LDBA: Lenguaje Distribuido Basado en Actores

Francisco J. Mendieta
Alejandro Quintero
Departamento de Ingeniería de Sistemas y Computación
Universidad de los Andes
Apartado Aéreo 4976 - Bogotá - Colombia
e-mail: aquinter@andescol.bitnet
fmendiet@andescol.bitnet

Resumen: La investigación en la programación orientada por objetos ha mostrado la factibilidad de distribuir la computación utilizando el paso de mensajes de este paradigma y la posibilidad de obtener concurrencia a nivel de métodos dentro de un mismo objeto y en distintos objetos a la vez. El modelo actor de programación concurrente, presenta una alternativa de diseño para lenguajes distribuidos pues trae consigo el concepto de entidades autónomas -**actores**- cuya única forma de comunicación es el paso de mensajes. A continuación presentamos un lenguaje orientado por objetos de tipo C++ cuyos objetos activos -**actores**- se implantan transparentemente en procesadores distribuidos.

Palabras claves: Programación Orientada por Objetos, Programación Concurrente, Lenguajes Actores, Lenguajes para Sistemas Distribuidos, Sistemas Distribuidos

1. Introducción

El desarrollo de nuevas metodologías para la construcción de sistemas distribuidos y paralelos, ha llevado al desarrollo de nuevas áreas de investigación como la programación concurrente orientada por objetos y los lenguajes orientados por objeto como mecanismos para su especificación y descripción. Los sistemas son descritos entonces como un conjunto de objetos que interactúan entre si de manera concurrente mediante protocolos uniformes de comunicación. Las ventajas que ofrece el paradigma de programación concurrente orientado por objetos son las siguientes: a) los lenguajes orientados por objetos permiten expresar más explícitamente las relaciones que existen entre elementos del mundo real y sus contrapartes en el software, b) la comunicación por medio del paso de mensajes básica en los sistemas distribuidos, es reflejada en el paradigma del paso de mensajes usado por los lenguajes orientados por objetos [1].

El modelo actor de programación concurrente introducido por Carl Hewitt [2], permite modelar objetos compartidos con cambios locales en su estado, reconfigurar dinámicamente la conectividad entre objetos y el procesamiento concurrente inherente a sus objetos -**Actores**-. Más recientemente, Gul Agha [3] definió un modelo actor con un número pequeño de primitivas muy poderosas que son la base de la construcción del lenguaje **ACT++** [1] y de nuestra implantación distribuida de este lenguaje llamada **LDBA** -Lenguaje Distribuido Basado en Actores- [4]. Este artículo describe el proyecto de implantación de LDBA en el cual intentamos verificar la utilidad de la programación de actores en lenguajes imperativos orientados por objetos, analizando las posibilidades de reutilización del software, y la factibilidad de construir jerarquías de herencia que nos permitirán lograr nuestro principal objetivo: la concurrencia.

La organización de este artículo es la siguiente: la sección 2 describe el lenguaje LDBA, la sección 3 presenta las conclusiones de nuestro proyecto de implantación del lenguaje y finalmente la sección 4 exhibe la bibliografía utilizada.

2. LDBA: Lenguaje Distribuido Basado en Actores

LDBA es una implantación distribuida del lenguaje ACT++ [1] el cual está basado en el modelo de Gul Agha [3]. En términos de implantación LDBA es una extesión de C++ [5] con una jerarquía de clases que permiten explotar el paralelismo intrínseco del modelo actor en una red local de máquinas Apollo (APOLLO COMPUTER INC.).

2.1. Primitivas de LDBA

El modelo actor es un modelo de computación concurrente en el cual la computación es llevada a cabo por actores que se comunican mediante el paso de mensajes. El modelo tiene cuatro elementos básicos: actores, colas de correo, comportamientos y relaciones.

Un **actor** es un objeto activo autocontenido. La interacción entre actores puede ocurrir solo a través del envío de mensajes. Cada actor está asociado a una única **cola de correo**, cuya dirección sirve como identificación del actor. Los mensajes enviados a un actor, son guardados en su cola de correo y son leídos uno a uno en estricto orden FIFO. Si la cola está vacía cuando el actor está listo para recibir el siguiente mensaje, el actor es bloqueado hasta que un mensaje llegue a la cola. Un actor A puede enviar un mensaje a un actor B, solo si A conoce la dirección de la cola de correo de B. Las direcciones de las colas de correo pueden ser enviadas como parte de un mensaje. Estas dos últimas propiedades permiten configurar dinámicamente la conectividad entre actores.

