Un Prototipo de Hidra

P. Galdámez Saiz, F.D. Muñoz Escoí, and J.M. Bernabéu Aubán
{pgaldam,fmunyoz,josep}@iti.upv.es

Instituto Tecnológico de Informática, Universidad Politécnica de Valencia,
Camino de Vera s/n, 46071 Valencia

Resumen En este trabajo describimos la implementación que hemos realizado de la arquitectura Hidra [3] en un prototipo simplificado. Hidra es una arquitectura para la construcción de clusters, y como tal una implementación definitiva deberá incluirse en el núcleo de un sistema operativo. Sin embargo, por su simplicidad y adecuación para el desarrollo de prototipos hemos elegido Java como lenguaje de implementación. El prototipo nos ha demostrado la viabilidad de los protocolos que planteamos en la arquitectura, nos ha exigido refinar las interfaces que teníamos diseñadas y nos ha forzado a revisitar el diseño interno de algunas componentes. Por tanto, gracias al esfuerzo de implementación que hemos realizado, también presentamos detalles adicionales de la propia arquitectura, en particular presentamos una descripción más actualizada del Gestor de Invocaciones a Objeto (ORB) de Hidra.

1 Introducción

Hidra [2,3] es una arquitectura diseñada para servir de soporte al desarrollo de servicios altamente disponibles en un cluster.

Un cluster Hidra (ver figura 1) estará formado por un conjunto de nodos interconectados con una red. Cada nodo ejecutará su propio núcleo de sistema operativo que ofrecerá unos servicios básicos a Hidra, e Hidra proporcionará a su vez servicios de alta disponibilidad a las aplicaciones que se construyan sobre ella. Las aplicaciones podrán residir en procesos de usuario o en el propio sistema operativo. Esta última posibilidad es la que permitirá construir servicios propios del sistema de forma altamente disponible.

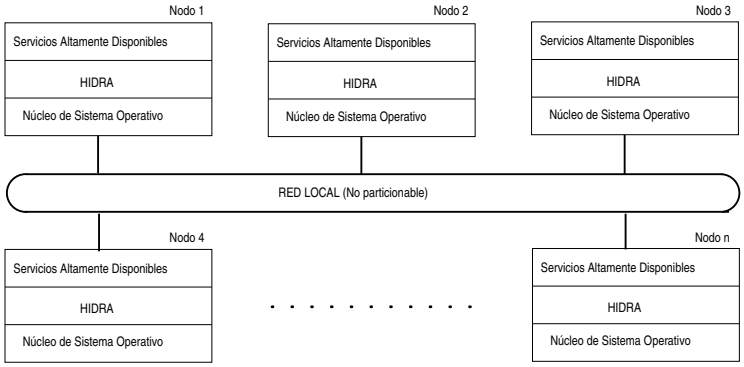

Figura1. Un cluster Hidra

Hidra se apoya sobre un modelo de sistema distribuido parcialmente síncrono, donde los mensajes pueden perderse, duplicarse, sufrir retrasos y llegar desordenados, donde los nodos pueden caer y donde adicionalmente asumimos que la red de interconexión no puede sufrir particiones.

Los servicios que necesita Hidra del núcleo del sistema operativo son los típicos ofrecidos por la mayoría de núcleos actuales. NanOS [8] es un ejemplo válido de núcleo sobre el que implantar Hidra. También se podrían utilizar sistemas operativos convencionales como Linux, Solaris, etc. En cualquier caso, los servicios mínimos que debe ofrecer el núcleo a Hidra son: el concepto de tarea, dominio de protección (o proceso), mecanismos de llamada al sistema (downcall) y de invocación desde el sistema operativo hacia los procesos (upcall), mecanismo de sincronización entre tareas, gestión de memoria, y primitivas de envío y recepción de mensajes por la red (transporte no fiable). Además es conveniente que se disponga de un mecanismo para incluir módulos en el propio núcleo, ya que Hidra en sí misma formará parte del núcleo para mejorar en eficiencia y aumentar la seguridad y estabilidad del sistema.

