

Un *Framework* de Extensión de Algoritmos de Exclusión Mutua Distribuida para Soportar Múltiples Peticiones por Nodo

Enrique Guadalupe[†] Jorge Pérez* Federico Meza*
egudalu@puc.cl jperez@utalca.cl fmeza@utalca.cl

[†]Depto. de Ciencia de la Computación, Pontificia Universidad Católica de Chile, Santiago - CHILE

*Depto. de Ingeniería de Sistemas, Universidad de Talca, Curicó - CHILE

Resumen

El problema de definir algoritmos de exclusión mutua distribuida con soporte para múltiples peticiones por nodo ha sido abordado principalmente mediante la extensión de algoritmos de exclusión mutua distribuida existentes. En este artículo presentamos una propuesta en marcha para la elaboración de un *framework* teórico para la construcción de estos algoritmos. Cada algoritmo se construye a partir de un algoritmo de exclusión mutua distribuida y de un módulo de exclusión mutua local. Ambos componentes se integran a través de una *llave* de extensión, y mediante la definición de interfaces claras de comunicación con esta *llave*. La interacción entre los componentes es transparente, de modo que los algoritmos involucrados no dependen unos de otros. El objetivo es preservar las propiedades de correctitud del algoritmo original, de forma que el algoritmo extendido resultante también sea correcto.

Palabras clave: Algoritmos, estructuras de datos y modelos de computación, exclusión mutua, exclusión mutua distribuida, *multithreading*, sección crítica.

1. Introducción

El problema de la sección crítica se presenta cuando varios procesos intentan acceder a recursos compartidos en forma concurrente [1]. Una solución para este problema debe garantizar la exclusión mutua en el tiempo para el acceso a las secciones críticas de los procesos concurrentes. En un monoprocesador o en un multiprocesador con memoria compartida, el problema se resuelve utilizando alguna primitiva de *hardware* sobre la cual se implementan mecanismos básicos de sincronización, como los semáforos, los *locks*, e incluso otros más sofisticados como los monitores.

En un sistema distribuido, la ausencia de un estado global que pueda ser consultado por los procesos, dificulta la implementación de una solución para el problema de la sección crítica. En este caso, la solución se obtiene mediante un *algoritmo de exclusión mutua distribuida*. En la literatura pueden encontrarse muchos de estos algoritmos, algunos centralizados y algunos distribuidos [1]. En los algoritmos centralizados existe un nodo coordinador que se encarga de otorgar el acceso a la sección crítica. En los algoritmos distribuidos, los nodos son simétricos y ninguno cumple una función especial. Un algoritmo distribuido puede estar basado en paso de *token* o en permisos [2].

La mayoría de los algoritmos de exclusión mutua distribuida no soportan la generación de múltiples peticiones por acceso a la sección crítica desde un mismo nodo. Esto es bastante restrictivo, principalmente en sistemas distribuidos con soporte para *multithreading*. En estos sistemas, los procesos que se ejecutan en cada nodo pueden estar compuestos por múltiples *threads* de ejecución concurrente. Esto ayuda a una mejor estructura de los programas y permite reducir los retardos provocados por la comunicación a través de la red, al traslapar computación con comunicación [3].

Así, el estudio de algoritmos de exclusión mutua distribuida con soporte para *multithreading*, se convierte en una necesidad para poder resolver el problema de la sección crítica para los distintos *threads* que se ejecutan en los distintos nodos de un sistema distribuido.

En la literatura se encuentran pocos algoritmos con estas características. El sistema DSM-PEPE implementa un algoritmo de exclusión mutua distribuida descentralizado con soporte para *multithreading* [4, 5], que puede considerarse una extensión del algoritmo presentado por Raymond [6]. Mueller presenta otro algoritmo de características similares [7]. Su algoritmo resulta ser una extensión para soportar *multithreading* del algoritmo presentado por Naimi [8].

El sistema de memoria compartida distribuida DSM-PEPE [4, 5] soporta *multithreading* e implementa un algoritmo de exclusión mutua distribuida descentralizado para garantizar acceso exclusivo a las secciones críticas de los distintos *threads*. La interfaz de programación incluye *locks* y barreras. Pérez, *et al.* presenta un estudio de extensión de varios algoritmos de exclusión mutua distribuida basados en paso de *token*, para incorporar el soporte para *multithreading* [9]. Su estudio incluye los dos algoritmos descritos anteriormente. Así, en general, la mayoría de los trabajos relacionados se han propuesto como extensiones de algoritmos de exclusión mutua distribuida capaces de atender sólo una petición por nodo.

