aproximación para los modelos coordinador-cohorte y pasivo. Siguiendo esta aproximación, la adición y eliminación de una réplica consistiría en una invocación ordinaria sobre el objeto replicado. Para ello, se utilizaría tanto el mecanismo de invocaciones fiables (ROIs), como el control de concurrencia distribuido (HCC) [108] de Hidra, con los que se estarían serializando las operaciones de gestión de réplicas junto con las demás operaciones que se efectúen sobre el objeto.

Adición de réplicas

El uso de los protocolos de comunicación a grupos o de los mecanismos de invocaciones atómicas disponibles según el modelo de replicación, sirven para garantizar que todas las réplicas ejecuten la operación de adición de una nueva réplica en el mismo orden respecto del resto de operaciones que se efectúen sobre el objeto. Gracias a esta garantía, lograda según el modelo de replicación por distintos medios, la nueva réplica tendrá una copia actualizada del estado del objeto antes de que cualquiera de las réplicas reciba nuevas invocaciones.

Sin embargo en Hidra tenemos la necesidad adicional de disponer de un tipo específico de objetos replicados: objetos replicados sin consistencia en el estado. Ejemplos de estos objetos los tenemos en los `CObj`, en los `RoiId` y en los `TObj` cuya función ya describimos en el apartado 2.3.6 dedicado al soporte Hidra del modelo coordinador-cohorte. Para estos objetos, el protocolo de adición de réplicas que utilizamos resulta relativamente sencillo:

1. El endpoint que se pretende unir al objeto replicado le envía un mensaje `JOIN` al endpoint principal.

2. El endpoint principal actualiza su localizador, incrementa el número de configuración del

objeto y difunde el mensaje al resto de réplicas incluyendo el nuevo número de configuración del objeto.

3. Los endpoints servidores que reciben el mensaje, actualizan el número de configuración del objeto y el localizador y responden al principal.

4. Cuando el endpoint principal recibe respuesta de todos los endpoints, contesta al endpoint que inició la petición.

5. Cuando el endpoint que inició la petición recibe el mensaje, da por finalizada la unión al objeto.

El tratamiento de los casos de fallo de este protocolo tampoco resulta demasiado complejo. Si falla un endpoint que no sea el principal o el iniciador del protocolo, lo único que debe hacerse es eliminar todo referencia al endpoint caído de los localizadores situados en los endpoints vivos. Esta fase de actualización de los localizadores se realiza durante la reconfiguración del ORB. Por su parte, si cae el endpoint principal, se realizarán dos fases en la reconfiguración del ORB. Primero se elegirá a un nuevo endpoint principal y después se permitirá al endpoint que inició la operación de adición, reintentarla. Al tratarse de una operación idempotente, no es necesario construir mecanismos que detecten la repetición de la misma. En lo que respecta a la elección del nuevo endpoint, ésta se realizará de forma determinista y local a cada endpoint servidor.

Eliminación de réplicas

A diferencia de la adición de réplicas, la eliminación de réplicas no genera inconsistencias en el estado de los objetos replicados aunque las distintas réplicas reciban la notificación de la eliminación en órdenes diferentes. Lo único que se debe exigir es que ninguna réplica reciba la orden de eliminar una réplica si esta réplica todavía está procesando invocaciones. El protocolo de eliminación de réplicas de Hidra es el siguiente:

1. Se bloquean futuras invocaciones sobre la réplica a eliminar, incluyendo los mensajes de actualización.

2. Se espera a que terminen las invocaciones en curso.

3. Se envía un mensaje LEAVE al endpoint principal. Si el endpoint principal es justamente el que se pretende eliminar, el endpoint actuará como si lo hubiera recibido.

4. Al recibir el mensaje LEAVE el endpoint principal incrementa el contador de configuraciones del objeto, elimina al endpoint del localizador y difunde el mensaje LEAVE a las demás réplicas, incluyendo el número de configuración del objeto.

5. Cada endpoint servidor que recibe el mensaje, elimina al endpoint correspondiente de su localizador y almacena el nuevo número de configuración del objeto. Una vez hecho esto contesta al endpoint principal.

6. El endpoint principal al recibir todas las confirmaciones, responde al endpoint que inició la eliminación.

7. El endpoint que inició la invocación elimina la réplica y elimina el endpoint (si se trata de un endpoint distinto del principal). Si le llegara alguna invocación al nodo que tenía este endpoint a partir de este instante, se responderá al origen de la invocación con una excepción que fuerce al reintento sobre otro destino.

Tal y como sucedía con el protocolo de adición de réplicas, un fallo en el nodo que mantuviera al endpoint principal provoca que se reinicie la petición de eliminación sobre el endpoint principal que haya sido elegido en sustitución del anterior. De igual manera, si cae un endpoint que no sea el principal ni el iniciador, bastará con que se eliminen todas las referencias a este endpoint en el resto del objeto.

Derivado de la ejecución posiblemente concurrente de múltiples adiciones y eliminaciones de réplicas sobre un mismo objeto, puede ocurrir que distintas réplicas reciban dichas operaciones en diferente orden. Si bien no es relevante que el orden no se respete respecto a las operaciones ordinarias del objeto, existen determinados casos que deben tratarse con especial cuidado. Supóngase por ejemplo que se pretende añadir cierta réplica y acto seguido se pretende eliminar. En este caso especial, si a determinado endpoint le llega antes la eliminación que la adición podría quedarse sin eliminar la réplica. Para prevenir estos casos los mensajes de eliminación que lleguen desordenados respecto del número de configuración del objeto, serán temporalmente bloqueados por los endpoints hasta que reciban y procesen los mensajes correspondientes a los números de configuración que falten por cubrir.

En lo referente a los posibles fallos de caida que puedan ocurrir durante la ejecución de este protocolo y de los demás protocolos que veremos en este capítulo, la aproximación que tomamos consiste en reconstruir su estado durante la reconfiguración del cluster. La reconstrucción de las referencias a objeto, como núcleo de toda la reconfiguración del ORB, la veremos en los apartados 4.5 y 4.6.6 del capítulo 4.

Por último, merece una atención especial el caso de que se pretenda eliminar la réplica asociada al endpoint principal. Cuando ocurra este caso, no se podrá eliminar el endpoint sin más, puesto que los demás endpoints lo estarán utilizando para ejecutar protocolos distribuidos. Para solucionar este caso específico y de especial relevancia para el protocolo de cuenta de referencias que detallaremos en el capítulo 4, incluimos el protocolo de migración del endpoint principal.

3.3.2 Migración del endpoint principal

Puede parecer razonable el intentar mantener el mismo endpoint principal durante toda la vida de los objetos. En este caso, el endpoint podría tener una réplica asociada o no tenerla. Por ejemplo, cierta tarea administrativa sobre el objeto podría acarrear que se eliminara la réplica situada sobre el endpoint principal, sin que ello provocara que se eliminase el endpoint. Si el endpoint principal almacenara en su localizador en todo instante la ubicación de los demás endpoints servidores, podría seguir utilizándose para coordinar los protocolos que se deban ejecutar sobre el objeto independientemente de si dispone de réplica.

Sin embargo, es conveniente estudiar los motivos que pueden llevar a un usuario a migrar un objeto de un nodo a otro, o a eliminar una réplica en particular de un objeto replicado. Una de las situaciones más claras aparece en las paradas planificadas de nodos. Es sobradamente conocido que el número de paradas de nodos debidas a tareas administrativas, supera con creces al número de paradas sufridas por los nodos debidas a fallos. Un usuario puede pretender detener un nodo para ampliar su memoria, sustituir un disco, añadir cualquier tipo de periférico, etc. En estos casos, al tratarse de una función planificada y no de un fallo, se le exigirá al sistema que proporcione mecanismos que permitan detener el nodo sin que el resto del sistema se degrade de forma significativa y sin que la parada ocasione indisponibilidad de los servicios. Es decir, un sistema altamente disponible debe disponer de mecanismos para detener la actividad de un nodo de forma habitualmente más eficiente de la que resultaría si sencillamente se detuviera el nodo simulando o inyectándole un fallo.

El mecanismo básico de Hidra para este tipo de paradas y en general el mecanismo básico que permitirá construir protocolos de migración de objetos y de eliminación de réplicas, es la migración del endpoint principal. Adicionalmente, todo protocolo que se construya utilizando al endpoint principal para romper la simetría, deberá considerar su funcionamiento concurrente con el protocolo de migración del endpoint principal.

El protocolo de migración del endpoint utiliza tres variables para representar su estado:

1. El localizador principal que posee todo endpoint.

2. Un contador que llevará la cuenta de cuántas migraciones del endpoint principal ha sufrido un objeto. A este contador lo llamaremos el *contador de migraciones*.

3. Una variable `rol` que podrá tomar uno de los siguientes valores: $MAIN$, $MOVING$, $MOVED$, REP, $MAINREQ$ o CEP. Diremos que los endpoints clientes son endpoints que desempeñan el rol CEP, los endpoints servidores que no sean el endpoint principal se encuentran con rol REP o $MAINREQ$, mientras que un endpoint servidor que ha sido o que es el endpoint principal puede encontrarse con rol $MAIN$, $MOVING$ o $MOVED$. Un endpoint servidor que no sea el endpoint principal se encontrará habitualmente con rol REP hasta que él mismo solicite convertirse en el endpoint principal. Si lo hace, pasará al rol $MAINREQ$ hasta que el endpoint principal le conceda la migración. El estado $MAINREQ$ sirve para impedir que el mismo endpoint solicite convertirse en el principal varias veces de forma consecutiva. Un endpoint se encontrará con rol $MAIN$ si se trata del endpoint principal, con rol $MOVING$ si estaba con rol $MAIN$ y algún otro endpoint servidor le ha solicitado sustituirle como endpoint principal. Por último, un endpoint servidor se encuentra con rol $MOVED$ cuando el endpoint que solicitó convertirse en endpoint principal ha confirmado que ha asumido el rol de endpoint principal. Por tanto una migración del endpoint principal provocará que el endpoint principal antiguo pase de rol $MAIN$ a rol $MOVING$ y de este rol, a rol $MOVED$. Por su parte el endpoint que adquiere el rol de endpoint principal habrá pasado del rol REP a $MAINREQ$ y de este rol, al rol $MAIN$.

El protocolo parte de una situación inicial en la que el endpoint principal se encuentra con rol $MAIN$ y los demás endpoints con rol REP o CEP dependiendo de si se trata de endpoints

servidores o clientes. A partir de este estado inicial, el funcionamiento del protocolo es el siguiente:

1. Cualquier endpoint servidor (endpoint que se encuentre como REP) puede enviarle un mensaje al endpoint principal cuando desee convertirse en el endpoint principal. Para ello le envía un mensaje de tipo $MOVE\_INI(l)$ al endpoint apuntado por su localizador principal, indicando como argumento el localizador que apunta a su propio endpoint (l). Al hacerlo, pasa automáticamente a rol $MAINREQ$. Cuando el mensaje llegue al endpoint destino éste puede que se encuentre desempeñando cualquiera de los roles $MAIN$, $MOVING$ o $MOVED$.

   (a) Un endpoint recibe un mensaje de tipo $MOVE\_INI(l)$ encontrándose con rol $MAIN$. Al recibirlo, pasa a rol $MOVING$ y toma como localizador principal el localizador recibido en el mensaje. Nada más recibir el mensaje, incrementa el contador de migraciones y contesta al nuevo endpoint principal con un mensaje $MOVE\_ACK_m$, donde m es el nuevo contador de migraciones.

   (b) Si un endpoint con rol $MOVING$ recibe un mensaje de tipo $MOVE\_INI(l)$, significa que algún endpoint servidor le está solicitando al antiguo endpoint principal el convertirse en el nuevo endpoint principal cuando una migración previa todavía no ha concluido. En este caso, el mensaje es retenido y encolado en el endpoint que recibe el mensaje hasta que se transite al rol $MOVED$.

   (c) Si un endpoint recibe un mensaje de tipo $MOVE\_INI(l)$ encontrándose con el rol $MOVED$, significa que un endpoint servidor le está solicitando convertirse en endpoint principal a un endpoint que fue el principal, pero que ya no lo es por haber migrado. En este caso el endpoint que recibe el mensaje envía un mensaje $MOVE\_NAK(l)_m$ al endpoint que ha pedido convertirse en el principal enviando como argumentos el localizador principal actual junto con el número de migración del objeto.

   Esta misma acción se realizará cuando un endpoint transite del rol $MOVING$ al rol $MOVED$. En este caso se enviarán los mensajes $MOVE\_NAK(l)_m$ a todos los endpoints que solicitaron convertirse en el principal mientras éste se encontraba con el rol $MOVING$.

2. Si un endpoint recibe un mensaje de tipo $MOVE\_ACK_m$, pasa del rol $MAINREQ$ al rol $MAIN$, actualiza su contador de migraciones con el contador presente en el mensaje y contesta al origen del mensaje con $MOVE\_END$.

3. Si un endpoint recibe un mensaje de tipo $MOVE\_NAK(l)_m$, actualiza su localizador y su número de migración. Posteriormente vuelve a enviar el mensaje $MOVE\_INI(l)$ pero esta vez al endpoint apuntado por el nuevo localizador principal.

Como puede observarse, el protocolo de migración del endpoint principal de Hidra es un sencillo protocolo distribuido que presenta cierta sincronía. Es síncrono en el sentido que los mensajes $MOVE\_INI(l)$ que llegan a un endpoint son encolados a la espera de que finalicen las migraciones iniciadas con anterioridad. Sin embargo, esta sincronía parece razonable puesto

que en general la migración del endpoint principal será una operación que bloqueará al código que la inicie, por lo que la espera a la compleción de la migración ocurrirá de igual forma. Adicionalmente, parece razonable partir de la hipótesis de que no ocurrirá con elevada frecuencia la situación en que dos o más endpoints soliciten al mismo tiempo convertirse en el endpoint principal.

Reconexión de las referencias

Cuando migra el endpoint principal, pueden aparecer referencias cuyos localizadores principales permanezcan un tiempo arbitrario apuntando al endpoint antiguo. Por este motivo, en principio no se puede permitir que un endpoint que haya sido principal, pase del rol $MOVED$ al rol REP. Esta transición sería deseable cuando se tuviera la certeza de que no existen referencias que apunten a la ubicación antigua del endpoint principal. Mediante el protocolo de cuenta de referencias que detallaremos en el capítulo dedicado a recolección de residuos, en el apartado 4.6.5 lograremos detectar esta situación, con lo que por el momento asumiremos que será detectada.

Por otra parte, cada vez que un cliente reciba una referencia a un objeto replicado que ya tuviera con anterioridad (encontrada por el OID), comprobará el contador de migraciones que recibe de la referencia con el contador que poseyera de antes. Si el contador recibido es mayor, se tomará como localizador principal el contenido en la referencia, mientras que éste no cambiará en otro caso.

Protocolos para objetos replicados y migrables

Cada protocolo que se construya para objetos replicados o migrables deberá considerar para cada mensaje propio del protocolo, el estado en que se encuentre el endpoint principal al recibirlo. El caso más corriente será que los mensajes alcancen al endpoint principal cuando este se encuentre con el rol $MAIN$. Cuando el endpoint reciba mensajes en este estado, ejecutará la parte correspondiente del protocolo que debe ejecutar por ser el endpoint líder. Para evitar inconsistencias en el protocolo, será conveniente bloquear todo mensaje $MOVE\_INI(l)$ hasta que el protocolo que utiliza al líder finalice.

Por contra, si un mensaje alcanza un endpoint cuando éste se encuentra con el rol $MOVED$, se podrá optar por que se reenvíe el mensaje al nuevo endpoint principal (al endpoint contenido en el localizador principal) o por responder al emisor del mensaje indicándole la ubicación del nuevo endpoint principal. La elección de una de estas dos alternativas dependerá del protocolo en particular que se esté implantando. De igual forma, un mensaje dirigido a un endpoint principal puede alcanzarlo con rol $MOVING$. En este estado, cada protocolo podrá elegir por implantar una opción entre al menos las siguientes tres:

1. El endpoint bloquea y encola el mensaje recibido hasta que el endpoint transite al rol $MOVED$, hecho que ocurrirá por el propio funcionamiento del protocolo de migración del endpoint.

2. El endpoint responde al emisor del mensaje para que reintente sobre el mismo endpoint después de que transcurra cierto intervalo de tiempo (polling).

3. El endpoint redirige el mensaje al endpoint que ha pedido convertirse en el principal. En este caso el mensaje incluye un mensaje de tipo $MOVE\_ACK$ para prevenir el caso de que este mensaje llegara antes al destino que el mensaje $MOVE\_ACK$ enviado por el propio protocolo de migración del endpoint. De esta forma, el endpoint que reciba este mensaje pasará a tener el rol $MAIN$ de igual forma a como hubiera ocurrido de recibir el mensaje enviado por el protocolo de migración.

3.4 Caída de un nodo

La caída de un nodo afecta al estado que mantienen los demás para cada una de sus referencias y objetos. Concretamente, aparecerán endpoints cuyo estado deberá modificarse. También será necesario detener, y en su momento reintentar los protocolos que estuvieran en funcionamiento en el momento de la caída.

Al caer un nodo se ejecutarán pasos síncronos de reconfiguración que permitirán que el ORB descarte todo el estado que haga referencia al nodo caído y que restablezca los invariantes de sus protocolos. Se procederá a elegir un nuevo endpoint principal para aquellos objetos cuyo endpoint principal hubiera caído y se procederá a reconectar a los clientes obsoletos con los endpoints principales. Después de esto se permitirá reanudar el funcionamiento de los protocolos, incluyendo el reinicio de aquellos que lo necesiten para hacer frente al fallo. En todo caso, los pasos de reconfiguración del ORB que aseguran un estado consistente después del fallo, tienen una relación fundamental con el protocolo de cuenta de referencias. Este protocolo lo detallaremos en todas sus variantes en el capítulo 4, mientras que los pasos de reconfiguración que se ejecutan para reconstruir las referencias de los objetos y la propia cuenta de referencias los mostraremos en ese mismo capítulo, en los apartados 4.5 y 4.6.6.

3.5 Trabajo relacionado

Muchos sistemas proporcionan soporte a objetos distribuidos de forma más o menos elaborada. Desde la enunciación del principio del proxy [129], hasta la elaboración completa del estándar CORBA en sus sucesivas versiones [113], han ido apareciendo multitud de sistemas que han proporcionado el concepto de objeto distribuido como la pieza básica con la que construir aplicaciones. Entre los sistemas más parecidos a Hidra tenemos Spring [67, 93] y posteriormente Solaris-MC [73, 8]. Spring es un sistema distribuido orientado a objetos que se estructura internamente de forma similar a como lo hace Hidra. En Spring aparecen los *Subcontracts* [66] que juegan un papel similar a los adaptadores de objeto de Hidra. Por su parte en Solaris-MC el papel de los adaptadores lo realizan los *handlers* y la funcionalidad de los endpoints es similar a la de las *xdoors* de Solaris-MC. La diferencia fundamental de Hidra respecto a estos sistemas radica en la estructura de las referencias y en su propósito de ofrecer un soporte mínimo a la replicación de objetos independiente del modelo de replicación.

Un ejemplo de soporte a objetos replicados se realizó sobre Spring [7], donde el soporte estaba basado en un *Subcontract* capaz de almacenar referencias a los objetos que forman el objeto replicado, sin embargo en este sistema no se acabó por incluir un soporte completo a la detección de fallos. Por contra, en Hidra se ha optado por bajar a nivel de endpoints esta

gestión, para incrementar la eficiencia de los protocolos que afectan a los objetos replicados y en particular para proporcionar un adecuado mecanismo de recolección de residuos.

Pocos sistemas, al margen de los que hemos descrito, y de las descripciones de los patrones de diseño de proxy [129], adaptador de objetos [56] y grupo de objetos [87] ofrecen detalle en cuanto a la forma de estructurar el soporte a los objetos dentro de un ORB. Los ORBs con soporte a replicación suelen ser sistemas comerciales poco documentados, o sistemas donde el énfasis se produce a nivel de la descripción de los protocolos distribuidos más que en la maquinaria de soporte a los objetos. En otros casos [62] las discusiones se centran en la complejidad de implantar protocolos de replicación o en la conveniencia o no de utilizar protocolos de comunicación ordenada.

Las descripciones con más detalle, aparecen al respecto de la estructura interna de las referencias a objeto cuando se trata de combinarlas con un mecanismo de recolección de residuos. Así por ejemplo tenemos SOR y SOUL [120] donde se describen los *entry points* y *exit points* que juegan un papel similar al que realizan los endpoints de Hidra, o los *Network Objects* [17] cuya estructura interna está enfocada a la recolección de residuos. Sin embargo, ninguno de estos sistemas describe el soporte para objetos replicados.

Finalmente, CORBA [113] ha servido desde su aparición a comienzos de los 90 hasta la actualidad, entre otras cosas, como recopilatorio del estado del arte en los sistemas de objetos distribuidos. Los conceptos de adaptadores de Hidra, los localizadores de objeto, o el propio modelo de objetos de Hidra han sido fuertemente influenciados por el estándar. En lo referente a tolerancia a fallos, CORBA [114] sugiere recientemente la posibilidad y la conveniencia de proporcionar soportes para la construcción de objetos distribuidos que permitan la adopción de diferentes modelos de replicación según se desee.

Capítulo 4

Recolección de residuos

“Desarrollo sostenible” es aquel que permita que se satisfagan las necesidades del presente sin comprometer la capacidad de las futuras generaciones para satisfacer las propias.

Comisión Mundial del Medio Ambiente y del Desarrollo.

Contenidos del capítulo

4.1 Introducción

Hidra es una arquitectura para la construcción de aplicaciones y sistemas que deben ejecutarse sin interrupción, incluso en presencia de fallos. Si no se detectan y se recolectan los residuos que vayan apareciendo, los recursos del sistema se irán agotando poco a poco, hasta el punto extremo en que todo el sistema se podría quedar bloqueado por no disponer de recursos para continuar su ejecución.

Si el programador es el responsable de detectar los residuos, se le estaría trasladando a cada aplicación un problema de solución costosa, compleja de implementar y tendente a errores. Cada programador optaría por un mecanismo diferente y en algunos casos se implantarían soluciones erróneas. Al tratarse de un sistema distribuido donde deben cooperar en la recolección de los residuos tanto los servidores como las aplicaciones clientes de tales servicios, deberían cooperar en la implantación del mecanismo de recolección de residuos tanto los programadores de los servicios, como los programadores de los clientes de cada servicio. El coste derivado de su implantación en cada aplicación sería difícilmente optimizable de forma global y su corrección estaría fuera del control de la arquitectura, con lo que finalmente lo más probable es que continuaran apareciendo residuos.

La alternativa que adoptamos en Hidra es incluir en el propio ORB un sistema de recolección y de detección de residuos, que libere al programador de la tarea de detectar residuos. En Hidra, la entidad principal a detectar como residuo son los objetos de aplicación, sea cual sea el dominio en el que residan.

Multitud de trabajos han abordado el desarrollo de sistemas de recolección de residuos en sistemas distribuidos [119, 1], la mayoría de los cuales revisaremos en el apartado 4.7 dedicado a trabajo relacionado. En este capítulo presentamos un protocolo distribuido de cuenta de referencias [51, 50] que presenta mejores características a los existentes: en particular, sin incrementar el coste logramos una solución más asíncrona que las soluciones propuestas hasta la fecha. Para demostrar la corrección de nuestro algoritmo, primero formalizamos el problema de la detección de residuos acíclicos, después describimos formalmente nuestro protocolo y finalmente demostramos sus propiedades de viveza y seguridad. Posteriormente describimos el protocolo de cuenta de referencias adaptado a objetos replicados, que hasta donde hemos podido investigar, constituye el primer protocolo de estas características dedicado a recolectar objetos replicados, para finalmente concluir el capítulo con una sección dedicada a trabajo relacionado.

4.1.1 Recolección local *vs* detección distribuida

El sistema de recolección de residuos de Hidra está compuesto fundamentalmente por dos componentes: una componente local y una componente distribuida. La componente local del sistema está constituida por implementaciones de mecanismos tradicionales de cuenta de referencias. En particular, en Hidra aparece cuenta local de referencias a nivel de *stubs*, a nivel de *adaptadores* y a nivel de *endpoints*.

El objetivo de las cuentas locales es permitir la liberación de las estructuras de datos internas al ORB una vez que ya no se necesiten, permitiendo hacer un uso eficiente de la memoria. La memoria se utiliza eficientemente por la propia naturaleza de la cuenta de referencias, la cual persigue compartir zonas de memoria, contando cuántos usos simultáneos se realizan, para

posteriormente liberar dichas zonas cuando el contador valga cero.

La cuenta de referencias local, efectuada como en nuestro caso en tres niveles distintos, y ejecutándose como es el caso de Hidra en un entorno multitarea donde pueden ejecutarse concurrentemente operaciones de descarte, copia y duplicación de referencias, da lugar a un sistema de implementación compleja y de costosa depuración. Sin embargo, en este capítulo estamos interesados en describir el sistema de recolección de residuos en su vertiente distribuida, por lo que asumiremos un modelo simplificado de sistema distribuido, en el que eliminaremos la descripción del trabajo que realizan los dominios y los nodos para mantener más de una referencia a un mismo objeto. Así pues, diremos que un dominio mantiene una referencia a un objeto o que un dominio mantiene referencias a un objeto de forma indistinta, para indicar que el dominio mantiene al menos una referencia a dicho objeto. Cuando digamos que un dominio recibe una referencia a un objeto distinguiremos entre que reciba la primera referencia al objeto y que reciba las siguientes referencias, en cuyo caso diremos explícitamente que el dominio recibe una referencia que ya poseía, en oposición al caso de recibir una referencia nueva. Diremos también que un dominio descarta todas sus referencias a un objeto o simplemente que descarta su referencia a un objeto para indicar que descarta la última referencia al objeto.

4.2 Planteamiento de objetivos

El objetivo fundamental del sistema de recolección de residuos de Hidra es emitir una notificación de no referenciado a cada objeto registrado en el ORB para el que no existan referencias válidas. Tal y como vimos al describir el modelo de objetos de Hidra en el apartado 2.2.3, la recolección de los recursos consumidos por el objeto la delegamos a la implementación del objeto, que podrá optar por efectivamente liberar los recursos consumidos o por realizar cualquier otra acción que desee efectuar al alcanzarse la situación de objeto no referenciado. Por contra, el soporte a los objetos y el soporte a las referencias a objeto internos a Hidra, sí debe eliminarse cuando los objetos o referencias ubicados en cada nodo ya no existan.

Por otra parte, no es necesario que se detecten ciclos de residuos por no ajustarse esta funcionalidad al modelo de objetos Hidra. La detección debe ser eficiente en mensajes emitidos y en espacio. Debe afectar lo mínimo posible al rendimiento del sistema, haciendo en la medida de lo posible que no se aprecien períodos de bajo rendimiento debidos a la recolección de residuos, es decir el sistema de recolección de residuos debe añadir poca variabilidad al rendimiento del sistema. La notificación de no referenciado debe emitirse a todas las implementaciones de los objetos, por tanto si un objeto es replicado, todas las réplicas deberán recibir la notificación. Por último, deben tolerarse los fallos que pueden ocurrir al nivel del ORB: fallos de parada, desorden en la entrega de los mensajes y retardos arbitrariamente largos en la entrega de mensajes.

4.2.1 Notificación de no referenciado

El ORB de Hidra deberá hacer un seguimiento a las referencias que apuntan a cada objeto para lograr decidir cuándo ya no existe ninguna. A partir de que ocurra esta situación de objeto no referenciado, el ORB de Hidra emitirá la notificación y se ejecutará el código correspondiente del objeto. El código a ejecutar, lo proporcionará el programador en la operación `unrefe-`

`renced()` del objeto. Este método, cuya implementación original no efectúa acción alguna, será invocado por el ORB al detectar la condición de no referenciado.

Al delegarse la recolección de cada residuo al programador de la aplicación, hablamos de detección de residuos y no de recolección de residuos. Sin embargo, durante la detección de los residuos de aplicación, se irán detectando y recolectando los residuos que aparecen dentro del propio ORB. En este caso, hablaremos del mecanismo de recolección de residuos de Hidra, el cual detecta y recolecta los residuos internos al ORB, mientras que detecta los residuos de aplicación a los cuales les notifica de este hecho.

4.2.2 Notificación de no referenciado para objetos replicados

Los objetos replicados en Hidra son objetos ordinarios, con la particularidad de disponer de más de una implementación en un determinado instante. Sin embargo, el objeto es uno sólo y por tanto los clientes del objeto mantienen una sola referencia al objeto replicado. Cuando no existan referencias que apunten al objeto, el objeto replicado será un residuo. Pretendemos que todas las implementaciones del objeto (todas las réplicas) reciban la notificación de objeto no referenciado cuando no existan clientes del objeto.

Por otra parte, pretendemos que el mecanismo de detección de residuos sea independiente del modelo de replicación, y que su coste dependa lo mínimo posible del número de réplicas que tenga cada objeto.

4.2.3 No detección de ciclos

Un residuo cíclico [123, 39] aparece cuando existe un conjunto de objetos $C = \{O_1, O_2, \ldots, O_n\}$ de forma que todo O_i, $1 \leq i < n$ mantiene una referencia al objeto O_{i+1}, y el objeto O_n mantiene una referencia al objeto O_1, y todo objeto que mantenga una referencia a un objeto de C, pertenece a C. Es decir, cuando existe un ciclo de referencias y no existen referencias al ciclo fuera del ciclo. Por definición, para detectar residuos cíclicos, las referencias a objeto deben estar asociadas con los objetos. En Hidra, de forma similar a como enuncia CORBA, las referencias existen en los dominios y cada dominio es libre de asociar las referencias con objetos o de no hacerlo. Por tanto, en Hidra, el hecho de que un dominio posea una referencia a objeto, implica que todos los objetos del dominio potencialmente pueden hacer uso de ella, con lo que la búsqueda de ciclos no tiene sentido.

Para solucionar este aspecto, podríamos habernos planteado el proporcionar cierta operación en la interfaz de Hidra para asociar referencias con objetos. Esta operación la invocaría un dominio para indicar que cierta referencia que se ha recibido, será utilizada únicamente desde las operaciones de cierto objeto, comprometiéndose la aplicación a no utilizarla desde otros objetos. Si hubiéramos proporcionado esta operación, estaríamos en condiciones de detectar ciclos pues ya podríamos hablar en términos de qué objetos poseen qué referencias a objeto. Nótese que en todo caso, habilitar la detección de ciclos de esta forma, limitaría en cierto grado la libertad del programador a la hora de utilizar las referencias.