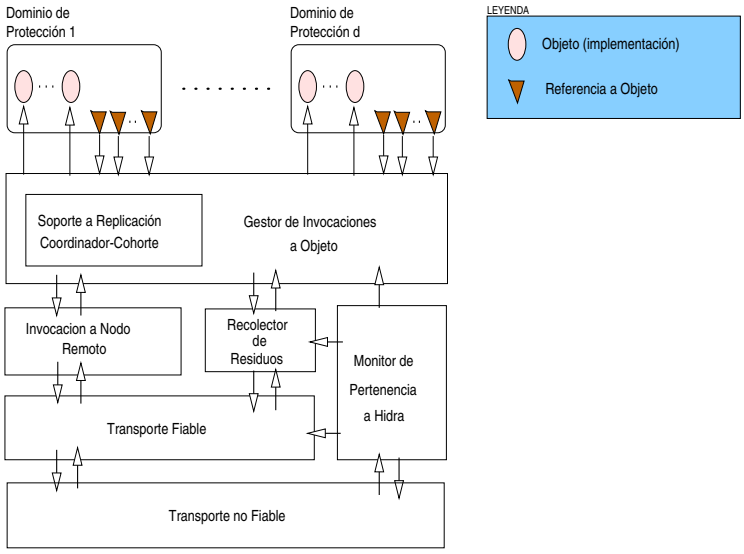

Figura2. La arquitectura Hidra

Tal y como puede observarse en la figura 2, la arquitectura Hidra está formada por varios niveles y componentes. Cada nivel utiliza los servicios de los niveles inferiores y ofrece servicios a los niveles superiores. A continuación describimos para cada uno de ellos, su diseño, funcionalidad y en su caso el modelo de sistema que asumen y el que proporcionan a los niveles superiores.

2 Transporte no fiable

Este nivel de transporte permite enviar mensajes a cualquier nodo del cluster Hidra utilizando su identificador de nodo. En este nivel, el modelo del sistema consiste en un conjunto reducido y numerado de nodos que pueden sufrir fallos de caída y donde los mensajes enviados entre todo par de nodos se pueden perder, pueden sufrir retrasos, pueden desordenarse y pueden duplicarse. Estrictamente hablando este nivel no forma parte de Hidra, sino que es, junto con el soporte del núcleo del sistema operativo, una componente necesaria para poder construir la arquitectura.

La interfaz ofrecida por este nivel tal y como está siendo utilizada por el último prototipo de Hidra es la que podemos observar en la figura 2.

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

```
public interface UnreliableTransport
{
    public void initialize (Node thisNode, Node [] preconfiguredNodes,
                            Properties transportProperties);

    public void start();

    public void send (Node destinationNode, OutgoingMessage message);

    public MessageHandlerRepository getMessageHandlerRepository();
    public MessageFactoryRepository getMessageFactoryRepository();
}
```

Figura3. Interfaz del Transporte no Fiable

Con el método `send()` se inicia el envío de un mensaje sin garantías de fiabilidad hacia el nodo `destinationNode`. Por su parte el método `start()` inicia la tarea de recepción de mensajes. Cada vez que se recibe un mensaje, se examina el tipo del mensaje que se encuentra codificado en su cabecera, y se invoca a la factoría de mensajes asociada a este tipo pasándole como argumento los bytes que forman el mensaje. Una vez creado el mensaje, se pasa el mensaje al manejador de mensajes registrado para este tipo. Los manejadores de mensajes y las factorías las registran los clientes del transporte utilizando los métodos `getMessageHandlerRepository()` y `getMessageFactoryRepository()` respectivamente.

Cualquier cliente del transporte no fiable que desee manejar sus propios mensajes se deberá registrar como manejador y creador de mensajes, implementando para ello las interfaces `MessageHandler` y `MessageFactory` respectivamente. Este es el caso de tanto del monitor de pertenencia a Hidra como del transporte fiable. Este último utilizará un sistema similar para permitir sus clientes manejar y crear mensajes fiables.

3 Monitor de pertenencia a Hidra

Sobre el nivel de transporte no fiable se apoya un protocolo de pertenencia a grupos [7] ampliado con servicios adicionales. Esta componente constituye el monitor de pertenencia a Hidra (HMM) [10]. Este monitor mantiene qué nodos forman parte del cluster Hidra, permitiendo que simultáneamente se produzcan nuevas incorporaciones al cluster y detectando qué nodos han caído. Ante cada reconfiguración del cluster (caídas o adiciones de nodos) el HMM genera un número de reconfiguración del cluster que será utilizado por los niveles superiores para enmascarar ciertos tipos de fallos. De igual forma, para proporcionar el modelo de fallo de parada [13], cada vez que un nodo se une al cluster, se le proporciona un número de encarnación nuevo de forma que aunque un mismo nodo se una al cluster y caiga de forma repetida, será considerado cada vez que lo haga como un nodo distinto.

La detección de caídas se realiza asumiendo que la red de comunicaciones es síncrona. Es decir, el protocolo de pertenencia está continuamente enviando mensajes de latido (heartbeat) y escuchándolos, sospechando de la caída de un nodo cuando éste deje de enviar sus mensajes de latido durante cierto período de tiempo.

