Memoria Compartida Distribuida en el Sistema Aleph

Gilberto Díaz - Leandro León
Centro de Estudios en Microelectrónica y Sistemas Distribuidos (CEMISID)
Centro Nacional de Cálculo Paralelo (CECALCULA)
Universidad de Los Andes
Mérida - Venezuela

Resumen

La memoria compartida distribuida es una abstracción que simula, en un ambiente multicomputador, la gestión de memoria de los sistemas multiprocesadores. Los factores más significativos en el desempeño de un sistema de memoria compartida distribuida son: el modelo de consistencia, la granularidad, el tipo de implantación y el protocolo. Los sistemas implementados hasta ahora han combinado estos factores según una calidad de servicio específica: desempeño o versatilidad. Este trabajo propone un sitema que conjuga estas dos características mediante la combinación de varios tipos de granularidad con un modelo de consistencia de entrada y con un protocolo de varios lectores y varios escritores completamente distribuido. La versatilidad es lograda a través de varios niveles de interfaz de granularidad de tamaño variable. Un desempeño eficiente se obtiene mediante un modelo de consistencia de entrada y un protocolo de múltiples lectores y múltiples escritores.

1 Introducción

Un sistema de memoria compartida distribuida (Distributed Shared Memory System, DSMS) gestiona un conjunto de procesos, cada uno con su propio espacio de memoria, en el cual una porción de memoria es manipulada en "modo compartido" por todos los procesos. El sistema implanta la ilusión de que cada proceso vea la porción compartida de manera coherente; es decir, si un dato compartido es modificado por algún proceso, el sistema garantiza que los demás procesos observarán la modificación de forma consistente.

Existe una muy amplia variedad de esquemas para implantar memoria compartida. Cada esquema es una combinación de niveles de los siguientes factores:

- **Granularidad:** La granularidad concierne el tamaño de la unidad de compartimiento. En este sentido, existen dos tipos básicos de granularidad:
  - **Paginada:** En este caso, la unidad de compartimiento son páginas de memoria virtual. La implantación se vale, pues, de los mecanismos de memoria virtual del hardware y del sistema operativo.

- **A objetos:** En este caso, la unidad de compartimiento es un objeto en el sentido tradicional de la programación orientada a objetos.

- **Modelo de consistencia:** El modelo de consistencia establece las reglas que deben seguir los procesos para observar los datos compartidos de manera coherente. Una amplia gama de modelos han sido propuestos, de los cuales cabe mencionar: **estricto, secuencial, causal, débil, liberado y de entrada**. En términos generales, estos modelos varían desde los menos restrictivos en la forma de programación, hasta los modelos que se alejan completamente de la programación secuencial tradicional. El desempeño de estos modelos es inversamente proporcional; es decir, los modelos cuya programación sea más cercana a la tradicional son los que exhiben menor desempeño.

- **Protocolo:** De manera tradicional, el protocolo establece el conjunto de mensajes, los componentes distribuidos y las reglas de comunicación que implantan la interfaz del sistema y el tipo de coherencia. En este sentido, podemos enunciar dos clases de protocolo: **centralizado y distribuido**. Cualquiera de estas clases puede combinarse en tres modalidades según la cantidad de lectores y escritores:

  - Un sólo lector, un sólo escritor.
  - Varios lectores, un sólo escritor.
  - Varios lectores, varios escritores.

- **Implantación:** Bajo la suposición de que el DSMS es implantado en software, las combinaciones de los tres factores anteriores pueden realizarse de las siguientes maneras: **en biblioteca, por el compilador e híbrido**. Las implantaciones en biblioteca son las más populares. Ellas pueden implantarse fácilmente para usarse en cualquier lenguaje de programación. Empero, las bibliotecas desechan información acerca del acceso a los datos que podría proporcionar el compilador. El conocimiento del compilador es importante para optimizar el desempeño. La gran desventaja del compilador es que éste se remite a un lenguaje particular, mientras que llamadas sistema a una implantación en biblioteca permiten adaptarla a cualquier lenguaje de programación.

1.1 El problema

Desde la perspectiva del desempeño, las experiencias en prototipos de investigación favorecen a los modelos de consistencia liberada o de entrada [11, 12, 18, 16]. A la excepción del modelo de consistencia, no existe un consenso acerca cuál es la combinación idónea de los factores restantes (granularidad, protocolo e implantación).

La tolerancia a fallas, el desempeño y la versatilidad hacen indispensable un protocolo distribuido; empero, la distribución total aumenta la complejidad de la implantación. Por esta razón, muchos sistemas han transado a favor de implantaciones parcialmente centralizadas con las consecuentes pérdidas en confiabilidad y desempeño [19, 2, 16]. Por la misma razón, muchos sistemas han favorecido protocolos de varios lectores y un sólo escritor [3, 19, 13, 5] en detrimento de la más versátil: varios lectores, varios escritores.

Cuanto al tipo de implantación, la alternativa más popular es en biblioteca. Sin embargo, algunas investigaciones, notablemente Orca [6, 9, 8, 7, 5], han demostrado que vale la pena aprovechar la información aportada por el compilador.

La incidencia de la granularidad en el desempeño está dada por la naturaleza de la aplicación. Consecuentemente, sería deseable disponer de un sistema que trabaje con varias granularidades, tal como la hace Linda [3]; sin embargo, este sistema no presenta un algoritmo de implementación óptimo.

Así pues, el problema que plantea este trabajo es realizar un DSMS que, en la mayor medida posible, combine los diferentes factores de manera tal de lograr el mejor rendimiento.