En este artículo presentamos una propuesta en marcha para la elaboración de un *framework* teórico para la construcción de algoritmos de exclusión mutua distribuida con soporte para múltiples peticiones por nodo. Cada algoritmo se construye a partir de un algoritmo de exclusión mutua distribuida y de un módulo de exclusión mutua local. Ambos componentes se integran a través de una *llave* de extensión, y mediante la definición de interfaces claras de comunicación con esta *llave*. El *framework* propuesto persigue definir pautas para la extensión de cualquier algoritmo de exclusión mutua distribuida. La idea es que la interacción entre los componentes sea transparente, de modo que los algoritmos involucrados no tengan consciencia entre ellos. El objetivo es conservar las propiedades de correctitud del algoritmo de exclusión mutua distribuida, de forma que el algoritmo resultante también sea correcto.

En adelante, usaremos la abreviatura *dmutex* para referirnos a exclusión mutua distribuida y *mutex* para referirnos a exclusión mutua local.

El artículo está organizado de la siguiente manera: En primer lugar, describimos generalidades acerca del *framework* propuesto, para luego presentar su formalización. Posteriormente discutimos alternativas para las interfaces y *llaves*. Finalmente, presentamos algunas conclusiones preliminares y el trabajo futuro.

2. Generalidades sobre el *Framework*

El trabajo que presentamos es un *framework* teórico de extensión de algoritmos *dmutex* para que puedan soportar múltiples peticiones por nodo. Nuestro enfoque es darle forma a un marco teórico que permita guiar la extensión de algoritmos *dmutex* existentes para incorporar el soporte de múltiples peticiones. En particular, el *framework* está orientado a sistemas con soporte para *multithreading*, ya que los requerimientos de los *threads* que solicitan acceso a la sección crítica dentro de un mismo nodo pueden ser vistos como múltiples peticiones desde el mismo nodo.

El desarrollo de algoritmos *dmutex* con soporte *multithreading* se ha basado, en su mayoría, en la extensión de algoritmos *dmutex* existentes. En general, la extensión consiste en unir un algoritmo *dmutex* que implemente la exclusión mutua distribuida entre los nodos, con un módulo *mutex* para administrar la atención de los *threads* locales una vez que la exclusión mutua ha sido otorgada al nodo. En el proceso de unión, cada uno de los mecanismos requiere ser modificado.

Un módulo *mutex* está formado por una política de atención y un algoritmo de exclusión mutua. La política de atención es, básicamente, una forma de encolar y atender requerimientos de acceso a la sección crítica que son locales a un nodo. Por su parte, el algoritmo de exclusión mutua se encarga de asegurar la exclusión mutua al requerimiento que está siendo atendido en el nodo.

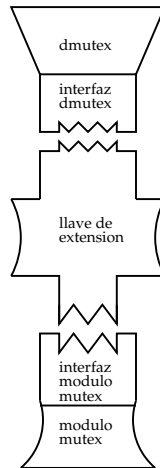


Figura 1: Esquema de Extensión

El *framework* tiene por objetivo dictar pautas que permitan unir, en forma transparente, un algoritmo *dmutex* que trabaja sobre los distintos nodos con el módulo *mutex* que opera a nivel local de un nodo atendiendo múltiples peticiones.

El algoritmo *dmutex* debe cumplir ciertas propiedades de seguridad –exclusión mutua y ausencia de *deadlock*– y de progreso –atención eventual y justicia– para que la extensión resultante sea correcta. Asimismo, el módulo *mutex* debe garantizar la exclusión mutua entre las peticiones locales al nodo en que se ejecuta.

La interacción entre el algoritmo *dmutex* y el módulo *mutex* se logra a través de un mecanismo de unión, al que llamaremos *llave de extensión* o simplemente *llave*. Asimismo, dos interfaces bien definidas permiten la comunicación entre la *llave* y el algoritmo *dmutex* y entre la *llave* y el módulo *mutex*. La Figura 1 muestra este esquema de interacción.