El HMM además de mantener de forma distribuida en todo momento el conjunto de nodos que pertenecen al cluster, permite ejecutar de forma virtualmente síncrona una serie de pasos a todos los nodos en cada reconfiguración del cluster. Muchos sistemas utilizan el concepto de sincronía virtual [1] en el contexto de tolerancia a fallos. Su utilización más frecuente aparece en sistemas donde se construyen protocolos de entrega de mensajes con orden total o causal. Gracias al concepto de sincronía virtual, sus aplicaciones pueden construirse como si el sistema distribuido [asíncrono] subyacente fuera un sistema síncrono. Nuestra aproximación, a diferencia de estas, únicamente utiliza cierta forma de sincronía en cada reconfiguración del cluster (caídas de nodos y/o adiciones de nodos), no para entregar mensajes, sino para permitirle a cada nodo ejecutar pasos de reconfiguración de forma sincronizada al resto de nodos. Durante la ejecución normal del cluster, la sobrecarga inherente a este tipo de protocolos es evitada.

```
public interface MembershipMonitor
    extends MessageHandler, MessageFactory
{

    public void initialize (UnreliableTransport unreliableTransport,
                            Node thisNode,
                            Node [] preconfiguredNodes);
    public Membership joinCluster ();
    public void leaveCluster ();
    public Membership getMembership();
    public void registerListener (MembershipListener listener,
                                  int stepNumber,
                                  int millisTimeout);

    // MessageHandler interface
    public void handleMessage (IncomingMessage message);

    // MessageFactory interface
    public IncomingMessage createMessage (byte messageType,
                                          byte [] bytes);
}
```

Figura4. Interfaz del Monitor de Pertenencia a Hidra

En la figura 3 mostramos la interfaz del monitor de pertenencia a Hidra. La componente es inicializada con el método `initialize()` al que se le proporciona el transporte a utilizar para el envío de mensajes, los nodos que pueden llegar a formar parte del cluster y el identificador del nodo local. Al invocarse el método `joinCluster()`, el monitor comienza a ejecutar un protocolo para

unirse al cluster que estuviera previamente formado. Si el monitor no encuentra a ningún nodo, se constituirá un cluster formado únicamente por el nodo local. El método retorna los nodos que forman parte del cluster junto con sus números de encarnación y el número de reconfiguración actual del cluster. Por su parte el método `leaveCluster()` puede ser invocado para abandonar de forma voluntaria el cluster, permitiendo de esta forma al resto del cluster reconfigurarse de forma más rápida a como se reconfiguraría en caso de una caída del nodo. El método `getMembership()` lo utiliza el resto de Hidra para averiguar la identidad de los nodos que forman el cluster en cualquier momento y finalmente `registerListener()` se emplea para registrar observadores interesados en actuar en las reconfiguraciones del cluster.

Cada componente de Hidra que desee realizar alguna acción para reconfigurar su estado de forma sincronizada con el resto de nodos, registra un objeto de tipo `MembershipListener` en el HMM. En el registro se indica qué número de paso de reconfiguración es el que está registrando y el tiempo máximo que dedicará a ejecutar su código. Por su parte el HMM de cada nodo ejecutará los pasos de reconfiguración respetando el orden impuesto por el número de reconfiguración, de forma que todos los nodos completarán las acciones registradas para el paso i antes de que alguno de ellos comience a ejecutar las acciones del paso $i + 1$. Si algún paso de reconfiguración tarda más tiempo en completarse del que fuera especificado con el argumento `millisTimeout` en su registro, el HMM abortará a su nodo. Esta estrategia permite asegurar que todo el cluster estará reconfigurado y dispuesto a volver a ejecutarse con normalidad consumiendo un tiempo acotado a priori. En la práctica rara vez se encontrará el caso de un nodo Hidra que no complete su trabajo en el tiempo indicado, puesto que el código se diseña y se prueba para ello. Sin embargo es una medida adicional que permite evitar a los nodos que por el motivo que sea estén funcionado de forma incorrecta. Una configuración típica de un cluster Hidra no deberá requerir mucho más de un segundo en reconfigurarse completamente con lo que las garantías de alta disponibilidad del cluster quedan satisfechas.