1.2 La propuesta

Este trabajo se enmarca en el sistema *Aleph* [1]. La meta de este trabajo es lograr un DSMS para *Aleph* que cumpla con los siguientes objetivos:

- Construir un sistema de compartimiento versátil. *Aleph* proporciona tres niveles de compartimiento que permiten modificar la granularidad. En cada nivel los usuarios cuentan con una granularidad diferente para el manejo de la memoria. También, se cuenta con un conjunto de bibliotecas de nivel de usuario que permiten un fácil manejo de la memoria compartida. Las bibliotecas desarrollas podrán ser utilizadas con lenguaje C y C++ lo cual ofrece la posibilidad de trasladar el sistema a diversas plataformas.

- Desarrollar un protocolo que permita múltiples escritores y múltiples lectores. Esto se logra mediante la asociación de las variables compartidas con variables de sincronización.

- Lograr un modelo de consistencia que garantice una gestión coherente y eficiente de los datos compartidos.

La gestión de la memoria compartida, en el primer nivel, utilizará como unidad funcional trozos continuos de memoria. La gestión y manejo de la consistencia de la memoria compartida se lleva completamente a cabo en este primer nivel. Por lo tanto, en los otros dos niveles la manipulación de la coherencia de los datos se hace de forma transparente. En el segundo nivel se implementarán mecanismos para gestionar objetos utilizando como base las primitivas desarrolladas en el primer nivel. En el tercer nivel se considerará una extensión del lenguaje de definición de interfaces CORBA IDL [14], de forma tal que el usuario pueda construir los instrumentos necesarios para realizar las peticiones a las implementaciones de los objetos compartidos.

El modelo de coherencia que se utiliza en *Aleph* es la consistencia de ingreso. La actualización y transferencia de los datos compartidos modificados por procesos remotos se realiza sólo cuando se va a ingresar a una sección crítica.

El protocolo de compartimiento de *Aleph* es totalmente distribuido. El algoritmo utilizado posee características de múltiple escritor - múltiple lector para proporcionar la capacidad de modificar simultáneamente variables compartidas diferentes. Esto se logra mediante la asociación de las regiones de memoria compartidas con variables de sincronización.

2 Interfaz

En esta sección, presentamos los tres niveles de interfaz disponibles en *Aleph*.

2.1 Nivel 1

En este primer nivel, trozos de memoria continua de tamaño arbitrario serán considerados la unidad de compartimiento del sistema. En principio estos trozos de memoria no deben solaparse entre ellos. Un nuevo trozo de memoria de tamaño *size* puede ser apartado mediante la primitiva:

`SharedMalloc(`*size, lock*`)`

La región de memoria apartada es asociada a la variable de sincronización *lock*. Esta zona de memoria puede ser liberada con la primitiva siguiente.

`SharedFree(`*lock*`)`

Cuando un proceso desee accesar un trozo de memoria que no se encuentre de forma local, y a través de algún tipo de mecanismo se decida que ese trozo debería estar localmente presente, se puede mapear localmente los datos utilizando la función:

`SharedMap(`*void *addr, size, lock*`).`

En principio, se pretende que el primer argumento **addr* sea el apuntador a la localidad de la memoria donde comenzará el trozo de memoria cuyo tamaño será de *size* bytes. Si no se puede utilizar la memoria de esa localidad, se tomarán medidas alternativas. Por último, *lock* es la variable de sincronización (cerradura) asociada al trozo de memoria en cuestión. Esta función puede ser ejecutada más de una vez utilizando una misma variable de sincronización para permitir la asociación de varios trozos de memoria a una misma cerradura. Una forma alternativa de esta función es:

`SharedMap(`*void *addr, size, lock, port*`).`

El argumento adicional, *port*, es una estructura de datos que contiene la localización de la máquina y del proceso propietario donde está ubicado el trozo de memoria requerido.

Un conjunto de funciones específicas será utilizado para permitir que las operaciones de lectura y escritura, sobre un trozo de memoria particular, sean llevadas a cabo de forma segura. Estas funciones indican el principio y el final de una sección crítica donde se podrá tener acceso a los trozos de memoria compartidos. Estas funciones deben hacer uso de la variable de sincronización para indicar los trozos de memoria sobre los cuales se realizarán las operaciones.

Para tener acceso de lectura sobre los trozos de memoria asociados a una cerradura *locl*, se debe utilizar las siguientes primitivas para delimitar la sección crítica:

`GetReader(`*lock*`), ReleaseReader(`*lock*`)`

Cuando se desea escribir o modificar el contenido de un trozo de memoria, es necesario delimitar la sección crítica, donde se colocan las instrucciones de escritura o actualización, con las siguientes primitivas:

`GetWriter(`*lock*`), ReleaseWriter(`*lock*`)`

2.2 Nivel 2

En este nivel, toda la manipulación de los datos compartidos se realiza a través de objetos. Se utiliza todo el desarrollo del nivel 1 como la base de esta nueva capa. En principio, toda clase implementada no debe ser modificada para indicar que las instancias de esa clase son compartidas.

Para lograr compartir datos a nivel de objetos, se utiliza una plantilla que encapsula la gestión de todo lo relacionado con el compartimiento de los objetos de una clase particular. En este ejemplo se utiliza la clase **Foo**.

```
SharedObject <Foo> sharedObj(referencia_a_foo);
```

Se proponen dos formas para indicar el tipo de acceso (lectura o escritura) que realiza cada método de la clase Foo a las instancias.

1. **Interfaces IDL:** El programador debe especificar mediante interfaces IDL el tipo de acceso realizado por los métodos de la clase. Luego, el compilador IDL se encarga de construir los *stubs* con los elementos necesarios para el control del compartimiento.