En el planteamiento del *framework* utilizamos los siguientes conceptos:

- **Petición:** Requerimiento de acceso a la sección crítica, generado localmente en un nodo, y que es manejado por el módulo *mutex* de ese nodo.
- **Solicitud:** Requerimiento generado por el nodo, al algoritmo *dmutex*, para obtener acceso exclusión mutua a nivel del sistema distribuido.
- **SC:** Abreviatura de sección crítica.

Es importante que las interfaces entre la *llave* y el algoritmo *dmutex*, y entre la *llave* y el módulo *mutex* sean limpias. De esta forma, la unión entre los componentes será transparente, es decir, existirá un alto grado de inconsciencia entre el algoritmo *dmutex* y el módulo *mutex*. Cada uno trabaja en forma independiente, desconociendo la existencia del otro componente y, como consecuencia, de la interacción entre ellos. Asimismo, las interfaces limpias permiten la independencia de la llave respecto a los otros componentes. Esto facilita el reemplazo de una llave por otra, con distinta funcionalidad.

La *llave* permite la interacción entre el algoritmo *dmutex* y el módulo *mutex* y determina la forma en que serán atendidas las peticiones generadas al interior del nodo. La *llave* recibe, a través de la interfaz, las peticiones que provienen del módulo *mutex*. A continuación, verifica la ausencia de una solicitud previa que se encuentre pendiente. En este caso, genera una solicitud de acceso a la SC, a través de la interfaz con el algoritmo *dmutex*. Cuando la solicitud sea atendida por el algoritmo *dmutex*, la *llave* recibe un mensaje –a través de la interfaz correspondiente– que indica que al nodo se le otorga el derecho de acceso a la SC. Luego, la *llave* ordena al módulo *mutex* que atienda peticiones, de acuerdo a algún criterio particular. La Figura 2 muestra la dinámica de interacción de *llave* con el resto de los componentes.

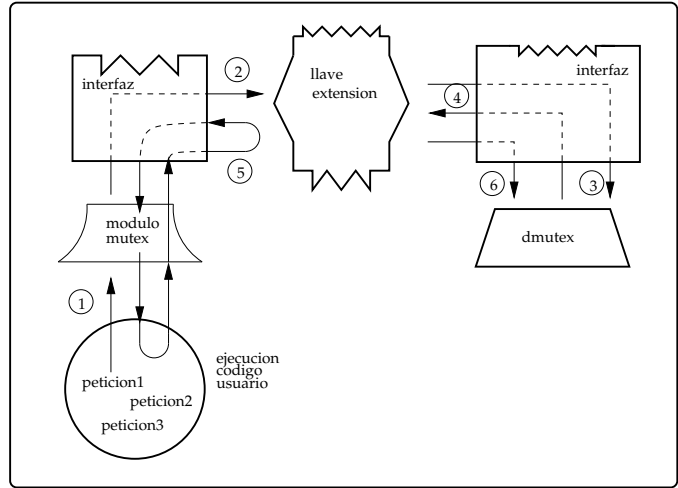


Figura 2: Tareas de la llave: 1) *Peticion a SC es generada y manejada por el módulo mutex.* 2) *Módulo mutex notifica a la llave que hay peticion a SC.* 3) *Llave notifica solicitud a algoritmo dmutex.* 4) *Algoritmo dmutex notifica a la llave que el nodo posee el derecho de acceso a la SC.* 5) *La llave lleva a cabo ciclo de atención de peticiones.* 6) *La llave notifica al algoritmo dmutex que puede renunciar al derecho de acceso a SC.*

3. Formalización del Framework

El *framework* establece que para la extensión de un algoritmo *dmutex* es necesario definir: (1) una interfaz entre el algoritmo *dmutex* y la *llave*, (2) una interfaz entre el módulo *mutex* y la *llave*, y (3) una *llave* de extensión. Este proceso puede ser independiente para cada algoritmo *dmutex* que tengamos y para distintos módulos *mutex*, con diferentes *llaves* definidas, lo que se puede observar en la Figura 3.

La definición de la interfaz entre el algoritmo *dmutex* y la *llave* involucra establecer las características más importantes del algoritmo y los eventos que lo involucran. Las características esquematizan la funcionalidad y los eventos permiten definir operaciones de la interfaz. Esto puede observarse en la Figura 3, donde las interfaces tienen diferentes formas según el algoritmo *dmutex*.