El **comportamiento** de un actor determina como reaccionar a los requerimientos especificados en el mensaje procesado. Un comportamiento o definición de clase actora , está definido por un cuerpo de código llamado **behavior script**. Este script contiene definiciones de métodos y **relaciones** o variables de instancia . Estas relaciones son los nombres de otros actores a los cuales el actor puede enviar mensajes -el modelo es homogéneo y las únicas entidades que pueden existir son actores-. Las relaciones del modelo de Agha son variables solo para lectura y no para escritura ya que la concurrencia inherente a nivel de instrucción no se puede garantizar si existe dependencia de datos entre instrucciones, pues cada sentencia es ejecutada concurrentemente -granuralidad fina- excepto cuando hay restricciones secuenciales por el orden casual. En LDBA las relaciones pueden ser modificadas gracias a la granuralidad gruesa de nuestra implantación -ejecución concurrente de métodos- haciendo el modelo más compatible con los lenguajes imperativos: de esta manera un actor puede recibir dinámicamente nombres de otros actores y cambiar mediante asignaciones sus variables de instancia.

En el procesamiento de un mensaje un actor puede hacer tres acciones: crear nuevos actores, enviar mensajes a otros actores y especificar un comportamiento de reemplazo. En el modelo de Gul Agha las dos primeras son opcionales y la tercera es obligatoria; sin embargo en LDBA el especificar un comportamiento de reemplazo solo debe hacerse cuando un actor ha modificado de alguna forma sus relaciones y quiere que este cambio en su estado sea permanente. A continuación detallaremos cada una de estas acciones.

2.1.1. La creación de actores

En LDBA podemos distinguir dos tipos de objetos: activos y pasivos. Los objetos activos son aquellos que de alguna forma son subclases de **Actor**, que es una clase definida en el lenguaje, mientras que los objetos pasivos constituyen todas las demás clases que el usuario defina. Los actores son creados dinámicamente en ejecución a medida que se necesitan mediante el operador **New** -no confundirlo con el new de C++-. Recibe como parámetro el nombre de la clase actora a crear, por ejemplo Mbox UnaMbox = New("UnaClaseActora") y retorna la cola de correo asociada al actor creado. Otros actores pueden enviar mensajes al nuevo actor usando esta cola de correo. Un ejemplo del encabezado de una clase actora sería:

```
class UnaClaseActora: public Actor {...};
```

2.1.2. El envío de mensajes

El invocar métodos de un objeto pasivo tiene la semántica de un llamado a una función y no lo discutiremos -podemos asociarlo con las transmisiones sincrónicas de ABCL/1 [6]-. La invocación de un método de un actor se hace mediante el paso de un mensaje asincrónico. El paso asincrónico de mensajes es soportado a través de objetos predefinidos como son las colas de correo, que también traen consigo la concurrencia inter-objeto: métodos de distintos objetos ejecutándose simultáneamente. Dos tipos de mensajes vamos a distinguir: **mensajes de requerimiento** y **mensajes de respuesta**.

Un mensaje de requerimiento es usado para invocar un método mientras un mensaje de respuesta es utilizado para entregar el resultado de la invocación de un método. Dos tipos de colas de correo están disponibles para soportar estos dos tipos de mensajes: las **Mboxes** y las **Cboxes**. Las Mboxes modelan las colas de correo de un actor mientras que las Cboxes permiten la programación con **futures** [7] que corresponden de alguna manera a una promesa de entrega: la computación es efectuada por un proceso paralelo que puede seguir activo después de que el future haya sido devuelto, los futures se confunden de esta manera con el valor de los datos a entregar.

Un mensaje de requerimiento consta del nombre del método a ser ejecutado por el actor receptor y los argumentos para el método. Si se quieren recibir respuestas de la invocación del método, se debe especificar una Cbox dentro de los argumentos como identificación del actor continuación [2], que es el actor que recibirá la respuesta del mensaje de requerimiento. Si no se

especifica el actor continuación tendremos asincronismo puro y si se especifica tendremos las transmisiones anticipadas de ABCL/1 [6]. Por ejemplo, si tenemos un actor **sumador** cuya Mbox es **mboxSum** y que posee un método **Sume** que nos permite sumar dos enteros, la invocación del método sería:

```
mboxSum << "sumador::Sume" << entero_1 << entero_2 << miCbox;
```

donde **miCbox** indica la Cbox donde se dejará el resultado de la suma.