2. **Nivel Usuario:** Permite al usuario tener independencia del lenguaje IDL. El programador dispondrá de mecanismos que le ayuden a especificar el tipo de las operaciones sobre los objetos que se comparten.

   La declaración o instanciación del objeto es de la forma:

   ```
   SharedObject shObj(Foo)
   ```

El parámetro **Foo** es una referencia a la clase que pertenece el objeto compartido.

La invocación de los métodos de las clases implementadas se hacen a través de un *wrapper*. Por ejemplo, un método M_1 de la clase Foo, se ejecutará utilizando los *wrappers*, **Reader** o **Writer**, según el tipo de acceso que realice M_1 al objeto.

```
shObj.reader(M1)(...  parámetros del método ...)
shObj.writer(M1)(...  parámetros del método ...)
```

La estructura de estos wrapper se muestra a continuación.

```
Template SharedObject < class T1, class T2, ... >
{
   reader ( Tr T::*X(T1, T2...){
      GetReader(lock);
         Tr retval = X(t1, t2, ... tn)
          ReleaseReader(lock);
   }

   writer ( Tr T::*X(T1, T2...){
      GetWriter(lock);
         Tr retval = X(t1, t2, ... tn)
          ReleaseWriter(lock);
   }
}
```

Los parámetros del *template* tienen la finalidad de cubrir todas las posibles formas de los métodos del objeto. Los objetos pueden estar conformados por un conjunto de elementos cuya ubicación en la memoria esté dispersa. Por lo tanto, es necesario implementar un mecanismo que permita asociar trozos de memoria no continuos a una misma variable de sincronización o cerradura. El aspecto fundamental es conseguir una forma de llamar a la función `SharedMalloc`, al momento de instanciar un objeto determinado, que realice la asociación de la cerradura a las distintas áreas de memoria que ocupará este objeto. Una solución propuesta, es capturar los eventos de asignación de memoria cuando se ejecuta el constructor del objeto y apartar la memoria utilizando la primitiva `ShareMalloc`.

La idea fundamental es la propiedad de poder llamar más de una vez a la función `SharedMalloc` utilizando una misma cerradura. De esta forma, puede ser asociada una cerradura a un objeto y así poder ser manejado apropiadamente cuando éste sea compartido.

2.3 Nivel 3

Una extensión al lenguaje CORBA IDL [14], permite especificar información acerca del compartimiento en tiempo de compilación de la interfaz. La figura 1 ilustra las extensiones en cuestión.

```
shared interface Foo
{
long bar_1(in long p1, in long p2, out long p3) reader;

void bar_2(in long p1, in long p2, out long p3) writer;
};
```

Figura 1: Extensión IDL

La palabra reservada `shared` especifica que el objeto definido por la interfaz `Foo` puede ser compartido. Esto permitiría al compilador IDL generar llamadas al nivel 2, en lugar de generar los respectivos stubs y skeletons de la interfaz. Las palabras reservadas `reader` y `writer` indican el tipo de acceso al objeto que efectuaría el método.

3 Protocolo de Compartimiento

El espacio de memoria compartida distribuida es manejado en el primer nivel utilizando regiones de memoria continua, de tamaño arbitrario, como unidad de compartimiento. Cada región tiene asociada una variable de sincronización (cerradura) que sirve para gestionar los accesos al contenido de las regiones. Como se ha mencionado anteriormente, los otros dos niveles de compartimiento se apoyan en este primer nivel.

3.1 Estado de una Región

Toda región posee una variable que almacena su estado. Los diferentes valores que puede tomar esta variable son: sólo lectura (READ), acceso exclusivo (WRITE) e inválido (INVALID). Cuando una región se encuentra en estado de sólo lectura, significa que se tiene una copia válida y el proceso que la maneja puede sólo puede realizar operaciones de lectura sobre el contenido de la región. Si el estado de la zona de memoria es de acceso exclusivo, significa que el proceso que la maneja es el propietario de ese trozo de memoria, y puede ejecutar operaciones de lectura o escritura. Por último, cuando el estado es inválido, el proceso no posee una copia actual de la región y si desea ejecutar alguna operación, entonces, debe pedir al propietario una copia válida, sea de sólo lectura o de acceso exclusivo.

3.2 Manejo del Propietario

Todo proceso que tenga una copia local de una región siempre sabe quién es el propietario de esa región. Esto se logra gracias a la difusión de esta información hacia el resto de los procesos cada vez que se cambia de dueño. Los procesos que ejecutaron algún requerimiento de propiedad sobre la zona de memoria, no son tomados en cuenta para el envío de esta difusión, debido a que estos son gestionados a través de otro mecanismo que controla la atención de tales requerimientos. Este mecanismo consiste de una lista que almacena, en orden de solicitud, todo proceso que manifieste el deseo de obtener la propiedad de la zona de memoria. El proceso que gestiona esta lista es el dueño actual de la región de memoria. El propietario agrega entradas a la lista cada vez que un proceso diferente haga una solicitud de propiedad. También, elimina el primero de la lista cuando se encuentre listo para ceder la propiedad al proceso que fue eliminado de la lista. Esta lista es enviada al nuevo propietario para que éste conozca cual es el próximo proceso (primero de la lista) que desea obtener la propiedad de la región. De esta forma, se garantiza que todo proceso selicitante obtenga la propiedad en algún momento.