Una interfaz limpia y bien diseñada puede servir para diversos algoritmos, como se muestra en la Figura 4.

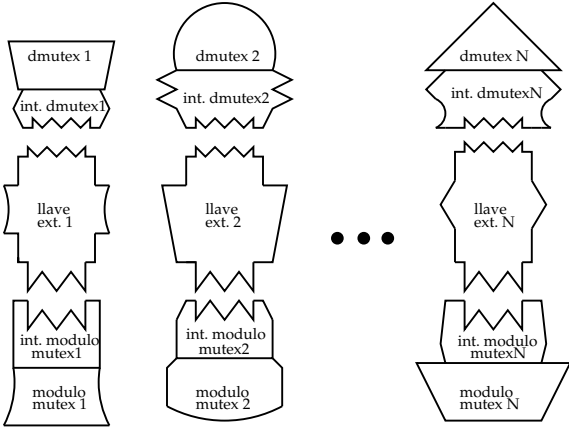


Figura 3: Aplicación del *framework*

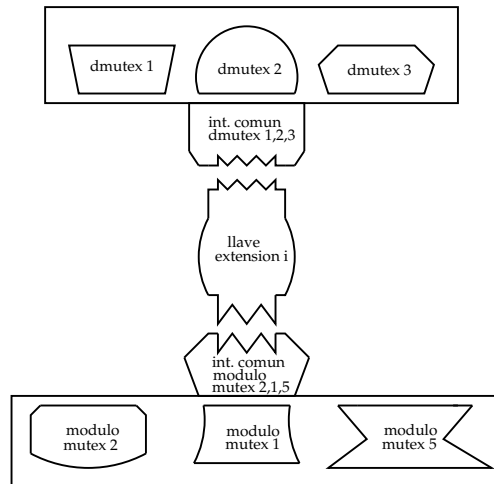


Figura 4: Potencialidades del *framework*

El mismo principio se puede aplicar para diversos módulos *mutex*. Una interfaz transparente permite en gran parte que la extensión correspondiente también lo sea, pero además, una buena interfaz puede servir de patrón de interfaces de algoritmos similares, facilitando la construcción de una única interfaz común que las agrupe. La principal tarea de la interfaz con el algoritmo *dmutex*, es comunicar y atender, junto a la *llave*, las solicitudes de acceso a la SC que un nodo genera. La solicitud es emitida por la *llave* a nombre del nodo y es enviada al algoritmo *dmutex* a través de la interfaz. El algoritmo *dmutex* otorga el derecho de acceso a la SC a la *llave* también a través de la interfaz. Una vez realizada la atención de peticiones locales del nodo, la *llave* notifica al algoritmo *dmutex* que el nodo ya accedió a la SC y la solicitud actual fue atendida, renunciando así al derecho de acceso a la SC por parte del nodo.

Además, la interfaz con el algoritmo *dmutex* debe poder entregar datos relevantes del manejo del algoritmo que pudieran ser de utilidad para la llave. Por ejemplo, los algoritmos *dmutex* basados en permisos a menudo utilizan *timestamps* para la serialización de los eventos. El algoritmo *dmutex* debería poner estos *timestamps* a disposición de la *llave* para su utilización.

La definición de la interfaz entre el módulo *mutex* y la *llave* involucra establecer un protocolo entre la *llave* y la política de atención de peticiones del módulo *mutex*. El módulo *mutex* depende para su funcionamiento de la *llave*, que le debe indicar cuándo atender la próxima petición en espera. Para esto, necesita seguir una política de atención guiada por la *llave* a través de la interfaz. Lo mismo sucede cuando el módulo *mutex* termina de atender una petición. En este caso, depende de la *llave* la decisión de atender otra petición o no.

La interfaz con el módulo *mutex* facilita a la *llave* la atención de las múltiples peticiones que surgen localmente en el nodo. Las peticiones se reciben en el módulo *mutex* quien, a través de la interfaz, le hace saber a la *llave* que existen peticiones pendientes. Eventualmente, la *llave* notifica al módulo *mutex* que el nodo posee el derecho de acceso a la SC por lo que le ordena atender la siguiente petición. También, mediante la interfaz con el módulo *mutex*, la *llave* se entera que una petición ya fue atendida.