Sin embargo todavía existen otros factores importantes que nos aconsejan evitar la detección de residuos cíclicos. Uno de ellos está relacionado con el tipo de objetos que contemplamos en Hidra, en particular con los *objetos fijos*. Como ya comentamos en el capitulo 3, todo dominio

puede crear en cualquier instante una referencia a un objeto fijo. Por tanto, si detectamos en cierto instante un ciclo de referencias, nunca podremos llegar a decidir si existen referencias fuera del ciclo, pues en cualquier instante, cualquier objeto del ciclo que estemos considerando puede construir una referencia a un objeto fijo y enviarle una referencia de sí mismo o de cualquiera de las referencias que posea. Por tanto, para considerar ciclos deberíamos volver a limitar o a controlar el tipo de acciones que pueden realizar los dominios, u optar por algún mecanismo transaccional que asegurara la existencia de cierto instante a partir del cual no se pudieran generar referencias a objetos fijos.

Otra alternativa para solucionar el problema que introducen los objetos de tipo fijo la encontramos en el soporte a tareas del sistema. Un objeto de cierto dominio puede crear una referencia a un objeto fijo porque cierta tarea ejecuta código dentro del objeto. Podríamos considerar para la detección de ciclos sólo aquellos objetos que no tengan tareas activas. Para lograrlo, deberíamos llevar control sobre qué tareas están ejecutando código dentro de qué objeto. De esta forma podríamos distinguir entre objetos pasivos (sin tareas ejecutando código) y objetos activos (con alguna tarea dentro de sus operaciones) y buscar ciclos sólo entre objetos pasivos. Lógicamente si encontramos un ciclo entre objetos pasivos, no aparecerá ninguna referencia a los objetos del ciclo en el exterior en ningún instante futuro, pues para que aparezcan referencias, debería existir alguna tarea capaz de enviarla. Otras alternativas podrían consistir en prohibir el uso de objetos de tipo fijo para aquellos objetos que desearan ser notificados en caso de pertenecer a un ciclo residual.

Como puede observarse, si pretendiéramos buscar residuos cíclicos, estaríamos alterando notablemente el modelo de objetos que nos planteamos proporcionar en Hidra, y estaríamos forzando a un tipo de estructuración del código, que en el caso específico de tratarse de código del sistema operativo, limitaría gran variedad de posibles optimizaciones.

4.2.4 Eficiencia y asincronía

De las posibles alternativas que existan para implantar un sistema de recolección de residuos, buscamos aquella que siendo correcta y completa, incurra en menores costes tanto espaciales como en número de mensajes. En cuanto a los mensajes, buscamos soluciones asíncronas y cuyos mensajes podamos empaquetar en lotes para enviarlos a su destino cuando sea menos perjudicial para el rendimiento del sistema.

En todo caso, buscamos una solución que pueda tener cierto coste en instantes de baja carga en el sistema, pero que suponga un coste despreciable cuando la carga del sistema aumente. Es decir, buscamos un sistema de recolección de residuos totalmente asíncrono, donde todos los mensajes puedan empaquetarse como información adicional a los mensajes ordinarios que se deban transmitir. Con ello se eliminará la necesidad de emitir mensajes en favor del sistema de recolección, limitándose el coste del sistema al incremento de tamaño producido en cada mensaje.

4.2.5 Reducida variabilidad del rendimiento

Existen mecanismos de recolección de residuos que por su propia concepción, introducen una notable carga en el sistema en determinados instantes, mientras que durante la mayoría del

tiempo no afectan al sistema en exceso. En estos casos el gasto medio en mensajes o en espacio relacionado con recolección de residuos puede ser muy bajo, dependiendo en gran medida del tiempo que transcurra entre los periodos de mayor actividad de la recolección. Sin embargo, pese a poder presentar un coste medio razonable, el rendimiento del sistema puede verse seriamente afectado mientras dure la fase más necesitada de recursos. En estos periodos puede ser notoria una cierta degradación del sistema, y de prolongarse durante un tiempo significativo, puede ser poco aconsejable su uso.

4.2.6 Tolerancia a fallos

Hidra es una arquitectura de alta disponibilidad, y como tal, pretende dar servicio incluso en presencia de fallos. El ORB de Hidra, y en particular el recolector de residuos debe hacer frente a fallos de parada y funcionar sobre un modelo de sistema totalmente asíncrono. Es decir, si el objetivo del recolector de residuos de Hidra es enviar la notificación de no referenciado a los objetos de aplicación correspondientes y eliminar los residuos intermedios que el ORB haya podido generar, estos objetivos deben satisfacerse incluso en casos de fallos de parada.

Sin embargo, no nos planteamos el desarrollo de un sistema de recolección de residuos que tolere más fallos de los que pueden ocurrir a nivel del ORB. Por ello, no contemplamos ni pérdidas en los mensajes, ni duplicidades, ni cualquier otro tipo de fallo que no sea fallo de parada en los nodos. Existen algoritmos (por ejemplo [128]) que pretenden tolerar más tipos de fallos, sin embargo, como ya detallamos en el capítulo 2, la propia arquitectura Hidra la hemos construido enmascarando fallos desde los niveles más bajos a los más elevados y limitando a fallos de parada los fallos que deben considerarse a nivel del ORB.

4.3 Cuenta distribuida y asíncrona de referencias

El sistema de detección de residuos de Hidra, al que llamaremos HGD[1] se basa en un algoritmo distribuido de cuenta de referencias. Solamente se detectan residuos acíclicos. Se trata de un mecanismo *vivo* y *seguro* en el sentido de que sólo se detectan como residuos aquellos que realmente lo sean y que todo residuo es eventualmente detectado. Es barato en espacio y en mensajes. Consume poco espacio, ya que solo requiere un contador como estado de cada colector, a diferencia de los sistemas basados en listas de referencias [89, 128, 17] que necesitan la lista de colectores cliente en cada colector servidor. Los mensajes del algoritmo son completamente asíncronos, por lo que podrán ser agrupados y por lo que no será necesario emitir mensajes dedicados en exclusiva a servir al algoritmo si en el sistema se transmiten otros mensajes. El sistema de recolección tampoco necesita que se ejecute ninguna tarea recolectora, salvo una pequeña tarea que garantice que los mensajes son eventualmente enviados a su destino. Esta tarea se utiliza para prevenir la situación en que no se emitan mensajes en el sistema durante cierto periodo de tiempo, lo cual podría provocar ausencia de viveza. Tampoco es necesario utilizar ningún tipo de difusión de mensajes ni mecanismo transaccional alguno.

[1] Acrónimo de Hidra Garbage Detector.

4.3.1 Descripción informal

Como todo algoritmo distribuido de cuenta de referencias, el algoritmo HGD intenta llevar la cuenta en el colector servidor de cuántos mutadores cliente existen en el sistema con referencias al objeto. Para ello, cada colector dispone de un contador. Todos los contadores inicialmente valen 0 y únicamente el servidor dispone de referencias al objeto, por lo que es el único capaz de enviar referencias.

Caso 1: el servidor envía una referencia

Cuando el servidor envía una referencia, antes de hacerlo, incrementa su contador.

Si el servidor siempre enviara referencias a mutadores que no dispusieran de referencias, el contador del servidor tendría el valor exacto. Sin embargo, cuando el servidor envía una referencia, no sabe si el receptor ya dispone de una referencia al objeto, por lo que no bastará con simplemente incrementar el contador.

Cuando un mutador recibe una referencia enviada por el servidor, comprueba si ya disponía de una referencia previa a ese objeto. En caso de tratarse de una referencia nueva para él, no hace nada, pues el servidor habrá incrementado el contador correctamente. Si ya disponía de una referencia, envía un mensaje DEC al servidor.

Cuando el servidor reciba el mensaje DEC, decrementará su contador.

Caso 2: un cliente envía una referencia a otro cliente

Cuando un cliente recibe una referencia, ya puede enviarla a otro cliente o al servidor. Si el cliente A envía una referencia al cliente B, el cliente A incrementa su propio contador de igual forma a como lo hubiera hecho el servidor. Cuando el cliente B reciba la referencia, comprueba si se trata de una referencia nueva para él. Si no es nueva, envía un DEC al cliente A. Cuando el cliente A reciba el DEC, decrementará su contador.

En cambio, si el cliente B que recibe la referencia no disponía de referencias al objeto, no basta con no hacer nada como ocurría cuando el emisor de la referencia era el servidor. En aquel caso, no era necesario hacer trabajo adicional porque el servidor había actualizado correctamente su contador, pero en este caso, la actualización se ha realizado en el cliente A. Lo que hace nuestro algoritmo en este caso es registrar el cliente B en el servidor de la siguiente forma. El cliente B cuando comprueba que la referencia es nueva para él, incrementa su contador y envía un mensaje $INC(A)$ al servidor, con el que le está diciendo que el cliente A ha incrementado su contador en beneficio suyo. Cuando el servidor reciba el mensaje $INC(A)$, incrementará su contador, y enviará un DEC al cliente A y otro al cliente B.

Caso 3: un cliente le envía una referencia al servidor

En este caso, el cliente incrementa su contador y el servidor le enviará un DEC cuando reciba la referencia.

Nótese que este modo de funcionar admite una optimización obvia: que al enviar la referencia al servidor no se incremente el contador y que el servidor tampoco envíe el mensaje DEC al recibir referencias. Esta optimización la tenemos realizada en el prototipo actual de Hidra

para la cuenta de referencias de los objetos básicos. En esta memoria asumiremos que no se realiza para simplificar la exposición y las demostraciones. Adicionalmente, la extensión del algoritmo para objetos móviles desaconseja la adopción de esta optimización.

Descartes

Cuando un cliente descarta sus referencias, le envía un DEC al servidor para indicarle que existe un nodo menos con referencias. Sin embargo, si en el momento de realizarse el descarte, el contador del cliente es mayor que cero, no se enviará el DEC y diremos que el cliente está *retenido*. El cliente permanecerá en estado retenido mientras su contador no alcance el valor cero y mientras no reciba referencias nuevas. Si el contador alcanza el valor 0 antes de que el cliente reciba más referencias, se enviará entonces el DEC al servidor, y diremos que el cliente ya no tiene referencias. Si el cliente recibe alguna referencia antes de que el contador alcance el valor 0, se le quitará al cliente la marca de retenido y se procederá como si la marca nunca hubiera sido puesta.

4.4 Formalización del algoritmo

Una vez que hemos descrito el algoritmo de cuenta de referencias de manera informal, en este apartado demostraremos su corrección. Para ello describiremos inicialmente la base matemática y los formalismos que utilizamos, después describiremos el entorno en el que debe ejecutarse el algoritmo, posteriormente lo enunciaremos formalmente, para finalizar demostrando sus propiedades de seguridad y viveza.

4.4.1 Modelo del sistema y base matemática

Consideramos un sistema distribuido formado por p procesadores (nodos) interconectados por una red de área local, donde los procesadores solo se comunican mediante paso de mensajes, donde no pueden ocurrir particiones y donde los mensajes no se pueden perder ni llegar duplicados, pero pueden sufrir retrasos arbitrarios y pueden llegar desordenados a su destinatario respecto del orden en que fueron enviados. Asumimos que los nodos tienen identificadores únicos y que los identificadores están totalmente ordenados. Los únicos fallos que pueden ocurrir en el sistema son fallos de parada en los nodos.

Nuestros algoritmos y el sistema los describimos utilizando el modelo atemporal de los autómatas de entrada/salida [85, 84]. Las componentes de este modelo transicional son las siguientes: conjunto de estados, conjunto de estados inicial formado por cierto subconjunto del conjunto de estados, una signatura que especifica acciones de entrada, de salida y acciones internas y un conjunto de transiciones (estado, acción, estado) que permite transitar de un estado a otro. Por último un conjunto de tareas permite especificar qué acciones se ejecutan concurrentemente.

A continuación citamos los términos y conceptos de este modelo que utilizaremos con más frecuencia en nuestro trabajo:

Un *fragmento de ejecución* de un autómata de entrada/salida es una secuencia donde se alternan estados y acciones y que se muestra consistente con la relación de transición del autómata. Los fragmentos de ejecución pueden ser secuencias finitas o infinitas. Una *ejecución* es un fragmento de ejecución que comienza por un estado inicial. Utilizamos *trazas* para capturar el comportamiento externo de las ejecuciones; por tanto, una *traza* de una ejecución α de una autómata A, denotada por $traza(\alpha)$, es la subsecuencia de α que consiste únicamente en todas las acciones externas de α, es decir una secuencia compuesta únicamente por acciones de entrada y de salida. Utilizaremos las trazas para formular propiedades de los algoritmos y de los sistemas. También utilizaremos el término *evento* para indicar la ocurrencia de una acción dentro de una secuencia.

Una característica fundamental del modelo de autómatas de entrada/salida radica en que las acciones de entrada siempre deben estar habilitadas, mientras que las de salida o las internas estarán habilitadas en función de sus precondiciones. Diremos que una tarea T está habilitada si existe alguna acción ejecutada por la tarea con su precondición habilitada. Decimos que un estado s alcanzado por el autómata A es *quiescente*, si las únicas acciones habilitadas son las de entrada. Diremos que un autómata es *cerrado* si no tiene acciones de entrada. Relacionado con estos términos, y de gran importancia para razonar acerca de las propiedades de viveza, utilizaremos el concepto de *justicia* y con él, los conceptos de ejecuciones justas, fragmentos de ejecución justos y trazas justas. Diremos que la secuencia (ejecución, fragmento o traza) α del autómata A es justa, si para toda tarea T del autómata A se cumple que:

- Si α es finita, entonces T no está habilitada.
- Si α es infinita, entonces α contiene infinitas acciones de T o infinitos estados en los que T no está habilitada.

Podemos entender esta definición informalmente diciendo que con infinita frecuencia, a cada tarea se le da la opción de ejecutarse.

Por último también utilizaremos la *composición de autómatas*, para formar autómatas más complejos en base a autómatas más sencillos, para lo cual exigiremos que sean *compatibles*.

4.4.2 El problema de la recolección de residuos acíclicos

En terminología de recolección de residuos [32], al proceso que computa, que pide recursos, que los utiliza y los libera se le conoce como *mutador*. Por su parte, al proceso, tarea o código que detecta y recolecta los residuos que generan los mutadores se le conoce como *colector*.

Asumimos un entorno distribuido donde cada nodo dispone de un mutador y de un colector, y donde ambos pueden interactuar entre ellos directamente, pero deben utilizar paso de mensajes si desean comunicarse con los mutadores o colectores de otros nodos.

A las entidades recolectables las denominamos *objetos*, los cuales son *creados* por un mutador en particular. El mutador que crea un objeto, recibe una *referencia* al objeto que ha creado y la mantiene hasta que la *descarte*. Los mutadores que mantienen una referencia a cierto objeto, la pueden utilizar como deseen, pudiendo realizar operaciones sobre ella. En particular los mutadores que mantienen alguna referencia, pueden *enviarla* a otro mutador y pueden *descartarla*.

Cuando un mutador envía una referencia decimos que el mutador envía una *copia* de la referencia, o que simplemente *copia* la referencia. Un mutador que recibe una referencia enviada por otro mutador también *mantiene* la referencia. Un mutador que descarta una referencia, pierde el derecho a realizar operaciones sobre la referencia, o lo que es lo mismo, deja de mantener la referencia. Nótese que la típica operación de *duplicación* de referencias sólo tiene impacto a nivel de la cuenta local de referencias, por lo que no la consideramos. Lo mismo ocurrirá con los descartes de las referencias que no sean la última referencia al objeto y con las recepciones de referencias para objetos para los que ya se mantuviera alguna referencia.

Para un objeto en particular, denominamos *mutador servidor* del objeto, al mutador que crea el objeto y *nodo servidor* al nodo donde se ubica dicho mutador. Por eliminación, llamaremos *mutadores cliente* y *nodos cliente* al resto de mutadores y nodos respectivamente. Para un objeto dado, todos los mutadores, tanto servidores como clientes, pueden mantener o no una referencia al objeto en cada instante de tiempo. Nótese que incluso el servidor puede disponer o no de referencias y que el hecho de disponer de referencias permite copiarlas y descartarlas. Cuando un objeto está no referenciado, es un residuo y los colectores deberán cooperar para detectarlo.

Decimos que un objeto está *no referenciado* si después de que el objeto haya sido creado, llega un instante en el que no exista ningún mutador que mantenga una referencia al objeto y que tampoco exista ningún mensaje en tránsito por la red que contenga referencias (ya que podría dar lugar a que algún mutador la recibiera). Podemos observar como la definición de objeto no referenciado y la definición de residuo acíclico coinciden.

Una vez que un objeto es un residuo, el recolector de residuos deberá reclamar los recursos que estén asignados al objeto. En contraste, un detector de residuos simplemente debe notificar acerca de esta situación. En este trabajo asumiremos que la notificación debe ser enviada al mutador servidor.

En la figura 4.1 mostramos un sistema distribuido formado por un conjunto P de p procesadores. En cada procesador se ejecuta un mutador y un colector, y una red de comunicaciones permite intercambiar mensajes entre los procesadores. Todas las componentes las describimos utilizando autómatas de entrada/salida. El procesador s ejecuta el mutador $SMUT$ y el colector $SCOL$. Por su parte, $CMUT$ es el autómata que ejecuta cada mutador cliente en los demás procesadores y $CCOL$ es el autómata colector cliente. Cuando queramos referirnos a los autómatas mutadores en general utilizaremos el nombre MUT. También utilizaremos MUT_i y $CMUT_i$ para referirnos respectivamente al mutador del procesador i y al mutador cliente del procesador i de forma individualizada. Utilizaremos la misma relajación en la nomenclatura para referirnos a los autómatas COL. Finalmente, el autómata $ARNET$ es la modelización que hacemos de la red asíncrona, fiable y no FIFO que interconecta los procesadores.

Nótese que en esta figura estamos describiendo los mutadores (colectores) para un objeto en particular. En realidad el mutador del procesador i debería ser el autómata compuesto por tantos autómatas $MUT_{i,o}$ como objetos o existan en el sistema. Por conveniencia de notación, no indicamos el objeto del que se trata en cada autómata ni en cada signatura. Este cambio de signatura debería hacerse, si hablamos de forma estricta, para habilitar la composición de los autómatas ubicados en cada nodo con el fin de modelar a todo el sistema de objetos.

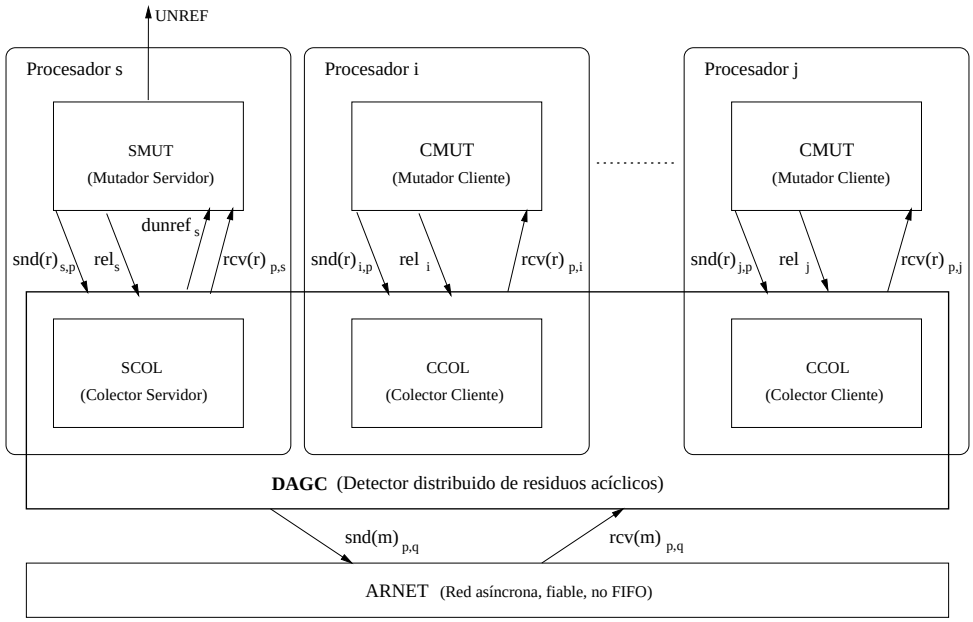

Figura 4.1: El problema de la detección de residuos acíclicos

Los mutadores

En las figuras 4.2 y 4.3 detallamos parcialmente el código que ejecutan los autómatas $SMUT$ y $CMUT$ respectivamente. Los autómatas no están completos, pues si pretendiéramos representar su código real, deberían incluir más estados, más transiciones y más código en cada transición, que dependería del código en particular que ejecuten. Con los autómatas MUT estamos únicamente interesados en modelar el entorno en el que se ejecutan los sistemas de detección de residuos. El objetivo de modelar su entorno, es restringir el comportamiento del entorno a aquellas situaciones que sean relevantes en ejecuciones reales.

Los autómatas son lo bastante genéricos como para no suponer restricción alguna al código de los mutadores, pero suficientes para describir las acciones relacionadas con la detección de residuos. Para cada autómata mostramos las signaturas de las acciones, las acciones, el estado y las tareas. Con los puntos suspensivos '...' indicamos condiciones o código que cada mutador en particular podría incluir de forma adicional al detallado en cada autómata, pero que no resulta relevante para el estudio de la detección de residuos. En cualquier caso, todo código adicional que se añadiera al autómata del mutador, no podrá hacer referencia al estado o a las acciones relacionadas con la detección de residuos, por lo que sin perder generalidad, y con el único motivo de simplificar la exposición, en lo sucesivo asumiremos que el código representado por los puntos suspensivos está vacío.

El autómata SMUT El mutador $SMUT$ comienza con al menos una referencia al objeto, hecho que correspondería a la situación inicial en la que el mutador acabara de crear el objeto. Como se puede observar, la creación del objeto le proporciona una referencia al mutador, hecho que queda representado con la variable $hold$. Mientras el mutador mantenga la referencia,

tendrá habilitadas las acciones $snd(r)$ y rel. La primera le permitirá enviar la referencia a cualquier procesador, mientras que rel la podrá utilizar para descartar sus referencias. Enviar una referencia al exterior queda reflejado en la variable $extref$, con la que indicamos que pueden existir referencias en otros mutadores (o en la red), por lo que necesitamos cooperación del resto del sistema para detectar la condición de no referenciado. Por su parte, si algún mutador nos envía una referencia, será recibida por la acción $rcv(r)$, cuyo efecto es garantizar que el mutador mantiene la referencia, que podría haberse perdido en caso de haber ejecutado rel.

SMUT: mutador servidor en procesador s

Signaturas

Entrada	Salida	Internas
$dunref$ $rcv(r)_{p,s}\ \ p \in P, p \neq s, r \in R$	$snd(r)_{s,p}\ \ p \in P, p \neq s, r \in R$ rel_s $UNREF$	

Estado

$hold \in \{true, false\}$, inicialmente $true$
$extref \in \{true, false\}$, inicialmente $false$
$unreferenced \in \{true, false\}$, inicialmente $false$

Transiciones

Entrada	Salida	Internas
$dunref$ Efecto: $extref \leftarrow false$ ... $rcv(r)_{p,s}$ Efecto: $hold \leftarrow true$ ...	$snd(r)_{s,p}$ Precondición: $hold = true$ ... Efecto: $extref \leftarrow true$ ... rel_s Precondición: $hold = true$ ... Efecto: $hold \leftarrow false$ $UNREF$ Precondición: $hold = false$ $extref = false$ $unreferencied = false$ Efecto: $unreferenced \leftarrow true$ ...	

Tareas

$\{snd(r)_{s,p}\ \ p \in P, p \neq s, r \in R\}$
$\{UNREF\}$
$\{rel_s\}$

Figura 4.2: Autómata SMUT ejecutado por el mutador servidor.

Cuando el sistema detecte que no existen otros mutadores con referencias al objeto ni existan referencias en tránsito por la red, el sistema deberá ejecutar la acción $dunref$, que es de entrada al mutador $SMUT$. Al ejecutarse esta acción, vuelve a quedar reflejado en la variable $extref$ que las únicas referencias posibles son locales al mutador servidor. Por último, cuando se cumpla que el mutador servidor no mantiene referencias locales, que no existen referencias fuera del mutador servidor y aún no se haya emitido la notificación de no referenciado, se podrá generar la acción $UNREF$. Esta acción sólo se ejecutará una vez y significará que el objeto es un residuo.

CMUT: mutador cliente en procesador i, $i \neq s$

Signaturas

Entrada	Salida	Internas
$rcv(p,r)_i \quad i,p \in P, i \neq p, r \in R$	$snd(r)_{i,p} \quad i,p \in P, i \neq p, r \in R$ $rel_i \quad i \in P$	

Estado

$hold \in \{true, false\}$, inicialmente $false$

Transiciones

Entrada	Salida	Internas
$rcv(r)_{p,i}$ Efecto: $hold = true$ ...	$snd(r)_{i,p}$ Precondición: $hold = true$ ... Efecto: ... rel_i Precondición: $hold = true$ ... Efecto: $hold \leftarrow false$ ...	

Tareas

$\{snd(r)_{i,p} \quad p \in P, p \neq i, r \in R\}$
$\{rel_i\}$

Figura 4.3: Autómata CMUT ejecutado por cada mutador cliente.

El autómata CMUT El autómata $CMUT$ es muy sencillo: tan sólo tres acciones y la variable $hold$ para indicar si el mutador mantiene referencias al objeto. Las acciones del mutador cliente son $snd(r)$, $rcv(r)$ y rel, cuyo significado es similar al de las acciones análogas del mutador $SMUT$, pero de implementación mucho más sencilla. La idea del código que ejecuta este mutador es simplemente recibir referencias y mientras el mutador no decida descartarlas, enviarlas a otro procesador o permitir el descarte.

La red de comunicaciones

En la figura 4.4 podemos observar el autómata $ARNET$, cuya relación con los demás autómatas ya quedó reflejada en la figura 4.1. Con $ARNET$ modelamos la red asíncrona de comunicaciones que interconecta a los procesadores de nuestro sistema. La red es asíncrona, fiable y no tiene porqué respetar el orden FIFO en la entrega de mensajes. La red la modelamos con dos canales de comunicaciones entre todo par de procesadores. Cada uno de los canales permitirá transmitir mensajes en uno de los dos sentidos posibles entre los dos procesadores conectados.

ARNET: red asíncrona, fiable, no FIFO		
Signaturas		
Entrada	Salida	Internas
$\forall i, j \in P,\ m \in M$ $snd(m)_{i,j}$	$\forall i, j \in P,\ m \in M$ $rcv(m)_{i,j}$	
Estado		
$\forall i, j \in P$ $buf_{i,j}$, bolsa de mensajes de M, incialmente vacía.		
Transiciones		
Entrada	Salida	Internas
$snd(m)_{i,j}$ Efecto: $buf_{i,j} \leftarrow buf_{i,j} \cup \{m\}$	$rcv(m)_{i,j}$ Precondición: $m \in buf_{i,j}$ Efecto: $buf_{i,j} \leftarrow buf_{i,j} - \{m\}$	
Tareas		
$\forall i, j \in P$ $\{rcv(m)_{i,j} : m \in M\}$		

Figura 4.4: Autómata ARNET que modela una red asíncrona fiable no FIFO

El estado de cada canal lo modelamos como una *bolsa* de mensajes. Denominamos *bolsa* a un conjunto que admite elementos repetidos. También podemos entender las bolsas como colas, donde las operaciones para añadir y eliminar elementos no acceden necesariamente a la cabeza ni al final de la cola, sino que acceden a cualquier parte de cola de forma arbitraria. Como operadores sobre las bolsas, utilizaremos los típicos operadores de conjuntos como $\cup$, $-$, $\in$, etc., extendidos a la manipulación de conjuntos donde se admiten elementos repetidos. En particular la unión (diferencia) de dos bolsas siempre resultará en una bolsa cuyo cardinal será la suma (resta) de los cardinales de las bolsas unidas (restadas). Por conveniencia de notación, denotaremos con M al conjunto finito de todos los posibles mensajes transmisibles por esta red. El autómata $ARNET$ tiene sólo una acción de entrada: $snd(m)_{i,j}$ y una acción de salida: $rcv(m)_{i,j}$. Con ambas se permite enviar mensajes desde cada procesador a cualquiera de los demás.

Teorema 1. *ARNET es una red asíncrona, fiable, no FIFO.*

> **Demostración informal.** El hecho de mantener una bolsa por cada canal y no una cola, implementa la ausencia de orden FIFO en los canales. La red no genera mensajes arbitrarios, hecho que queda descrito al exigir como precondición en la entrega de mensajes que el mensaje exista previamente en el canal, ni duplicados pues todo mensaje que entra a la red, o sale una sola vez o se queda en la red. Por último, las tareas del autómata, junto con el concepto de ejecuciones justas garantizan que no podrá ocurrir el caso en que un mensaje permanezca durante tiempo infinito en la red, con lo que se garantiza la asincronía, es decir, que todo mensaje es eventualmente entregado sin que existan cotas al tiempo que tarde en efectuarse dicha entrega. □

Los colectores y la compatibilidad entre componentes

Para que diversos autómatas se puedan componer, el modelo de autómatas de entrada/salida exige que sean compatibles. Un conjunto de autómatas son compatibles si se cumplen tres condiciones:

1. Las acciones internas de todo autómata tienen signaturas que no aparecen como acciones de ningún otro autómata.

2. No existen dos autómatas que tengan la misma acción de salida.

3. No existen acciones que aparezcan en un subconjunto infinito del conjunto de autómatas.

De las tres condiciones, la primera será sencilla de lograr, pues ninguno de los autómatas que hemos modelado hasta ahora tiene acciones internas, y la tercera se cumple simplemente si consideramos un conjunto finito P de procesadores. Sin embargo, la segunda propiedad puede ser problemática si permitimos a los colectores COL enviar mensajes por la red haciendo uso de la acción snd, pues esta acción ya es una acción de salida de los autómatas MUT.