Existe un caso especial donde puede generarse una confusión al tratar de pedir la propiedad de una región. Cuando un propietario va a ceder la región para acceso exclusivo a otro proceso, primero ejecuta la difusión que indica al resto de los procesos que el dueño ha cambiado, espera por los reconocimientos de ese mensaje por parte de los otros procesos, y luego hace el envío de la propiedad junto con la lista de procesos solicitantes. Puede ocurrir, que en el lapso de tiempo comprendido entre la espera de los reconocimientos y el envío de la propiedad otro proceso, que ya conozca la identidad del nuevo dueño, ejecute una solicitud de propiedad a ese nuevo dueño. El nuevo propietario no ha recibido aún la región con acceso exclusivo y debe utilizar un mecanismo adicional para manipular tales peticiones. Este mecanismo consiste de una lista local donde se va almacenando, en orden de llegada, todas las solicitudes de propiedad. Cuando este proceso obtiene al fin la propiedad de la región, la lista local es agregada al final de la lista principal, la cual debe haberse recibido junto con la propiedad.

3.3 Algoritmo de Implementación

En esta parte se presentan los detalles de las operaciones de lectura y escritura que pueden ejecutar los procesos sobre las regiones de memoria compartida.

Antes de entrar a una sección crítica, se debe indicar de forma explícita el deseo de ejecutar tal sección crítica, mediante anotaciones en el programa (*GetWriter, GetReader, ReleaseWriter, ReleaseReader*).

```
GetWriter(lock);                    GetReader(lock);

   SeccionCritica();                   SeccionCritica();

ReleaseWriter(lock);                ReleaseReader(lock);
```

Si en el transcurso de la sección crítica sólo se realizarán operaciones de lectura, entonces los accesos de adquisición (GetReader) se realizarian sin demora alguna; a menos que no exista una copia local de los datos requeridos, en cuyo caso, se envía un requerimiento solicitando al propietario una copia de la región.

Si dentro de la sección crítica alguna variable compartida sufrieran modificaciones, entonces el acceso de adquisición (GetWriter) debe ubicar el propietario para pedirle una copia de los datos actualizados con acceso exclusivo. Este es el punto donde se realiza la transmisión de las actualizaciones de las variables compartidas según la consistencia de ingreso.

3.3.1 Mensajes

Los diferentes procesos que acceden a la memoria compartida distribuida deben establecer un intercambio de información entre sí, para poder mantener la consistencia de los datos. Este intercambio se lleva a cabo a través del envío de mensajes entre los procesos involucrados. El protocolo diseñado es capaz de manejar fludo de mensajes desordenado, aunque este se apoya en un sistema de comunicación de procesos confiable que soporta pérdida de mensajes y además los entrega de forma ordenada. Es fácil modificar este diseño para delegar la gestión de errores al sistema de comunicación utilizado. A continuación se describe los diferentes tipos de mensajes que se utilizan en el protocolo.

- `Rgn_Property_Request` Este mensaje se utiliza para ejecutar una solicitud de propiedad, es decir, realizar un requerimiento de región con acceso exclusivo para poder llevar a cabo operaciones de escritura o modificación sobre el contenido de una determinada zona de memoria. El mensaje lo envía el proceso que ejecuta un *GetWriter* y lo debe recibir el dueño de la región. Dentro de este mensaje se envía el identificador del proceso solicitante.

- `Rgn_Property_Reply` Este mensaje es utilizado para transferir una región con acceso exclusivo. Lo envía el dueño de la región y lo recibe el próximo propietario. Su contenido es conformado por el identificador del expropietario, la región de memoria y la lista de procesos que esperan por la propiedad.

- `Ack_Property_Reply` El nuevo propietario envía este mensaje al expropietario para hacerle saber que recibió exitosamente la propiedad de la región. El contenido de este mensaje es sólo un bit de control.

- `Rgn_Reader_Request` Este tipo de mensaje se utiliza para realizar requerimientos de regiones con acceso de sólo lectura. El mensaje lo envía el proceso que ejecutó un *GetReader* y lo debe recibir el dueño de la región. El contenido de este mensaje está conformado por el identificador del proceso solicitante.

- `Rgn_Reader_Reply` Este mensaje se usa para el envío de una copia válida con acceso de sólo lectura a los procesos que ejecuten un *GetReader*; es decir, procesos que deseen ejecutar operaciones de lectura sobre una región de memoria determinada.

El envío, por supuesto, lo hace el propietario de la región. El contenido del mensaje consta de una copia de la región solicitada y del identificador del dueño.

- `Ack_Reader_Reply` Los procesos lectores envían un mensaje de este tipo al propietario de la región, para reconocer que han recibido la copia de la región con éxito. Este mensaje consta de un bit de control.

- `New_Owner_Not` Este mensaje se utiliza para dar a conocer que ha cambiado el dueño de una región. Lo envía el propietario actual y lo reciben todos los procesos excepto aquellos procesos que han solicitado la propiedad de la región. Este mensaje esta conformado por el identificador del nuevo propietario.

- `New_Owner_Ack` Todos los procesos que reciben una notificación de cambio de propietario deben enviar un mensaje de este tipo al dueño actual de la región con la finalidad de reconocer el cambio de propietario. Su contenido consta de un bit de control.

- `New_Owner_Property_Pending` Este mensaje se utiliza para darle a conocer al dueño actual que una solicitud de propiedad se encuentra pendiente. Si un proceso recibe una notificación de nuevo propietario y ha solicitado la propiedad de tal región con anterioridad, entonces, el proceso solicitante responde con este tipo de mensaje en lugar de enviar un mensaje New_Owner_Ack. El contenido de este mensaje es el identificador del proceso solicitante de la propiedad.