Además, la interfaz con el módulo *mutex* debe brindar datos sobre las peticiones actuales del nodo, para uso de la *llave*. El contar con esta información permite tomar buenas decisiones sobre la forma en que las peticiones son atendidas y aprovechar características particulares del algoritmo *dmutex* o del algoritmo de exclusión mutua del módulo *mutex*.

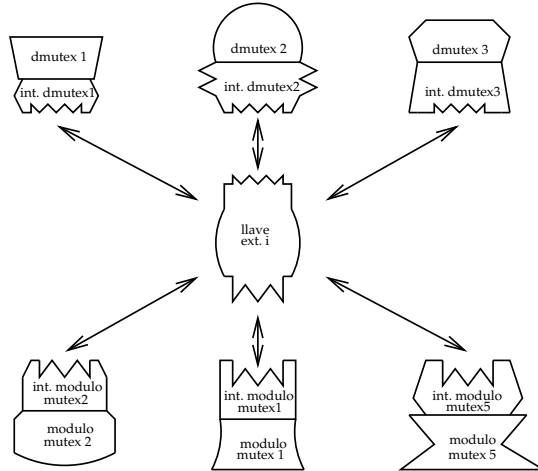


Figura 5: Definición de la *llave*

La definición de la *llave* involucra construir un mecanismo de interacción entre el algoritmo *dmutex* y el módulo *mutex*, manejar las solicitudes y traducir peticiones a través de las interfaces correspondientes, como se muestra en la Figura 5. La *llave* se encarga de obtener el derecho de acceso a la SC para, una vez obtenido, permitir que las peticiones locales sean atendidas por el módulo *mutex*.

La *llave* lleva a cabo cuatro tareas principales: (1) obtener el derecho de acceso a la SC, (2) determinar la forma en que las peticiones son atendidas, (3) atender las peticiones locales del nodo, y (4) devolver al algoritmo *dmutex* el derecho de acceso a la SC.

1. Para obtener el derecho de acceso a la SC para un nodo, la *llave* debe saber que hay peticiones pendientes. Luego emite la solicitud y espera que el algoritmo *dmutex* le otorgue el derecho de acceso a la SC. La cronología de eventos involucrados puede verse en la Tabla 1. Primero se genera una petición en un nodo, de lo cual notifica el módulo *mutex* a la *llave*. Al recibir el mensaje que notifica la existencia de una petición, la *llave* crea la solicitud de acceso a la SC y la envía, a través de la interfaz, al algoritmo *dmutex*, confirmando antes que no hay una solicitud pendiente. Finalmente, la *llave* espera asincrónicamente que el algoritmo *dmutex* otorgue el derecho de acceso a la SC.
2. La *llave* determina la forma en que las peticiones serán atendidas en el nodo. El método para determinarlo puede variar desde un criterio sencillo hasta una sofisticada heurística que aproveche características del algoritmo *dmutex* o del módulo *mutex*. Para tomar una buena decisión, la *llave* necesita saber, por ejemplo, el número de peticiones pendientes en el módulo *mutex*. Esta información debe estar disponible a través de la interfaz.
3. La *llave* atiende las peticiones locales del nodo cuando el algoritmo *dmutex* le otorga el derecho de acceso a la SC. Entonces le indica al módulo *mutex* que atienda la próxima petición pendiente. Esta secuencia puede verse en la Tabla 2. Primero, el algoritmo *dmutex* avisa mediante la interfaz a la *llave* que se ha obtenido el derecho de acceso a la SC. Luego, la *llave* ordena al módulo *mutex*, mediante la interfaz, que atienda la próxima petición. Esto lo hará tantas veces como peticiones determine la *llave* que se deben atender. Por cada atención de petición que el módulo *mutex* efectúa, le indica a la *llave* que ha terminado y procede a esperar la próxima orden de atención por parte de la *llave*.
4. La *llave* instruye al algoritmo *dmutex* para que renuncie al derecho de acceso a la sección crítica, cuando ha atendido las peticiones que consideró (observar Tabla 3). Esto debe ser consistente con la estrategia implementada por la llave para la atención de las peticiones. Luego, la *llave* debe verificar si todavía existen peticiones pendientes en el módulo *mutex* y si es así, debe volver a emitir una nueva solicitud de acceso a la SC.