Como formalidad para permitir la composición de autómatas asumiremos que la red $ARNET$ tiene acciones de entrada distintas para permitir el envío de mensajes a los autómatas MUT sin que coincidan sus signaturas con las correspondientes a los envíos que efectúen los autómatas COL. Nótese que este cambio de signatura no impide que los autómatas COL tengan como acciones de entrada, las acciones de envío de mensajes de los autómatas MUT. De esta forma podemos implantar en los autómatas COL mecanismos para interceptar los mensajes que envían los mutadores, sin por ello impedir ni retrasar su envío.

Especificación formal del problema

Una vez hemos detallado los autómatas que modelizan tanto a los mutadores, como a la red de comunicaciones, podemos enunciar formalmente qué deben cumplir los autómatas $SCOL$ y $CCOL$ para resolver el problema de la detección de residuos acíclicos. Llamemos A al autómata resultante de componer los autómatas MUT, COL y $ARNET$. El sistema de detección de residuos acíclicos $DAGD = \prod_{i \in P} COL_i$ es correcto si, siendo compatible con los autómatas de los mutadores $\prod_{i \in P} MUT_i$ y con la red de comunicaciones $ARNET$, cumple con las siguientes propiedades de seguridad y viveza.

Propiedad de Seguridad.

$$\alpha \cdot UNREF \in execs(A) \Rightarrow \begin{cases} \alpha.MUT_i.hold = false & \forall i \in P \\ & \wedge \\ \alpha.REFS = 0 \end{cases}$$

Propiedad de Viveza.

$$\forall \alpha \in fairexecs(A) : (\alpha = \alpha_1 \cdot s \cdot \alpha_2) \wedge (\nexists\, \alpha_1', \alpha_1'' : \alpha_1 = \alpha_1' \cdot UNREF \cdot \alpha_1''),$$
$$si \begin{cases} s.MUT_i.hold = false & \forall i \in P \\ & \wedge \\ s.REFS = 0 \end{cases} \Longrightarrow \alpha_2 = \alpha_3 \cdot UNREF \cdot \alpha_4$$

En las propiedades hemos denotado por $execs(A)$ al conjunto de todas las ejecuciones del autómata A y por $fairexecs(A)$ al conjunto de todas las ejecuciones justas del autómata A. Hemos utilizado el símbolo '$\cdot$' como operador de concatenación de secuencias. Por su parte hemos utilizado el símbolo '.' como notación para indicar que accedemos al estado de cierto autómata y también para acceder al estado de un autómata en particular desde el estado global del autómata A. Por último, con $REFS$ indicamos el número de referencias a objeto en tránsito por la red, es decir, la diferencia entre el número de eventos $sref$ y el número de eventos $rref$ que aparecen en α_1.

Informalmente la propiedad de seguridad la podemos enunciar diciendo, que si el autómata genera la acción $UNREF$, es porque el objeto es un residuo, es decir, porque no existe ningún mutador con referencias al objeto, ni ninguna referencia en la red. Análogamente la propiedad de viveza la enunciaríamos diciendo que siempre que se alcance un estado en el que ningún mutador tenga referencias al objeto y en el que no hayan referencias en tránsito por la red, se producirá eventualmente la acción UNREF, es decir, se emitirá la notificación de no referenciado.

4.4.3 Formalización del algoritmo distribuido de cuenta de referencias

Llamamos SEP (Server Endpoint) al autómata $SCOL$ que ejecuta el colector servidor de Hidra y CEP (Client Endpoint) al autómata $CCOL$ que ejecuta cada colector cliente en Hidra.

En la figura 4.5 podemos observar cómo estos autómatas son compatibles con los autómatas MUT y $ARNET$. Con objeto de simplificar la exposición del algoritmo, hemos reescrito la signatura de las acciones snd y rcv por $sref$ y $rref$. De hecho debe entenderse que $sref$ lo utilizan los mutadores para enviar un mensaje de tipo referencia y $rref$ sirve para entregar este tipo de mensajes. Por su parte, las acciones $sinc(q)$ y $sdec$ las utilizan los colectores para enviar mensajes de tipo $INC(q)$ y DEC respectivamente, mientras que las acciones $rinc(q)$ y $rdec$ se utilizan para entregar estos mensajes. Con estos cambios de signatura, no estamos alternado el autómata $ARNET$ ni los autómatas MUT, que serán los mismos que ya describimos en la figura 4.1. Simplemente hemos desglosado las acciones relacionadas con el paso de mensajes, en tantas acciones distintas como tipos de mensajes aparecen en nuestro sistema.

En la figura 4.5, también podemos observar cómo las acciones encargadas de enviar referencias son interceptadas por nuestros colectores. De esta forma, la acción de salida $sref$ de los mutadores, es de entrada tanto de los colectores como de la red. Algo similar ocurre con la

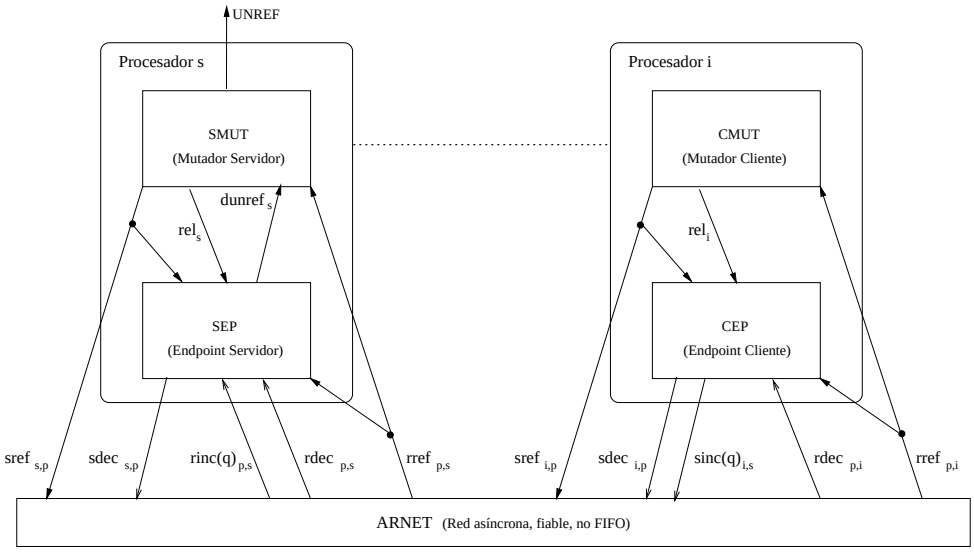

Figura 4.5: La detección de residuos acíclicos en Hidra

acción $rref$, que siendo de salida para la red, es de entrada tanto para los mutadores como para los colectores. Esta es la manera con la que modelamos el hecho de que los colectores están interesados en saber cuándo un mutador envía referencias al exterior y cuándo las recibe.

El endpoint servidor

En la figura 4.6 tenemos detallado el código del autómata SEP. Podemos comprobar cómo el estado del autómata está formado por el contador $count$, la variable de estado $status$ y la secuencia $Decs$. A efectos de una implementación real, el único estado sería el contador, pues tanto la secuencia $Decs$ como la variable $status$, más que estado del algoritmo, son modelizaciones de ciertos aspectos del sistema y no variables propiamente dichas. La secuencia $Decs$ se utiliza en el autómata para desincronizar aquellos puntos donde se pretende enviar un mensaje DEC, de la acción que se encarga de efectuar el envío. La hemos modelado como una secuencia para simplificar la exposición, pero podíamos haberlo hecho perfectamente con una bolsa o conjunto que admita repeticiones, ya que el orden de los envíos no tiene relevancia tal y como dicta la red $ARNET$. Por su parte, la variable $status$ podrá tomar uno de tres valores. Tomará el valor $null$ para representa que no existe el endpoint, y valdrá $localref$ para indicar que existen referencias locales, es decir que el endpoint tiene asociado algún adaptador. Finalmente $status = terminated$ indicará que la notificación de no referenciado ya se ha emitido.

SEP: Endpoint servidor en procesador s

Signaturas

Entrada	Salida	Internas
rel_s $sref_{s,p} \quad p \in P,\ p \neq s$ $rref_{p,s} \quad p \in P,\ p \neq s$ $rdec_{p,s} \quad p \in P,\ p \neq s$ $rinc(q)_{p,s} \quad p, q \in P,\ p \neq s \neq q$	$sdec_{s,p} \quad p \in P, p \neq s$ $dunref$	

Estado

$status \in \{null, localref, terminated\}$, inicialmente $localref$.
$count \in \mathbb{N}$, inicialmente 0.
$Decs$: secuencia de números de procesador $p \in P$, inicialmente vacía.

Transiciones

Entrada	Entrada	Salida

rel_s
Efecto:
$--$
 	
$sref_{s,p}$
Efecto:
$count \leftarrow count + 1$
$status \leftarrow localref$
 	
$rref_{p,s}$
Efecto:
$Decs \leftarrow Decs \cdot p$
$status \leftarrow localref$
	$rdec_s$
	Efecto:
	$count \leftarrow count - 1$
	if $(count = 0)$ then
	$status \leftarrow null$
	endif
	
	$rinc(q)_{p,s}$
	Efecto:
	$count \leftarrow count + 1$
	$Decs \leftarrow Decs \cdot p$
	$Decs \leftarrow Decs \cdot q$
	endif
		$dunref$
		Precondición:
		$status = null$
		Efecto:
		$status \leftarrow terminated$
		
		$sdec_{s,p}$
		Precondición:
		$Decs = p \cdot \alpha$
		Efecto:
		$Decs \leftarrow \alpha$

Tareas

$\{dunref\}$
$\{sdec_{s,p}\} \quad \forall p \in P,\ p \neq s$

Figura 4.6: Autómata SEP ejecutado por el endpoint servidor.

Por último, cabe resaltar que hemos construido el autómata sin acciones internas, y con únicamente dos acciones de salida: una para enviar mensajes DEC y la otra para emitir la notificación de no referenciado, por lo que las acciones controladas por el autómata son muy limitadas.

El endpoint cliente

CEP: Endpoint cliente en procesador i, $i \neq s$

Signaturas

Entrada	Salida	Internas
$rel_i \quad i \in P$ $sref_{i,p} \quad p \in P,\ p \neq i$ $rref_{p,i} \quad p \in P,\ p \neq i$ $rdec_{p,i} \quad p \in P,\ p \neq i$	$sinc(q)_{i,s} \quad q \in P,\ q \neq s \neq i$ $sdec_{i,p} \quad p \in P, p \neq i$	

Estado

$status \in \{null, localref, retained\}$, inicialmente $null$.
$count \in \mathbb{N}$, inicialmente 0.
$Decs$: secuencia de números de procesador $p \in P$, inicialmente vacía.
$Incs$: secuencia de números de procesador $p \in P$, inicialmente vacía.

Transiciones

Entrada	Entrada	Salida

$sref_{i,p}$
Efecto:
$count \leftarrow count + 1$
 	
$rref_{p,i}$
Efecto:
if $(status \neq null)$ then
$Decs \leftarrow Decs \cdot p$
else
if $(p \neq s)$ then
$count \leftarrow count + 1$
$Incs \leftarrow Incs \cdot p$
endif
endif
$status \leftarrow localref$
	rel_i
	Efecto:
	if $(count == 0)$ then
	$Decs \leftarrow Decs \cdot s$
	$status \leftarrow null$
	else
	$status \leftarrow retained$
	endif
	
	$rdec_i$
	Efecto:
	$count \leftarrow count - 1$
	if $(count = 0) \wedge$
	$(status = retained)$ then
	$status \leftarrow null$
	$Decs \leftarrow Decs \cdot s$
	endif
		$sdec_{i,p}$
		Precondición:
		$Decs = p \cdot \alpha$
		Efecto:
		$Decs \leftarrow \alpha$
		
		$sinc(q)_{i,s}$
		Precondición:
		$Incs = q \cdot \alpha$
		Efecto:
		$Incs \leftarrow \alpha$

Tareas

$\{sinc(q)_{i,s}\} \quad \forall q \in P : q \neq i \neq s$
$\{sdec_{i,p}\} \quad \forall p \in P : i \neq p$

Figura 4.7: Autómata CEP ejecutado por cada endpoint cliente.

Respecto al autómata SEP, el autómata CEP (ver figura 4.7), incorpora como estado la secuencia $Incs$. Su propósito es similar al de la secuencia $Decs$, pero en este caso para albergar temporalmente los mensajes INC que se desean enviar. Nótese que todos los mensajes INC se envían al servidor, por lo que sólo es necesario almacenar el número de procesador que

enviaremos como argumento de los mensajes INC. En lo referente al estado del endpoint reflejado por la variable $status$, podemos observar cómo deja de utilizarse el estado $terminated$, mientras que aparece el estado $retained$. Este último estado significa que el endpoint ya no está asociado al adaptador, pero no puede recolectarse porque todavía le deben llegar mensajes DEC enviados por otros endpoints.

También tiene interés comentar las acciones rel y $rdec$, que en ambos casos pueden producir que se le envíe un DEC al servidor. Esto ocurrirá cuando el endpoint vaya a ser eliminado, para advertirle al servidor de que existe un nodo menos con referencias al objeto.

4.4.4 Corrección del algoritmo

Demostrar que el algoritmo HGD es correcto, consiste en demostrar que el autómata $A = \prod_{i \in P} MUT_i \times EP \times ARNET$, donde $EP = SEP \times \prod_{i \in P, i \neq s} CEP_i$, cumple con las propiedades de seguridad y viveza que vimos en el apartado 4.4.2.

Lema 2. Valor de los contadores.

$\forall \alpha \in execs(A)$,

$$\sum_{i \in P} EP_i.count = \sum_{i \in P} (CEP_i.status \neq null) + REFS_{x,y} + DECS_{x,y} + INCS_{x,y}$$

Donde $\sum_{i \in P}(CEP_i.status \neq null)$ significa el número de mutadores cliente con referencias más el número de endpoints retenidos, o lo que es lo mismo el número de endpoints cliente que existen en el sistema. $REFS_{x,y}$ significa el número de referencias en tránsito en la red. La notación $REF_{x,y}$ significa mensajes de tipo REF en tránsito desde un nodo x cualquiera, a otro nodo y cualquiera. Análogamente utilizamos $INCS_{x,y}$ y $DECS_{x,y}$ para indicar respectivamente el número de mensajes de tipo INC (DEC) en tránsito. Nótese que decimos que un mensaje DEC o INC está en tránsito, si está en la red $ARNET$, o si todavía permanece en la secuencia correspondiente ($Incs$ o $Decs$) pendiente de envío a la red.

Demostración. La demostración del lema la hacemos por inducción según la longitud de las ejecuciones.

Si $|\alpha| = 0$ el invariante es trivialmente cierto. Suponiendo que sea cierto para ejecuciones de longitud $n - 1 \geq 0$, tenemos que demostrar que lo es para ejecuciones de longitud n. Para ello detallaremos todas las posibles acciones que pueden constituir la acción n-ésima. Veremos cómo después de ejecutarse cada acción, la igualdad se habrá preservado, pues ambas partes de la igualdad habrán sido incrementadas o decrementadas en la misma cantidad.

sref: Si se ejecuta esta acción, se incrementa el contador del colector que lo ejecuta, y pasa a haber una referencia más en tránsito, luego el invariante sigue siendo cierto si lo era antes de la ejecución de la acción.

rref: Si se ejecuta esta acción hay varias posibilidades:

1. La acción la ejecuta un cliente que ya tenía referencias o cuyo endpoint estaba retenido: se pone el estado del endpoint a $localref$, es decir el cliente pasa a tener referencias y se envía un DEC, con lo que la igualdad se habrá modificado en que habrá una referencia menos en tránsito y que habrá un DEC más.

2. La acción la ejecuta un cliente que no tenía referencias ni tenía el endpoint retenido y el emisor fue el servidor: se pasa el el estado del cliente a $localref$, con lo que habrá una referencia menos en tránsito y un nodo más con endpoint.
3. La acción la ejecuta un cliente que no tenía referencias ni tenía el endpoint retenido y el emisor de la referencia fue otro cliente: se incrementa el contador y se envía un INC, entonces tendremos una referencia menos en tránsito, un nodo más con endpoint, un contador incrementado en una unidad y un INC más en tránsito.
4. La acción la ejecuta el servidor: el servidor envía un DEC al cliente, por lo que habrá una referencia menos en tránsito y un DEC más.

Por tanto en caso de ejecutarse $rref$ se preserva el invariante.

sdec, sinc, dunref, UNREF: no se altera ningún término de la igualdad que estamos demostrando.

rdec: Se plantean varios casos:

1. Lo ejecuta el servidor: se decrementa un contador, y desaparece un DEC que estaba en tránsito.
2. Lo ejecuta un cliente cuyo contador vale 1 y cuya referencia estaba siendo retenida para evitar que el efecto del descarte llegara al servidor: se decrementa el contador, se envía un DEC y se descarta la referencia, con esto tendremos un DEC menos en tránsito, habremos enviado otro DEC, se habrá decrementado el contador y contaremos con un nodo menos con endpoint.
3. Lo ejecuta un cliente que no ha intentado descartar su referencia o cuyo contador es mayor que 1: se decrementa el contador y se elimina un DEC en tránsito.

En todos los casos el invariante se mantiene cierto.

rinc: Se incrementa un contador y se envían dos DEC, con lo que habrá un INC menos en tránsito, un contador habrá aumentado y tendremos dos DEC en tránsito.

rel: Sólo afecta al invariante si lo ejecuta un cliente. Si el cliente tiene el contador a 0 cuando ejecuta esta acción, se envía un DEC al servidor y el nodo elimina su endpoint, con esto se incrementa el número de mensajes DEC en tránsito y se decrementa el número de nodos con endpoint. Por contra si lo ejecuta un cliente que está protegido contra descartes, es decir que tiene el contador mayor que 0, no se hace nada que altere el invariante. □

Lema 3. Valor de los contadores de los clientes.

$$\forall \alpha \in execs(A),\ \sum_{i \in P} CEP_i.count = REFS_{c,x} + DECS_{x,c} + (2 \times INCS_{c,s})$$

Donde c representa a cualquier cliente, s al nodo servidor y x a cualquier nodo. Donde $REFS_{c,x}$ representa el número de referencias enviadas por los clientes y no entregadas todavía, $DECS_{x,c}$, el número de mensajes DEC enviados hacia los clientes y que todavía permanecen en tránsito y donde $INCS_{c,s}$, indica el número de mensajes INC en tránsito.

Demostración. La demostración la haremos por inducción en la longitud de las ejecuciones.

Si $|\alpha| = 0$ el invariante es trivialmente cierto. Veamos si el cierto para ejecuciones de longitud n, suponiendo que lo sea para ejecuciones de longitud $n - 1 \geq 0$.

sref: Sólo se altera algún término del invariante si esta acción la ejecuta un cliente. En este caso, se incrementa el contador y se envía la referencia a la red: por tanto el lema permanece cierto, pues habremos incrementado en una unidad ambas partes de la igualdad.

rref: Sólo pueden alterar algún término de la igualdad las recepciones de mensajes que no provengan del servidor:

1. La acción la ejecuta un cliente que ya tenía referencias, o cuyo endpoint estaba retenido: se envía un DEC al cliente emisor, con lo que la igualdad se habrá modificado únicamente en que habrá una referencia menos en tránsito y que habrá un DEC más.
2. La acción la ejecuta un cliente que no tenía referencias ni tenía el endpoint retenido y el emisor fue el servidor: este caso no se altera ningún término de la igualdad.
3. La acción la ejecuta un cliente que no tenía referencias ni tenía el endpoint retenido y el emisor de la referencia fue otro cliente: se incrementa el contador y se envía un INC, entonces tendremos que el lado izquierdo de la igualdad se ha incrementado en una unidad, y el lado derecho se incrementa en dos (por el INC) y se ha decrementado en uno (una referencia menos en tránsito), con lo que la igualdad sigue cumpliéndose.
4. La acción la ejecuta el servidor: el servidor envía un DEC al cliente, por lo que habrá una referencia menos en tránsito y un DEC más.

Por tanto, en caso de ejecutarse $rref$ se preserva el invariante.

sdec, sinc, dunref, UNREF,rel: no se altera ningún término de la igualdad que estamos demostrando.

rdec: Se plantean varios casos:

1. Lo ejecuta el servidor: no se altera ningún término de la igualdad.
2. Lo ejecuta un cliente cuyo contador vale 1 y cuyo endpoint estaba retenido: se decrementa el contador, se envía un DEC al servidor y se descarta la referencia, con esto tendremos un DEC menos en tránsito, y se habrá decrementado el contador.
3. Lo ejecuta un cliente que no ha intentado descartar su referencia, o cuyo contador es mayor que 1: se decrementa el contador y se elimina un DEC en tránsito.

En todos los casos se aprecia que el invariante se mantiene cierto.

rinc: Esta acción sólo la ejecuta el servidor, que enviará dos DEC, con lo cual la igualdad se preserva, pues habrán dos DEC más y un INC menos. □

Lema 4. Valor del contador de cada cliente.

$$CEP_i.count = REFS_{i,x} + DECS_{x,i} + INCS(x)_{i,s} + INCS(i)_{x,s} \quad \forall i \in P - \{s\}$$

Donde i representa al cliente i-ésimo para el cual describimos la expresión, s representa al nodo servidor y x a cualquier nodo. Donde $REFS$ y $DECS$ son abreviaciones de notación con el mismo significado que describimos en el lema 3, y donde finalmente representamos con $INCS(j)_{p,q}$ al número de mensajes INC enviados desde el nodo p al q con argumento j.

Demostración. La demostración de este lema es prácticamente idéntica a la demostración del lema 3, donde la diferencia fundamental radica en el desglose de los mensajes INC.

Cuando un nodo envía un INC mediante $sinc$ pueden darse dos casos para que este envío afecte a la igualdad que pretendemos demostrar:

- El envío lo hace el mismo nodo cliente (el nodo i) para el que estamos demostrando el lema: el contador se incrementa y se envía en INC, con lo cual se habrán incrementado en 1 ambas partes de la igualdad, preservando por tanto el invariante.
- El envío del INC lo hace un nodo que acaba de recibir una referencia que ha enviado el nodo i: tendremos una referencia menos en tránsito y un $INC(i)$ más en tránsito, luego también se preserva el invariante.

Las demás acciones que deben detallarse para demostrar el paso de inducción, tienen un efecto similar al descrito en el lema 3 pero particularizando a un cliente en concreto. □

Lema 5. Valor del contador del servidor.

$$\forall \alpha \in execs(A),\ SEP.count = \sum_{i \in P}(CEP_i.status \neq null) + REFS_{s,x} + DECS_{x,s} - INCS_{x,x}$$

Demostración. Trivial, restando los invariantes obtenidos por los lemas 2 y 3. □

Lema 6. $CEP_i.count >= 0 \quad \forall i \in P$

Demostración. Los contadores de los autómatas CEP son mayores o iguales a 0, pues por el lema 4 vemos que el valor del contador de todo endpoint cliente es la suma de varios términos que son siempre mayores o iguales a 0. □

Lema 7. $CEP_i.status = localref \iff CMUT_i.hold = true \quad \forall i \in P - \{s\}$

Demostración. Trivial, por construcción de las acciones $rref$ y rel de los autómatas $CMUT$ y CEP. □

Lema 8. $CEP_i.status = null \implies CEP_i.count = 0 \quad \forall i \in P$
Es decir, si un endpoint cliente no existe, entonces su contador es 0. También podemos reformular este lema utilizando la regla de contraposición y la desigualdad del lema 6:

$$CEP_i.count > 0 \implies CEP_i.status \neq null \quad \forall i \in P$$

Es decir, si el contador de un endpoint cliente es mayor que cero, el endpoint existe (el nodo tiene referencias o el endpoint está retenido).

Demostración. Demostramos el lema por inducción en la longitud de las ejecuciones. En caso de una ejecución de longitud 0, el lema es cierto, pues todo endpoint cliente es $null$ y todo contador de los endpoints cliente vale 0.

Suponiendo el lema cierto para ejecuciones de longitud $n - 1 \geq 0$, demostraremos si se cumple en ejecuciones de longitud n. Para ello nos centramos en un cliente C arbitrario. Las únicas acciones que pueden alterar los términos del lema son los que ejecuten los clientes, y en nuestro caso las que ejecute el cliente C.

sref: Si se ejecuta $sref$ es porque el mutador correspondiente tiene la variable $hold$ a cierto, luego por el lema 7 deducimos que $status = localref$ antes y después de ejecutarse la acción. Por tanto después de ejecutarse $sref$ sabemos que el contador es mayor que 0 y que $status \neq null$.

rref: En cualquier caso, después de ejecutarse esta acción, tendremos que $status = localref$, luego el lema es cierto.

rel: Si el contador es 0, $status$ pasará a valer $null$, haciendo que el lema sea cierto. Si el contador no es 0, $status$ no será $null$.

rdec: Se decrementa el contador y si alcanza el valor 0, en ciertos casos $status$ pasará a ser $null$. Por tanto sigue cumpliéndose el lema. □

Lema 9. $CEP_i.count = 0 \Longrightarrow CEP_i.status \neq retained \quad \forall i \in P$

Demostración. Demostramos de nuevo por inducción en la longitud de las ejecuciones. Demostraremos que en ningún caso puede alcanzarse un estado donde el contador sea 0 y donde $status = retained$. El caso base es trivial. Para la inducción, basta con que observemos las acciones $sref$, $rref$, $sdec$ y $rdec$, pues las demás no afectan a términos del lema. Después de ejecutarse $rref$, $status$ siempre será $localref$, por lo que se cumplirá el lema. Después de ejecutarse $sref$ se habrá incrementado el contador. Por el lema 6 sabemos que antes de ejecutarse esta acción, el contador era mayor o igual que cero, por lo que después de ejecutarse será mayor o igual a 1, lo que hace cierto el lema. Si se ejecuta la acción $rdec$ se decrementa el contador y en caso de alcanzarse el valor 0 y sólo si $status$ era $retained$ previamente, se le asigna entonces a $status$ el valor $null$, por tanto, en el único caso en el que podría ocurrir que el contador fuera 0 y $status = retained$, se asigna a $status$ el valor $null$. Por último, si la acción que se ejecuta es $sdec$, vemos que el contador no cambia, y que $status$ o bien se pone a $null$ (lema cierto) o se pone a $retained$ en caso de que el contador no fuera 0 (lema cierto). □

Lema 10. Registro de clientes.

$$\forall \alpha \in execs(A) \quad , si\ INCS = n,\ n > 0 \Longrightarrow \sum_{i \in P} (CEP_i.status \neq null) >= n + 1$$

Es decir, si hay algún INC en tránsito, entonces siempre hay más endpoints cliente que mensajes INC. Nótese que consideramos que cierto endpoint no existe, si su variable $status$ vale $null$.

Demostración. Por inducción en n. Si n=1, entonces tan sólo hay un INC en la red. Sea A el cliente que envía el INC y sea B el nodo que le envió la referencia al nodo A causando que éste enviara el INC. Por tanto tenemos el mensaje $INC(B)_{A,S}$ en la red. Por el lema 4 sabemos que $CEP_A.count >= 1$ y $CEP_B.count >= 1$. Estas conclusiones unidas al lema 8 nos llevan a ver que tanto el endpoint del nodo A como el del nodo B existen.

Paso de inducción: si hay n mensajes INC en tránsito, entonces también hay $n - 1$ mensajes en tránsito, por lo que al menos habrán n endpoints. Sea $INC(C1)_{C2,S}$, el mensaje n-ésimo que queda en tránsito después de que hubieran $n-1$ en tránsito. Por los lemas 4 y 8, y utilizando el mismo argumento que el caso base, sabemos que el endpoint del nodo $C2$ existe, pues el valor de su contador es mayor que cero. Supongamos que ya existiera antes de que se enviara el mensaje $INC(C1)_{C2,s}$, entonces cuando la referencia de $C1$ llegara a $C2$, éste no habría enviado el mensaje INC (ver acción $rref$), con lo que no se habría podido enviar el mensaje $INC(C1)_{C2,s}$. Hemos alcanzado una contradicción, por lo que inferimos que el endpoint del nodo $C2$ no podía existir antes del envío del INC, con lo que al menos existirán $n + 1$ endpoints. □

Lema 11. Invariante de seguridad.

$$\forall\, \alpha \cdot s \in execs(A),\ si \begin{cases} s.CMUT_i.hold = true & i \in P : i \neq s \\ & \vee \\ REFS_{x,y} > 0 \end{cases} \implies s.SEP.count > 0$$

Es decir, si existe al menos un mutador cliente con referencias al objeto o si existen referencias en tránsito, se garantiza que el contador del colector servidor será mayor que 0.

Demostración. Lo demostraremos por reducción al absurdo. Supongamos que la premisa es cierta:

$$REFS_{x,y} > 0 \quad \vee \quad \exists i \in P\ : CMUT_i.hold = true \tag{1}$$

Para alcanzar contradicción, supongamos que:

$$SEP.count = 0 \tag{2}$$

Utilizamos el lema 7 para reescribir (1), después mediante el lema 2 deducimos que:

$$\sum_{i \in P} EP_i.count > 0 \tag{3}$$

Uniendo las expresiones (2) y (3), obtenemos que:

$$\sum_{i \in P} CEP_i.count > 0 \tag{4}$$

Por el lema 3, escribimos la expresión (4) de la siguiente forma:

$$REFS_{c,x} + DECS_{x,c} + 2 \times INCS_{c,s} > 0 \tag{5}$$

Para que (5) sea cierta, contemplamos dos casos: que $INCS_{c,s} > 0$ ó $INCS_{c,s} = 0$.

$INCS_{c,s} > 0$**:** Considerando la expresión del lema 5 y el lema 10, alcanzamos contradicción, pues resultaría que $SEP.count > 0$.

$INCS_{c,s} = 0$**:** Por la expresión (4), sabemos que existe algún endpoint con contador mayor que cero. Esto unido al lema 8 nos conduce a deducir que existe algún endpoint. Sabiendo que existe algún endpoint y que no existen mensajes INC en tránsito, por el lema 5 deducimos que $SEP.count > 0$, con lo que alcanzamos una contradicción. □

Teorema 12. El algoritmo HGD es seguro.

$$\alpha \cdot UNREF \in execs(A) \Rightarrow \begin{cases} \alpha.MUT_i.hold = false & \forall i \in P \\ & \wedge \\ \alpha.REFS = 0 \end{cases}$$

Demostración.