4 Transporte fiable

El cliente más importante del HMM es el nivel de transporte fiable. Este nivel, elimina duplicidades en los mensajes y evita las pérdidas de mensajes de forma similar a como lo haría cualquier protocolo de transporte orientado a conexión (p.e. TCP [12]). Adicionalmente, apoyándose en el HMM, este transporte proporciona un nivel de fiabilidad adicional: Los mensajes que un nodo N envía a un nodo M, son reintentados hasta que la entrega sea confirmada o hasta que el HMM decida, de forma consensuada con el resto de nodos, excluir al nodo M del cluster. Conviene destacar que no se impone ningún tiempo de espera máximo con lo que teóricamente los mensajes pueden reintentarse durante tiempo ilimitado. Estas garantías de entrega fuertes, permiten al resto de niveles de Hidra confiar en que los mensajes siempre llegarán (a no ser que el nodo destino falle en cuyo caso se podrán ejecutar acciones de recuperación de forma sincronizada).

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

Con esto el nivel de transporte fiable de Hidra proporciona un modelo de sistema formado por un conjunto reducido de nodos donde los nodos pueden sufrir fallos de parada, y donde los mensajes pueden desordenarse y pueden sufrir retrasos arbitrarios, pero cuya entrega tendrá garantías fuertes.

```
public interface ReliableTransport
    extends MessageHandler, MembershipListener
{

    public initialize (Node thisNode,
                       UnreliableTransport ut,
                       MembershipMonitor mm);

    public MessageHandlerRepository getMessageHandlerRepository();
    public MessageFactoryRepository getMessageFactoryRepository();

    public void send (Node destination,
                      ReliableOutgoingMessage msg);

    public void multicast (Node [] destination,
                           ReliableOutgoingMessage msg);

    // Message Handler interface
    public void handleMessage (IncomingMessage message);

    // MembershipListener interface
    public void doReconfigurationStep(MembershipChanges membershipChanges,
                                      int stepNumber,
                                      int millisTimeout);
}
```

Figura5. Interfaz del Transporte Fiable

De la interfaz de este nivel (ver figura 4), destacan los métodos `send()` y `multicast()`. Ambos permiten enviar mensajes con garantías fuertes de entrega. El método para realizar multienvíos, a diferencia de lo que podemos encontrar en la mayoría de arquitecturas para alta disponibilidad no implementa ningún tipo de orden en la entrega de los mensajes, tan sólo garantiza su entrega con la misma semántica que resultaría de invocar a `send()` tantas veces como nodos haya en el destino del multienvío. Este tipo de multienvíos es útil en Hidra para enviar mensajes de tipo *checkpoint* a un conjunto de réplicas.

En lo que respecta al manejo de mensajes, el transporte fiable se registra como manejador de mensajes del transporte no fiable. Los mensajes que manejará son aquellos a los que se desea dotar de fiabilidad, como por ejemplo invocaciones y respuestas de invocaciones. El método `handleMessage()` es el responsable de esta tarea y será invocado por el transporte no fiable cuando lleguen sus mensajes. Por otra parte, de igual forma a como procede el transporte no fiable, el fiable, permite a sus clientes registrar manejadores y creadores de mensajes, de forma que los niveles superiores utilizarán al transporte fiable de forma similar a como el transporte fiable utiliza al no fiable o como el HMM utiliza al transporte no fiable.

Por último el método `doReconfigurationStep()` es el que será invocado por el HMM cada vez que ocurra una reconfiguración del cluster. Para que esto suceda, el transporte no fiable se registrará durante su inicialización como observador de los cambios en el cluster. La tarea principal que realiza el transporte fiable para reaccionar ante cambios en el cluster, será precisamente actualizar sus datos internos para filtrar los mensajes provenientes de nodos que hubieran caído.

5 Gestor de Invocaciones a Nodo Remoto

Por encima del nivel de transporte fiable, una fina capa implementa un nivel de llamada a nodo remoto sencillo. No llega a ser llamada a procedimiento remoto [15] pues este nivel no se encarga de empaquetar ni de desempaquetar argumentos. Tan sólo se encarga de enviar un mensaje a un destino, bloqueando a la tarea invocadora hasta que llegue la respuesta.

```
public interface RemoteNodeCallBroker
    extends MessageHandler, MessageFactory
{
    public void initialize (ReliableTransport rt);
    public IncomingReply invoke (OutgoingInvocation msg);
    public void multiInvoke (OutgoingInvocation msg);

    // MessageHandler, MessageFactory interfaces
    ...
}
```

Figura6. Interfaz del Gestor de Invocaciones a Nodo Remoto

La misión de invocar se realiza con las operaciones `invoke()` y `multiInvoke()`. Este último método es equivalente a realizar tantas invocaciones ordinarias como destinos estén codificados en la invocación. Empleamos las invocaciones múltiples para enviar los mensajes de *checkpoint* a las réplicas de los objetos replicados. La diferencia fundamental entre `multiInvoke()` y la operación `multicast()` del

nivel de transporte fiable, es que los destinos de `multiInvoke()` están especificados como ubicaciones de objetos y no como números de nodo. Gracias a esta diferencia `multiInvoke()` proporciona orden FIFO en la entrega de los mensajes que vayan dirigidos a las réplicas de un mismo objeto. Esta funcionalidad es útil para el caso de los `checkpoints` que se envían a los objetos replicados para que las réplicas actualicen su estado. Gracias al orden FIFO, cada réplica recibirá las actualizaciones emitidas desde un nodo en el mismo orden.