Tabla 1: Acciones e interacciones al surgir una petición en un nodo

Evento Principal	Evento Dependiente	Acción DMutex	Llave	Acción Módulo Mutex
Surge petición de acceso a la SC	Llega aviso de petición pendiente a la <i>llave</i>		<p>Verifica si hay solicitud pendiente. if hay solicitud pendiente Seguir ejecución normal else - enviar solicitud a <i>dmutex</i> - escuchar por derecho de acceso a SC entregado endif</p>	Guarda petición y avisa a la <i>llave</i> que hay una petición pendiente
	Llega solicitud a <i>dmutex</i>	Ejecuta código local para que el nodo obtenga derecho de acceso a SC, y se bloquea en espera de la autorización.		

Tabla 2: Acciones e interacciones al obtener el nodo el derecho de acceso a SC

Evento Principal	Evento Dependiente	Acción DMutex	Llave	Acción Módulo Mutex
Nodo recibe de <i>Dmutex</i> autorización para el acceso a la SC	Llega a la <i>llave</i> la notificación de autorización para que el nodo acceda a la SC	Avisa a la <i>llave</i> el arribo de la autorización, y se bloquea en espera de que la <i>llave</i> termine su ciclo de atención de peticiones	<p>Establece estrategia de atención y lleva a cabo ciclo de atención de peticiones. for (cada petición que se decide atender) - enviar permiso de atención de petición a módulo <i>mutex</i> - esperar por aviso de petición servida de módulo <i>mutex</i> endfor</p>	

Tabla 3: Acciones e interacciones al atender una petición

Evento Principal	Evento Dependiente	Acción DMutex	Llave	Acción Módulo Mutex
M. <i>mutex</i> recibe aviso de servir próxima petición de la <i>llave</i>	<p>Llega aviso de servicio de petición terminado a la <i>llave</i></p> <p><i>Llave</i> termina ciclo de atención</p>		<p><i>Llave</i> ejecuta siguiente iteración del ciclo de atención de peticiones</p> <p><i>Llave</i> avisa a <i>dmutex</i> que suelte el derecho de acceso a SC. Luego chequea si todavía existen peticiones pendientes. Si las hay, envía nueva solicitud a <i>dmutex</i></p>	M. <i>mutex</i> atiende la próxima petición. Al terminar avisa a la <i>llave</i>

Toda *llave* debe cumplir una propiedad fundamental. Cuando se limita el número de peticiones locales a una, el algoritmo extendido debiera comportarse en forma similar al algoritmo *dmutex* original. Esto es necesario para poder demostrar que las extensiones mantienen las propiedades de los algoritmos originales.

Una característica importante del *framework* propuesto es que no altera las propiedades ni la forma original del algoritmo *dmutex* durante el proceso de extensión, manteniendo así un alto nivel de transparencia en la unión con el módulo *mutex*.

A pesar de que no lo hemos demostrado formalmente, intuitivamente podemos afirmar que el algoritmo extendido resultante, hereda las propiedades de correctitud del algoritmo *dmutex* original. Nótese que el algoritmo *dmutex* otorga exclusión mutua a lo más a uno de los nodos, en donde el módulo *mutex* del nodo garantiza la exclusión mutua entre las peticiones locales. Un razonamiento similar puede aplicarse a otras propiedades del algoritmo original.

4. Propuesta para interfaces y llaves

En esta sección mostramos, a modo de ejemplo, algunas propuestas de interfaces, tanto la que opera con el algoritmo *dmutex* como la que opera con el módulo *mutex*. Asimismo, presentamos algunas propuestas sencillas para la *llave* de extensión.

4.1. Propuesta para Interfaz entre Algoritmo *dmutex* y *llave*

Para implementar el acceso y la salida a la SC, los algoritmos *dmutex* cuentan con un protocolo de entrada y de salida, respectivamente. El protocolo de entrada se encarga de obtener el derecho de acceso a la SC, comunicándose para ello con otros nodos en el sistema distribuido. La forma en que el derecho es tramitado depende de los detalles específicos de cada algoritmo *dmutex*. El protocolo de salida se encarga de renunciar al derecho de acceso a la SC por parte del nodo. De esta forma, el algoritmo *dmutex* puede otorgar el derecho a otro nodo que lo esté requiriendo.