Si se ha ejecutado la acción $UNREF$, es porque tanto $hold$, como $unreferenced$ como $extref$ valen $false$. Si $hold = false$, es porque α contiene rel y después de esta acción no hay ninguna recepción de referencias mediante $rref$. Por su parte, si $unreferenced = false$, es porque α no contiene ninguna acción $UNREF$.

Si $extref = false$, es por una de dos posibilidades:

1. No hay ningún $sref$ en toda la ejecución α.

   En este caso, el teorema es cierto pues inicialmente ningún mutador cliente tiene referencias, el mutador servidor ha descartado las suyas y no hay referencias en tránsito pues no se ha llegado a enviar ninguna.

2. Hay alguna acción $sref$ en α antes de que se ejecute la acción rel.

   Posteriormente a rel lógicamente no podrá aparecer ninguna acción $sref$ adicional. Adicionalmente, después de la última acción $sref$ aparece la acción $dunref$ en α. Esta acción es la que pondrá $extref = true$.

   Por cada acción $sref$, el contador del servidor se ha incrementado en 1, con lo que al menos ha llegado a valer 1 y si posteriormente a la última acción $sref$ se ha ejecutado $dunref$, es porque el contador ha pasado de 1 a 0. Por contraposición del lema 11, tenemos que si $SEP.count = 0$ ningún mutador cliente tiene referencias, ni tampoco existen referencias en tránsito.

Por tanto, si se emite $UNREF$ es porque no hay referencias en los mutadores ni referencias en tránsito. □

Teorema 13. El algoritmo HGD es vivo.

$$\forall \alpha \in fairexecs(A) : (\alpha = \alpha_1 \cdot s \cdot \alpha_2) \ \wedge \ (\nexists\, \alpha_1', \alpha_1'' \ : \ \alpha_1 = \alpha_1' \cdot UNREF \cdot \alpha_1''),$$
$$si \begin{cases} s.MUT_i.hold = false & \forall i \in P \\ & \wedge \\ s.REFS_{x,y} = 0 \end{cases} \Longrightarrow \alpha_2 = \alpha_3 \cdot UNREF \cdot \alpha_4$$

Demostración. En función de si α_1 contiene o no acciones $sref_{s,x}$ tenemos dos posibilidades. Si α_1 no contiene ninguna acción $sref_{s,x}$ entonces tenemos que $SMUT.hold = false$, $SMUT.extref = false$ y $SMUT.unreferenced = false$. Mirando las precondiciones de las acciones observamos que la única acción habilitada es $UNREF$, lo que nos lleva a deducir que toda ejecución justa que parta del estado s, acabará ejecutando $UNREF$.

El segundo caso que debemos contemplar es que α_1 contenga al menos una acción $sref_{s,x}$. En este caso, sabemos que en el estado s no hay ninguna referencia en tránsito, por lo que en la red sólo podrán haber mensajes INC y mensajes DEC (también podemos tener secuencias $Incs$ y $Decs$ con varios elementos). Si observamos qué acciones pueden estar habilitadas a partir de alcanzarse el estado s, comprobamos que las únicas acciones posibles son: $sinc$, $rinc$, $sdec$, $rdec$, $dunref$ y $UNREF$.

Si estuviera habilitada la acción $dunref$, sería porque $SEP.status = null$ y puede verse fácilmente que esta acción permanecerá habilitada hasta que se ejecute. Como el autómata SEP tiene una tarea encargada de ejecutar esta acción, sabemos que toda ejecución justa deberá acabar ejecutándola, por lo que eventualmente se habilitará la acción $UNREF$. Una vez que $UNREF$ se habilite, de igual forma a como ocurriría si ya estuviera habilitada en s, la tarea que ejecuta la acción $UNREF$ garantiza que $UNREF$ terminará por ejecutarse.

Por tanto sólo queda por demostrar que $UNREF$ eventualmente se ejecutará en toda ejecución justa que parta de s, siendo s un estado en el que no hay referencias en tránsito ni mutadores con referencias, donde $dunref$ y $UNREF$ no están habilitadas y donde al menos se envió una referencia antes de alcanzarse s.

Si tenemos en s un número finito de mensajes INC en tránsito, sabemos que todos ellos eventualmente llegarán a su destino, pues el concepto de ejecución justa y las tareas del autómata CEP y de la red $ARNET$ lo garantizan.

Sea s' el estado alcanzado una vez se hayan entregado todos los mensajes INC que estuvieran en tránsito en s. Por cada mensaje INC entregado, viendo la acción $rinc$, sabemos que se habrán generado dos mensajes DEC adicionales. Sin embargo no se habrá generado ninguna referencia adicional, ni ningún mensaje INC adicional. Por tanto, sabemos que toda ejecución justa que parta de s alcanzará cierto estado s' donde las únicas acciones habilitadas serán $sdec$ y $rdec$. En s' están en tránsito aquellos mensajes DEC que estando en tránsito en s no hubieran sido entregados hasta alcanzarse s' más aquellos mensajes DEC generados por las entregas de los mensajes INC, que tampoco hubieran sido entregados. En cualquier caso, en s' sólo habrá un número finito de mensajes DEC en tránsito.

Sea s'' el estado en el que todos los mensajes DEC que estuvieran en tránsito en s' y que estén dirigidos haca algún endpoint cliente hayan sido entregados. Sabemos que este estado se alcanzará, por las tareas de los autómatas CEP, SEP y $ARNET$ y por el concepto de ejecuciones justas. Cuando todos hayan sido entregados, viendo la acción $rdec$ del autómata CEP podemos observar cómo la entrega tiene como efecto decrementar el contador, y en algún caso se genera un nuevo DEC en este caso dirigido al servidor. Por tanto en s'' todavía podrán existir mensajes DEC en tránsito hacia el servidor. Sin embargo no habrán ni referencias en tránsito, ni mensajes INC, ni mensajes DEC dirigidos a clientes.

Sea s''' el estado que se alcance desde s'' en el que no queden mensajes DEC en tránsito, por haberse completado la entrega de todos los que quedaban en tránsito dirigidos al servidor. Con un razonamiento análogo al expuesto para los demás mensajes, sabemos que dicho estado s''' será eventualmente alcanzado gracias a la ejecución de las tareas de los autómatas CEP y $ARNET$. Observando la acción $rdec$ del autómata SEP comprobamos cómo cada recepción no genera ningún otro mensaje. Por tanto hemos deducido que a partir de s, eventualmente se alcanzará el estado s''' en el que no hay ni referencias en tránsito, ni tampoco mensajes INC ni DEC en tránsito.

Por el lema 4 sabemos que $CEP_i.count = 0 \ \forall i \in P$. Por el lema 9 podemos deducir que $CEP_i.status \neq retained \ \forall i \in P$. También sabemos que en s no había mutadores con referencias, y que desde s hasta s''' no se han transmitido referencias, por lo que en s''' no hay mutadores con referencias. Esto último implica por el lema 7 que $CEP_i.status \neq localref \ \forall i \in P$. Uniendo las deducciones que hemos realizado sobre las variables $status$, inferimos que $CEP_i.status = null \ \forall i \in P$. Aplicando el lema 5 sabemos que $s'''.SEP.count = 0$.

Como estamos analizando el caso de las ejecuciones que parten de s, siendo s un estado alcanzado tras ejecutar α y donde α contiene al menos un evento $sref_{s,x}$, sabemos que el contador del autómata SEP ha llegado a valer 1. Mirando el autómata SEP vemos fácilmente que el contador sólo se incrementa o se decrementa de una unidad en una unidad, por lo que si en s''' vale 0, sabemos que habrá transitado de 1 a 0 en cierto instante. Sabemos que la transición de 1 a 0 del contador habrá sido realizada por la ejecución de $rdec$, ya que ésta es la única que decrementa el contador. Mirando la acción $rdec$, observamos que al decrementar el contador y alcanzarse el valor 0, se habrá modificado la variable $status$ asignándosele el valor $null$. Podemos observar cómo a partir de este instante la acción $dunref$ queda habilitada y cómo quedará habilitada hasta que sea ejecutada. Por el concepto de ejecuciones justas y por las tareas de los autómatas SEP y $SMUT$, y siguiendo un razonamiento análogo al que hemos seguido al comenzar la demostración, podemos deducir que la acción $UNREF$ será eventualmente ejecutada. □

4.5 Reconstrucción ante fallos

El mayor problema de los algoritmos de recolección de residuos basados en cuenta distribuida de referencias consiste en su escasa tolerancia a fallos. No se toleran pérdidas de mensajes, duplicidades de mensajes, ni caídas de nodos. Los dos primeros tipos de fallos no son importantes en el caso de un sistema como Hidra, que construido en niveles, incorpora en los niveles inferiores mecanismos para enmascarar los errores que puedan ocurrir en las comunicaciones. Sin embargo, los fallos de los nodos sí pueden ocurrir, y al ser Hidra una arquitectura de alta disponibilidad, son de especial relevancia.

Si tenemos el algoritmo HGD funcionando y en cierto instante falla un nodo, podemos apreciar claramente cómo los invariantes pueden empezar a no ser ciertos. El nodo que ha fallado puede que tuviera referencias al objeto, o puede que tuviera el endpoint retenido o puede que no tuviera endpoint. Además es muy probable que existieran un número indeterminado de mensajes en tránsito enviados por o hacia el nodo caído. Nuestro algoritmo, en su pretensión de consumir pocos recursos, no guarda registro en ningún otro nodo del estado local a cada nodo relativo a los invariantes. Por tanto, es necesario que se ejecute algún protocolo para restablecer los invariantes del algoritmo tan pronto como se detecte que un nodo ha fallado.

Es interesante darse cuenta que las caídas de nodos afectan a la viveza del algoritmo y no a su seguridad. Esta afirmación puede corroborarse al apreciar que los invariantes del algoritmo siempre mantienen en los contadores un valor superior o igual al número real de nodos con referencias. En caso de una caída, será posible que los contadores ya nunca decrezcan lo suficiente como para alcanzar el valor cero, pero no ocurrirá el caso de que alcancen el valor cero habiendo nodos con referencias. Es decir, una caída de un nodo (suponiendo fallo de parada) no emitirá mensajes arbitrarios que puedan hacer disminuir algún contador de forma artificiosa.

Otro aspecto que debe tenerse en cuenta, es el hecho de que puede caer el nodo con el servidor. En este caso, es notorio que todos los endpoint cliente de los objetos situados en el nodo caído son residuos. También podemos observar cómo esta detección la podríamos retrasar hasta que los nodos con los endpoints inválidos intentaran invocar al servidor. En el instante de la invocación, se podrían emitir las correspondientes excepciones de nodo caído a las aplicaciones al tiempo que se recolectan los endpoints ya detectados como residuos.

4.5.1 El algoritmo de reconstrucción

Para restablecer los invariantes del algoritmo HGD, cada vez que se detecte una caída de nodo, en Hidra se ejecuta el protocolo de reconstrucción de la cuenta de referencias. El objetivo del algoritmo es alcanzar la situación en la que el contador del servidor tenga el valor exacto del número de nodos con referencias al objeto. Cuando se alcance esta situación, serán residuos aquellos objetos cuyos contadores en el servidor valgan cero, mientras que el resto de los objetos no lo serán. El mayor problema que nos encontramos al diseñar la reconstrucción de la cuenta de referencias radica en la asincronía de las comunicaciones, por la que los mensajes pueden tardar un tiempo arbitrario en entregarse. Es decir, pueden existir mensajes en tránsito por la red que dificulten o imposibiliten la reconstrucción de la cuenta.

Por una parte, es necesario que los mensajes INC y DEC del algoritmo no afecten a la reconstrucción. Estos mensajes incrementan y decrementan los contadores, y su efecto puede

ser indeseable durante el cálculo del número de nodos con referencias. Adicionalmente, una vez que la reconstrucción termine, se deberá ejecutar de nuevo el algoritmo de detección de residuos, y el funcionamiento de éste no deberá verse afectado por los mensajes que hubieran podido generarse por el algoritmo antes de la reconstrucción. Por otra parte, para calcular el número de nodos con referencias a cada objeto es necesario que no existan referencias al objeto en tránsito por la red. Si las hubiera, podría darse el caso de que se considerara a un objeto como residuo cuando realmente no lo fuera.

El algoritmo de reconstrucción de referencias es un sencillo protocolo de marcado y barrido [91, 32] ampliado con dos rondas iniciales y una final. Las rondas iniciales eliminarán los mensajes que pudieran estar en tránsito o harán que los mensajes no tengan efecto. Por su parte la ronda final estará dedicada a habilitar el funcionamiento ordinario del sistema. Las cuatro rondas del algoritmo se ejecutarán en cada uno de los nodos cuando se detecte una caída. A continuación mostramos las cuatro rondas y más adelante en este apartado veremos con detalle las acciones que se ejecutan en cada una de ellas:

Las cuatro rondas del algoritmo

1. Bloqueo de referencias y de los mensajes del HGD.
2. Entrega de referencias en tránsito.
3. Marcado y barrido.
4. Desbloqueo de referencias y de los mensajes del HGD.

Ronda 1.- Bloqueo de referencias y de los mensajes del HGD. Esta ronda comienza al detectarse la caída de un nodo y su objetivo es garantizar que no se emitan nuevas referencias, que no se emitan nuevos mensajes del HGD y que no se entreguen los mensajes del HGD que estuvieran en tránsito.

1. Se instala el nuevo número de reconfiguración del cluster como número que etiquetará a todo mensaje que el algoritmo HGD emita en el futuro.
2. Todo mensaje del algoritmo HGD (INC ó DEC) que llegue al nodo con un número de reconfiguración del cluster previo al que se acaba de instalar, será descartado.
3. Se deshabilita el envío de los mensajes del algoritmo HGD, es decir, las acciones rel, $rref$, $rdec$ y $rinc$, en lugar de añadir los mensajes del HGD a las secuencias $Incs$ y $Decs$, los descartarán.
4. Las colas $Incs$ y $Decs$ son vaciadas.
5. Se impide todo envío de referencias. Es decir, toda tarea que intente ejecutar $sref$ quedará bloqueada. Nótese que esta acción no bloquea las invocaciones sobre operaciones que no tengan argumentos de tipo objeto. El hecho de bloquear los envíos de referencias, bloquea la creación de nuevos endpoint servidores. Por otra parte, las acciones descritas

hasta ahora en esta misma ronda garantizan que tampoco se eliminará ningún endpoint servidor. Por tanto, se podrá recorrer la lista de endpoints servidores sin que aparezcan condiciones de carrera.

6. **Inicialización de la fase de marcado:** El contador de todo endpoint servidor se inicializa con el valor 0. Nótese que los mensajes del HGD están deshabilitados, por lo que el contador del servidor no podrá transitar de 1 a 0 y por tanto no será eliminado.

Ronda 2.- Entrega de referencias en tránsito. Con la ronda previa se ha logrado que no aparezcan nuevas referencias en la red, sin embargo, todavía puede haber cierto número de referencias en tránsito. Cuando finalice esta segunda ronda se garantizará que todas las referencias en tránsito habrán sido entregadas. Lograr que se entreguen las referencias va a consistir simplemente en esperar hasta que se hayan entregado las referencias. Si transcurre cierta cantidad de tiempo sin que se entreguen, habrá que considerar que ha ocurrido un fallo de caída en alguno de los dos nodos involucrados. Esperar a que se entreguen las referencias que permanezcan en tránsito, implica saber cuántas referencias han sido enviadas desde cada nodo a cada uno de los demás nodos y cuántas han sido entregadas.

Para implementar esta ronda, cada nodo dispone de dos vectores de contadores y de un objeto de tipo fijo. Cada una de los dos vectores tiene tantos elementos como nodos haya en el sistema. Un vector almacenará el número de referencias enviadas por el nodo a cada uno de los demás nodos y el segundo vector almacenará el número de referencias recibidas de cada uno de los demás nodos. Los contadores se actualizan en cada envío y recepción de referencias, utilizándose el mismo contador para todos los objetos. Por su parte, el objeto de tipo fijo, al que llamaremos *ayudante de referencias* servirá para almacenar los dos vectores de contadores. Gracias a él, todo nodo que lo desee, podrá invocar al objeto correspondiente ubicado en otro nodo, para bloquearse hasta que todas las referencias emitidas desde el nodo que invoca, lleguen al nodo invocado.

1. Todo nodo invoca en paralelo al objeto ayudante de cada uno de los los demás nodos. A cada invocación se le pasa como argumento el número de referencias enviadas al nodo que se invoca.

2. Cada vez que el objeto ayudante de un nodo recibe una invocación, espera hasta que el contador de referencias recibidas del nodo que realiza la invocación alcance al número de invocaciones que se recibe como argumento de la invocación.

3. Si la invocación efectuada sobre el objeto ayudante excede cierta cantidad de tiempo sin que se complete, el nodo que recibe la invocación será excluido del cluster.

Al acabar la ronda, tendremos la garantía de que no existen referencias en tránsito. De la primera ronda, se mantiene el hecho de que no se envían referencias, ni mensajes del HGD, mientras que los mensajes del HGD que estuvieran en tránsito no tendrán efecto.

El mayor problema que plantea esta ronda es la posibilidad de que el fallo en un nodo pueda provocar que varios nodos sean excluidos del cluster. Esta situación podría llegar a ocurrir si cierto nodo tuviera incorrectos los contadores que mantienen el número de referencias enviadas.

Sin embargo, dado el modelo de fallos del que partimos, asumimos que este tipo de fallos no aparecerá. De aparecer, el sistema reaccionará expulsando del cluster a nodos correctos y forzando a que éstos se reintegren de nuevo.

Ronda 3.- Marcado y barrido. Al comenzar esta ronda no existen referencias en tránsito y el contador de todos los endpoints servidores es cero. Ahora es posible ejecutar un protocolo de marcado y barrido para detectar residuos.

1. Se bloquean los descartes de endpoints, es decir se bloqueará toda tarea que intente ejecutar la acción rel. Con esta medida, logramos que no se destruya ningún endpoint cliente mientras ejecutamos el resto de este paso.

2. **Fase de marcado I:** El nodo envía a los demás nodos un mensaje. El mensaje contiene la lista de referencias que existen en el nodo emisor cuyo servidor se encuentra ubicado en el nodo destino del mensaje.

3. **Fase de marcado II:** El nodo espera un mensaje del resto de nodos. Al recibir cada mensaje, se recorre la lista de referencias contenida en él y se incrementa el contador del endpoint servidor correspondiente.

   Cuando se ha recibido un mensaje de cada nodo, el contador del servidor será el número exacto de nodos con endpoints cliente.

4. **Fase de barrido:** Cada nodo recolecta todo endpoint servidor cuyo contador sea 0. Al objeto asociado se le envía la correspondiente notificación de objeto no referenciado.

Ronda 4.- Desbloqueo de referencias y de los mensajes del HGD. Al finalizar la tercera ronda, se han recolectado los residuos del ORB y se han detectado las situaciones de objeto no referenciado. Todo lo que queda por hacer es permitir que el ORB reanude su funcionamiento normal.

1. Se habilita el envío de los mensajes del HGD.

2. Se desbloquean las acciones rel.

3. Se desbloquea el paso de referencias.

4.5.2 La reconstrucción de la cuenta de referencias como pasos de reconfiguración

Tal y como comentamos en el capítulo 2, el monitor de pertenencia a Hidra, al detectar la caída de un nodo, coordina la ejecución de pasos “síncronos” entre el conjunto de nodos vivos. Este es el mecanismo que utilizamos para implantar el concepto de las rondas con el que hemos descrito el algoritmo de reconstrucción. Como hemos visto, utilizamos cuatro pasos “síncronos” de reconfiguración. En caso de fallos de caída durante la ejecución de cualquiera de los pasos, el algoritmo será reiniciado desde la primera ronda.

4.6 Recolección de residuos para objetos replicados y móviles

Hasta ahora hemos descrito los algoritmos de cuenta de referencias y de reconstrucción de las cuentas para el caso de objetos ordinarios de implementación única. Si consideramos la posibilidad de que los objetos migren de una ubicación a otra o de que tengan más de una implementación, aparecen nuevos aspectos que deberán ser tenidos en cuenta.

4.6.1 Notificación de no referenciado para objetos replicados

La primera cuestión que debe abordarse es el tipo de notificación de no referenciado que se desea implantar. Una posibilidad sería implantar un mecanismo de notificación dependiente del modelo de replicación. Para el caso de replicación pasiva, la notificación debería dirigirse a la réplica primaria y para replicación coordinador-cohorte a la coordinadora. En ambos casos, la réplica que recibiera la notificación realizaría el trabajo que el programador hubiera especificado, enviándose posteriormente algún mensaje de actualización a las demás réplicas para que éstas pudieran actuar de forma consistente. Para el caso de replicación activa, la notificación de no referenciado debería dirigirse a todas las réplicas, utilizando para ello el transporte de mensajes ordenados.

Sin embargo, en Hidra optamos por implantar un mecanismo de notificación de no referenciado independiente del modelo de replicación. De hecho, el mecanismo de la detección de residuos constituye uno de los elementos comunes a todo objeto replicado, y su implementación es independiente del modelo de replicación. Cuando un objeto sea detectado como residuo, se emitirá la notificación de no referenciado a todas sus implementaciones. Cada una de ellas será responsable de eliminar los recursos que ella consuma, o de proceder de cualquier otra forma con la que se desee reaccionar ante la notificación de no referenciado. Las notificaciones se emitirán eventualmente después de transcurrido cierto tiempo arbitrariamente largo desde que el objeto sea un residuo. Las notificaciones se entregarán sin respetar ningún orden preestablecido entre las réplicas, y podrá transcurrir un tiempo arbitrariamente largo entre la entrega de cada una de las notificaciones.

Actualmente tenemos implementada gran parte de la arquitectura Hidra [52] junto con algunas aplicaciones de prueba, y no hemos encontrado ninguna situación para la que se demande un tipo de notificación distinto al que adoptamos en Hidra. Sin embargo, no excluimos la posibilidad de que en el futuro aparezca la necesidad de proporcionar, para los modelos pasivos, una notificación de no referenciado con semántica de ejecución de exactamente una vez. Para implantar este tipo de notificaciones de debería enviar tan sólo una notificación a la réplica primaria de forma similar a como se le envía una invocación. Sin embargo para el caso de la notificación de no referenciado, se admiten diversas optimizaciones respecto a las invocaciones, que dependen de sus características especiales:

- No existe cliente que inicie la invocación, o más precisamente el cliente es el ORB, y el ORB no cae a no ser que falle todo el sistema, por lo que se elimina la necesidad de contemplar los casos de caída de clientes.

- Dado que cada objeto sólo debe esperar una notificación de no referenciado, es sencillo trasladar a la aplicación la detección de repeticiones de la notificación. Este aspecto

simplificaría enormemente la gestión de la notificación, pues el ORB se podría limitar a reiniciar la notificación sobre una réplica nueva en caso de caídas. Es decir, el ORB podría tratar la notificación como una operación idempotente, y la responsabilidad de detectar notificaciones duplicadas quedaría en parte trasladada a la aplicación, que debería comprobar duplicados en base a los mensajes de actualización que hubiera recibido.

De cualquier forma, en Hidra sólo proporcionamos la notificación de no referenciado como una notificación que llegará eventualmente a todas las réplicas del objeto y otro tipo de notificación queda fuera de los objetivos de nuestro mecanismo.

4.6.2 El algoritmo para objetos replicados y móviles

Los objetos replicados o móviles se diferencian de los objetos que no son replicados ni móviles en que los primeros pueden tener simultáneamente más de un endpoint servidor. Ya vimos en el capítulo 3 dedicado al ORB de Hidra, que uno de los endpoints servidores detenta el rol de endpoint principal, y que esta asimetría entre los endpoints servidores se proporcionaba para facilitar la implantación de protocolos distribuidos.

Gracias al endpoint principal, el algoritmo de detección de residuos es prácticamente el mismo para los objetos replicados y móviles, que para los objetos ordinarios. Considérese el algoritmo HGD que describimos en las secciones precedentes y supóngase que el autómata SEP lo ejecuta el endpoint principal, mientras que tanto los endpoints servidores que no sean el principal, como los endpoints cliente ejecutan el autómata CEP. Tendríamos que el algoritmo prácticamente funcionaría a excepción de ciertos aspectos:

- Con el algoritmo sin variaciones, sólo recibiría la notificación de no referenciado la réplica que estuviera ubicada sobre el endpoint principal.

- Puede que no exista réplica sobre el endpoint principal. Esta situación puede ocurrir si ésta ha sido eliminada por la correspondiente operación administrativa sobre el objeto.

- Los endpoints servidores que no sean el principal, están realmente asociados a endpoints servidores. Es decir, todo endpoint servidor existe fundamentalmente para admitir invocaciones que lleguen al mutador desde nodos remotos o para entregar los mensajes de actualización. Si no modificamos el algoritmo HGD, los mutadores servidores que no sean el principal serán recolectados como residuos cuando su mutador descarte sus referencias y el endpoint no esté retenido. Lógicamente este modo de funcionamiento no es admisible para objetos replicados, pues los mutadores servidores deben ser accesibles a través de sus endpoints para recibir invocaciones desde el exterior aunque el mutador no tenga referencias al objeto. Por tanto, ningún endpoint servidor deberá ser recolectado hasta que el objeto sea un residuo.

4.6.3 Algoritmo sin considerar migración del endpoint principal

A continuación describimos las modificaciones que incluimos sobre el algoritmo HGD para soportar objetos replicados y móviles suponiendo que los objetos mantienen el mismo endpoint

principal durante toda su existencia. Llamemos MEP al autómata que ejecuta el endpoint principal, CEP al que ejecutan los clientes y REP al que ejecutan los endpoints servidores que no sean el principal.

- El autómata CEP no sufre alteraciones. La única consideración radica en que los mensajes que anteriormente decíamos que se enviaban al servidor, diremos ahora que serán enviados al endpoint principal.

- El autómata REP es idéntico al autómata CEP con las siguientes salvedades:

  – En los lugares donde se eliminaba el endpoint (asignándole a $status$ el valor $null$) en el autómata CEP, haremos que $status$ valga $waiting$ en REP. Este estado significa que el mutador no tiene referencias, que el endpoint no está retenido, pero que continuará existiendo hasta que se reciba un mensaje de objeto replicado no referenciado. Nótese que el mensaje DEC sí habrá sido enviado. En este estado, el endpoint podrá recibir una nueva referencia mediante $rref$, actuando en este caso de igual forma a como procedía el autómata CEP al recibir una referencia estando en estado $null$, es decir, tratándose como el caso de un cliente que recibe una referencia que antes no tenía.

  – Añadimos la acción de entrada $runref$ que será ejecutada cuando se le entregue al endpoint un mensaje de tipo $UNREF$. Esta acción destruye el endpoint haciendo que $status$ sea $null$. Nótese que en estos casos el autómata estaría previamente en estado $waiting$.

  – Añadimos la acción de salida $dunref$ cuyo funcionamiento será idéntico al de la acción $dunref$ del autómata SEP.

- El autómata MEP es idéntico al autómata SEP salvo para el código de la acción $rref$. Cuando el contador del endpoint principal transite de 1 a 0 en la acción $rref$, se enviará un mensaje $UNREF$ a los demás endpoints servidores. Después de realizar esto, se cambiará el valor de la variable $status$ de igual forma a como se realizaba anteriormente.

- Por lo que respecta a los mutadores, lógicamente tendremos mutadores servidores conectados a los autómatas REP y MEP, mientras que seguiremos teniendo mutadores cliente asociados a los autómatas CEP. Nótese que los mutadores servidores asociados a los autómatas REP partirán de un estado inicial sin referencias al objeto ($hold = false$).

Es sencillo observar cómo se preservan las propiedades de seguridad y viveza con estas modificaciones. Las acciones que modifican el valor de los contadores, las que procesan referencias y las que tratan los mensajes INC y DEC no han sido modificadas, por lo que los invariantes seguirán manteniéndose. La única diferencia radica en que los autómatas REP, en lugar de destruirse cuando el mutador local libere sus referencias, se quedarán en estado $waiting$ hasta que reciban un mensaje $UNREF$ o hasta que reciban otra referencia. Es trivial demostrar que el primer caso ocurrirá si y solo si el objeto es un residuo.

4.6.4 El problema de la migración

Si el endpoint principal puede migrar, debemos considerar qué efecto debe producir en la cuenta de referencias la migración, y posteriormente qué acciones deberán ejecutarse cuando los mensajes del algoritmo alcancen a un endpoint, que habiendo sido el principal, haya migrado o haya iniciado el proceso de migración. De igual forma, será posible que le llegue a un endpoint no principal un mensaje proveniente de un nodo cuyo localizador esté más actualizado o más obsoleto que su propio localizador.

El problema fundamental que nos encontramos en nuestro protocolo para afrontar las situaciones de migración, consiste en lograr un intercambio de roles entre el endpoint principal antiguo y el nuevo, que no viole los invariantes y que garantice viveza. Ya que el sistema no es síncrono, el intercambio de roles no puede ser atómico y por tanto hará falta realizarlo modificando sustancialmente el protocolo de cuenta de referencias actual.

La aproximación que adoptamos se basa en mantener en cada endpoint cliente (o réplica) una etiqueta que indique si el endpoint tiene la certeza de que el endpoint principal que él conoce, le tiene incluido en su contador. Gracias a esta etiqueta, cuando el endpoint migre a otra ubicación, podremos transformar el rol del antiguo endpoint principal, desde rol de principal a rol de cliente, y podremos transformar el rol del nuevo endpoint principal, desde el rol de cliente a rol de principal. Para ello incrementaremos el contador del nuevo endpoint principal, reflejando con ello que el contador incluye al endpoint principal antiguo como si éste último hubiera recibido una referencia del nuevo principal. De igual forma, deberá decrementarse el contador del endpoint principal antiguo, pero sólo cuando se tenga la certeza de que su contador incluye al endpoint principal.