- `New_Owner_Region_Pending` Este mensaje se utiliza para darle a conocer al dueño actual que una solicitud de región con acceso de sólo lectura se encuentra pendiente. Si un proceso recibe una notificación de nuevo propietario y ha solicitado una copia de tal región para lectura con anterioridad, entonces, el proceso solicitante responde con este tipo de mensaje en lugar de enviar un mensaje New_Owner_Ack. El contenido de este mensaje es el identificador del proceso solicitante de la copia de la región de memoria.

- `Invalidation_Request` Este mensaje es utilizado por parte del propietario de una región para darle a conocer a los procesos lectores (procesos que tienen una copia de sólo lectura) que deben invalidar su copia pues el dueño desea entrar a sección crítica.

- `Invalidation_Reply` Este mensaje es utilizado por parte de los procesos lectores para enviarle un reconocimiento de invalidación al propietario de una región.

3.3.2 Procesamiento de Operaciones de Escritura

Cuando un proceso desee ejecutar una sección crítica donde el contenido de una región de memoria será modificado, este debe llamar al procedimiento *GetWriter*. Este procedimiento debe verificar el estado de la copia local. Si el estado es inválido o de sólo lectura, entonces debe realizar una petición de propiedad al dueño actual de la región y esperar que la propiedad le sea cedida. Si el propietario todavia se encuentra dentro de la sección crítica, entonces, el proceso solicitante deberá esperar hasta que el propietario finalice. Una vez que el propietario termina la ejecución de la sección crítica, hace el envío de las actualizaciones al proceso solicitante. Al recibir la propiedad, el proceso solicitante envía un reconocimiento al dueño. Cuando el proceso propietario recibe el reconocimiento del envío, cambia el estado y dueño a los nuevos valores. Por su parte, el proceso solicitante modifica los valores del estado a modo exclusivo y se coloca el como dueño de los datos. Ver figura 2.

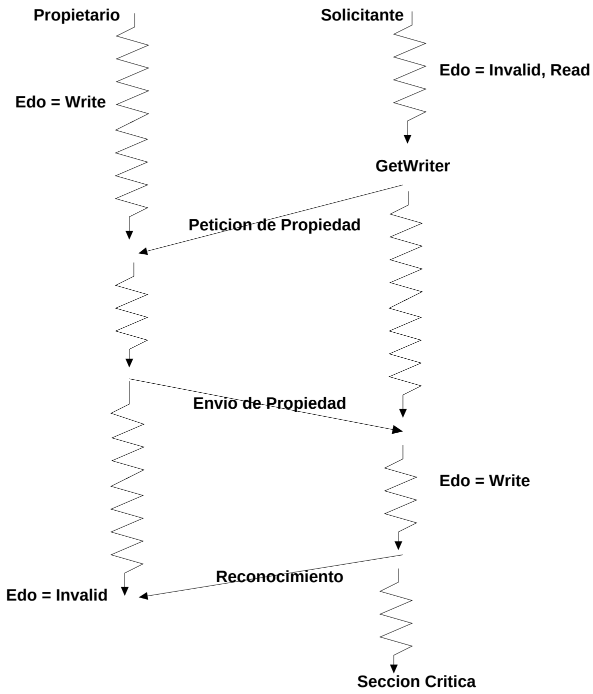

Figura 2: Solicitud de Propiedad (El edo. copia del solicitante es inválido)

Si el procedimiento *GetWriter* es ejecutado y el estado de la copia local es de acceso exclusivo, se debe enviar a todos los procesos que poseen una copia de sólo lectura una invalidación. Si no hay ningún proceso al cual enviarle invalidaciones o si se recibieron todos los reconocimientos de las invalidaciones, entonces se entra a la sección crítica. Si existen procesos con copias válidas, se debe esperar por los reconocimientos de invalidación de la región por parte de los lectores. Ver figura 3.

3.3.3 Procesamiento de Operaciones de Lectura

Si un proceso desea ejecutar una sección crítica con operaciones de lectura y el estado de la copia local es inválido, se debe hacer la petición de una copia válida al dueño de la

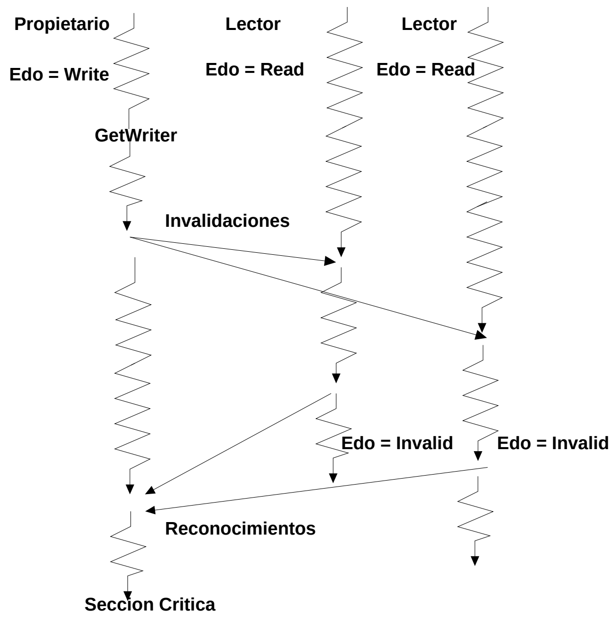

Figura 3: Entrada a sección crítica (El edo. de la copia local es de acceso exclusivo)

región, y esperar por ella. Al recibir la copia se envía un reconocimiento al dueño y se entra después a la sección crítica. Ver figura 4.

Si el estado de la región es de sólo lectura o de acceso exclusivo, entonces se entra de inmediato a la sección crítica.