6 Recolector de residuos

Una de las principales características de Hidra es su facilidad para recolectar residuos. Hidra incluye un sistema de recolección de residuos para objetos tanto de implementación única, como para objetos replicados. El sistema garantiza que cualquier implementación de objeto[1], recibirá una notificación de *objeto no referenciado* cierto tiempo después de que no existan referencias que apunten al objeto. Para lograr este objetivo, Hidra incluye un protocolo distribuido de cuenta de referencias para objetos replicados y un protocolo de marcado y barrido para recuperar los invariantes del protocolo en caso de fallos de los nodos [4].

Adicionalmente todas las estructuras de datos internas del ORB que se mantienen para soportar tanto objetos como referencias a objeto son liberadas utilizando algoritmos de cuenta de referencias. El objetivo de esta cuenta local de referencias es garantizar que los recursos del sistema no se agotan conforme el sistema esté funcionando.

```
public interface GarbageCollector
       extends MessageFactory, MessageHandler, MembershipListener
{
    public void initialize (ReliableTransport rt,
                            ORBinternal orb,
                            Node [] nodes);
    public void addInc (ClusterSlot dest, ClusterSlot origin)
    public void addDec (ClusterSlot dest);
    public void addMig (ClusterSlot dest, Locator newLocator);
    public void addUnref (ClusterSlot dest);

    public byte [] bytes getPiggyMessage (Node destination);
    public void setPiggyMessage (Node origin, byte [] bytes);

    // MessageFactory, MessageHandler, MembershipListener interfaces
    ...
}
```

Figura7. Interfaz del Recolector de Residuos

[1] Decimos implementación de objeto y no simplemente objeto, para contemplar el caso de los objetos replicados donde todas las réplicas (implementaciones) recibirán la notificación.

La interfaz del recolector de residuos es la que se proporciona en la figura 6. Las operaciones `addInc()`, `addDec()`, `addMig()` y `addUnref()` las utiliza el ORB para enviar los mensajes propios del protocolo de cuenta de referencias a los demás nodos. Todos los mensajes se envían de forma asíncrona, lo que permite que almacenemos estos mensajes en memoria temporal, y cada cierto período de tiempo se envíen los mensajes a su destino. El método `start()` inicia una tarea que se encarga de realizar este volcado periódico de los mensajes. Por último los métodos `getPiggyMessage()` y `setPiggyMessage()` son utilizados por el transporte fiable para transmitir los mensajes del recolector de residuos como información adicional a los mensajes que maneje (piggybacking).

En lo que respecta a la reconfiguración del cluster, el recolector de residuos se registra como observador en el HMM para participar en el protocolo de reconstrucción de la cuenta de referencias. Este protocolo es necesario para restablecer los invariantes del protocolo ordinario de cuenta de referencias en caso de fallos.

```
public interface ORBinternal {
...
    public void Inc (ClusterSlot dest, ClusterSlot origin)
    public void Dec (ClusterSlot dest);
    public void Mig (ClusterSlot dest, Locator newLocator);
    public void Unref (ClusterSlot dest);
...
}
```

Figura8. Interfaz interna del ORB

Por su parte el recolector de residuos interactúa con el ORB para informarle de los mensajes que debe procesar. En la figura 6 tenemos la interfaz que ofrece el ORB al recolector de residuos para este fin. Todos los métodos son utilizados exclusivamente por el recolector de residuos y permiten al ORB tratar la cuenta de referencias distribuida de forma similar a como trata la cuenta de referencias local de las demás entidades.

7 Gestor de invocaciones a objeto

El diseño del ORB de Hidra toma diversas ideas de diferentes sistemas, de entre las que destacan fundamentalmente dos: CORBA[11] y Solaris-MC[6]. Sin embargo Hidra presenta aspectos diferenciadores con ambos. No se trata de un ORB que cumpla con el estándar por tener unos objetivos distintos, fundamentalmente alta disponibilidad, recolección de residuos y eficiencia, y no forzosamente estandarización ni interoperabilidad. En cambio el modelo de objetos es similar al modelo de CORBA y a nivel arquitectónico los adaptadores de objeto de CORBA y los interceptores han sido tenidos en cuenta en el diseño de sus homólogos en Hidra. De Solaris-MC, el ORB de Hidra toma determinados aspectos

funcionales de los xdoors y de los handlers, comparables con los endpoints y los adaptadores de Hidra. El diseño del orb lo podemos observar en la figura 9.