4.6.5 Algoritmo de cuenta de referencias para objetos móviles

En cierto instante tenemos un conjunto de endpoints cliente, y un conjunto de endpoints servidores. De entre los servidores, como máximo uno será el endpoint principal. Es decir, como máximo uno tendrá su variable rol a $MAIN$. Cada endpoint que no sea principal, tiene un localizador principal que apunta al endpoint al que él considera como principal. Estos endpoints apuntados como si fueran el principal, podrán estar con rol $MAIN$, con rol $MOVING$ o con rol $MOVED$. El objetivo de este algoritmo consiste en garantizar seguridad y viveza en las notificaciones de unreferenced y en garantizar seguridad y viveza en la recolección de los endpoints que se encuentren con rol $MOVED$. La recolección consistirá en hacer que transiten del rol $MOVED$ al rol REP o CEP en función de si tienen réplicas del objeto o no. Una vez que hayan dejado de estar con el rol $MOVED$ podrán recolectarse de la misma forma como se realizaba en el algoritmo sin considerar migraciones.

Estado del algoritmo y contenido de los mensajes

Todo endpoint tiene además del contador de referencias, del localizador, del contador de configuraciones, del contador de migraciones y del OID, la *etiqueta de registrado* y la *lista de cancelaciones de registro* o lista de cancelaciones para abreviar. La etiqueta de registrado se utilizará en los endpoints que no sean el principal para indicar si se tiene constancia de que el principal tiene una unidad de su contador debida a la existencia del propio endpoint. Por su

parte, la lista de cancelaciones se mantiene en cada endpoint no principal para almacenar los registros iniciados desde el endpoint local, cuyo efecto debe deshacerse en el endpoint remoto. Este "des-registro" se realizará cuando el endpoint remoto indique que ha finalizado el registro mediante el correspondiente mensaje de decremento.

Recordemos que el registro se produce cuando un nodo recibe su primera referencia y el emisor no era el principal. En este caso, el receptor de la referencia le envía un INC al principal indicando como argumento el endpoint que envió la referencia. El registro finaliza cuando el principal envía un DEC tanto al donante de la referencia como al receptor de la misma. Ahora estamos interesados en distinguir las finalizaciones de los registros de los demás mensajes DEC, por lo que utilizaremos mensajes diferentes. Ahora diremos que el principal responde con DEC al endpoint indicado como argumento del INC, pero que responde con $IDEC$ al endpoint que se registra como nuevo cliente.

Los mensajes INC, $IDEC$ y DEC y los mensajes $MOVE\_ACK$ y $MOVE\_NAK$ incorporan el número de migración del endpoint principal. Gracias a este número, un receptor de cualquiera de estos mensajes podrá distinguir entre mensajes enviados en favor de líderes obsoletos, del líder actual, o de lideres para los cuales es necesario registrarse.

El algoritmo

- **Caso 1.- Un endpoint envía una referencia**

  Simplemente se incrementa el contador local.

- **Caso 2.- Un nodo recibe una referencia nueva**[2]

  Se crea el correspondiente endpoint cliente si no existe.

  - 2a.- El emisor de la referencia es el endpoint principal que figura en la propia referencia:

    Se inicializa la etiqueta de registrado a cierto. Esta etiqueta indicará que el endpoint principal de la migración actual tiene en su contador una unidad por cuenta del endpoint local.

  - 2b.- El emisor de la referencia no es el endpoint principal:

    Se emite un mensaje $INC(b)$ hacia el endpoint principal, indicando como argumento el endpoint del que se ha recibido la referencia.

    Se incrementa el contador local.

    Se inicializa la etiqueta de registrado a falso. Hasta que no finalice el registro con el servidor, la etiqueta permanecerá a falso, indicando que no existe certeza de que el servidor haya incrementado su contador para reflejar la existencia del endpoint que recibe la referencia.

- **Caso 3.- Un nodo recibe una referencia que ya tiene**

  Se envía un DEC al emisor.

[2] O lo que es lo mismo, un endpoint en estado $waiting$ o $null$ recibe una referencia.

- **Caso 4.- Un nodo recibe** DEC

  Se decrementa el contador. Si el contador llega a cero, pueden ocurrir varias situaciones:

  - 4a.- El endpoint es un endpoint cliente:

    Si no hay referencias locales, se recolecta el endpoint y se envía un DEC al endpoint principal.

    Si hay referencias locales, no se hace nada.

  - 4b.- El endpoint es un endpoint servidor no principal:

    Si no hay referencias locales, se envía un DEC al endpoint principal y el estado pasa a ser $waiting$. La etiqueta de registrado pasa a tomar el valor falso.

    Si hay referencias locales, no se hace nada.

  - 4c.- El endpoint es el principal:

    El objeto replicado es un residuo: se emite $UNREF$ a todos los endpoints servidores y se recolecta el propio endpoint.

- **Caso 5.- Se descarta la última referencia**

  Si el contador es 0, se realiza la misma tarea que en el caso 4,

  Si el contador es mayor que 0 y el endpoint es de tipo cliente, el endpoint quedará en estado retenido.

- **Caso 6.- Un nodo recibe** $INC(D)$

  Es fácil observar que se tratará de un endpoint en estado $MAIN$, en estado $MOVING$ o en estado $MOVED$. Se incrementa el contador, y se envía un mensaje DEC al destino D. Al emisor del mensaje INC ahora no se le enviará un DEC, sino un mensaje $IDEC$, que tendrá un comportamiento ligeramente diferente a los mensajes DEC.

- **Caso 7.- Un nodo recibe** $IDEC$

  Este mensaje lo recibe un endpoint cuando ha finalizado el proceso de registro con el endpoint principal. Sigue tratándose de un mensaje de decremento, por lo que se decrementa el contador.

  Si la lista de cancelaciones de registros contiene el contador de migraciones incluido en el mensaje $IDEC$, se envía un DEC al emisor del $IDEC$ y se elimina dicho contador de migraciones de la lista de cancelaciones. En esto consiste el "des-registro": en deshacer el incremento en el servidor, del que ya se tiene constancia.

  Hasta el momento no hemos visto en qué momento se introducen elementos en la lista de cancelaciones. Esto ocurrirá en determinadas situaciones provocadas por la ejecución concurrente del protocolo de migración que vimos en el apartado 3.3.2.

- **Caso 8.- Un nodo envía MOVE_INI**

  Este mensaje se envía para solicitar convertirse en endpoint principal.

Para evitar que el endpoint sea recolectado mientras dura la migración, se incrementa el contador de referencias.

El rol pasa a $MAINREQ$

- **Caso 9.- Un nodo recibe MOVE_INI**

  Si se encuentra con rol $MOVED$, responde con $MOVE\_NAK(l)$ enviando como argumento el nuevo endpoint principal. Previamente el contador habrá sido incrementado, pues el envío del $MOVE\_NAK$ en este caso es similar al envío de una referencia.

  Si se encuentra con rol $MOVING$, bloquea el mensaje hasta que se transite al rol $MOVED$.

  Si se encuentra con rol $MAIN$, incrementa su contador para evitar que el endpoint sea recolectado, se pasa al rol $MOVING$ y se responde con $MOVE\_ACK$. Previamente se habrá puesto el valor de la etiqueta de registrado a cierto. Con la etiqueta se indica que el incremento que efectuó el endpoint que pide ser el principal al enviar el $MOVE\_INI$ no será deshecho.

- **Caso 10.- Un nodo recibe MOVE_ACK**

  - 10a.- Si la etiqueta de registrado está a falso:

    La etiqueta a falso significa que anteriormente iniciamos el registro con el endpoint principal que nos acaba de conceder el rol de principal, y que éste todavía no ha contestado con el correspondiente $IDEC$. Debido a esto, no tenemos certeza de que el endpoint principal antiguo tenga en su contador una unidad correspondiente a la existencia del endpoint local.

    Se añade el contador de migraciones a la lista de cancelaciones de registros.

  - 10b.- Si la etiqueta de registrado está a cierto:

    La etiqueta a cierto significa que la primera referencia que ocasionó la creación del endpoint la recibimos del endpoint que ahora deja el rol de principal, o que el registro que iniciamos enviándole un INC finalizó al recibirse el correspondiente $IDEC$.

    Se envía un DEC al endpoint que deja el rol de principal. Con este DEC estamos desregistrando al endpoint que va a ser el principal del endpoint principal antiguo.

  Se transita al rol $MAIN$.

  Se instala como endpoint principal, el endpoint local y como contador de migraciones el contador que se haya recibido en el mensaje.

  Se emite un mensaje $MOVE\_END$ al emisor del $MOVE\_ACK$

- **Caso 11.- Un nodo recibe MOVE_NAK**

  Significa que no se ha concedido la migración por haber sido concedida a otro endpoint. En este caso se deben realizar dos funciones: primero, desregistrar al endpoint local del que hasta ahora era el principal. El desregistro va a ser similar al efectuado al recibirse

el mensaje $MOVE\_ACK$. Posteriormente se iniciará el registro con el nuevo endpoint principal y a continuación se solicitará a este endpoint el rol de principal.

- 11a.- Si la etiqueta de registrado está a falso:
  Se añade el contador de migraciones a la lista de cancelaciones de registros.
- 11b.- Si la etiqueta de registrado está a cierto:
  Se envía un DEC al emisor del $MOVE\_NAK$. Con este DEC estamos desregistrando al endpoint local.

Se envía un $INC(O)$ al nuevo endpoint principal. El argumento es el endpoint principal que nos ha respondido con $MOVE\_NAK$. Con este envío estamos registrando al endpoint local con el nuevo endpoint principal.

Se pone la etiqueta de registrado a falso. Con la etiqueta indicamos que no hemos recibido el correspondiente $IDEC$.

Instalamos como localizador principal y como contador de migraciones, el localizador y el contador que se han recibido en el mensaje $MOVE\_NAK$.

Se le solicita al nuevo endpoint principal el rol de principal. Para ello, se envía el correspondiente mensaje $MOVE\_INI$ al nuevo destino. Nótese que en el envío previo del mensaje $MOVE\_INI$ ya se habrá incrementado el contador y se habrá transitado al rol $MAINREQ$, por lo que ahora no debe hacerse.

- **Caso 12.- Un nodo recibe** $MOVE\_END$

  Se decrementa el contador local y el endpoint pasa al rol $MOVED$.

El algoritmo se ha complicado sustancialmente por la ejecución concurrente del protocolo de migraciones. Sin embargo, sigue tratándose de un algoritmo distribuido de cuenta de referencias eficiente en espacio y mensajes. Lo habitual para cualquier objeto es que no ocurran demasiadas migraciones y que todo lo más estas sucedan de una en una, con lo que las listas de cancelaciones estarán habitualmente vacías y en muy pocas ocasiones tendrán un elemento. Para que tengan más de un elemento, debe ocurrir, no sólo que ocurra más de una migración sobre un objeto en un intervalo pequeño de tiempo, sino además debe suceder que los mensajes $IDEC$ tarden más en llegar a los endpoints que inician la migración, que los mensajes $MOVE\_NAK$ o $MOVE\_ACK$, lo que generalmente tampoco ocurrirá.

Por tanto en ejecuciones ordinarias, el estado que mantiene cada endpoint sigue siendo prácticamente constante.

La seguridad y viveza del algoritmo puede deducirse de un análisis detallado de casos similar al que empleamos para demostrar el algoritmo de cuenta de referencias sin migración de objetos.

Reconexión de los clientes

Para lograr que un cliente obsoleto se reconecte con el nuevo endpoint principal es necesario que el cliente obsoleto reciba una referencia actualizada. Esto podrá ocurrir de forma natural,

simplemente porque coincida que algún nodo se la envíe, o forzada, porque el nodo obsoleto emita algún mensaje que alcance a algún nodo actualizado. En este segundo caso, el nodo actualizado le enviará un mensaje de tipo $NREF$ al nodo obsoleto para que actualice su referencia. El mensaje $NREF$ tendrá un efecto similar al que tendría el envío de una referencia que el receptor descartara a continuación.

El procedimiento de reconexión es similar al que se realiza al recibir un mensaje de tipo $MOVE\_NAK$.

Un cliente que recibe una referencia con contador de migraciones más reciente que el contador de migraciones local, realiza de forma consecutiva los procesos de des-registro del endpoint principal antiguo y de registro con el nuevo endpoint principal.

- **Des-registro**:
  - A.- Si la etiqueta de registrado está a falso:

    Se añade el contador de migraciones a la lista de cancelaciones de registros con lo que se desregistrará al endpoint local cuando se reciba el $IDEC$.
  - B.- Si la etiqueta de registrado está a cierto:

    Se envía un DEC al endpoint principal antiguo. Con este DEC estamos desregistrando al endpoint local.
- **Registro**: Se envía un $INC(O)$ al nuevo endpoint principal. El argumento es el endpoint principal que nos ha enviado la referencia. Con este envío estamos registrando al endpoint local con el nuevo endpoint principal.

  Se pone la etiqueta de registrado a falso. Con la etiqueta indicamos que no hemos recibido el correspondiente $IDEC$.

  Instalamos como localizador principal y como contador de migraciones los que se han recibido en el mensaje $MOVE\_NAK$.

4.6.6 Reconstrucción de la cuenta ante fallos

En el apartado 4.5 detallamos las rondas que se ejecutan después de cada caída para reconstruir la cuenta de referencias de los objetos básicos. Para los objetos móviles, la estrategia que adoptamos va ser similar, pero en este caso, prestando especial atención a la elección de un endpoint principal para cada objeto.

Las primeras dos rondas de la reconstrucción no sufren apenas alteraciones: Tal y como se hacía para el caso de referencias de objetos básicos, se impide el envío de mensajes de cuenta de referencias, se anula la recepción de mensajes de cuenta de referencias originados antes de la caída, se detiene el paso de referencias y se fuerza la entrega de las referencias que estuvieran en tránsito por la red.

Además de estas acciones, es necesario realizar algunas tareas adicionales para tratar la migración del endpoint principal y ciertas diferencias de funcionamiento entre los objetos replicados y los no replicados.

Al comienzo de la ronda 1, se realizan los siguientes pasos:

- Se bloquea la recepción de mensajes $MOVE\_INI$, $MOVE\_NAK$ y $MOVE\_END$. Con esto se logra detener las migraciones en curso, excepto aquellas que habiendo alcanzado al líder vayan a tener éxito.

- Se ignorarán los mensajes $MOVE\_END$ que hayan sido emitidos y que todavía no hayan sido entregados. Para ello se utiliza la misma técnica que la empleada para descartar los mensajes antiguos de cuenta de referencias. Mediante el número de reconfiguración del cluster, estos mensajes simplemente serán descartados.

- Se espera a que terminen las acciones relativas a las recepciones de los mensajes $MOVE\_INI$, $MOVE\_END$ y $MOVE\_NACK$. Esta espera puede provocar que se envíen mensajes $MOVE\_ACK$ y $MOVE\_NAK$. Una vez que esto ocurra, ya no se emitirán nuevos mensajes $MOVE\_ACK$ ni $MOVE\_NAK$, puesto que están bloqueada la recepción de mensajes $MOVE\_INI$. Ahora tenemos la garantía de que los mensajes $MOVE\_ACK$ están en camino de su destino.

Al comienzo de la ronda 2, se realizan las siguientes acciones:

- Utilizando la misma técnica que la empleada para las referencias, se espera a que todos los mensajes $MOVE\_ACK$ lleguen a su destino y sean procesados. Gracias a este trabajo, tendremos exactamente un endpoint principal en todos los casos, excepto si la caída ha afectado directamente al principal.

Una vez finalizadas las 2 rondas, de forma concurrente a la ronda 3 del algoritmo de reconstrucción de referencias para objetos básicos, se ejecuta la siguiente ronda para los endpoints de objetos móviles.

Ronda 3: reconstrucción de referencias a objetos móviles

- Cada objeto *ayudante de referencias* espera una invocación de cada uno de los demás nodos. Recordemos que el objeto ayudante de referencias es un objeto fijo presente en todo nodo. El argumento de la invocación es una lista de tuplas <`OID`, `localizadorPlano`, `contadorDeMigraciones`, `rol`>, donde `localizadorPlano` es el localizador de cierto endpoint ubicado en el nodo emisor del mensaje, `OID` es el identificador del objeto representado por dicho endpoint, `contadorDeMigraciones` es el contador de migraciones del endpoint y `rol` una variable que puede tomar uno de cuatro valores: cliente, replica sin referencias, replica con referencias y principal. La variable `rol` será "cliente" cuando el endpoint identificado por `localizadorPlano` sea un endpoint cliente, en otro caso y dependiendo de si se trata de un endpoint servidor con referencias locales o sin referencias locales tomará los demás valores.

  El tamaño de la lista de tuplas que un nodo envía a un ayudante, es el número de endpoints de objetos móviles que existen en el nodo emisor, cuyo OID es *próximo* al número de nodo donde se encuentra el ayudante. La relación *"próximo"* que utilizamos para asociar números de nodo a OIDs, es una sencilla función que aplicada al OID retorna un único nodo. En todo caso la función próximo debe retornar el nodo contenido en el OID si

dicho nodo está vivo, y otro nodo cualquiera en caso contrario, pero calculado de forma determinista y local. Por ejemplo se podría retornar el nodo inmediatamente superior al nodo contenido en el OID que se encuentre vivo (o el menor si no existe ninguno mayor).

- Cuando un objeto ayudante recibe uno de los mensajes, almacena la tupla indexada por OID.

- Cuando un objeto ayudante ha recibido un mensaje de todos los nodos, recorre la lista de tuplas por OID. El número de tuplas almacenada para cada OID indicará el número de endpoints que tiene el objeto. Por su parte el contador de referencias será sencillo de calcular. Si el principal no ha caído, será el número de tuplas que tengan el `rol` distinto de "replica sin referencias" menos uno. Sino, una unidad podrá variar dependiendo de si el endpoint elegido como nuevo principal tenía referencias locales al objeto o no.

  Posteriormente, si existe una tupla con rol "principal", el endpoint correspondiente será el elegido para desempeñar el rol de líder. Si ninguna tupla tiene el rol de "principal", se seleccionará para desempeñar el rol de líder al endpoint correspondiente a la tupla que tenga el mayor contador de migraciones y que tenga el `rol` distinto de "cliente". En este caso, en el que el endpoint principal resulta elegido por no existir previamente, el contador de migraciones se incrementa en una unidad. Si sólo existen tuplas con `rol` igual a "cliente", no existirá objeto, por lo que estas referencias serán recolectadas tan pronto como se intenten utilizar.

  Una vez elegido el nuevo líder, se le envía el contador de referencias que se ha calculado y el contador de migraciones. A los demás endpoints se les envía la identidad del endpoint principal y el contador de migraciones.

- Las respuestas del objeto ayudante, se realizan como respuesta a la invocación que cada nodo ha efectuado sobre el. Al recibir la respuesta a cada tupla, se actualiza el endpoint identificado en ella. Si la respuesta contiene la identidad del endpoint principal y el contador de migraciones, se actualizan estos valores en el endpoint. Por contra, si la respuesta del objeto ayudante contiene el contador de referencias, se instala el rol $MAIN$ en el endpoint y se almacena el contador de referencias y el contador de migraciones.

- Cuando cada nodo ha recibido respuesta de todos los objetos ayudante, se da por finalizada la ronda en dicho nodo.

- Al llegar a este momento, los endpoints principales tendrán la cuenta exacta de nodos con referencias, con lo que a continuación se procederá a emitir las notificaciones de no referenciado

En la cuarta ronda, se realizarán las mismas acciones que para las referencias a objetos básicos, es decir desbloquear la cuenta de referencias y el paso de referencias, pero habiendo habilitado previamente los mensajes del protocolo de migración del endpoint principal.

4.7 Trabajo relacionado

Existen fundamentalmente dos técnicas de recolección de residuos: las basadas en cuenta de referencias [25] y las basadas en trazas de referencias [91, 32]. Ambas técnicas surgieron inicialmente para sistemas centralizados y posteriormente, con el desarrollo de los sistemas distribuidos, han ido apareciendo variaciones de los algoritmos originales, para considerar las particularidades de los sistemas distribuidos: coste de las comunicaciones, fallos y asincronía en los mensajes.

En esta sección describiremos los algoritmos y sistemas más relevantes de recolección de residuos, centrándonos en exclusiva en su adaptación a los sistemas distribuidos, y prestando especial atención a la recolección de residuos acíclicos en general, y a los protocolos de cuenta de referencias en particular.

Para describir algunos de los protocolos haremos uso de la nomenclatura Hidra, y llamaremos endpoint servidor al soporte que se mantiene en el nodo donde reside cada objeto, y endpoint cliente al soporte que encontramos en los nodos que mantienen referencias remotas.

4.7.1 Cuenta de referencias

La cuenta de referencias [25] se basa en mantener en cada objeto un contador que indicará cuántas referencias le apuntan. Cada vez que se duplica una referencia, se incrementa el contador y cada vez que se descarta, se decrementa el contador. Cuando el contador llega a cero, el objeto es un residuo.

La adaptación de la cuenta de referencias a entornos distribuidos debe afrontar la problemática que aparece por el hecho de tener desacoplados a los objetos de sus referencias. En un sistema distribuido, cada objeto reside en un nodo y ese nodo o cualquier otro nodo pueden disponer de referencias al objeto. Cualquier nodo que disponga de una referencia puede duplicarla y enviarla a otro nodo, lográndose de esta forma que este último nodo también disponga de la referencia. Para lograr que la cuenta de referencias se mantenga actualizada en el objeto, ya no bastará con realizar incrementos y decrementos locales, se deberán enviar mensajes desde los nodos remotos al nodo servidor del objeto. Si tenemos un objeto O que reside en el procesador S_O y otro procesador P duplica una referencia del objeto O para enviarla al procesador Q, será necesario enviar un mensaje de incremento al procesador S_O. De igual forma, cuando se descarte una referencia remota, se deberá enviar un mensaje de decremento a S_O. Sin embargo, esta simple extensión de la cuenta de referencias no preserva el invariante de seguridad de la cuenta de referencias que existía en la versión para uniprocesadores. Los problemas que aparecen son fundamentalmente dos:

- Si el procesador P que duplica la referencias para enviarla a Q, descarta su referencias inmediatamente después de haberla duplicado, podrá ocurrir que el mensaje de decremento que se envía causado por el descarte llegue al objeto O antes de que le llegue el mensaje de incremento causado por la duplicación de la referencia. Este problema aparecerá si el canal de comunicación no respeta el orden FIFO.

- Si el procesador P duplica la referencia y se la envía a Q y el procesador Q nada más recibir la referencia, la descarta, podrá ocurrir que el mensaje de decremento causado por

el descarte le llegue antes al objeto O que el mensaje de incremento enviado por P con motivo de la duplicación. En este caso, el problema no aparece por ausencia de orden FIFO, sino por la propia naturaleza asíncrona de los sistemas distribuidos, y en particular por la naturaleza causal [77] de las comunicaciones.

En [80], se soluciona el segundo problema mediante un proceso de registro más elaborado que el mero envío de un incremento. El proceso de registro de referencias funciona de la siguiente forma: cada endpoint servidor mantiene el contador de referencias y además todo endpoint, incluido el servidor, tiene dos contadores adicionales. Los dos contadores adicionales se utilizan durante el registro de las referencias y se llaman iR y aR. El contador aR lleva la cuenta del número de referencias recibidas que han sido confirmadas por el servidor, mientras que iR lleva la cuenta del número de reconocimientos que faltan por llegar. El contador iR podrá tener un valor negativo si los mensajes de reconocimiento llegan al endpoint antes que las propias referencias. Cuando un nodo A le envía una referencia a un nodo B de un objeto situado en S, El nodo A le envía en ese mismo instante un mensaje $AR(B)$ (petición de confirmación) al nodo S. Cuando el servidor reciba el mensaje $AR(B)$, incrementa el contador de referencias y envía un mensaje ACK al nodo B. Cuando B recibe una referencia, incrementa el contador aR si iR es negativo y en cualquier caso incrementa iR. Cuando B recibe un mensaje ACK, si iR es positivo incrementa aR, y en cualquier caso se decrementa iR. En lo que se refiere a los descartes, estos son retrasados hasta que el contador aR sea mayor que cero. En ese instante se decrementará aR y se enviará un mensaje de decremento al servidor. Este protocolo tiene fundamentalmente dos desventajas: la primera, que requiere dos contadores en cada endpoint cliente, y la segunda, que no soluciona el primer problema, es decir, exige orden FIFO en la red de comunicaciones.

Más recientemente, junto a la descripción del prototipo de sistema operativo Solaris-MC, se describió un protocolo de cuenta de referencias que plantea un mecanismo de registro diferente [8, 73]. Este algoritmo no exige orden FIFO en los mensajes. El problema de la duplicación remota de referencias se soluciona en Solaris-MC de la siguiente forma: todos los endpoints disponen de un contador inicializado a cero. Si un nodo P le envía una referencia del objeto O al nodo Q, siendo S_O el nodo servidor de O, el contador de P se incrementa en una unidad para proteger al endpoint contra descartes. Cuando el endpoint del nodo Q recibe la referencia, incrementa su contador y envía un mensaje INC a S. El incremento del contador local se hace de nuevo para proteger al endpoint contra descartes. Cuando S recibe el mensaje, incrementa su contador responde con un ACK. Cuando Q recibe el ACK, decrementa su contador y le envía un DEC al nodo P, que al recibirlo decrementará su contador. Este esquema tiene la desventaja de que los nodos que reciben referencias, deben recordar la identidad de los nodos que les ceden las referencias hasta que el servidor responda con el correspondiente ACK.

Otra variación aparecida recientemente [94] también incorpora un mecanismo de registro, pero al igual que sucedía con el primer algoritmo que hemos descrito, continúa exigiendo orden FIFO. En este algoritmo, el receptor de la referencia enviada por otro cliente, envía un mensaje $INC - DEC$ al servidor, el cual enviará un DEC al nodo que envió la referencia. Puede observarse que este esquema soluciona el segundo de los problemas que presenta la cuenta distribuida de referencias, pero continúa exigiendo orden FIFO.

Por último, la última variación de la cuenta de referencias, la presentamos en esta tesis como

parte de la arquitectura Hidra [50]. Nuestra aproximación radica en un proceso de registro asíncrono. De igual forma a como se realiza en Solaris-MC, todos los endpoints disponen de un contador inicializado a cero. Si un nodo P le envía una referencia del objeto O al nodo Q, siendo S_O el nodo servidor de O, el contador de P se incrementa en una unidad para proteger al endpoint contra descartes. Cuando el endpoint del nodo Q recibe la referencia, incrementa su contador y envía un mensaje $INC(Q)$ a S. El incremento del contador local se hace de nuevo para proteger al endpoint contra descartes. Cuando S recibe el mensaje, incrementa su contador responde con dos mensajes DEC, uno dirigido a Q y otro a P. Cuando P y Q reciben los mensajes DEC, decrementan sus respectivos contadores.

Como puede observarse nuestro esquema es el más asíncrono de todos los estudiados, tiene un consumo de espacio muy reducido: tan sólo un entero por endpoint y no exige orden FIFO en los mensajes.

Cuenta de referencias sin mensajes de incremento

Otras variantes de los protocolos de cuenta de referencias intentan solucionar el problema de la duplicación remota de referencias evitando los mensajes de incremento. Esta es la aproximación que se sigue, tanto en los protocolos de cuenta ponderada de referencias [10, 135], como en los protocolos de cuenta de referencias generacional [59], como en la cuenta de referencias indirecta [117].

En la cuenta de referencias ponderada [10, 135], se evitan los incrementos restringiendo el número de referencias que pueden transitar por la red. Para ello, al crearse un endpoint servidor, se inicializa su contador a cierto valor w que restringirá el número de duplicaciones que se podrán efectuar. Cada vez que un endpoint duplique una referencia para enviarla, su contador se dividirá por dos y se enviará la referencia al destino con la otra mitad del contador. Cuando se descarte una referencia, se le enviará el contador del endpoint al servidor, que incrementará su propio contador en tantas unidades como indique el contador que reciba. Cuando el contador del servidor alcance el valor w, el objeto será un residuo. Cuando el contador de cierto endpoint alcanza el valor 1, no podrá enviar más referencias al exterior. Como una posible solución, se plantea la posibilidad de crear en estos casos un nuevo endpoint servidor con un nuevo contador inicializado con el valor w. El endpoint servidor se ubicaría en el mismo nodo donde no se pudo duplicar la referencia. De esta forma, se podrán enviar más referencias, al precio de consumir más recursos y crear un nivel de indirección adicional.

La cuenta de referencias generacional [59], se basa en una idea similar a la presentada por la cuenta ponderada de referencias. En la cuenta generacional, el endpoint servidor tiene un vector de enteros llamado *ledger* y todo endpoint, incluido el servidor tiene un contador y un número de generación. Cuando un objeto O se crea en S_O, su contador y el número de generación se inicializan a cero. Por su parte todas las componentes del vector se inicializan a cero excepto la primera que se inicializa a 1. Cada componente i-ésima del vector *ledger* mantiene información sobre el número de referencias de la generación i-ésima que existen. El funcionamiento del protocolo es el siguiente: Cuando el procesador P duplica una referencia para enviarla a Q, crea la copia de la referencia, inicializando su generación a la inmediatamente siguiente a la generación del endpoint de P e inicializa el contador a 0. Por su parte el contador del endpoint de P se incrementa en 1. Cuando se descarta una referencia, se envía un mensaje de descarte al

endpoint servidor incluyendo la generación del endpoint que se destruye junto con su contador. Cuando el mensaje alcanza al servidor, éste actualiza su estado de la siguiente forma: sea i la generación enviada en el mensaje de descarte y c el contador enviado, entonces se decrementa la componente $ledger[i]$ en una unidad y se incrementa la componente $ledger[i+1]$ con el valor del contador recibido. Un objeto será residuo si y solo si todas las componentes de ledger valen 0. La desventaja de este algoritmo respecto a la cuenta ponderada de referencias radica en su mayor consumo de espacio. De forma similar a como ocurre con la cuenta ponderada, también se impone un límite a la cantidad de duplicaciones de referencias que se pueden efectuar. En este caso resulta incluso más complejo calcular y por tanto ajustar esta cota.