3.3.4 Otras Operaciones

Sólo un proceso propietario de una región puede recibir peticiones de propiedad o peticiones de regiones para sólo lectura.

Cuando se recibe una solicitud de región para sólo lectura o de acceso exclusivo se puede presentar dos casos. En el primer caso, ningún otro proceso ha solicitado la propiedad de la región para uso exclusivo con anterioridad, entonces, se le envía la copia o la propiedad al proceso solicitante. En la figura 2, se muestra como ejemplo una petición de propiedad. En el segundo caso, algún otro proceso ya solicitó la propiedad (acceso exclusivo), entonces, se anota el proceso solicitante en una cola donde esperará hasta que el nuevo dueño de la región le envíe la solicitud requerida. Esta cola transmitida hacia el proximo propietario de la región de manera tal que tenga en cuenta todos los procesos que han realizado solicitudes de acceso a la región. El propietario actual entregará la copia para acceso exclusivo o de sólo lectura al proceso que ocupa la primera posición en la cola. Con la implementación de esta cola se garantiza que todo proceso obtendrá la región en algún momento, evitando así, la hambruna.

Otra de las posibles solicitudes que puede recibir un proceso es un requerimiento de invalidación. Si el proceso no va a hacer uso de la región, entonces, éste devuelve un

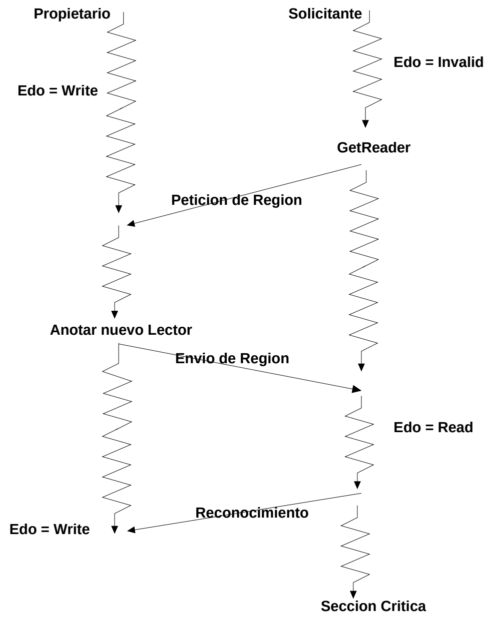

Figura 4: Petición de Región (El edo. de la copia del solicitante es inválido)

reconocimiento a la invalidación.

Cuando un proceso recibe una notificación de nuevo propietario y no se ha requerido el uso de la región, entonces se envía un reconocimiento a tal notificación. Si el proceso que recibe la notificación hizo un requerimiento de uso de la región, ya sea de uso exclusivo o de sólo lectura, entonces, este responde con un mensaje que indica su estado de espera.

4 Prototipo

El protocolo de compartimiento de *Aleph* se ha modelado con el lenguaje de especificación de sistemas concurrentes Promela [15]. Se ha utilizado, además, la herramienta spin [15] para realizar simulaciones del protocolo y recorrer todos los posibles estados en busca de errores de diseño. El protocolo fue completamente verificado a nivel sintáctico y semántico.

La implantación en *Aleph* se encuentra en fase de desarrollo. Un prototipo beta estará disponible prontamente.

5 Antecedentes

La mayoría de los DSMS implementados en software utilizan las páginas de memoria como unidad de compartimiento [4], como ejemplo, podemos mencionar a Ivy [19], Midway [12] y Treadmarks [18]. A pesar de que este método tiene varias ventajas, posee ciertos inconvenientes. Particularmente, muchos de los lenguajes de programación los datos son organizados en paquetes, módulos, estructuras diversas u objetos. Si un programa requiere un componente de un objeto, se necesitará el objeto completo. Por esta razón, es conveniente transportar los datos a través de la red como unidades de objetos en lugar de unidades de páginas. El método de variables compartidas que se utilizan en otros de los DSMS conocidos, por ejemplo Munin [10] y Midway, es un adelanto en una mejor estructuración de la memoria compartida. Si se desea un modelo de programación de alto nivel, la programación de los DSMS puede ser más sencilla y menos propensa a errores. Sistemas como CRL [17] manejan regiones arbitrarias de memoria como unidad de compartimiento. Esto proporciona versatilidad a la hora de definir los datos que se van a compartir.

Compartir objetos es otra política adoptada por algunos DSMS, los cuales proporcionan al usuario una forma más natural de intercambiar información entre sus procesos. *Aleph* ofrecerá soporte para regiones de memoria de tamaño arbitrario. Utilizando estas regiones como base se pueden obtener objetos completos como unidad de compartimiento pués todo el espacio de memoria que ocupa un objeto determinado puede ser cubierto por una región de memoria con el tamaño adecuado. Munin es un sistema DSM que se basa fundamentalmente en objetos del software, pero que puede colocar cada objeto en una página aparte ,de modo que el hardware MMU pueda utilizarse para detectar el acceso a los objetos compartidos. El objetivo de Munin es el de tomar los programas multiprocesadores existentes, realizarles cambios menores y hacer que se ejecuten de manera eficiente en los sistemas con multicomputadoras que utilicen una forma de DSM. Las modificaciones a los programas consisten en anotar las declaraciones de las variables compartidas con la palabra reservada *shared*, de modo que el compilador las reconozca.