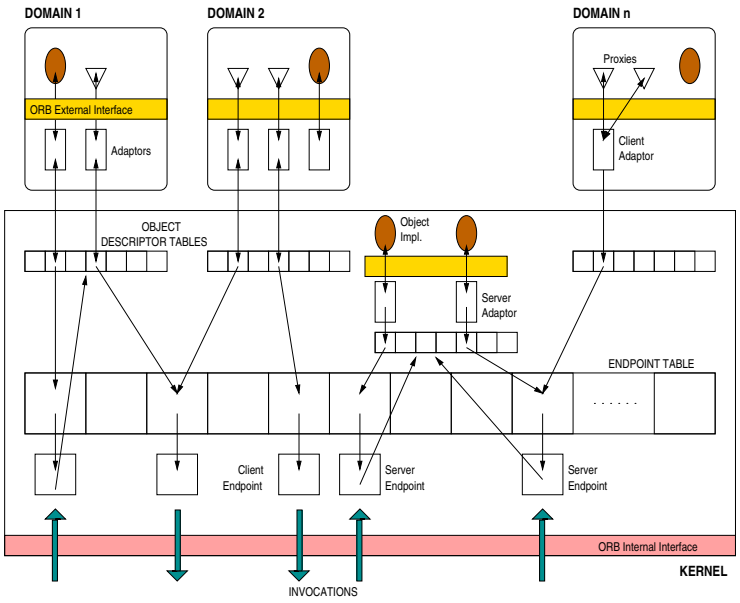

Figura9. El Gestor de Invocaciones a Objeto de Hidra

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

El nivel de *descriptores de objeto* contiene una tabla de descriptores de objeto por cada dominio de protección (incluido el propio núcleo del sistema operativo). La funcionalidad de este nivel proporciona a cada dominio de protección acceso a los objetos para los que el dominio posea referencias utilizando capacidades. De forma simétrica, proporciona al núcleo acceso a los objetos que residen en el dominio utilizando punteros a los adaptadores.

El tercer nivel, el *nivel de adaptadores*, ya reside en cada dominio de protección y consiste en una serie de adaptadores de objeto que están asociados, o bien

a implementaciones de objeto (adaptadores servidores), o bien a referencias a objeto (adaptadores clientes). Los adaptadores contienen un descriptor de objeto para permanecer conectados al ORB del núcleo. La diferencia entre adaptadores servidores y adaptadores clientes consiste en que los primeros contienen un único puntero para acceder a la implementación del objeto, mientras que los segundos contienen una tabla de punteros para permanecer conectados a los proxies del objeto.

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

7.1 Interfaz externa del ORB

```
public interface ORBexternal {
    public void initialize (String [] args);

    public BOA getBOA();
    public ROA getROA();
    public FOA getFOA();

    public NameServer getNameServer ();
    public void release (HIDRAObjectImpl obj);
    public HIDRAObjectImpl duplicate (HIDRAObjectImpl obj);
}
```

Figura10. Interfaz externa del ORB

La interfaz que ofrece el ORB de Hidra a la aplicaciones es la que podemos observar en la figura 7.1. El método *initialize()* configura el ORB proporcionando al menos el número de nodos que forman parte del cluster, y cómo asociar identificadores de nodo a las direcciones de nodo que utilice el transporte. Nótese que este método sólo es necesario para configurar el ORB dentro del sistema operativo y no será visible a las aplicaciones de usuario. Los métodos `release()` y `duplicate()` permiten descartar y duplicar respectivamente referencias a objeto. En particular el método `release()` tiene especial relevancia pues puede desencadenar la liberación tanto de las estructuras de datos internas del ORB que soportan al objeto, como desencadenar la ejecución de parte del protocolo distribuido de cuenta de referencias.

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

Figura11. Interfaces de los Adaptadores del ORB

Los demás métodos, sirven para obtener los adaptadores[2] del ORB. Estos adaptadores son similares en esencia a los adaptadores CORBA, y permiten registrar diferentes tipos de objetos en el ORB. Hasta el momento tenemos diseñados e implementados tres adaptadores del ORB: el FOA, el BOA y el ROA. Las interfaces de estos adaptadores pueden observarse en la figura 7.1. El FOA permite registrar objetos de tipo *fijo*, el BOA objetos básicos y el ROA objetos replicados.