La cuenta indirecta de referencias [117] se basa en mantener en cada endpoint un contador y un puntero que apunte al endpoint que le envió la referencia. A este puntero hacia detrás se le denomina *parent* y con él se mantiene el árbol invertido generado por las difusiones de referencias. Cada vez que se envía una referencia, se incrementa el contador del endpoint que envía la referencia. Cuando un nodo recibe una referencia, comprueba si se trata de una referencia nueva para el nodo o si por el contrario ya existía el endpoint correspondiente a dicho objeto previamente. Si la referencia es nueva, se crea un endpoint cliente y se almacena en su puntero *parent* la identidad del endpoint que envió la referencia. Si el endpoint ya existía, se envía un mensaje de decremento al endpoint que ha enviado la referencia. Cuando un nodo descarta sus referencias, sólo destruye el endpoint si el contador correspondiente es cero. Si es mayor que cero, el endpoint quedará retenido. Cuando el contador de un endpoint cliente sea cero y no tenga referencias locales, se recolectará el endpoint y se enviará un decremento al endpoint registrado como donante de la primera referencia en el endpoint. Cuando el contador del endpoint servidor llegue a cero, el objeto será un residuo. La clave de este algoritmo consiste en mantener el árbol resultante de los envíos de referencias efectuados. El árbol se irá recolectando conforme los nodos de las hojas descarten sus referencias. La desventaja de este algoritmo radica en la generación de bastantes residuos intermedios con el consumo de espacio que esto supone.

Listas de referencias

Otro grupo de algoritmos intentan paliar la deficiente tolerancia a fallos que presenta la cuenta de referencias. En la cuenta de referencias, incluyendo las variaciones que no utilizan mensajes de incremento, no se toleran caídas de nodos, pérdidas de mensajes, ni duplicaciones de los mensajes.

Para solucionar este aspecto, varios algoritmos se han planteado [128, 17] basándose en la sustitución del contador del endpoint servidor por una lista de localizadores. Cada localizador apuntará a cada nodo que disponga de endpoints clientes y que haya sido registrado con el servidor. Gracias a mantener la lista de ubicaciones de los clientes, el servidor podrá comprobar si los clientes están vivos. Adicionalmente, el hecho de mantener la lista completa de clientes en el servidor convierte en idempotentes los mensajes de decremento, con lo que también se argumenta su mejor tolerancia a pérdidas y duplicaciones de mensajes.

Respecto a los mensajes de incremento, sigue apareciendo el mismo problema que describimos como problema 2 en la cuenta de referencias: es necesaria alguna forma de registro en el servidor de los clientes que reciben sus referencias de otros clientes. En los *Network Objects*

[17] se opta por una solución síncrona. El emisor de una referencia, retiene su endpoint hasta que recibe un mensaje de reconocimiento del nodo receptor de la referencia. Este reconocimiento lo enviará el receptor de la referencia al registrarse con una invocación síncrona llamada *dirty*. Por tanto en esta solución, se mantienen sincronizadas las acciones de los mutadores con la actividad del recolector de residuos, restándole asincronía.

En el mecanismo *Stub Scion Pairs Chains* (SSPC) [128], se opta por una solución comparable a la proporcionada por la cuenta de referencias indirecta [117]. La diferencia fundamental radica en que en lugar de mantenerse el árbol de difusión de referencias invertido, se mantiene el conjunto completo de cadenas generadas al difundirse las referencias. Cuando se exporta una referencia a objeto se crea un *scion* y el receptor de la referencia crea un *stub*. Si este último nodo transmite la referencia a otro nodo, se crea otro scion y el receptor creará su stub adicional y así sucesivamente. Gracias a que se mantiene la cadena completa de pares *stub-scion*, no es necesario el registro de las referencias en el servidor. El precio que se paga por ello, consiste en la creación de múltiples entidades intermedias, cuya existencia puede ser simplemente debida a la necesidad de no romper la cadena. Por tanto, de forma similar a como ocurre con la cuenta de referencias indirecta, se crean residuos intermedios que escapan a los objetivos del recolector. La ventaja que se obtiene por este mayor consumo de espacio, consiste en la ausencia de protocolo de registro, con lo que la tolerancia a fallos en los mensajes es total.

4.7.2 Trazas de referencias

El defecto principal que se achaca a los protocolos de cuenta de referencias en cualquiera de sus variaciones radica en su incapacidad para detectar ciclos de residuos. Los protocolos basados en trazas de referencias [39], al basarse en una relación de alcanzabilidad, son capaces de recolectarlos. Existen dos tipos de protocolos basados en trazas: los de marcado y barrido [32, 40] y los basados en *backtracking* [88]. Los primeros parten de un conjunto de objetos que forma el conjunto raíz y a partir de este conjunto, se marcan los objetos alcanzables desde ellos. Cuando se hayan examinado todos los posibles caminos desde el conjunto raíz, los objetos que no estén marcados serán residuos. Por su parte los protocolos basados en backtracking parten de cierto objeto y a partir de él se retrocede hasta que se alcance algún objeto raíz o hasta que se agoten todos los caminos hacia detrás. En el primer caso, el objeto será alcanzable y en el segundo será un residuo. Si en el segundo caso, alguno de los caminos hacia detrás alcanza al propio objeto, además el objeto estará formando parte de un residuo cíclico.

La diferencia fundamental que existe entre ambos esquemas radica en que los protocolos de marcado y barrido intentan detectar todos los residuos y los de backtracking intentan discernir si un objeto dado es residual.

Por último, otra característica fundamental de estos protocolos consiste en su intrínseca tolerancia a fallos. Gracias a que cada ejecución de los protocolos de marcado y barrido no depende de ejecuciones previas, pueden reiniciarse sin problema alguno cuando aparezca un fallo de caída.

4.7.3 Soluciones híbridas

Hay varios sistemas que han combinado protocolos de cuenta de referencias con protocolos de trazas de referencias [30, 123]. Con esta combinación, se logra la eficiencia de la cuenta de referencias y la tolerancia a fallos y/o la capacidad de recolección de residuos cíclicos de los esquemas de trazas de referencias.

En nuestro caso, Hidra hereda la aproximación planteada en Solaris-MC [8] por la que se combina un protocolo de cuenta de referencias con otro de marcado y barrido para recuperar los invariantes del protocolo de cuenta de referencias en caso de fallos. El modelo de objetos de Hidra desaconseja la detección de residuos cíclicos, que en caso de requerirse se lograría con escaso coste adicional.

4.7.4 Recolección de residuos para objetos móviles

No demasiados trabajos describen sistemas de recolección de residuos para objetos que puedan migrar. Encontramos excepciones en [117, 128]. En todo caso, no hemos encontrado ningún trabajo que muestre un protocolo de cuenta de referencias basado en simples contadores junto con un protocolo de migración.

En [118] se argumenta cómo los protocolos indirectos [117] soportan la migración. La idea de este argumento radica en que los protocolos indirectos mantienen el árbol invertido generado por las difusiones de referencias desde el nodo propietario del objeto a los demás nodos. Gracias a los punteros hacia el donante de la referencia, la migración de un objeto de un nodo a otro consiste en asignar el puntero *parent* de la antigua ubicación del objeto a la nueva, incrementar el contador de la nueva ubicación, decrementar el contador de la vieja ubicación y asignar el valor `null` al puntero *parent* de la nueva ubicación. El intercambio de una unidad en los contadores se realiza mediante un sencillo mensaje de decremento enviado desde la nueva ubicación del objeto a la previa. Para que la nueva ubicación del objeto conozca la ubicación previa del objeto, se argumenta el uso de un segundo puntero, llamado *owner* ubicado en todo endpoint, que apuntará siempre con la máxima precisión posible a la ubicación del servidor. Este puntero es similar al mecanismo de punteros indirectos presentado por Fowler [43], que también fue adoptado por los SSPC y que resulta similar al que hemos descrito para Hidra. El puntero *owner* apuntará siempre a la ubicación del objeto, a no ser que ocurra una migración. De ocurrir, los clientes detectarán esta situación al realizar una invocación al objeto, momento en el que reasignarán su puntero *owner*.

En el mecanismo SSPC [128], la migración se logra extendiendo todas las cadenas que comienzan en el servidor hacia la nueva ubicación. Pero en este caso, el crecimiento de la cadena se produce en sentido opuesto al que ocurre con la difusión de referencias. En la nueva ubicación se creará un scion y en la vieja ubicación un stub apuntará al scion recién creado.

Uno de los mayores problemas que aparece con la migración de objetos, es el tratamiento ante fallos. En Hidra lo solucionamos mediante un protocolo de reconstrucción de referencias basado en OIDs únicos, que solapa la reconstrucción de las cuentas de referencias con la elección de un propietario para el objeto (si había caído) y con la reconexión de los clientes con el servidor. Ninguna de los dos trabajos propuestos afronta los casos de caídas de nodos con profundidad, y todo lo más en [128], se reconoce que pese a la vocación de tolerancia a fallos

del mecanismo SSPC, cuando ocurren migraciones y fallos, se requiere el uso de un protocolo de búsqueda exhaustiva de referencias, que no se describe, y que en nuestra opinión, tendrá un coste comparable al nuestro.

Capítulo 5

Depuración basada en eventos

El primer día de la primavera es una cosa y el primer día primaveral, otra diferente. Frecuentemente la diferencia entre ellas es más de un mes.

Henry van Dyke

Contenidos del capítulo

Un *Gestor de Invocaciones a Objeto* es una capa de software distribuido que proporciona el modelo de objetos distribuidos a las aplicaciones que se construyen sobre ella. Como cualquier aplicación distribuida, los ORB's están condicionados por diversas características que provocan que sea especialmente difícil su depuración.

Las técnicas y herramientas tradicionales utilizadas para depurar aplicaciones secuenciales suelen basarse en la posibilidad de detener la ejecución de los programan en cualquier instante y en la facilidad de analizar el estado completo de la memoria. Para ello se hace un uso en general de características propias del hardware subyacente. Estas características han permitido la construcción de herramientas visuales que proporcionan un modelo de depuración cíclica: en cada iteración del proceso de depuración se permite la selección de diversos puntos de interrupción, se permite el análisis de la memoria, y la ejecución selectiva del código. Incluso la depuración de códigos más sofisticados, como por ejemplo el código multitarea de los propios sistemas operativos está solucionado de forma bastante satisfactoria por depuradores nativos que permiten detener tareas, reanudarlas y analizar la memoria cuando aparecen sospechas de código incorrecto.

En los sistemas distribuidos, aspectos tales como la falta de un reloj global, los retrasos arbitrarios que pueden afectar a los mensajes y la ausencia de un estado global consistente accesible de forma sencilla, son fuentes bien conocidas de dificultades [57], que resaltan la necesidad de una estrategia de depuración adecuada.

Las aproximaciones que se han venido tomando para depurar sistemas distribuidos las podemos agrupar fundamentalmente en dos categorías:

- Técnicas orientadas a permitir la ejecución cíclica del sistema.
- Técnicas de evaluación de predicados sobre el estado global del sistema.

Las primeras intentan proporcionar un entorno que permita aplicar la misma aproximación iterativa utilizada para depurar programas secuenciales. Para ello se construyen mecanismos que permitan ejecutar el mismo sistema tantas veces como se desee, analizando en cada iteración los resultados que vaya proporcionando el sistema. Las técnicas de depuración basadas en evaluación de predicados persiguen capturar instantáneas del estado global del sistema [5] y evaluar predicados sobre ellas. Los predicados consistirán en invariantes, precondiciones y postcondiciones que permitirán encontrar condiciones excepcionales, errores y situaciones anómalas. Los dos grupos de técnicas, que revisaremos con mayor detalle en la última sección de este capítulo (sección 5.6 de la página 139) no son incompatibles entre sí, sino más bien complementarias.

Este capítulo lo dedicamos a exponer el mecanismo de depuración que hemos utilizado en Hidra, fundamentalmente para depurar el soporte a objetos. En Hidra, hemos incluido en la propia arquitectura una herramienta de depuración [54, 53], a la que llamamos *Fielt*[1] basada en la evaluación de predicados sobre el estado global. La herramienta permite realizar un análisis *postmortem* del comportamiento del sistema tanto en condiciones normales como frente a casos de fallo. Esto último es especialmente relevante en un entorno como el nuestro donde la corrección de la mayoría de componentes y algoritmos viene determinada por la corrección de la recuperación frente a fallos. La herramienta Fielt proporciona un medio integrado para simular

[1] Acrónimo inglés **F***ault* **I***njection and* **E***vent* **L***ogging* **T**ool.

e inyectar fallos al sistema, de forma que se puedan evaluar predicados sobre el estado global resultante de la aparición de tales fallos.

5.1 Planteamiento de objetivos

Dos son los objetivos principales que hemos perseguido con el diseño de la herramienta Fielt.

- *Ejercitación del ORB:* Pretendemos que el mecanismo de depuración sirva fundamentalmente para ejercitar el ORB y mostrar cómo funciona éste ante situaciones de carga elevada, ante la aparición de errores y ante combinaciones de fallos, tanto en los nodos como en la red de comunicaciones.

- *Monitorización del ORB:* El mecanismo debe servir para monitorizar el comportamiento del ORB en todo instante y particularmente durante la ejercitación del ORB. Gracias a la capacidad de monitorización, deben hacerse patentes los errores que se hayan cometido tanto en fase de diseño como en la misma implementación. Adicionalmente, dado que el número de eventos que puede generar el ORB durante su ejecución puede ser elevado, es necesario que se permita la selección dinámica de qué eventos se van a registrar en cada momento.

Por otra parte, como requisitos técnicos adicionales que debe satisfacer Fielt, nos planteamos la necesidad de que presente un *efecto de pruebas (Probe-effect)*[2] tan pequeño como sea posible. El mecanismo debe ser sencillo de mantener en paralelo al mantenimiento que se realice sobre el código y debe evitar que el programador se confunda con una instrumentación excesiva del software que suele aparecer como código no esencial y como fuente de distracción respecto de la propia funcionalidad que debe implementarse.

La herramienta Fielt logra estos objetivos por medio de técnicas de instrumentación del software aplicadas tanto a la inyección de fallos como a la generación de eventos. La instrumentación del código se realiza de una forma sencilla, poco intrusiva y extensible. El diseño de nuestra herramienta está basado en distintas observaciones que hemos realizado sobre la naturaleza específica de la computación que efectúa un ORB para servir objetos distribuidos.

5.2 Un ORB como sistema distribuido específico

5.2.1 Ejecución de un ORB

Un ORB es una capa distribuida de software que proporciona un modelo de computación basada en objetos a los dominios que lo utilizan. Teniendo un ORB como el mecanismo subyacente, los dominios se construyen como colecciones de objetos que se ejecutan tanto por código local, como invocando operaciones exportadas por objetos que residen en otros dominios. Cada objeto puede ser invocado o puede invocar a otros objetos, actuando respectivamente como objeto

[2] Este efecto no deseable puede ser definido como las perturbaciones que el proceso de depuración introduce en el sistema que se pretende analizar.

servidor o como cliente. El ORB proporciona este mecanismo de invocación de objetos con transparencia de ubicación por medio del concepto de referencia a objeto. Para que un dominio pueda invocar cierta operación sobre un determinado objeto, el dominio debe poseer previamente una referencia válida de este objeto. Las referencias a objeto son interpretadas por el ORB, el cual ejecutará el código y los protocolos adecuados para ocultar la naturaleza distribuida del sistema tanto a los servidores como a los clientes.

Para completar el soporte a objetos, el ORB proporciona además del mecanismo de invocación, otras operaciones que pueden efectuarse sobre las referencias a objeto. Las dos más importantes son: primero una operación para pasar referencias de un dominio a otro y segundo, una operación para descartar referencias. El paso de referencias se logra pasando referencias a objeto como argumentos de las operaciones; como argumentos *in* para los casos en que el cliente que invoca la operación envía referencias al dominio servidor, o como argumentos *out* o como retorno de las operaciones para los casos en que un objeto retorna referencias a objeto al dominio cliente que lo invoque. Por otra parte, cuando un dominio descarta una referencia a objeto, lo que está realizando es informar al ORB de que ya no la va a utilizar más. El objetivo de ambas operaciones, paso de referencias y descarte de referencias, es básicamente permitirle al ORB gestionar las referencias: poder hacer un seguimiento de qué referencias son válidas en cada instante y crear los enlaces adecuados entre referencias a objeto y sus ubicaciones de forma que se permita la invocación remota.

5.2.2 Observaciones sobre la ejecución de un ORB

Invocar un objeto, esperar su respuesta, enviar una referencia a otro dominio, recibir una referencia, descartar una referencia y cualquier otra operación válida sobre referencias a objeto, requiere que se ejecute código distribuido. El código que se ejecutará estará íntimamente relacionado con el objeto asociado a la referencia involucrada en la operación. Más aún, en general, todo código ejecutado por un ORB, se ejecuta para modificar o consultar el estado distribuido asociado a un objeto o al soporte del objeto presente en el propio ORB. Es decir, el ORB puede considerarse como código pasivo que reacciona y es ejecutado ante las operaciones que los distintos dominios efectúan sobre referencias a objeto.

Dado que un ORB se encuentra distribuido entre un conjunto de dominios y de nodos, el código a ejecutar para atender a las operaciones que efectúan los dominios, se encuentra distribuido en los distintos nodos y dominios. De igual forma, el estado necesario para mantener los objetos y las referencias a objeto se encuentra también distribuido. En general, ejecutar una operación sobre una referencia a objeto en particular, implica que se ejecute código y que se acceda a estado distribuido cuya existencia está asociada a la existencia de las referencias y del propio objeto al que éstas apuntan.

Basándonos en estas observaciones de la forma en que se ejecuta un ORB definimos los conceptos de representaciones de los objetos.

5.2.3 Representaciones de los objetos

Definimos el término *representación de un objeto en un dominio* o para abreviar *representación de un objeto*, como el código y el estado que mantiene el ORB en el nodo donde se ubica ese

dominio para manejar dicho objeto.

Los dominios cliente para una objeto en particular, decimos que tienen una representación cliente del objeto, mientras que aquellos que contienen una implementación del objeto decimos que tienen una representación servidora del objeto. De forma análoga, ante cualquier variación en el rol que desempeñe una representación de objeto para un dominio definimos el correspondiente término de representación de objeto. Por ejemplo utilizamos el término representación primaria de un objeto a la representación del objeto asociada a la réplica primaria del modelo pasivo. Análogamente utilizaremos los términos representación secundaria, coordinador, cohorte y principal.

Estas definiciones nos permiten acotar qué código y qué estado es necesario para que el ORB atienda las distintas operaciones que debe ejecutar como respuesta a las peticiones efectuadas por los dominios. Por ejemplo, en el caso sencillo de una invocación a un objeto de implementación única, observamos como tan sólo se ejecuta el código de las dos representaciones del objeto involucradas en la invocación: la representación cliente que inicia la invocación y espera respuesta, y la representación servidora que recibe la petición, la envía a la implementación y envía la respuesta al cliente. Esta observación de que sólo se ejecuta el código de las representaciones de los objetos, la encontramos no sólo en las invocaciones, sino en toda operación que ejecuta el ORB. Esta observación es crucial para el modelo que subyace al esquema de depuración que proponemos en nuestro trabajo.

5.3 Modelo de depuración

La herramienta Fielt está compuesta por dos elementos conceptualmente distintos: un sistema de inyección de fallos que permite la simulación de un amplio abanico de condiciones excepcionales, y una herramienta de monitorización que facilita el análisis post-mortem del comportamiento del sistema. Dado que cada operación que ejecuta el ORB la realiza alguna representación de objeto, para producir eventos con suficiente información para representar las ejecuciones del ORB, cada evento debe incluir información para identificar la representación del objeto que se ejecuta. Esta apreciación nos conduce a registrar trazas de eventos y a inyectar fallos con una aproximación basada en objetos individuales.

5.3.1 Trazas de eventos

Computaciones distribuidas y causalidad

Consideramos que un sistema distribuido está formado por un conjunto de procesos que se comunican con el resto únicamente por medio de paso de mensajes. Cada proceso dispone de su propia memoria local, un reloj local, y es capaz de ejecutar instrucciones de forma secuencial. Las instrucciones que puede ejecutar, pueden ser o bien simples instrucciones que afecten únicamente al estado local, o una de las dos primitivas básicas de paso de mensajes: `send()`, `receive()`. Cada una de estas tres acciones son consideradas como eventos, y las denominaremos respectivamente *pasos de ejecución* (o *pasos* para abreviar), *send* y *receive*. Adicionalmente consideramos que acciones consecutivas ejecutadas por un proceso se ejecutan en pulsos estrictamente crecientes del reloj local del proceso.

Grabando los eventos que se producen durante la ejecución del ORB somos capaces de analizar el comportamiento real del sistema. Para lograrlo, los eventos deben ser ordenados respetando la relación *ocurrió antes* de Lamport. Este orden parcial ordena los eventos, para observar aquellos que puedan haber sido la causa de que se produjeran otros eventos.

De un modo informal definimos el orden causal de la siguiente forma: decimos que un evento producido en un nodo es siempre una causa potencial de todos los eventos que se produzcan con posterioridad en el mismo nodo; un evento de tipo *send* producido en un nodo es la causa potencial del respectivo *receive* que se produce en el nodo al que se dirige el mensaje. Finalmente, decimos que un evento $e1$ producido en el nodo 1, es causa potencial del evento $e2$ producido en el nodo 2, si existe una cadena de eventos que comienza en el evento $e1$ y finaliza en el evento $e2$, de forma que todo evento es causa potencial del evento que le sigue en la cadena.

Eventos

Si tenemos todos los eventos que produce un ORB ordenados respecto al orden causal, es factible construir una instantánea [20] distribuida del estado del sistema e incluso la matriz completa de estados consistentes. Como nuestra aproximación es únicamente habilitar el análisis post-mortem, nos basta con incluir en cada evento un escalar de forma muy similar a como se describe en [42]. Utilizar únicamente un escalar en cada evento proporciona un ahorro considerable respecto a tener que almacenar el *vector tiempo* [90] que sería necesario si nos planteáramos calcular instantáneas consistentes del estado global en tiempo de ejecución. Dado que el análisis post-mortem no es una tarea crítica desde el punto de vista temporal, el coste computacional necesario para construir el vector tiempo de cada evento en base a los escalares que almacenamos en cada evento y el posterior cálculo de los estados consistentes es completamente asumible.

Como conclusión disponemos de un sistema de registro de eventos que requiere almacenar poca información, pero que será suficiente para realizar el análisis del sistema. La información que incorporamos en los eventos es la siguiente:

- Para cada evento producido en un nodo, almacenamos el identificador del evento, que estará formado por el par identificador de nodo y número local de evento. El número local de evento sirve el mismo propósito que el valor escalar que se incluye en los mensajes descritos en [42].

- Para cada evento *receive*, almacenamos el identificador de evento del evento *send* correspondiente al nodo emisor del mensaje.

El modelo

El modelo de ejecución que consideramos aplicable a todo ORB está basado en en el modelo de sistema distribuido que acabamos de describir unido a las observaciones que hemos comentado en el apartado 5.2.2. El modelo lo detallamos enumerando las siguientes características:

- Un objeto está formado por una *red de nodos*, con tantos nodos como dominios haya en el sistema.

- Dentro de esta red, los dominios que mantengan una implementación del objeto diremos que son *nodos servidores*.

- Los dominios que no tengan una implementación del objeto diremos que son *nodos clientes*. Un nodo será cliente tanto si tiene referencias al objeto como si no las tiene.

Los *nodos cliente* pueden ser reales o potenciales. Los nodos cliente reales son dominios que se están ejecutando y que mantienen referencias al objeto, mientras que los nodos cliente potenciales no mantienen tales referencias o pueden incluso no existir en cierto instante de tiempo. Los clientes potenciales se convierten en clientes reales cuando reciben una referencia al objeto. De forma similar los clientes reales se convierten en clientes potenciales si descartan todas las referencias del objeto que mantuvieran o cuando terminan de ejecutarse. Nótese que considerar cada objeto como una red, nos lleva a disponer de tantas redes como objetos existan en el sistema.

En el resto del capítulo utilizaremos el término *red* con letras itálicas para referirnos al conjunto de nodos que constituyen un objeto de la forma en que acabamos de definir, análogamente utilizaremos el término *nodo* con letras itálicas para referirnos a cada uno de los dominios que que forman parte de esta red. Por su parte seguiremos utilizando los términos red y nodo sin tipografía especial para referirnos a las acepciones usuales de ambos términos.

Para completar el modelo, instanciamos de forma natural el resto de los componentes del modelo de computación con los componentes análogos para un objeto en particular:

- Pasos de computación son aquellos pasos de ejecución que consultan o modifican el estado de un *nodo*.

- Los eventos *send* y *receive* son aquellos mensajes transferidos por ORB entre un par de *nodos* de una misma *red*. Es decir, los mensajes transferidos entre los distintos dominios que estén involucrados en la ejecución de una operación sobre una referencia a objeto. En el ORB de Hidra como ejemplos de estos mensajes tenemos los siguientes:

  - Dos mensajes por cada invocación a objeto: la invocación propiamente dicha y la respuesta.
  - Los mensajes asíncronos enviados entre las distintas representaciones del objeto para ejecutar el protocolo de cuenta de referencias.
  - Los mensajes enviados para reconstruir la cuenta de referencias en caso de fallos de caída de nodos.
  - Los mensajes relativos a la migración del endpoint principal.

Nuestro modelo asume que dos representaciones de objeto no comparten estado y que el ORB no ejecuta otro código que no sea servir las operaciones que los distintos dominios efectúen sobre referencias a objeto. Si el ORB necesitara ejecutar otras operaciones, al menos se debe garantizar que éstas no interfieren con las representaciones de los objetos.

Estas asunciones son ciertas para el ORB de Hidra[3] y la propia naturaleza de los ORBs en general nos hace pensar que seguramente serán ciertas en la mayoría de casos.

Por otra parte, el hecho de tener el conjunto de los eventos que genera el ORB particionado en tantas partes como objetos existan, reduce la cantidad de información a manipular cuando se trata de detectar errores. Por ejemplo si una respuesta a una invocación no llega al emisor de la invocación, si se pierde alguna referencia a objeto, si se duplica por error alguna otra referencia, etc, tenemos que la cantidad de eventos a analizar para detectar cuando y donde se produce el error se reduce considerablemente.

5.3.2 Inyección de fallos

Estamos interesados en ejercitar nuestro ORB de forma que, simulando los fallos que puedan ocurrir en un sistema Hidra, nos permita estudiar las acciones de recuperación que ejecute el ORB de manera que podamos evaluar su corrección.

El ORB debe reaccionar correctamente ante fallos. Cuando ocurre un fallo, como por ejemplo la caída de un nodo, cierto código se ejecuta de forma distribuida para recuperar la consistencia en el estado del ORB. Comprobar, depurar y validar éste código, encargado de recuperar al sistema frente a fallos, merece atención especial por diferentes motivos. Primero, porque el código de tratamiento de fallos suele ser más complejo que el código que habitualmente podemos encontrar en las aplicaciones, incluido el código de un ORB que ignore la aparición de fallos. Este código de aplicación suele centrarse en ofrecer cierta funcionalidad de alto nivel, asumiendo que no existen fallos o que estos serán convenientemente tratados. Segundo, debido a que el código de aplicación se basa en que el código de tratamiento de fallos funciona correctamente, un pequeño error en la recuperación de un fallo, puede invalidar todo el código de la aplicación, que en nuestro caso será todo el ORB. Finalmente, como tercer y último factor que resalta la necesidad de una depuración especial del código de tratamiento de errores, tenemos que éste se ejecutará muy pocas veces. Sólo se ejecutará cuando ocurra algún error y es de esperar que los errores no ocurran con demasiada frecuencia. Esto conduce a un código que no se ejecutará en condiciones normales, y que por tanto será más probable que contenga errores latentes. Resumiendo estos factores, tenemos un código complejo, crítico y básico para todo el sistema que necesita ser ejercitado para validarlo y donde su ejercicio consistirá en provocar o simular fallos.

Nuestra aproximación es instrumentar el código del ORB para forzar a que se ejecute el código de tratamiento de fallos en un entorno controlado. Forzaremos el flujo de instrucciones para que se alcancen las partes del código que tratan los fallos. Como el código que queremos comprobar está de nuevo relacionado con referencias a objeto y por tanto incluido en las representaciones de los objetos, podremos integrar la inyección de fallos con la generación de trazas de eventos para su posterior análisis. Con este objetivo instalaremos en cada *nodo* qué fallos queremos inyectar, y en qué momento. Esta especificación de los fallos será almacenada en cada nodo en el propio ORB y permitirá decidir en tiempo de ejecución si se inyectan o no los fallos en función de distintos parámetros. Cada fallo inyectado será considerado como

[3]Las representaciones de los objetos están formadas fundamentalmente por los *endpoints* y los *adaptadores* de objeto.

un evento, con lo que podremos evaluar predicados globales para verificar si el ORB funcionó correctamente o no al ejecutar la recuperación.

5.4 Diseño del mecanismo de depuración

Hemos incluido un objeto dentro del ORB de cada dominio. Este objeto, al que llamaremos `FielM`[4], es un objeto CORBA con su correspondiente interfaz IDL. El objeto FielM será por tanto accesible de forma remota, habilitando de esta forma su control desde aplicaciones externas. Sin embargo, para permitir que el código del ORB acceda a sus servicios de forma rápida y para evitar en lo posible el *efecto de pruebas*, el objeto será accedido por el código del ORB a través del mecanismo propio del lenguaje en que esté implementado el mismo ORB. Se trata por tanto de un objeto integrado en el propio ORB, íntimamente relacionado con el resto del código del ORB pero con la particularidad de ser accesible de forma remota.

A cierto nivel de abstracción podemos describir al objeto FielM como una tabla con cierto número de componentes, donde cada componente está dedicada a una representación de objeto (*nodo* dentro de una *red*) dentro del dominio donde reside el objeto FielM. Por tanto un único FielM, permite monitorizar todas las *redes* a las que pertenece un dominio, es decir todas las *redes* para las que el dominio actúa como un *nodo*. El análisis de cada una de las representaciones de objeto para un FielM puede ser activado o desactivado en tiempo de ejecución. Es más, debido a que el número de objetos manejados por un ORB puede ser muy grande, y con ello según nuestro modelo el número de *redes*, el modo de funcionamiento del objeto FielM, permite instalar en él tan sólo un subconjunto de los objetos totales y de entre ellos, activar o desactivar con libertad los que se desea monitorizar.

A cada entrada en el objeto FielM, la llamamos *especificación Fiel*. Cada una de estas especificaciones almacena el último evento acontecido para el *nodo* asociado a dicha especificación, y una especificación de los fallos que serán inyectados en el *nodo*. Cada especificación Fiel está identificada por un *identificador Fiel* o *FielId* o más precisamente, cada FielId identifica a una *red*. Por tanto, el mismo identificador FielId en diferentes dominios, referencia a las distintas representaciones de un mismo objeto, o lo que es lo mismo el identificador FielId está asociado a un objeto CORBA que se desea monitorizar.