Aleph utiliza un esquema parecido a Munin, en el sentido que se intenta minimizar las modificaciones a código existente para lograr el compartimiento de objetos. Al ejecutarse un programa en Munin, éste se inicia como proceso raíz en uno de los procesadores. Este, puede crear nuevos procesos en otros procesadores. La comunicación entre estos procesos se lleva a cabo mediante las variables compartidas, al igual que lo hacen los programas para multicomputadores normales. Una vez iniciado un proceso en un procesador particular, éste no se puede mover. Al contrario de Munin, *Aleph* permite que un proceso pueda ser trasladado a otro nodo de la red. Esto brinda la ventaja de poder balancear carga o soportar fallas de algún nodo.

Munin proporciona las herramientas para que los usuarios estructuren sus programas en torno a las regiones críticas. Las regiones críticas son definidas de manera dinámica mediante las llamadas de adquisición (entrada) y liberación (salida). Mientras un proceso está activo dentro de una región crítica, el sistema no garantiza la consistencia de las variables compartidas, pero cuando sale de ésta, las variables compartidas modificadas desde la última liberación son actualizadas en todas las máquinas. En *Aleph* se utiliza el modelo de consistencia de ingreso, el cual representa una ventaja debido a que las actualizaciones son postergadas hasta el momento en que otro proceso vaya a utilizar el contenido de una región de memoria. Además, se ahorra tráfico de información, pues las actualizaciones no son transmitidas a los procesos que no lo requieren.

Cuando se inicia un proceso en Munin, el proceso raiz posee todas las variables compartidas. Cuando el proceso P_1 ejecuta una referencia posterior a una variable compartida, ocurre una falta de página, lo cual trae como consecuencia que se envíe un mensaje hacia el proceso raiz para solicitar la página requerida. El proceso raiz proporciona la página deseada y anota que P_1 es el propietario de tal página. Si P_2 solicita la página, la raiz le responde que probablemente P_1 sea el propietario. Cuando P_2 solicita la variable a P_1, la obtiene. Si P_2 desea escribir o la página es migratoria, P_2 se convierte en el nuevo propietario y P_1 anota que P_2 es el propietario. *Aleph* asigna como propietario inicial de una región de memoria al proceso que crea la región, sin embargo, esta propiedad cambia a medida que la región sea transferida a otros procesos con uso exclusivo de la misma. En la sección 3.2 se explicó con detalle la gestión de la propiedad en *Aleph* , evidenciando claramente la superioridad de éste esquema sobre los procedimientos utilizados en Munin, particularmente, la tolerancia a fallas en caso de que uno de los nodos intermedios incluidos dentro de la cadena de búsqueda del dueño en Munin falle.

Otro de los DSMS bastante conocido es Midway. Este es un sistema de programación paralela de memoria distribuida compartida. Su objetivo principal es compartir estructuras de datos individuales. Midway utiliza un modelo de programación MIMD [1] dentro de una memoria global compartida. Este sistema utiliza el paquete de hilos C de Mach para expresar el paralelismo y el compartimiento de variables es controlado a través de candados y barreras (*locks y barriers*). Los programas en Midway son básicamente programas convencionales escritos en lenguajes C o C++ con cierta información adicional proporcionada por el programador. *Aleph* sigue de cierta manera la filosofía de Midway en el sentido de que proporciona bibliotecas para la programación en C y C++. Midway también hace uso del modelo de consistencia de ingreso igual que en *Aleph*.

La búsqueda del propietario en Midway se realiza de forma tal que cada procesador lleva una lista de registros de los propietarios probables de cada cerradura. Las solicitudes de propiedad se envían al propietario probable, quien las retransmite si la propiedad ha cambiado. Si la propiedad cambia varias veces, el requerimiento tendrá que enviarse también varias veces. Al inicio de la ejecución y después de cada n cambios de propiedad, se transmite el estado actual de los propietarios para que todos los procesadores actualicen las tablas de propietarios probables. Este esquema es bastante parecido al manejo de propietarios en Munin. *Aleph* sigue presentando ventajas en la gestión de los dueños de regiones ante estos dos DSMS.

La mayoria de los lenguajes de programación organizan los datos en objetos, paquetes, módulos u otras estructuras de datos. Por lo general, cuando un proceso acceda a una parte de un objeto, este proceso necesitará eventualmente todo el objeto. Por tal razón, tiene sentido realizar la transferencia de los datos, a través de la red, utilizando unidades de objetos y no de páginas. En el método de las variables compartidas, el programador debe proporcionar información acerca de la variables que se comparten y las que no. Además, los errores en estas anotaciones pueden generar errores de gran proporción. *Aleph* proporciona un nivel de compartimiento de objetos para brindar al programador todas las ventajas inherentes a un modelo de más alto nivel, donde la programación de los DSM puede ser más sencilla y menos propensa a errores. Los sistemas de memoria

[1] Multiple Instruction, Multiple Data

compartida distribuida basados en objetos ofrecen ventajas como:

- La modularidad es mayor que las técnicas anteriores.
- La implantación es más flexible debido al control de los accesos.
- La sincronización y el acceso se pueden integrar de manera limpia.

Una de desventaja de esta técnica es que no puede ser utilizada para ejecutar programas que no estén escritos en programación orientada a objetos como sucede con antiguos programas. Otra desventaja es que se incurre en un costo adicional al tener que realizar los accesos a los objetos a través de llamadas a sus métodos.

Una de las implementaciones de DSM basados en objetos es el sistema Orca. Este permite a los procesos de diferentes máquinas el acceso controlado a una memoria distribuida compartida consistente en objetos protegidos.