Uno de los objetivos que debe cumplir la interfaz entre el algoritmo *dmutex* y la *llave* es controlar el otorgamiento del derecho de acceso a la SC para un nodo particular. Cuando se genera una petición local al nodo, la interfaz permite accionar el protocolo de entrada a la SC del algoritmo *dmutex*. Asimismo, cuando se han terminado de atender las peticiones locales pertinentes, la interfaz permite ejecutar el protocolo de salida del algoritmo *dmutex*.

Dado que la comunicación con el algoritmo *dmutex* se lleva a cabo a través de una interfaz bien definida y siguiendo un protocolo establecido, las diferencias de implementación de los distintos algoritmos *dmutex* no es central al proceso de extensión. Por ejemplo, en el caso de los algoritmos *dmutex* descentralizados basados en *tokens*, el evento de recibir el *token* involucra recibir el derecho de acceso a la SC. En el caso de los algoritmos *dmutex* descentralizados basados en permisos, el evento equivalente es la recepción de los permisos necesarios para obtener el derecho de acceso a la SC. En ambos casos, la *llave* sólo necesita enterarse de que al nodo le ha sido entregado el derecho de acceso a la SC.

La interfaz mínima entre el algoritmo *dmutex* y la *llave* involucra dos métodos que son exportados por el algoritmo *dmutex* y un evento al que la *llave* se suscribe y cuya ocurrencia es notificada por el algoritmo. A saber:

- `DMUTEX_REQUEST()`: Método para solicitar exclusión mutua distribuida.
- `DMUTEX_RELEASE()`: Método para renunciar a la exclusión mutua distribuida.
- `ON_DMUTEX_GRANTED`: Evento que representa el otorgamiento de la exclusión mutua distribuida al nodo.

4.2. Propuesta para interfaz entre Módulo *mutex* y *llave*

La interfaz entre el módulo *mutex* y la *llave* debe permitir a la *llave* enterarse de la ocurrencia de nuevas peticiones locales al interior del nodo. Asimismo, la *llave* debe ser capaz de ordenar la atención de peticiones locales, una vez que ha obtenido la exclusión mutua distribuida para el nodo.

La interfaz mínima comprende un método que es exportado por el módulo *mutex* y dos eventos a los que la *llave* se suscribe y cuyas ocurrencias son notificadas por el módulo. A saber:

- `SERVE_LOCAL_REQUEST()`: Método para ordenar la atención de una petición local al nodo.
- `ON_NEW_LOCAL_REQUEST`: Evento que representa la llegada de una nueva petición local.
- `ON_LOCAL_REQUEST_SERVED`: Evento que representa el término de atención de una petición local.

Cuando el módulo *mutex* recibe una orden para atender petición, a través del método `SERVE_LOCAL_REQUEST()`, activa la política de atención de peticiones locales, y sólo en este caso. De esta forma se garantiza la exclusión mutua a nivel local, una vez que un nodo ha recibido la exclusión mutua distribuida.

4.3. Propuesta para *llaves*

La *llave* de extensión es un mecanismo de unión entre el algoritmo *dmutex* y el módulo *mutex*. Sin embargo, es una pieza independiente de éstos dos componentes.

Junto con establecer la unión, la *llave* determina la forma en que las peticiones son atendidas, siendo ésta la parte principal de su definición. Asimismo, la *llave* decide en qué momento dejar de ordenar al módulo *mutex* que atienda peticiones, para poder así renunciar a la exclusión mutua distribuida que hasta ese momento estaba en posesión del nodo.

Una forma para determinar cuándo dejar de atender peticiones, es que la *llave* determine de antemano el número de peticiones a servir de aquellas peticiones pendientes en el nodo. Esto puede ser calculado según un criterio sencillo o una heurística sofisticada.

Así, tres criterios generales y sencillos para determinar cuántas peticiones serán atendidas son:

- **Servir hasta N peticiones:** Atender una petición por nodo ($N=1$), obligando al nodo a pedir nuevamente el acceso a la sección crítica si posee más de una petición al momento de solicitar el acceso. Cabe mencionar que esta idea es propuesta por la mayoría de los autores como forma de extensión de algoritmos *dmutex* para soporte de *multithreading*, sin comprobar que sea eficiente.