5.4.1 Interfaz

Como podemos observar en la figura 5.1, la interfaz del objeto FielM está dedicada fundamentalmente a instalar especificaciones de fallos para un FielId en particular, y para activar y desactivar de forma selectiva la monitorización de las *redes* identificadas por cada FielId.

El método `setFailure()` se utiliza para instalar una especificación de fallo, la cual permitirá inyectar o simular el correspondiente fallo en un *nodo*. La información que contiene cada especificación de fallo instalada en una especificación Fiel contiene los siguientes campos:

- *Punto de Fallo (failurePoint):* Este campo identifica un punto del código donde se inyectará el fallo.

[4]Del inglés **F**ault **I**njection and **E**vent **L**ogging **M**anager.

- *Condición de Fallo (whenSpec):* Se trata de un registro formado por la agregación de una identificación de tipo de condición, más información dependiente del tipo de condición de fallo, que permitirá su evaluación en tiempo de ejecución. Una vez que se instale una condición de fallo, ésta será evaluada en el punto de fallo cada vez que el código alcance dicho punto, y cada vez que la condición de fallo se evalúe a cierto, el fallo será inyectado.
- *Acción de Fallo (actionSpec):* Acción a ejecutar para simular o inyectar el fallo. Este registro contiene todos aquellos parámetros que se necesiten para inyectar el fallo.

```
interface fielm {
        void setFailure (in long fielId,
                         in long failurePoint,
                         in condition whenSpec,
                         in failureAction actionSpec);
            raises (invalidFielId, invalidFailurePoint, failuresAreActivated);
        void enableOne (in long fielId) raises (invalidFielId);
        void disableOne (in long fielId) raises (invalidFielId);
        void activate ();
        void deactivate ();
};
```

Figura 5.1: Interfaz del objeto FielM

Por su parte los métodos `enableOne()` y `disableOne()` permiten activar y desactivar respectivamente la monitorización y la inyección de fallos de un *nodo* (identificado por el propio dominio que contiene al objeto FielM, de una *red* (identificada por fielId). Para terminar, los métodos `activate()` y `deactivate()` activan o desactivan el funcionamiento del objeto FielM, es decir activan o desactivan el modo de depuración e inyección de fallos para un dominio.

5.4.2 Asignación de los identificadores FielId

Las representaciones de los objetos acceden a sus correspondientes especificaciones Fiel utilizando los identificadores FielId. Con este propósito, hemos modificado el ORB, tal y como lo habíamos descrito hasta ahora en los capítulos 3 y 4, para que las representaciones de los objetos contengan el identificador FielId. La asignación del identificador FielId se realiza cuando se crea la representación del objeto. Cuando se crea un objeto, o más precisamente cuando se crea la primera implementación de un objeto, se debe especificar explícitamente el identificador FielId a utilizar. Si no se indica ningún FielId, será asignado un identificador especial que indica que no se desea asociar este objeto con ningún FielId, que será equivalente a no monitorizar la ejecución de las representaciones de dicho objeto. Por otra parte, cuando un dominio cliente recibe una referencia a un objeto, también recibe el identificador Fiel, haciendo posible que la representación cliente pueda instalar el identificador cuando se cree la representación cliente.

Nótese que no estamos interesados en una asignación de los identificadores FielId que pueda acarrear conocimiento distribuido ni sobrecostes significativos a la ejecución ordinaria de las

aplicaciones. Por ello hacemos que la asignación de los FielId parta del propio programador, o más precisamente del personal encargado de la depuración. Este esquema puede provocar que a dos objetos distintos se les asigne el mismo FielId, coincidencia que puede ser relativamente frecuente durante un proceso de depuración amplio. En estos casos la única facilidad que proporciona el sistema es la detección de la colisión para permitir la resolución manual del conflicto.

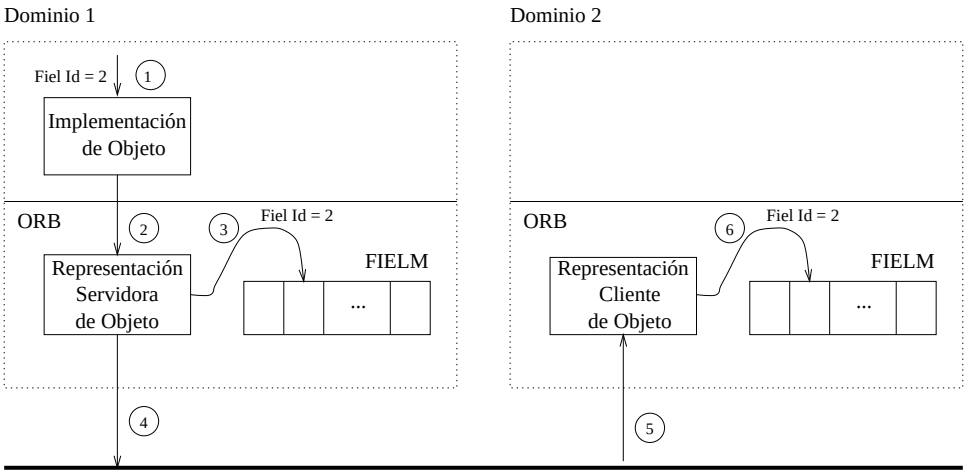

Figura 5.2: Asignación de los identificadores Fiel.

La figura 5.2 muestra los pasos que se siguen para completar la asignación de los identificadores Fiel. Cuando un objeto se crea, se especifica su identificador Fiel (paso 1), y el identificador se almacena en la representación servidora del objeto (paso 2) cuando esta sea creada. Al disponer del identificador Fiel, las representaciones de objeto pueden acceder a sus correspondientes especificaciones Fiel para realizar tareas relacionadas con inyección de fallos y depuración (paso 3). Cuando se empaqueta (marshal) una referencia a objeto para enviarla a otro dominio, el identificador Fiel formará parte de la versión empaquetada de la referencia (paso 4), de forma que el dominio receptor, podrá extraer el identificador Fiel al desempaquetar la referencia (paso 5). En este instante, si el dominio receptor era un dominio cliente potencial, creará la representación cliente del objeto y asociará el identificador Fiel a ella (pasos 6 y 7).

Nótese que en cada dominio existen las especificaciones Fiel incluso antes de que el dominio reciba una referencia al objeto, es decir mientras sea un dominio cliente potencial. Esto ayuda a la tarea de depuración en el sentido de que será posible monitorizar eventos tales como la propia creación de las representaciones de objeto o inyectar fallos como por ejemplo el agotamiento de la memoria en instantes tales como la creación de las representaciones a objeto.

5.4.3 Instrumentación del código

La generación de las trazas de eventos y la inyección de fallos se logra añadiendo sentencias específicas en el propio código del ORB. Cuando el código alcanza una de estas sentencias, en caso de estar ejecutándose una representación de objeto cuya monitorización estuviera habili-

tada, se ejecutará la acción correspondiente, es decir, se registrará un evento o se inyectará un fallo.

Las sentencias de registro de eventos son simples invocaciones directas al objeto FielM del ORB. Este objeto, cuya interfaz ya mostramos en la figura 5.1, proporciona métodos que serán invocados directamente desde el ORB sin hacer uso del mecanismo de invocación a objetos del ORB. Las operaciones más importantes que serán ejecutadas de esta forma las podemos observar en la figura 5.3.

```
class FielM {
        ...
        EventCounter logEvent (ObjectRepresentation &or, EventType et, ...);
        Boolean injectFault (ObjectRepresentation &or, FailurePoint fp,
                             FailureAction &fa, ...);
        ...
};
```

Figura 5.3: Operaciones del objeto FielM para instrumentar el ORB

La operación `logEvent()` se utiliza para registrar eventos, mientras que la operación `injectFault()` será empleada para inyectar y simular fallos.

Registro de eventos

Para registrar cierto evento, colocaremos en el código del ORB una invocación al método `logEvent()` al que le pasaremos como argumento la representación[5] del objeto que se encuentra en ejecución y la información relativa al evento que queremos registrar. Al ser invocado, el objeto FielM comprobará si la monitorización del objeto asociado a la representación del objeto está habilitada, en cuyo caso registrará el evento. Para ello realiza atómicamente las siguientes acciones:

- Incrementar el contador de eventos de la especificación Fiel.
- Guardar en almacenamiento estable toda la información asociada al evento.

Para cada evento, se salva la siguiente información:

- *Identificador Fiel:* El número de la *red* en ejecución.
- *Identificador de dominio:* Servirá para identificar a un *nodo* dentro de la *red*.
- *Reloj local:* Contador local de eventos interno a la especificación Fiel.
- *Marca temporal:* Información utilizada para realizar mediciones de rendimiento. Estrictamente hablando no se trata de información relevante para el evento.

[5] En nuestra implementación actual, realmente pasamos como primer argumento, el adaptador del objeto o su endpoint que constituyen en Hidra la representación del objeto.

- *Número de Tarea:* El identificador de tarea del sistema operativo que ejecuta el código.
- *Tipo de Evento:* El tipo de evento tal y como fue introducido por el programador como argumento a la sentencia de instrumentación.
- *Información del Evento:* Toda aquella información que el programador haya considerado relevante para el evento que se desea registrar.

Inyección de fallos

De forma similar a como lo realizamos para el registro de eventos, en aquel trozo de código donde deseemos inyectar un fallo, introduciremos una invocación al método `injectFault()`. Cuando este método sea invocado, se comprobará si la representación del objeto que se encuentra en ejecución está habilitada. Si se encuentra habilitada, se accederá al estado de la especificación Fiel, para evaluar la condición de inyección del fallo. La condición será accedida gracias al argumento `failurePoint`, que permitirá extraer un objeto de la especificación Fiel que será invocado para determinar si debe inyectarse el fallo o no. En caso de que el resultado indique que se debe inyectar el fallo, se invocará al objeto `FailureAction` pasado como último argumento de la sentencia de instrumentación. Este objeto se encargará de ejecutar el código necesario para inyectar o simular el fallo.

5.5 Caso de estudio

Uno de los algoritmos distribuidos más complejos del ORB de Hidra es el protocolo de cuenta de referencias. Este protocolo, tal y como describimos en el capítulo 4 asegura que eventualmente se notificará mediante la notificación de no referenciado a todo objeto para el que no existan referencias en el sistema. Un problema práctico aparece cuando se desarrolla una aplicación distribuida en la que se espera que cierto objeto reciba la notificación de no referenciado, y este objeto simplemente no recibe la notificación. Es complejo saber si el objeto no ha recibido la notificación por errores de diseño o implementación en la propia aplicación que no descartan correctamente todas las referencias, o si por el contrario existe un error en el ORB que impide el envío de la notificación de no referenciado en algún caso.

Dada la importancia de este protocolo en Hidra, lo presentamos como caso de estudio para mostrar cómo nuestro sistema de depuración monitoriza los eventos que ocurren durante su ejecución utilizando nuestro modelo basado en redes de representaciones de objetos.

El protocolo, tal y como ya indicamos, debe además reaccionar convenientemente gran variedad de fallos, tanto transitorios, como permanentes. Por ejemplo, si un dominio recibe una referencia a objeto nueva, y no puede crear la correspondiente representación cliente del objeto debido a que no queden recursos suficientes, el protocolo debe reaccionar y ejecutar código alternativo. En este caso, en lugar de enviar el mensaje de tipo `INC` para registrarse como nuevo cliente, debe enviarse un `DEC` para indicarle al emisor de la referencia que actúe como si no se la hubiera generado.

Por tanto, para depurar correctamente todo el código relacionado con el protocolo de cuenta de referencias, monitorizaremos los eventos que se generen, e inyectaremos fallos tanto tran-

sitorios como permanentes para permitir que se ejecute el código de recuperación que no se ejecutaría en condiciones normales.

5.5.1 Los eventos del protocolo

Cuando monitorizamos los eventos producidos al ejecutar el protocolo de cuenta de referencias, la cantidad de eventos a almacenar puede ser enorme. Para mostrar cómo nuestra herramienta ayuda a analizarlos, comparamos los diagramas temporales que obtendríamos al analizar una ejecución ejemplo del ORB mediante dos modelos. Primero, aplicando el modelo tradicional y genérico de computaciones distribuidas y después utilizando nuestro modelo específico de ejecuciones de los ORB basado en objetos.

La ejecución ejemplo que utilizaremos para comparar ambos diagramas se corresponde al siguiente escenario:

- Hay tres dominios en el sistema: A, B y C.
- Estos dominios mantienen cierto número de implementaciones de objetos. A cada uno de estos objetos les damos un nombre de la forma X_i, donde X es el nombre del dominio y donde i identifica correlativamente cada implementación dentro de su correspondiente dominio.
- Inicialmente cada dominio mantiene una referencia a los objetos que residen en él. Adicionalmente, el dominio A mantiene una referencia a los objetos B_1, B_2 y C_1, y el dominio C mantiene una referencia a los objetos A_1 y B_1.
- El dominio A invoca una operación sobre el objeto B_1, pasándole una referencia al objeto A_1 como argumento de entrada y recibiendo B_2 como un argumento de salida. Después de realizar esa petición, el mismo dominio A invoca al objeto C_1 pasándole una referencia al objeto B_2 como argumento de entrada y recibiendo una referencia al objeto C_2. Después de recibir la respuesta de esta última invocación, el dominio A falla en el intento de obtener los recursos necesarios para crear la representación del objeto C_2. Finalmente, el dominio A descarta todas las referencias que mantenía del objeto B_2.
- El dominio B simplemente contesta a las peticiones de los demás dominios.
- El dominio C invoca al objeto B_1 pasándole una referencia al objeto A_1 como argumento de entrada y recibiendo una referencia al objeto B_2 como retorno de la invocación.

En la figura 5.4 tenemos el diagrama temporal que resultaría al considerar el modelo tradicional y genérico de ejecuciones distribuidas. Por contra, en la figura 5.5 tenemos el diagrama temporal que resulta de aplicar nuestro modelo a la misma ejecución.

Comparando ambos diagramas podemos observar cómo se logra disminuir notablemente con nuestro modelo el número de eventos a procesar, sin perder por ello información significativa. Resultará más sencillo aplicar predicados, inyectar fallos y evaluar la reacción del código ante cada uno de ellos, haciéndolo sobre cada una de las *redes* representadas por los distintos diagramas de la figura 5.5 en lugar de hacerlo sobre la totalidad de eventos mostrados en la figura 5.4.

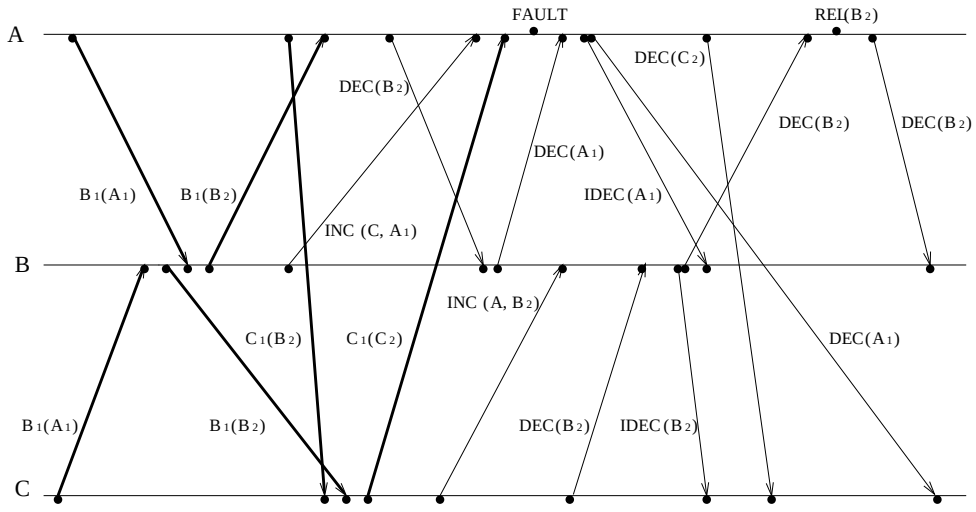

Figura 5.4: Diagrama de eventos para la ejecución completa del ORB.

5.6 Trabajo relacionado

Una de las tendencias actuales de investigación en el área de depuración de aplicaciones distribuidas, consiste en construir mecanismos que permitan aplicar el mismo proceso cíclico de construcción del software que se aplica habitualmente en el desarrollo de aplicaciones secuenciales. Dentro de este proceso de desarrollo tradicional de aplicaciones la fase de depuración se desarrolla asimismo de forma cíclica: ante la detección de cierto síntoma de error o anomalía, se inicia una serie sucesiva de ejecuciones de la aplicación. En cada iteración se pretende reproducir la aparición del defecto, restringiendo sucesivamente la zona de código responsable. Finalmente, en la última iteración, es deseable encontrar el punto o puntos exactos que provocaron la situación anómala.

Por ejemplo, [79] asume que cada proceso del sistema distribuido se comporta de forma determinista, y proporciona un mecanismo para reproducir completamente la computación distribuida. En cada repetición de la ejecución, se introducen los mismos estímulos de entrada al sistema y cada proceso es forzado a acceder a la misma versión de los datos compartidos a la que accedió en la primera ejecución. Por su parte el trabajo descrito en [58] basa la repetición de las ejecuciones en disponer de un proceso gemelo para cada uno de los procesos del sistema distribuido. Para estos procesos gemelos, no es necesario que se vuelvan a evaluar las elecciones no deterministas que hubieran hecho los procesos originales cuando se repita la ejecución del sistema. También existen otras aproximaciones que hacen uso de *mensajes de actualización* (checkpoints) [110] para almacenar el estado global del sistema, permitiendo la reanudación de las ejecuciones desde cualquiera de los estados globales salvados previamente.

Desafortunadamente, parece que las estrategias basadas en repetición de las ejecuciones son difícilmente realizables para depurar software de sistema como es el caso de un ORB, si se pretende evitar el rediseño y la reimplementación de todo el sistema. Nuestro objetivo ha sido desarrollar un sistema de depuración capaz de se utilizado para depurar Hidra y en particular para depurar el ORB de Hidra. La necesidad de asumir comportamiento determinista en un

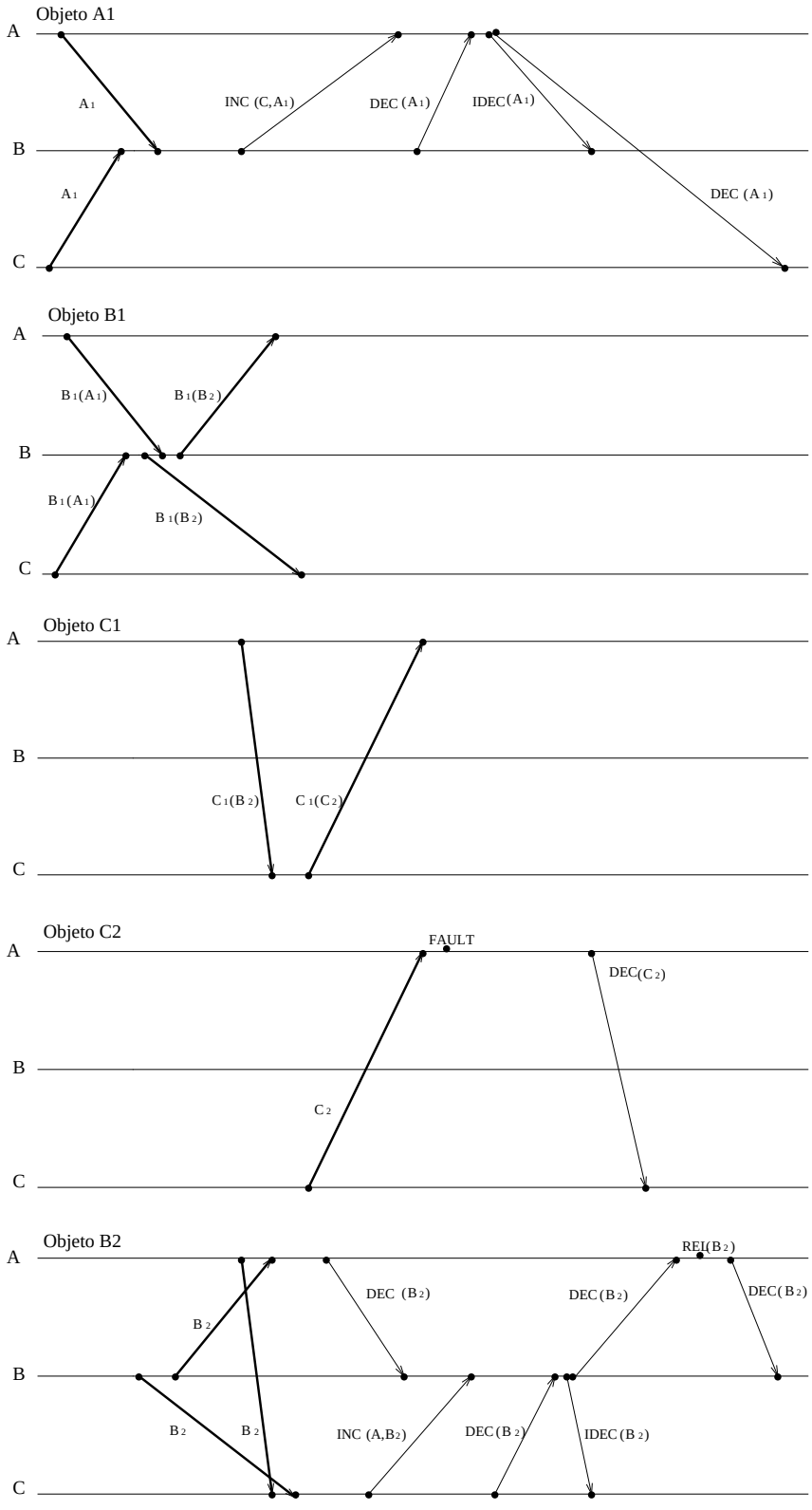

Figura 5.5: Diagramas de eventos desglosado por cada objeto.

entorno multitarea como el nuestro, donde gran parte del código tiene una estrecha relación con el sistema operativo e incluso gran parte forma parte de él, o de utilizar *checkpoints* donde muchas de las acciones a ejecutar no pueden retrasarse, o de emplear procesos gemelos para cada una de las tareas que se crean en el ORB de forma dinámica, nos han inclinado a apreciar la necesidad de seguir una aproximación distinta a la depuración cíclica basada en repetición de las ejecuciones.

La segunda aproximación generalmente propuesta para facilitar la depuración de sistemas distribuidos, se basa en la monitorización de los eventos generados por los sistemas. El objetivo de estas técnicas es comparar los modelos de comportamientos esperados con los comportamientos reales de los sistemas durante su ejecución capturados en trazas de ejecución [6]. Para ello, las ejecuciones distribuidas se modelan como secuencias de eventos producidas por los distintos procesos que forman el sistema. Los eventos son grabados durante las ejecuciones del sistema y las trazas se comparan con modelos previamente especificados de comportamientos correctos.

El comportamiento de un programa se dice que es correcto si satisface un conjunto de predicados globales [5] que son evaluados sobre la secuencia de eventos. Dado en orden impredecible que siguen las distintas componentes distribuidas de un sistema para ejecutar instrucciones y ante la ausencia de un observador omnisciente, se debe tener especial cuidado en asegurar que la secuencia de eventos grabados realmente representa un estado consistente del sistema. La monitorización de los eventos se basa en recolectar los eventos de los procesos involucrados en la computación y el calcular instantáneas consistentes de tales eventos tal y como se describe por ejemplo en [20]. Cada instantánea del estado global contiene una subsecuencia de la secuencia de eventos grabada, que respeta la relación *ocurrió antes* de Lamport [77]. Por tanto se garantiza que cierto observador externo podría haber observado la computación tal y como queda reflejada en la instantánea. Tal y como se indica en [5], en general la evaluación de predicados sobre instantáneas solo permite la detección de propiedades estables [20] como la detección de terminación o la detección de interbloqueos. Por contra la detección de propiedades inestables de las computaciones distribuidas puede resultar prohibitiva en coste. Se requiere, a menos que los predicados satisfagan ciertas restricciones [133], el cálculo de la matriz completa de estados consistentes y la evaluación de predicados globales sobre ellos [5].

La construcción de la matriz y la evaluación de predicados globales sobre ellos es muy costosa y conduce a implementaciones poco prácticas y que modifican enormemente el propio comportamiento del sistema que se pretende analizar, es decir, presentan un efecto de pruebas tan notorio que la ejecución del sistema con el sistema de depuración habilitado difiere de forma muy significativa respecto a la ejecución del sistema sin tal sistema de depuración.

Como consecuencia del estudio de las diversas técnicas de depuración y las características propias de los ORBs, hemos desarrollado un sistema de depuración específico para los ORB, basado en la monitorización de eventos que permite inyección de fallos y habilita el estudio de los predicados sobre estados globales durante un análisis post-mortem del sistema.

Capítulo 6

Conclusiones

A veces sentimos que lo que hacemos es tan solo una gota en el mar, pero el mar sería menos si le faltara una gota.

Madre Teresa de Calcuta

Contenidos del capítulo

Se está consolidando un interés creciente tanto de la industria como desde el punto de vista de la investigación hacia los sistemas altamente disponibles. Se trata de sistemas complejos, difíciles de construir y de verificar, pero que proporcionan a los programadores de aplicaciones indudables ventajas. El desarrollo de estas aplicaciones podrá centrarse en los aspectos más esenciales de su funcionalidad, apoyándose en los sistemas subyacentes para aumentar su disponibilidad.

Del amplio abanico de soluciones que podemos encontrar, en este trabajo nos hemos centrado en los sistemas que dotan de alta disponibilidad a las aplicaciones y sistemas orientados a objeto y hemos profundizado en el soporte básico que estos sistemas deben ofrecer. Hemos descrito el soporte de Hidra, mostrando la estructura de sus referencias y los protocolos básicos que operan de forma independiente al modelo de replicación.

A nivel arquitectónico, hemos descrito los niveles de la arquitectura Hidra, y en particular hemos enfatizado el enmascaramiento jerárquico de fallos que ocurre en cada nivel. Este aspecto ha sido especialmente relevante a la hora de centrarnos en el desarrollo de los protocolos a nivel del soporte a objetos, que han podido asumir con todo realismo la única existencia de fallos de parada.

6.1 Soporte a objetos

El soporte a objetos de Hidra sigue la tendencia de sistemas como Spring [67, 93] y Solaris-MC [8], en los cuales se prima la flexibilidad para poder implantar semánticas diferentes a los objetos. El concepto de adaptador de objetos se nos antoja de gran utilidad para este fin y gracias a él, podemos considerar objetos con semánticas tan diferentes como los objetos fijos, los básicos y los replicados.

A nivel de la estructura interna de las referencias, hemos optado por una estructura basada en localizadores que flexibiliza la adopción en principio de cualquier modelo de replicación. El localizador principal, juega un papel similar al puntero débil de otros sistemas [43, 128], en el sentido de que sirve para localizar la ubicación del endpoint principal en ausencia de migración. En caso de migraciones, se podrá alcanzar a la ubicación definitiva a través del camino formado por los localizadores principales.

Las referencias ocupan poco espacio comparado con el que se requiere para otros sistemas comparables [17, 128, 73]. Utilizamos identificadores únicos de objeto, cuestión que parece inevitable si consideramos la posibilidad de migraciones junto con caídas de nodos.

6.2 Recolección de residuos

Nuestro detector de residuos únicamente detecta residuos acíclicos. Sin embargo este aspecto es más una característica que un defecto, pues el modelo de objetos de Hidra lo hace desaconsejable. El mecanismo que proponemos es vivo y seguro, es barato en espacio y en mensajes. A nivel espacial solamente se necesita un contador por endpoint y si consideramos migración una lista de localizadores que en general tendrá un tamaño de entre cero y un elementos. Todos los mensajes del detector de residuos son asíncronos, con lo que se puede desacoplar completamente la ejecución del recolector, de la ejecución de los mutadores. No es necesario ningún

mensaje en primer plano, ni ninguna tarea que sobrecargue el sistema. Todos los mensajes pueden incluirse como información adicional a los mensajes propios del sistema y no se requiere de ningún mecanismo transaccional ni de comunicación ordenada.

Nuestro algoritmo, es el más asíncrono entre todos los algoritmos de cuenta de referencias que hemos encontrado, quedano la asincronía únicamente limitada por el tamaño de los contradores. Los algoritmos que limitan el número de posibles duplicaciones nos parecen poco adecuados a nuestro sistema, los que se basan en listas de referencias consumen demasiado espacio, y los basados en trazas demasiado coste computacional y de mensajes.

Respecto a la migración, se puede comprobar como la intención de nuestro algoritmo consiste en evitar la aparición de residuos intermedios. También utilizamos los propios mensajes del recolector de residuos para acortar rápidamente los caminos indirectos hasta el endpoint principal que puedan aparecer por cada migración.

La tolerancia a fallos la logramos con un sencillo protocolo de marcado y barrido, siguiendo la misma aproximación de construir sistemas híbridos que podemos encontrar en otros sistemas [123, 30, 8]. Una característica adicional de nuestro protocolo de marcado y barrido es que realiza de una sola vez la regeneración de los invariantes del recolector de residuos, junto con la búsqueda de los objetos migrados y la reconexión de los clientes a sus ubicaciones.

6.3 Depuración de ORBs

Nuestra aproximación para depurar Hidra ha consistido en el desarrollo de una herramienta sencilla, que sin embargo se muestra lo bastante potente como para ejercitar y monitorizar la gran variedad de algoritmos que aparecen en el ORB. La implementación es lo bastante sencilla como para poderla mantener mientras el ORB evoluciona, y no impone un efecto de pruebas significativo. Hemos combinado el registro de eventos con la inyección de fallos, ya que ambos mecanismos deben trabajar conjuntamente en un sistema como el nuestro, donde el comportamiento del sistema incluye la reacción ante fallos.

Hemos presentado un modelo de ejecución para los ORBs. El modelo permite que se seleccione qué eventos se desea monitorizar basándonos en los objetos que los van a producir. Resulta claramente necesario que dispongamos de una herramienta para analizar los eventos que se producen durante la monitorización del sistema [29, 28]. Sin embargo, pese a no disponer de tal herramienta, la mera inspección de los registros de eventos ha demostrado ser útil en variedad de situaciones, encontrando algunos errores de implementación que hubieran sido difíciles de encontrar sin esta ayuda.