El sistema Orca consta de un lenguaje de programación, el compilador y un sistema de tiempo de ejecución, quien controla realmente los objetos compartidos durante la ejecución. Aunque todos estos componentes fueron diseñados para trabajar juntos, el sistema de tiempo de ejecución es independiente del compilador y se podría utilizar también para otros lenguajes. La ventaja de *Aleph* respecto a Orca es que el soporte en la programación esta hecha para lenguajes conocidos como lo son C y C++

Las operaciones sobre los objetos son atómicas y secuencialmente consistentes. El sistema garantiza que las operaciones son realizadas de forma secuencial cuando dos o más procesos ejecutan operaciones simultáneas sobre un mismo objeto. En *Aleph* tales protecciones se llevan a cabo mediante la utilización de las anotaciones o procedimientos *GetWriter, ReleaseWriter, GetReader* y *ReleaseReader*.

Toda la gestión sobre los objetos compartidos en Orca se realiza mediante el sistema de tiempo de ejecución. Este se encarga de controlar las múltiples copias, migración, consistencia y llamadas a métodos de objetos. Podemos encontrar los objetos en dos estados: copia única o duplicado. En el primero de los estados el objeto se encuentra en una sola máquina. Los objetos duplicados pueden estar presentes en todas las máquinas donde se está ejecutando un proceso que se encuentre utilizándolo. Los objetos pueden pasar de un estado a otro en el transcurso de la ejecución.

Otra consideración importante trata sobre la decisión de duplicar o no un objeto. Inicialmente un programa Orca consta de un sólo proceso el cual contiene a todos los objetos. Una vez que este proceso genera otro, el resto de las máquinas son informadas de este evento y reciben las copias actuales de todos los parámetros compartidos del proceso hijo. Luego, cada sistema calcula el costo esperado por duplicar o no, cada objeto. Este cálculo es realizado basándose en la proporción esperada de lecturas y escrituras. Esta proporción es obtenida por el compilador quien realiza una estimación mediante la revisión del programa. El compilador toma en cuenta que las lecturas realizadas dentro de un lazo tienen mayor peso, y las lecturas que se encuentran dentro de una decisión simple, tienen un menor peso. Los costos de comunicación también son considerados en el cálculo. En *Aleph* se solventa el problema de la decisión de duplicación mediante la transferencia de una copia válida a los procesos que la requieran. Esta es una forma mucho más sencilla de manejar este problema.

6 Conclusiones

El modelo de consistencia de ingreso utilizado en *Aleph* provee un mejor rendimiento frente a otros esquemas, pues posterga la transferencia de información sólo hasta el momento en que sea estrictamente necesario. Esto reduce la transferencia de información de actualizaciones, lo cual ahorra tiempo en el procesamiento de envío de mensajes. El número de copias de objetos o regiones de memoria es optimizado, pues sólo aquellos procesos que requieran la ejecución de operaciones sobre el contenido de una zona de memoria compartida son los que poseen copias válidas. El manejo del propietario de regiones de memoria optimiza el flujo de tráfico necesario para el mantenimiento de esta información y, además, ofrece mejor soporte a la tolerancia de fallas, pues evita la búsqueda a través de una cadena de nodos.

Los tres niveles de compartimiento que proporciona *Aleph* le da característica de versatilidad pués el programador puede elegir uno de los tres niveles para llevar a cabo el desarrollo de su implementación.

La elección de regiones de memoria de tamaño arbitrario permite una mejor gestión de toda la memoria compartida distribuida. También, sirve como base eficiente para el compartimiento de objetos, abriendo la posibilidad de tener un nivel más alto y sencillo de programación. Esto reduce la posibilidad de que los usuarios cometan errores de lógica dentro de su código.

En general, el sistema de compartimiento de *Aleph* presenta características importantes que lo hacen un DSMS, todo en software, eficiente y capaz de proporcionar rendimientos muy aceptables en su ejecución.

Referencias

[1] Aleph: Cálculo portátil de alto rendimiento sobre redes de e staciones de trabajo. Technical report, Universidad de Los Andes.

[2] V. Abrossimov, F. Armand, and M. I. Ortega. A distributed consistency server for the chorus system. In *Proc. of the Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS-III)*, pages 129–148, March 1992.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[4] Tanenbaum Andrew. *Distributed Operating Systems*. Prentice Hall, Agosto 1994.

[5] H. E. Bal and M. F. Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *Proc. of the Eighth Annual Conf. on Object-Oriented*

Programming Systems, Languages, and Applications (OOPSLA'93), pages 162–177, September 1993.

[6] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with distributed programming in orca. In *Proc. of the 1990 Int'l Conf. on Computer Languages*, pages 79–89, March 1990.

[7] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. on Software Engineering*, 18(3):190–205, March 1992.

[8] H. E. Bal and A. S. Tanenbaum. Distributed programming with shared data. *Computer Languages*, 16(2):129–146, 1991.

[9] H. E. Bal, A. S. Tanenbaum, and M. F. Kaashoek. Orca: A language for distributed programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[10] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors. Technical Report COMP TR89-91, Dept. of Computer Science, Rice University, April 1989.

[11] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, March 1990.

[12] B.Ñ. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[13] R. C. Chen and P. Dasgupta. Implementing consistency control mechanisms in the *clouds* distributed operating system. In *Proc. of the 11th Int'l Conf. on Distributed Computing Systems (ICDCS-11)*, May 1991.

[14] Object Managment Group. *The Common Object Broker: Arquitecture and Specification*, 2.3 edition, jul 1999.

[15] Gerard Holzmann. *Design and Validatiosn of Computer Protocols.* Prentice Hall, Marzo 1991.

[16] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 213–228, December 1995.

[17] Kirk L. Johnson. High-performance all-software distributed shared memory.

[18] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[19] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.