Otra idea es atender el número de peticiones presentes al momento de obtener la exclusión mutua distribuida. Esto puede denominarse como L_{qSc} , donde qSc es la cola de peticiones pendientes al momento de obtener la exclusión mutua distribuida y L el largo. Luego, $N=L_{qSc}$.

La *llave* con $N=0$ peticiones impide que se cumpla la propiedad de progreso, provocando inanición.

- **Servir hasta que queden N peticiones:** La llave que atiende hasta que queden $N=0$ peticiones pendientes, favorece al nodo que posee la exclusión mutua distribuida, y podría disminuir la comunicación entre nodos pues favorece la localidad. Sin embargo, esto se produce a costo de eventuales problemas de inanición.
- **Servir peticiones dentro de un marco de tiempo Δt :** Se atienden peticiones mientras el marco de tiempo no se cumpla y queden peticiones pendientes. Esto asegura retardos máximos por nodo.

Estas *llaves* son relativamente sencillas y fáciles de implementar, pero no aprovechan características del algoritmo *dmutex* particular. Para aprovechar estas características, una *llave inteligente* podría suscribirse a eventos específicos del algoritmo particular, y utilizar esta información para tomar mejores decisiones.

Por ejemplo, una *llave* podría aprovechar la información relacionada con los *timestamps* que habitualmente acompañan a los requerimientos en los algoritmos *dmutex* basados en permisos. Así, sería posible combinar en un orden global las peticiones locales y las peticiones remotas de las que la *llave* se va enterando.

5. Conclusiones y trabajo futuro

Hemos presentado los adelantos de una propuesta en marcha para la elaboración de un *framework* teórico que permita extender algoritmos de exclusión mutua distribuida para soportar múltiples peticiones por nodo. El *framework* tiene especial aplicación a sistemas distribuidos con soporte para *multithreading*.

Los algoritmos que se pretende obtener a partir de las pautas dictadas, son construidos definiendo *llaves* de extensión e interfaces entre estas *llaves*, un algoritmo de exclusión mutua distribuida y un módulo de exclusión mutua local. Las extensiones facilitan una integración transparente entre los componentes y permiten preservar las propiedades de correctitud del algoritmo de exclusión mutua distribuida sobre el que se basan. Una buena definición de la *llave* y las interfaces permite la aplicación de una misma *llave* a distintos algoritmos de exclusión mutua distribuida. A modo de ejemplo, presentamos propuestas simples para *llaves* y para las interfaces.

Actualmente nos encontramos refinando la propuesta del *framework*, y pretendemos definir claramente los eventos y los métodos involucrados en la integración de una amplia gama de algoritmos, con distintas características. Pretendemos formalizar las interfaces y definir llaves que nos permitan obtener un conjunto representativo de algoritmos extendidos, y llevar a cabo estudios estadísticos de desempeño basados en simulación para estos algoritmos.

Referencias

- [1] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.
- [2] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25(2):47–50, 1991.
- [3] Kritchalach Thitikamol and Peter J. Keleher. Per-Node Multithreading and Remote Latency. *IEEE Transactions on Computers*, 47(4):414–426, 1998.
- [4] Federico Meza, Alvaro E. Campos, and Cristian Ruz. On the Design and Implementation of a Portable DSM System for Low-Cost Multicomputers. In *International Conference on Computational Science and Its Applications, ICCSA 2003*, number 2667 in Lecture Notes in Computer Science, pages 967–976, Montreal, Canada, May 2003. Springer-Verlag.
- [5] Alvaro E. Campos and Federico Meza. DSM-PEPE: Un Sistema de Memoria Compartida Distribuida para Multicomputadores de Bajo Costo. In *Anales del VIII Congreso Argentino de Ciencias de la Computación, CACIC 2002*, pages 205–216, Buenos Aires, Argentina, October 2002. Article-C163.
- [6] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1):61–77, 1989.
- [7] F. Mueller. Decentralized synchronization for multithreaded dsms, 2000.
- [8] Mohamed Naimi, Michel Trehel, and André Arnold. A log (n) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1):1–13, 1996.
- [9] Jorge Pérez. Extending Distributed Mutual Exclusion Algorithms to Support Multithreading. Master's thesis, Departamento de Ciencia de la Computación, P. Universidad Católica de Chile, July 2004.