Finalmente, es interesante observar que la herramienta que hemos presentado, también puede utilizarse para medir el rendimiento del sistema. Ya que los eventos que almacenamos en el registro contienen una marca de tiempo, con la que podremos realizar mediciones de los aspectos que consideremos más interesantes.

6.4 Estado de la implementación

Hemos implementado [52] todos los niveles de la arquitectura Hidra y varios programas distribuidos de prueba que ejercitan la mayoría de componentes. En el momento de redactar este

trabajo tan sólo resta integrar el soporte al modelo de replicación coordinador-cohorte en el prototipo, que ya implementamos con anterioridad sobre el ORB de Visibroker [134, 106].

La implementación del prototipo nos ha servido fundamentalmente para validar y refinar el diseño en general de Hidra y corregir y ampliar el diseño de algunas de sus componentes. En particular el diseño del ORB ha sido mejorado y el prototipo nos ha permitido probar protocolos que aún no teníamos implementados. En particular hemos podido probar el recolector de residuos para objetos replicados, comprobando de forma experimental su corrección y la poca sobrecarga que introduce en el sistema.

Hasta ahora, el prototipo está siendo de gran utilidad, pero su objetivo principal es servir de primera implementación de Hidra antes de migrar el código al núcleo del sistema operativo. Planeamos implementar Hidra en C++ e integrarlo en Linux, para posteriormente desarrollar software de sistema en el propio kernel que utilice las facilidades de alta disponibilidad de Hidra. Puede parecer sorprendente que entonces hayamos elegido Java y no C++ para el prototipo, pero hemos llegado a esta elección por varios motivos:

- Sintácticamente Java es similar a C++. De hecho existen traductores de libre uso para migrar código Java a C++.
- Para la gran mayoría de errores en tiempo de ejecución, la máquina virtual Java muestra la pila de llamadas efectuadas hasta el momento del fallo. Con C++ lo habitual en caso de este tipo de fallos es que la aplicación termine sin más datos que una mera información de acceso ilegal a la memoria. Si la aplicación a desarrollar es distribuida, la importancia de esta facilidad aumenta por la ausencia en general de depuradores adecuados [54].
- Que el recolector de residuos de Java recolecte los objetos automáticamente, simplifica la implementación, permitiendo centrarse en los aspectos funcionales, olvidándose de gran parte de aspectos de asignación y liberación de memoria.
- La cantidad de clases de utilidad probadas y verificadas en Java e incluidas de forma estándar, tales como tablas hash, listas, pilas, etc. también facilitan enormemente cualquier desarrollo de software de sistema, permitiendo centrarse en lo más esencial.
- Los monitores de Java, obligan a razonar en entornos multitarea en términos de operaciones que pueden proceder en paralelo o que deben ser serializadas. En nuestra opinión, es más sencillo y lleva a mejores diseños utilizar esta facilidad que razonar en términos de cerrojos o de cualquier otra primitiva que disponga el sistema para controlar la concurrencia. Si bien es cierto que con la mayoría de primitivas se puede implementar lo mismo que con monitores (incluso los mismos monitores), el hecho de que Java obligue a usarlos, es ventajoso en nuestra opinión en etapas de diseño.

Para desarrollar el prototipo hemos tenido especial cuidado de llevar la cuenta de referencias para todos los objetos que forman parte del ORB: casos de los endpoints, adaptadores, proxies y descriptores de objeto. Es decir, para ellos no utilizamos las facilidades de Java de recolección de residuos, sino que llevamos la cuenta de las referencias para liberarlas manualmente cuando sea necesario. Esto lo hemos realizado así porque la recolección de residuos es una parte esencial de Hidra, y sólo pretendíamos que Java nos “ayudara” en el tratamiento de los objetos y entidades temporales, pero no para aquellos que forman parte del soporte básico de Hidra.

6.5 Posibles líneas de trabajo futuro

Por lo que hemos podido encontrar, aparte de Hidra, no existe ningún otro sistema que incorpore un mecanismo de recolección de residuos centrado en objetos replicados. Parece interesante estudiar cómo se podrían adaptar otros protocolos a entornos con soporte a replicación, especialmente si consideramos la replicación en redes de área extensa en lugar de sobre redes de área local tal y como tenemos en Hidra.

En lo que respecta a la implementación de Hidra, necesitamos disponer de aplicaciones que ejerciten nuestro sistema, lo que en sí mismo puede dar lugar a gran cantidad de trabajos interesantes. Tenemos la sensación de que el desarrollo de cualquier componente de un sistema operativo basado en Hidra puede dar lugar a conclusiones atractivas.

En lo que respecta a la validación de la arquitectura Hidra, sería muy recomendable que afrontáramos la implementación del modelo de replicación activo en nuestro esquema. Tenemos el modelo coordinador-cohorte, por lo que el modelo pasivo supondría una simplificación del soporte actual. Sin embargo el modelo activo puede aportar nuevas dificultades.

Por lo que hemos podido comprobar, no existe ningún trabajo que haya estudiado la recolección de residuos asumiendo un modelo de comunicación orientado a grupos en lugar de basado en paso de mensajes, lo que por tanto parece un reto. Por otra parte, antes de afrontar la implementación del modelo activo en Hidra, deberíamos estudiar la adecuación del protocolo de cuenta de referencias a este tipo de comunicaciones.

Para el prototipo de Hidra, hemos optado por una estructura de los localizadores sencilla. Una posible línea de trabajo puede consistir en comparar el rendimiento que pueden ofrecer los distintos modelos de replicación y los protocolos necesarios para la gestión de las réplicas dependiendo de la estructura de los localizadores.

Otra posible vía de trabajo la tenemos en los protocolos de gestión de réplicas. En Hidra hemos optado por cederle al propio modelo de replicación la sincronización que se requiere entre las peticiones de adición de réplicas a un objeto replicado y las operaciones que se estén efectuando concurrentemente sobre el objeto. Sería muy beneficioso el disponer de un protocolo de adición y de eliminación de réplicas lo bastante general como para no depender del modelo de replicación y que no suponga mayor coste del derivado del uso del propio modelo de replicación.

Bibliografía

[1] Saleh E. Abdullahi y Graem A. Ringwood. Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373, Septiembre 1998.

[2] Ole Agesen, David Detlefs y J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. *j-SIGPLAN*, 33(5):269–279, Mayo 1998.

[3] A. Avizienis. The N-version approach to fault-tolerance software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[4] Ö. Babaoğlu y S. Toueg. Non-blocking atomic commitment. En S. J. Mullender, editor, *Distributed Systems*, capítulo 6, págs. 147–168. Addison-Wesley, Wokingham, UK, 2nd edición, 1993.

[5] Özalp Babaoğlu y Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. Informe técnico UBLCS-93-1, University of Bologna, Department of Computer Science, January 1993.

[6] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):11–22, January 1995.

[7] Ganesha Beedubail, Udo Pooch y Peter Kessler. Object replication in spring using subcontracts. Technical Report TR95-041, Texas A&M University, 1995.

[8] J. M. Bernabéu, Y. A. Khalidi, V. Matena, K. Shirriff y M. Thadani. The design of solaris mc: A prototype multi-computer operating system (2nd edition). Informe técnico, SMLI-94-492, Sun Microsystems Laboratories Inc., Mountain View, CA, June 1995.

[9] J. M. Bernabéu-Aubán, V. Matena y Y. A. Khalidi. Extending a traditional OS using object-oriented techniques. En *2nd Conf. on Object-Oriented Technologies & Systems, Toronto, Canada*, págs. 53–63, Berkeley, CA, USA, Junio 1996. USENIX.

[10] D. I. Bevan. Distributed garbage collection using reference counting. En A. J. Nijman J. W. de Bakker y P. C. Treleaven, editores, *Proceedings of the Conference on Parallel Architectures and Languages Europe (PARLE). Volume II: Parallel Languages*, volumen 259 de *LNCS*, págs. 176–187, Eindhoven, The Netherlands, Junio 1987. Springer.

[11] K. 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.

[12] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12), Diciembre 1993.

[13] K. P. Birman, T. Joseph y T. Raeuchle. Concurrency control in resilient objects. Informe técnico, TR 84-622, Dept. of Computer Science, Cornell Univ., Ithaca, NY, Julio 1984.

[14] K. P. Birman, T. Joseph, T. Raeuchle y A. El Abbadi. Implementing fault-tolerant distributed objects. *IEEE Trans. on SW Eng.*, 11(6):502–508, Junio 1985.

[15] K. P. Birman y T. A. Joseph. Exploiting virtual synchrony in distributed systems. *Proc 11th ACM Symposium on OS Principles, Austin, TX, USA*, págs. 123–138, Noviembre 1987.

[16] Ken Birman. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. *sigops*, 28(1), Enero 1994.

[17] A. Birrell, D. Evers, G.Ñelson, S. Owicki y E. Wobber. Distributed garbage collection for network objects. Informe técnico 116, DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA, USA, Diciembre 1993.

[18] A. Borg, W. Blau, W. Graetsch, F. Herrmann y W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Febrero 1989.

[19] N. Budhiraja, K. Marzullo, F. B. Schneider y S. Toueg. The primary-backup approach. En S. J. Mullender, editor, *Distributed Systems*, capítulo 8, págs. 199–216. Addison-Wesley, Wokingham, UK, 2nd edición, 1993.

[20] K. Mani Chandi y Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(2):63–75, February 1985.

[21] TD Chandra y S Toueg. Unreliable failure detectors for asynchronous systems. En *PODC91 Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, págs. 325–340, 1992.

[22] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg y Bernadette Charron-Bost. On the impossibility of group membership. En *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC'96)*, págs. 322–330, New York, USA, Mayo 1996. ACM.

[23] David R. Cheriton y Dale Skeen. Understanding the limitations of causally and totally. *ACM Operating Systems Review, SIGOPS*, 27(5), Diciembre 1993.

[24] David D. Clark. The Structuring of Systems Using Upcalls. En *Proceedings of the Tenth Symposium on Operating Systems Principles*, Shark Is., WA, 1985.

[25] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Diciembre 1960.

[26] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 6(4):175–187, 1991.

[27] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–68, Febrero 1991.

[28] Janice E. Cuny, George Forman, Alfred Hough, Joydip Kundu, Calvin Lin y Lawrence Snyder. The ariadne debugger: Scalable application of event-based abstraction. En Barton P. Miller y Charles McDowell, editores, *Proceedings of the Workshop on Parallel and Distributed Debugging*, volumen 28_12 de *ACM SIGPLAN Notices*, págs. 85–95, New York, NY, USA, Mayo 1993. ACM Press.

[29] Peter Dauphin y Richard Hoffman. Hasse: a tool for analyzing causal relationships in parallel and distributed systems. *Lecture Notes in Computer Science*, 977, 1995.

[30] Peter Dickman. *Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures*. Ph.D. thesis, Cambridge University Computer Laboratory, Cambridge, England, UK, 1992.

[31] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, Noviembre 1974.

[32] Edsger W. Dijkstra, Leslie Lamport, Alain J. Martin, C. S. Scholten y E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, Noviembre 1978. Also E. W. Dijkstra Note EWD496, June 1975.

[33] D. Dolev y D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, Abril 1996.

[34] Cynthia Dwork, Nancy Lynch y Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Abril 1988.

[35] Guy Eddon y Henry Eddon. *Inside Distributed COM*. Microsoft Programming Series. Microsoft Press, Redmond, WA, 1998.

[36] P. Felber, R. Guerraoui y A. Schiper. Replicating objects using the CORBA event service. En *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, págs. 14–19, Tunis, Tunisia, Octubre 1997.

[37] Pascal Felber. *The CORBA Object Group Service. A service approach to object groups in CORBA.* Tesis doctoral, École Polytechnique Fédérale de Lausane, Octubre 1998.

[38] Pascal Felber, Rachid Guerraoui y Andre Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[39] Fabrice Le Fessant. Detecting distributed cycles of garbage in large-scale systems. En *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC'01)*. ACM, Aug 2001.

[40] Fabrice Le Fessant, Ian Piumarta y Marc Shapiro. An implementation of complete, asynchronous, distributed garbage collection. En *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, págs. 152–161, 1998.

[41] Michael J. Fischer, Nancy A. Lynch y Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374–382, Abril 1985.

[42] J. Fowler y W. Zwaenepoel. Causal distributed breakpoints. En *Proc. 10th Intl. Conf. Distributed Computing Systems (ICDCS-10)*, págs. 134–141, Paris, France, 1990. IEEE.

[43] Robert Joseph Fowler. The complexity of using forwarding addresses for decentralized object finding. En Joseph Halpern, editor, *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, págs. 108–120, Calgary, AB, Canada, Agosto 1986. ACM Press.

[44] R. Friedman y R. van Renesse. Strong and weak virtual synchrony in horus. *Technical Report TR95-1537, Dept. of Computer Science, Cornell University, Ithaca, NY*, pág. 27 pgs, August 1995.

[45] P.D. Ezhilchelvan G. Morgan, S.K. Shrivastava y M.C. Little. Design and implementation of a corba fault-tolerant object group service. En *Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems*, Helsinki, Junio 1999.

[46] P. Galdámez, F. D. Muñoz-Escoí y J. M. Bernabéu-Aubán. Extending an ORB to support high availability. En *Proc. of the V Jornadas de Concurrencia, Gandía, Spain*, págs. 349–360, Junio 1997.

[47] P. Galdámez, F. D. Muñoz-Escoí y J. M. Bernabéu-Aubán. HIDRA: Architecture and high availability support. Informe técnico, DSIC-II/14/97, Univ. Politècnica de València, Spain, Mayo 1997.

[48] P. Galdámez, F. D. Muñoz-Escoí y J. M. Bernabéu-Aubán. High availability support for distributed object systems. En *Workshop on CORBA, ECOOP'97, Jyväskylä, Finland*, Junio 1997.

[49] P. Galdámez, F. D. Muñoz-Escoí y J. 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, Czech Republic*, volumen 1338 de *LNCS*, págs. 407–414. Springer Verlag, Noviembre 1997.

[50] P. Galdámez, F. D. Muñoz-Escoí y J. 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, Czech Republic*, volumen 1725 de *LNCS*, págs. 373–380. Springer Verlag, Noviembre 1999.

[51] P. Galdámez, F. D. Muñoz-Escoí y J. M. Bernabéu-Aubán. Garbage collection in a kernel-based ORB. En *Proc. of the VII Jornadas de Concurrencia, Gandía, Spain*, págs. 139–150, Junio 1999.

[52] P. Galdámez, F. D. Muñoz-Escoí y J. M. Bernabéu-Aubán. Un prototipo de Hidra. En *Proc. of the IX Jornadas de Concurrencia, Sitges, Spain*, Junio 2001.

[53] P. Galdámez, D. Murphy, F. D. Muñoz-Escoí y J. M. Bernabéu-Aubán. Debugging an object request broker. En *Proc. of the V Jornadas de Concurrencia, Gandía, Spain*, págs. 333–348, Junio 1997.

[54] P. Galdámez, Declan Murphy, José M. Bernabéu-Aubán y Francesc D. Muñoz-Escoí. Event-based techniques to debug an object request broker. *The Journal of Supercomputing*, 13(2):133–149, Marzo 1999.

[55] Doreen L. Galli. *Distributed Operating Systems: Concepts and Practice*. Prentice Hall, Inc, Upper Saddle River, New Jersey, USA, Marzo 1993.

[56] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[57] Hector Garcia-Molina, Frank Germano, Jr. y Walter H. Kohler. Debugging a distributed computing system. *IEEE Transactions on Software Engineering*, SE-10(2):210–219, March 1984.

[58] O. Gerstel, S. Zaks, M. Hurfin, N. Plouzeau y M. Raynal. On-the-fly replay: A practical paradigm and its implementation for distributed debugging. En *Proceedings of the 6th Symposium on Parallel and Distributed Processing*, págs. 266–272, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[59] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. *SIGPLAN Notices*, 24(7):313–321, Julio 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[60] J.Ñ. Gray. Notes on database operating systems. En G. Goos y J. Hartmanis, editores, *Operating Systems: An Advance Course*, volumen 60 de *Lecture Notes in Computer Science*, págs. 393–481. Springer-Verlag, 1978.

[61] Jim Gray y Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[62] Rachid Guerraoui, Patrick Eugster, Pascal Felber, Benoît Garbinato y Karim Mazouni. Experiences with object group systems. *Software Practice and Experience*, 30(12):1375–1404, Octubre 2000.

[63] Rachid Guerraoui, Pascal Felber, Benoit Garbinato y Karim Mazouni. System support for object groups. En *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volumen 33, 10 de *ACM SIGPLAN Notices*, págs. 244–258, New York, Octubre 18–22 1998. ACM Press.

[64] Rachid Guerraoui, Benoît Garbinato y Karim R. Mazouni. Garf: A tool for programming reliable distributed applications. *IEEE Concurrency*, 5(4):32–39, Octubre/Diciembre 1997.

[65] V. Hadzilacos y S. Toueg. Fault tolerant broadcasts and related problems. En S. J. Mullender, editor, *Distributed Systems*, capítulo 5, págs. 97–145. Addison-Wesley, Wokingham, UK, 2nd edición, 1993.

[66] G. Hamilton, M. L. Powell y J. J. Mitchell. Subcontract: A flexible base for distributed programming. En B. Liskov, editor, *Proc. of the 14th Symp. on Operating Systems Principles*, págs. 69–79, New York, NY, USA, Diciembre 1993. ACM Press.

[67] Graham Hamilton y Panos Kougiouris. The Spring nucleus: A microkernel for objects. Informe técnico TR-93-14, Sun Microsystems Laboratories, Inc., Mountain View , CA , USA, Abril 1993.

[68] Elliotte Rusty Harold. *Java Network Programming Guide*. O'Reilly & Associates, Inc., 103a Morris Street, Sebastopol, CA 95472, USA, Tel: +1 707 829 0515, and 90 Sherman Street, Cambridge, MA 02140, USA, Tel: +1 617 354 5800, 1997.

[69] R. John M. Hughes. A distributed garbage collection algorithm. En Jean-Pierre Jouannaud, editor, *Record of the 1985 Conference on Functional Programming and Computer Architecture*, volumen 201 de *Lecture Notes in Computer Science*, págs. 256–272, Nancy, France, Septiembre 1985. Springer-Verlag.

[70] IONA. *Orbix 2.2 Programming Guide*. IONA Technologies Ltd, 1997.

[71] R. A. Johnson. The ups and downs of object oriented systems development. *Communications of the ACM*, 43(10):69–73, Octubre 2000.

[72] M. F. Kaashoek y A. 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, 1991.

[73] Y. A. Khalidi, J. M. Bernabéu-Aubán, V. Matena, K. Shirriff y M. Thadani. Solaris MC: A multi computer OS. En *Proceedings of the USENIX 1996 annual technical conference, San Diego, California, USA*, págs. 191–203. USENIX, Enero 1996.

[74] K. H. Kim y H. O. Welch. Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, 38(5):626–36, 1989.

[75] R. Ladin y B. Liskov. Garbage collection of a distributed heap. En *12th International Conference on Distributed Computing Systems*, págs. 708–715, Yokohama, Japan, Junio 1992. IEEE.

[76] R. Ladin, B. Liskov, L. Shrira y S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. on Comp. Sys.*, 10(4):360–391, Noviembre 1992.

[77] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[78] Bernard Lang, Christian Queinnec y José Piquer. Garbage collecting the world. En *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, págs. 39–50, Albequerque, New Mexico, 1992.

[79] Thomas J. LeBlanc y John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[80] C.-W. Lermen y Dieter Maurer. A protocol for distributed reference counting. En *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming*, ACM SIGPLAN Notices, págs. 343–350, Cambridge, MA, Agosto 1986. ACM Press.

[81] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, Marzo 1988. Comment 1 by Schlenk, Thu Jun 30 16:12:36 1988. Full scale transactions are supported on persistent objects. CLU is used to program the objects.

[82] M. C. Little y S. K. Shrivastava. Replicated k-resilient objects in Arjuna. En *Proc. of IEEE Workshop on the Management of Replicated Data*, págs. 53–58, Houston, Texas, USA, Noviembre 1990.

[83] M. C. Little y S. K. Shrivastava. Object replication in Arjuna. Informe técnico, BROADCAST TR-50, Dept. of Comp. Sc., Univ. of Newcastle, Newcastle upon Tyne, UK, 1993.

[84] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[85] Nancy A. Lynch y Mark R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Septiembre 1989.

[86] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. Tesis doctoral, Dept. of Computer Science, University of Zurich, Febrero 1995.

[87] Silvano Maffeis. The object group design pattern. En USENIX Association, editor, *2nd Conference on Object-Oriented Technologies & Systems (COOTS), June 17–21, 1996. Toronto, Canada*, págs. 155–163, Berkeley, CA, USA, Junio 1996. USENIX.

[88] U. Maheshwari y B. Liskov. Partitioned garbage collection of a large object store. En *Proc. of the ACM SIGMOD International Conference on Management of Data*, volumen 26 de *SIGMOD Record*, págs. 313–323, New York, Mayo 1997. ACM Press.

[89] U. Maheshwari y B. H. Liskov. Fault-tolerant distributed garbage collection in a client-server object-oriented database. En *Parallel and Distributed Information Systems (PDIS '94)*, págs. 239–248, Los Alamitos, Ca., USA, Septiembre 1994. IEEE Computer Society Press.

[90] Friedmann Mattern. Virtual time and global states of distributed systems. En *Parallel and Distributed Algorithms*, págs. 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[91] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3:184–195, April 1960.

[92] S. Mishra, L. L. Peterson y R. D. Schlichting. Implementing fault-tolerant replicated objects using psync. En *Proc. 8th Symp. on Reliable Distributed Systems (SRDS-8)*, págs. 42–52, Seattle, Washington, USA, 1989. IEEE Computer Society Press.

[93] James G. Mitchell, Jonathan Gibbons, Graham Hamilton, Peter B. Kessler, Yousef Y. A. Khalidi, Panos Kougiouris, Peter Madany, MichaelÑ. Nelson, Michael L. Powell y Sanjay R. Radia. An overview of the Spring system. En *COMPCON*, págs. 122–131, 1994.

[94] Luc Moreau y Jean Duprat. A construction of distributed reference counting. *Acta Informatica*, 37(8):563–595, 2001.

[95] L. E. Moser, Y. Amir, P. M. Melliar-Smith y D. A. Agarwal. Extended virtual synchrony. En *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems, Poznan, Poland*, págs. 56–65, Junio 1994.

[96] L. E. Moser, P. M. Melliar-Smith, D. 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.

[97] Louise. E. Moser, P. Michael Melliar-Smith y Priya Narasimhan. A fault tolerance framework for CORBA. En *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, Washington, DC, Junio 1999. IEEE Computer Society.

[98] J. E. B. Moss. Nested transactions: An approach to reliable distributed computing. Informe técnico, MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[99] Sape J. Mullender. Introduction. En S. Mullender, editor, *Distributed Systems*, págs. 3–18. acm Press, Wokingham, 1989.

[100] Sape J. Mullender. Kernel Support for Distributed Systems. En Sape Mullender, editor, *Distributed Systems*, capítulo 15. Addison-Wesley, 2nd edición, 1993.

[101] F. D. Muñoz-Escoí y J. M. Bernabéu-Aubán. The NanOS microkernel: A basis for a multicomputer cluster operating system. En H. R. Arabnia, editor, *Proc. of the 2nd International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA*, págs. 127–135. CSREA, Julio 1997.

[102] F. D. Muñoz-Escoí, J. M. Bernabéu-Aubán y P. Galdámez. Fault handling in distributed systems with group membership services. Informe técnico, DSIC-II/8/97, Univ. Politècnica de València, Spain, Mayo 1997.

[103] F. D. Muñoz-Escoí, P. Galdámez y J. 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, USA*, págs. 29–35, Octubre 1998.

[104] F. D. Muñoz-Escoí, P. Galdámez y J. 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, Slovakia*, volumen 1521 de *LNCS*, págs. 389–398. Springer Verlag, Noviembre 1998.

[105] F. D. Muñoz-Escoí, P. Galdámez y J. 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, USA, 1999.

[106] F. D. Muñoz-Escoí, P. Galdámez y J. M. Bernabéu-Aubán. Uso de interceptores CORBA para dar soporte a objetos replicados. En *Proc. of the VII Jornadas de Concurrencia, Gandía, Spain*, págs. 209–222, Junio 1999.

[107] F. D. Muñoz-Escoí, O. Gomis Hilario, P. Galdámez y J. M. Bernabéu-Aubán. HMM: A membership protocol for a multi-computer cluster. En *Appendix to Proc. of the VIII Jornadas de Concurrencia, Cuenca, Spain*, Junio 2000.

[108] Francisco Daniel Muñoz Escoí. *Hidra: Invocaciones Fiables y Control de Concurrencia*. Tesis doctoral, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Julio 2001.

[109] V. P. Nelson. Fault-tolerant computing: Fundamental concepts. *IEEE Computer*, 23(7):19–25, Julio 1990.

[110] Robert H. B. Netzer y Mark H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. En *Proceedings of the Conference on Programming Language Design and Implementation*, págs. 313–325, New York, NY, USA, Junio 1994. ACM Press.

[111] OMG. *Garbage Collection of CORBA Objects, Request for Proposals*. Document: orbos/97-08-08, Object Management Group, Agosto 1997.

[112] OMG. *CORBAservices: Common Object Services Specification*. Document: formal/98-12-09, Object Management Group, Diciembre 1998.

[113] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Julio 1999. Revision 2.3.

[114] OMG. *Fault Tolerant CORBA Specification V1.0*. Document: pct/2000-04-04, Object Management Group, Abril 2000.

[115] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater y Mark C. Little. The design and implementation of Arjuna. En USENIX, editor, *Computing Systems, Summer, 1995.*, págs. 255–308, Berkeley, CA, USA, Summer 1995. USENIX.

[116] Gregory F. Pfister. *In search of clusters: The Ongoing Battle in Lowly Parallel Computing, 2nd ed.* Prentice Hall, Englewood Cliffs, NJ, 1995 edición, 1998. :1998 edition: ISBN 0-13-899709-8, IBM.

[117] José M. Piquer. Indirect reference counting: A distributed garbage collection algorithm. En Aarts et al., editores, *PARLE'91 Parallel Architectures and Languages Europe*, volumen 505 de *Lecture Notes in Computer Science*, págs. 150–165. Springer-Verlag, Junio 1991.

[118] José M. Piquer. Indirect distributed garbage collection: Handling object migration. *ACM Transactions on Programming Languages and Systems*, 18(5):615–647, Septiembre 1996.

[119] D. Plainfossé y M. Shapiro. A survey of distributed garbage collection techniques. En H. Baker, editor, *Proc. of International Workshop on Memory Management*, volumen 986 de *LNCS*, Kinross, Scotland, Septiembre 1995. Springer-Verlag.

[120] David Plainfossé. *Distributed Garbage Collection and Referencing Management in the SOUL system*. Tesis doctoral, L'Université Pierre and Marie Curie., Junio 1994.

[121] Jon Postel, Editor. Transmission Control Protocol — DARPA Internet Program Protocol Specification. Internet Request for Comment RFC 793, Internet Engineering Task Force, Septiembre 1981.

[122] A. Ricciardi y K. P. Birman. Consistent process membership in asynchronous environments. En K. P. Birman y R. van Renesse, editores, *Reliable Distributed Computing with the Isis Toolkit*, capítulo 13, págs. 237–262. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[123] Helena C. C. D. Rodrigues y Richard E. Jones. A cyclic distributed garbage collector for Network Objects. En Ozalp Babaoglu y Keith Marzullo, editores, *Tenth International Workshop on Distributed Algorithms WDAG'96*, volumen 1151 de *Lecture Notes in Computer Science*, Bologna, Octubre 1996. Springer-Verlag.

[124] J. H. Saltzer, D. P. Reed y D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Noviembre 1984.

[125] R. D. Schlichting y F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant systems. *ACM Trans. on Computer Sys.*, 1(3), Agosto 1983.

[126] F. B. Schneider. Replication management using the state-machine approach. En S. J. Mullender, editor, *Distributed Systems*, capítulo 7, págs. 166–197. Addison-Wesley, Wokingham, UK, 2nd edición, 1993.

[127] M. D. Schroeder. A state-of-the-art distributed system: Computing with bob. En S. J. Mullender, editor, *Distributed Systems*, capítulo 7, págs. 1–16. Addison-Wesley, Wokingham, UK, 2nd edición, 1993.

[128] M. Shapiro, P. Dickman y D. Plainfossé. Robust distributed references and acyclic garbage collection. En M. Herlihy, editor, *Proc. of the 11th Annual Symposium on Principles of Distributed Computing*, págs. 135–146, Vancouver, BC, Canada, Agosto 1992. ACM Press.

[129] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. En *Proceedings of the 6th International Conference on Distributed Computer Systems*, págs. 198–205, Washington, 1986. IEEE Press.

[130] A. S. Tanenbaum y R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):579–625, Diciembre 1985.

[131] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen y G. van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, Diciembre 1990.

[132] R. van Renesse, K. P. Birman y S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, Abril 1996.

[133] S. Venkatesan y Brahma Dathan. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*, 21(2):163–176, February 1995.

[134] Visigenic. *Visibroker C++ 3.0 Programming Guide*. Visigenic Software, Inc., 1997.

[135] P. Watson y I. Watson. An efficient garbage collection scheme for parallel computer architectures. En J. W. de Bakker, A. J. Nijman y P. C. Treleaven, editores, *PARLE: Parallel Architectures and Languages Europe (Volume 2: Parallel Languages)*, págs. 432–443. Springer-Verlag, Berlin, DE, 1987. Lecture Notes in Computer Science 259.

[136] W. E. Weihl. Remote Procedure Call. En Sape J. Mullender, editor, *Distributed Systems*, págs. 37–64. ACM Press, 1990.

[137] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. En Yves Bekkers y Jacques Cohen, editores, *International Workshop on Memory Management*, volumen 637 de *Lecture Notes in Computer Science*, University of Texas, USA, Septiembre 1992. Springer-Verlag.

[138] Mark D. Wood. Replicated RPC using amoeba closed group communications. En *International Conference on Distributed Computing Systems*, págs. 499–507, 1993.