7.2 Tipos de objeto

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

Los objetos básicos, son los objetos ordinarios del modelo de objetos CORBA, con la salvedad que serán recolectados como residuos cuando no existan referencias que les apunten, y por su parte los objetos replicados son aquellos que pueden tener más de una implementación.

Para los objetos replicados, el ROA proporciona además del método de registro de objetos, las operaciones `joinObject()` y `leaveObject()` para permitir añadir y eliminar réplicas a objetos replicados. Estas operaciones deberán ser ejecutados por el dominio que posea la implementación que se desee añadir o eliminar del objeto replicado y su invocación no terminará hasta que todos los nodos hayan actualizado el estado de las réplicas en lo referente a su número y ubicación.

Al registrar un objeto en el ORB utilizando alguno de los adaptadores del ORB, se crea el adaptador de objeto correspondiente y éste se asocia al objeto.

[2] Los adaptadores del ORB pueden verse como métodos de clase proporcionados por los adaptadores de objeto.

Por tanto disponemos de adaptadores de objeto distintos según a qué tipo de objeto se asocie. Algo similar ocurrirá con los endpoints que se creen para conectar a los adaptadores, se creará un tipo de endpoint distinto (fijo, básico o replicado) según sea el tipo del adaptador servidor que desencadene su creación.

7.3 Interfaz interna del ORB

Adicionalmente a los métodos dedicados a la recolección de residuos, expuestos en la sección 6, la interfaz interna del ORB proporciona otros métodos relacionados tanto con el HMM como con el tratamiento de las invocaciones. Para simplificar la exposición, baste comentar que el ORB se registrará como observador de las reconfiguraciones del cluster y que proporciona métodos para poder acceder a los endpoints utilizando identificadores únicos de ubicación de endpoints (ClusterSlots).

Para reaccionar ante los fallos, participa en el protocolo de reconstrucción de la cuenta de referencias e implementa un protocolo de reconstrucción de objetos replicados.

8 Estado de la implementación y Conclusiones

Hemos implementado todos los niveles de la arquitectura Hidra y varios programas distribuidos de prueba que ejercitan la mayoría de componentes. En el momento de redactar este trabajo tan sólo resta integrar el soporte al modelo de replicación coordinador-cohorte en el prototipo, que ya implementamos con anterioridad sobre el ORB de Visibroker [14,9].

La implementación del prototipo nos ha servido fundamentalmente para validar y refinar el diseño en general de Hidra y corregir y ampliar el diseño de algunas de sus componentes. En particular el diseño del ORB ha sido mejorado y el prototipo nos ha permitido probar protocolos que aún no teníamos implementados. En particular hemos podido probar el recolector de residuos para objetos ordinarios y para objetos replicados, comprobando de forma experimental su corrección y la poca sobrecarga que introduce en el sistema. Para ello hemos desarrollado una aplicación distribuida ejemplo que hace uso extensivo de objetos y operaciones sobre los objetos para ejercitar el recolector de residuos de forma insistente.

Hasta ahora, el prototipo está siendo de gran utilidad, pero su objetivo principal es servir de primera implementación de Hidra antes de migrar el código al núcleo del sistema operativo. Planeamos implementar Hidra en C++ e integrarlo en Linux, para posteriormente desarrollar software de sistema en el propio kernel que utilice las facilidades de alta disponibilidad de Hidra. Puede parecer sorprendente que entonces hayamos elegido Java y no C++ para el prototipo, pero hemos llegado a esta elección por varios motivos:

- Sintácticamente Java es similar a C++. De hecho existen traductores de libre uso para migrar código Java a C++.

- Para la gran mayoría de errores en tiempo de ejecución, la máquina virtual Java muestra la pila de llamadas efectuadas hasta el momento del fallo. Con C++ lo habitual en caso de este tipo de fallos es que la aplicación termine sin más datos que una mera información de acceso ilegal a la memoria. Si la aplicación a desarrollar es distribuida, la importancia de esta facilidad aumenta por la ausencia en general de depuradores adecuados [5].
- Que el recolector de residuos de Java recolecte los objetos automáticamente, simplifica la implementación, permitiendo centrarse en los aspectos funcionales, olvidándose de gran parte de aspectos de asignación y liberación de memoria.
- La cantidad de clases de utilidad probadas y verificadas en Java e incluidas de forma estándar, tales como tablas hash, listas, pilas, etc. también facilitan enormemente cualquier desarrollo de software de sistema, permitiendo centrarse en lo más esencial.
- Los monitores de Java, obligan a razonar en entornos multitarea en términos de operaciones que pueden proceder en paralelo o que deben ser serializadas. En nuestra opinión, es más sencillo y lleva a mejores diseños utilizar esta facilidad que razonar en términos de cerrojos o de cualquier otra primitiva que disponga el sistema para controlar la concurrencia. Si bien es cierto que con la mayoría de primitivas se puede implementar lo mismo que con monitores (incluso los mismos monitores), el hecho de que Java obligue a usarlos, es ventajoso en nuestra opinión en etapas de diseño.