Un actor puede referirse a su Mbox, mediante la variable predefinida **MiMbox**. Siguiendo nuestro ejemplo, el método **Sume** del actor sumador obtendría los parámetros de su invocación así:

```
MiMbox >> par_1 >> par_2 >> UnaCbox;
```

Como el nombre del método fue utilizado para seleccionar su invocación correspondiente, en MiMbox solo se reciben los parámetros. El número de parámetros en una invocación a un método puede ser fácilmente parametrizado, ya que los parámetros son en realidad enviados a un objeto **Correo** de las Mboxes el cual es retornado cuando la Mbox recibe el nombre del método a invocar.

Los mensajes de respuesta como CboxResp << suma, son colocados en las Cboxes que implantan la espera por necesidad. Un actor puede recibir de una Cbox mediante el operador >> y enviar a una Cbox con el operador <<. Si en una recepción hay un resultado para ser entregado al actor, le es entregado inmediatamente, de lo contrario el actor se bloquea hasta que un mensaje llegue. Un actor puede revisar si un mensaje de respuesta ha llegado a una Cbox usando la operación **in()** sobre la Cbox.

Las razones para la existencia de las Cboxes son principalmente el evadir el desdoblamiento de un actor que necesita una respuesta, en múltiples continuaciones que esperan por ella y el permitir una implantación eficiente de un actor que no puede proceder hasta que una respuesta sea recibida.

2.1.3. Especificación de un comportamiento de reemplazo

Un comportamiento es escrito usando las primitivas de actores y las construcciones imperativas de C++. Las instrucciones en un comportamiento son ejecutadas secuencialmente. La operación

llamada **become** permite cambiar el comportamiento de un actor dinámicamente en ejecución. Un actor puede ser asociado a distintos comportamientos en diferentes instantes del tiempo. Estos comportamientos podrían no tener los mismos métodos o estructuras de datos internas. La única entidad que no cambia en la vida de un actor, es la dirección de su cola de correo.

En el modelo de Agha, la operación become necesita un nombre de un comportamiento y una lista de relaciones -que puede ser muy larga si el actor está en lo profundo de una jerarquía de herencia-, haciendo que las estructuras de datos utilizadas por los actores sean pequeñas. En nuestro modelo, la operación become recibe un apuntador a un nuevo actor o la pseudo-variable **Self**, haciendo la estructura interna de los actores más flexible y compacta. La pseudo-variable Self, se utiliza para especificar como comportamiento de reemplazo al actor mismo.

La operación become juega dos papeles importantes en el modelo como son: es el único medio que tiene un actor para cambiar su estado, es decir, para hacer permanente cualquier cambio en sus relaciones y mantiene la consistencia de las relaciones del actor, ya que junto con los mensajes guardados en la cola de correo permite tratar el siguiente mensaje no procesado en forma serial.

En nuestra implantación, el usuario es el responsable de indicar si un actor es serializado -ejecuta un método a la vez- o no, especificando en el método constructor de la clase la constante **SERIALIZADO** como argumento para la construcción de la parte actora del objeto. Se sabe que un actor debe ser serializado, si en algún momento modifica sus relaciones y hace este cambio de estado permanente mediante una operación become. Por ejemplo, si sabemos que el actor **Almacén** al tratar el mensaje **retire**, decrementa su relación **cantidad** en el valor solicitado, su constructora seguiría el siguiente esquema:

```
Almacen::Almacen (void)
:Actor (SERIALIZADO)
{cantidad = MAX_ARTICULOS;}
```

y su método **retire** -teniendo en cuenta que hay artículos suficientes- sería:

```
void Almacen::retire (void)
{int pedido;
MiMbox >> pedido;
cantidad -= pedido;
become (Self);
}
```

Si no se especifica nada al construir la parte actora de Almacén -mediante :Actor(..)-, el actor Almacén sería por defecto **NO_SERIALIZADO**, es decir, permitiría el tratamiento simultáneo de métodos correspondientes al actor Almacén sin ningún mecanismo de sincronización, con la posibilidad de inconsistencias en el estado del actor cuando se hagan cambios en el valor de sus relaciones.

Un resultado significativo de la operacón become, es el potencial de concurrencia dentro del actor. La concurrencia intra-objeto -varios métodos de un objeto ejecutándose al mismo tiempo- naturalmente aparece cuando el comportamiento corriente continúa después de especificar su comportamiento de reemplazo.

