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 76
4.2 Planteamiento de objetivos 77
4.3 Cuenta distribuida y asíncrona de referencias 80
4.4 Formalización del algoritmo 82
4.5 Reconstrucción ante fallos 102
4.6 Recolección de residuos para objetos replicados y móviles 106
4.7 Trabajo relacionado 117

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.

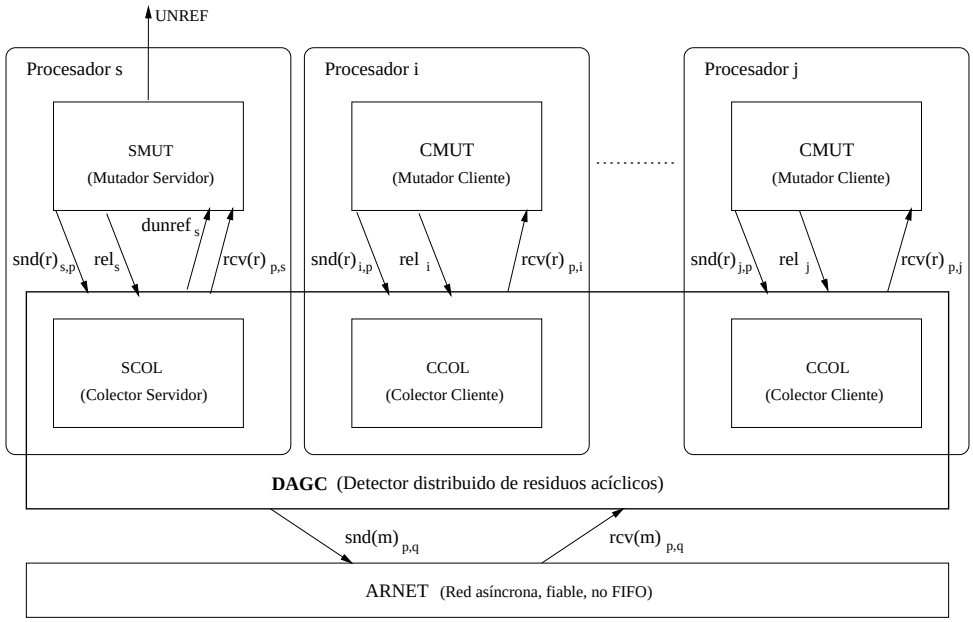

Figura 4.1: El problema de la detección de residuos acíclicos

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

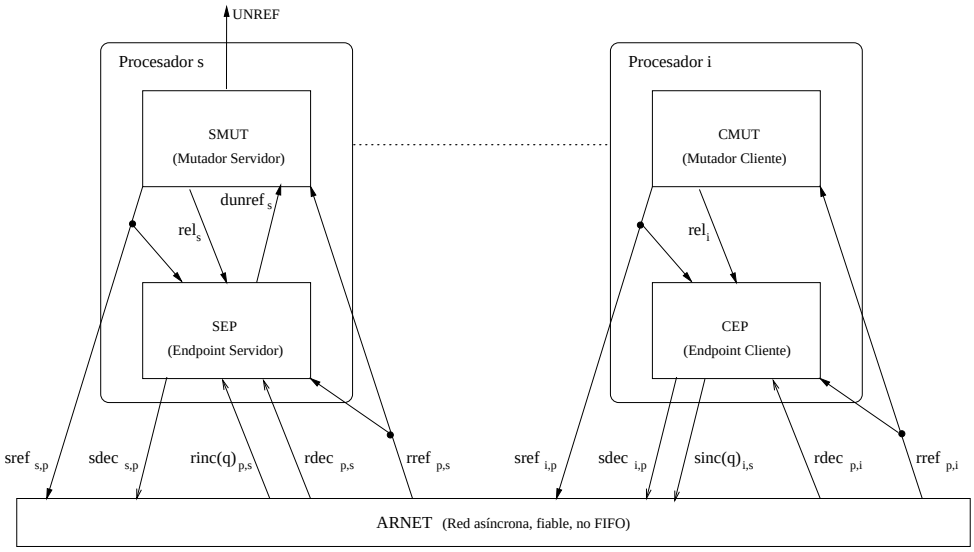

Figura 4.5: La detección de residuos acíclicos en Hidra

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. $CEP_i.status = null \Longrightarrow CEP_i.count = 0 \quad \forall i \in P$
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:

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

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.