Para desarrollar el prototipo hemos tenido especial cuidado de llevar la cuenta de referencias para todos los objetos que forman parte del ORB: casos de los endpoints, adaptadores, proxies y descriptores de objeto. Es decir, para ellos no utilizamos las facilidades de Java de recolección de residuos, sino que llevamos la cuenta de las referencias para liberarlas manualmente cuando sea necesario. Esto lo hemos realizado así porque la recolección de residuos es una parte esencial de Hidra, y sólo pretendíamos que Java nos "ayudara" en el tratamiento de los objetos y entidades temporales, pero no para aquellos que forman parte del soporte básico de Hidra.

Referencias

1. K. P. Birman and T. A. Joseph. Exploiting virtual synchrony in distributed systems. *Proc 11th ACM Symposium on OS Principles, Austin, TX, USA*, pages 123–138, November 1987.
2. P. Galdámez, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. HIDRA: Architecture and high availability support. Technical report, DSIC-II/14/97, Univ. Politècnica de València, Spain, May 1997.
3. P. Galdámez, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. High availability support in CORBA environments. In F. Plášil and K. G. Jeffery, editors, *24th Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic*, volume 1338 of *LNCS*, pages 407–414. Springer Verlag, November 1997.
4. P. Galdámez, F. D. Muñoz-Escoí, and J. M. Bernabéu-Aubán. Garbage collection for mobile and replicated objects. In J. Pavelka, G. Tel, and M. Bartosek, editors, *26th Seminar on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic*, volume 1725 of *LNCS*, pages 373–380. Springer Verlag, November 1999.
5. Pablo Galdámez, Declan Murphy, José M. Bernabéu-Aubán, and Francesc D. Muñoz-Escoí. Event-based techniques to debug an object request broker. *The Journal of Supercomputing*, 13(2):133–149, March 1999.
6. Y. A. Khalidi, J. M. Bernabéu-Aubán, V. Matena, K. Shirriff, and M. Thadani. Solaris MC: A multi computer OS. In *Proceedings of the USENIX 1996 annual technical conference, San Diego, California, USA*, pages 191–203. USENIX, January 1996.
7. F. D. Muñoz-Escoí, J. M. Bernabéu-Aubán, and P. Galdámez. Fault handling in distributed systems with group membership services. Technical report, DSIC-II/8/97, Univ. Politècnica de València, Spain, May 1997.
8. F. D. Muñoz-Escoí, P. Galdámez, and J. M. Bernabéu-Aubán. The NanOS cluster operating system. In R. Buyya, editor, *High Performance Cluster Computing*, volume 1, chapter 29, pages 682–702. Prentice-Hall PTR, Upper Saddle River, NJ, USA, 1999.
9. F. D. Muñoz-Escoí, P. Galdámez, and J. M. Bernabéu-Aubán. Uso de interceptores CORBA para dar soporte a objetos replicados. In *Proc. of the VII Jornadas de Concurrencia, Gandía, Spain*, pages 209–222, June 1999.
10. F. D. Muñoz-Escoí, O. Gomis Hilario, P. Galdámez, and J. M. Bernabéu-Aubán. HMM: A membership protocol for a multi-computer cluster. In *Appendix to Proc. of the VIII Jornadas de Concurrencia, Cuenca, Spain*, June 2000.
11. OMG. *The Common Object Request Broker: Architecture and Specification.* Object Management Group, July 1999. Revision 2.3.
12. Jon Postel, Editor. Transmission Control Protocol — DARPA Internet Program Protocol Specification. Internet Request for Comment RFC 793, Internet Engineering Task Force, September 1981.
13. R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant systems. *ACM Trans. on Computer Sys.*, 1(3), August 1983.
14. Visigenic. *Visibroker C++ 3.0 Programming Guide.* Visigenic Software, Inc., 1997.
15. W. E. Weihl. Remote Procedure Call. In Sape J. Mullender, editor, *Distributed Systems*, pages 37–64. ACM Press, 1990.