2.2. Operaciones remotas

La existencia de las colas de requerimiento -Mboxes- y de las colas de respuesta -Cboxes- nos hace pensar en una implantación distribuida del modelo actor como la hace LDBA, que aprovecha el paso de mensajes entre actores para implantar tanto su creación como su ejecución remota de métodos, ambas en forma transparente para el usuario. Para lograr ésto, hemos definido un lenguaje muy sencillo con la misma filosofía del lenguaje de definición de interfaces que utiliza **RPC** para sus operaciones remotas. En este lenguaje se declaran para cada clase actora, los métodos de su interfaz que son requeridos para la computación. Con esta información y la ayuda de la herramienta **LDBA_parser** podemos generar el código necesario para la creación de actores por nombre -note que el argumento del New es una cadena de caracteres- y para la ejecución de los métodos de actores por procesos en máquinas remotas -que también son identificados por nombre-.

El archivo que describe los métodos de las clases actoras que se invocarán, presenta la siguiente forma:

```
methods <nombre_clase_actora> in <archivo_encabezado_clase> {
    void <nombre_metodo_1> ();
    ...
    void <nombre_metodo_n> ();
}
```

donde:

- nombre_clase_actora es el nombre de la clase actora cuyos métodos pueden ser accesados remotamente por medio de mensajes de requerimiento.

- archivo_encabezado_clase es el archivo de encabezado de la clase -.h- donde se definen de acuerdo a la sintaxis de C++ los métodos y los atributos de la clase actora.

- nombre_metodo_i es el nombre de un método de la interfaz del actor que se da a conocer para que pueda ser accesado por un mensaje de requerimiento.

La distribución de la computación es posible debido a la creación remota de actores y a la posibilidad de comunicarse entre ellos por medio de mensajes. La creación de actores en las diferentes máquinas del sistema se hace en forma balanceada por una autoridad central que decide la máquina de creación con base en el número de actores presentes en las máquinas en un instante determinado. Esta autoridad central se determina mediante la función **start**() que debe colocarse al principio del **main**() de la aplicación: está encargada de sincronizar los procesos remotos que intervendrán en el desarrollo del programa, asignando los papeles de autoridad central o servidor y de esclavos a las distintas máquinas del sistema. De igual forma para terminar la ejecución global del programa se debe invocar la función **end**() al final del mismo.

Finalmente en cada una de las máquinas del sistema, se ejecutan procesos de comunicación encargados de la creación de actores por nombre -New()-, el envío y la recepción de mensajes de respuesta -manejo de las Cboxes-, el envío de mensajes de requerimiento -ejecución de métodos por nombre- y la recepción de parámetros para los mismos -manejo de las Mboxes-, consultas a las Cboxes -in()-, y especificación de comportamientos de reemplazo -become()-.

3. Conclusiones

- En LDBA la operación become es usada la mayor parte del tiempo para la sincronización sobre un objeto compartido. Sinembargo puede resultar un caos en la estructura del programa puede, ya que un actor es libre de tener un comportamiento con muy poca o ninguna relación con el comportamiento actual. El uso indiscriminado del la operación become puede ser tan dañino como el uso indiscriminado del **goto**.

- La versión actual de LDBA, tiene aún restricciones que serán mejoradas con el tiempo; el desarrollo de una interfaz que permita la entrada-salida desde cualquier método en ejecución y un manejo de errores distribuidos apropiado, son las siguientes etapas del desarrollo. En el

futuro se considerará la tolerancia a fallas y se tendrá un modelo de control completamente distribuido.

- La implantación de la concurrencia intra-actor e inter-actor, hace que el uso de la creación de procesos pesados pueda degradar la eficiencia del sistema y en ocasiones limitar el tamaño de los problemas a solucionar. Sinembargo, la creación de estos procesos pesados trae consigo la uniformidad y la transportabilidad de la implantación actual a distintas máquinas con sistema operacional UNIX. En el futuro se trabajará en una implantación eficiente que utilice **threads**.

4. Bibliografía

[1] D. Kafura, "ACT++: Building a Concurrent C++ using Actors", Journal of Object-Oriented Programming, 1990.

[2] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages", Journal of Artificial Intelligence, Junio, 1977.

[3] G. Agha, "A Model of Concurrent Computation in Distributed Systems", MIT Press, Cambridge, MA, 1986.

[4] F. J. Mendieta, "LDBA: Lenguaje Distribuído Basado en Actores", Tesis de Grado, Universidad de los Andes, SantaFe de Bogotá, 1991.

[5] B. Stroustrup, "The C++ Programming Language", Addison-Wesley, Menlo-Park, CA, 1986.

[6] A. Yonezawa, Shibayama, "Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1", Object-Oriented Concurrent Programming, MIT Press, Cambridge, MA, 1987.

[7] H. Lieberman, "Concurrent Object-Oriented Programming in ACT 1", Object-Oriented Concurrent Programming, MIT Press, Cambridge, MA, 1987.