Universidad Nacional del Sur

Tesis de Magister en Ciencias de la Computación

Coordinación y Sincronización en Aplicaciones Distribuidas

Karina M. Cenci

Bahía Blanca Argentina

2009

Prefacio

Esta tesis es presentada como parte de los requisitos para optar para el grado académico de *Magister en Ciencias de la Computación*, de la Universidad Nacional del Sur, y no ha sido presentado previamente para la obtención de otro título en esta universidad u otras. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el Departamenteo de Ciencias e Ingeniería de la Computación, durante el período comprendido entre abril de 2001 y julio de 2009, bajo la dirección del Mg. Jorge R. Ardenghi, Profesor Titular del Departamento de Ciencias e Ingeniería de la Computación.

Karina M. Cenci

UNIVERSIDAD NACIONAL DEL SUR
Secretaría General de Posgrado y Educación Continua

La presente tesis ha sido aprobada el .... / .... / ....., mereciendo la calificación de ..... (.....)

Agradecimientos

Quiero expresar mi agradecimiento especialmente a mi director de tesis, el Mg. Ing. Jorge R. Ardenghi, por la formación brindada, acompañando en el proceso de desarrollo del trabajo, dándome la oportunidad de crecer, de tomar decisiones y respetando mis tiempos. Al Departamento de Ciencias de la Computación por el lugar de trabajo, la libertad y el apoyo para el desarrollo del mismo. A mis compañeros de trabajo, en especial, a Mercedes Vitturini, con quien he compartido muchas horas de trabajo y siempre me estimuló para que culmine con mi trabajo. Y por último, a mi familia, en especial mi hermano, que me apoyó y siempre confía en mi trabajo.

Resumen

Con la propagación de los ambientes distribuidos, las aplicaciones distribuidas y las tareas cooperativas van en crecimiento y popularidad. Estas tareas requieren la utilización de recursos, en algunos casos los compartirán y en otros los utilizarán en forma excluyente, para ello se requieren de protocolos que sincronicen y coordinen el accesos a los mismos. En los sistemas uno de los problemas es el acceso exclusivo de un proceso hacia un recurso, y en los ambientes cooperativos varios procesos comparten la utilización del recurso para resolver una tarea. Para estas tareas es necesario contar con protocolos que garanticen exclusión mutua para procesos y para grupos de procesos. En el análisis del problema de la exclusión mutua para un proceso, se considera al sistema compuesto por n procesos, p_0, p_1, ..., p_{n-1}, que compiten por acceder a un recurso. Las soluciones a este problema pueden estar basadas en memoria compartida, pasaje de mensajes basados en quorum ó basados en token. En ambientes distribuidos, algunos compiten por los recursos y otros los *comparten*, cooperando en la realización de su tarea. Esta característica motivó el estudio del problema de exclusión mutua para grupos de proceso. Se considera que el sistema está formado por un conjunto de n procesos, p_0, p_1, ..., p_{n-1}; donde los procesos pueden participar de cualquiera de los diferentes m grupos, G_0, G_1, ..., G_{m-1}. Las soluciones al problema de la exclusión mutua para grupos de procesos se las clasifica de acuerdo a su diseño en: modelos basados en un actor y modelos basados en dos actores. A partir del estudio y análisis de la problemática de utilización de los recursos en forma excluyente y compartida, se propone en esta tesis un modelo general de diseño simple y claro para la exclusión mutua de grupos de procesos.

Abstract

Distributed environments, distributed applications and collaborative tasks are growing and increasing in popularity. These tasks requiere the utilization of resources, in some cases they shared and in others compete to use in a exclusive mode. So, we need protocols that synchronize and coordinate the access to resources. One of the problems in systems is the exclusive access of a process to a resource, and in cooperative environments several processes share the utilization of the resource to solve a tasks. For these tasks are required a protocol that guarantee individual mutual exclusion and group mutual exclusion. In the analysis of the mutual exclusion problem, we consider the system with n processes, p_0, p_1, ..., p_{n-1}, that compite for access to the resource. The solutions to this problem may be based on shared memory, messages based on quorum or based on token. In distributed environments, some ones compite for the resources and others share them, cooperating in solving a task. Let be a set of n processes, p_0, p_1, ..., p_{n-1}; where the processes can participate of any of the m groups, G_0, G_1, ..., G_{m-1}. The solutions for the group mutual exclusion problem can be classified in accordance with his design: models based in one actor and models based in two actors. Further the study and analysis of the problematic of utilization of the resources in exclusive and shared mode, we propose in this thesis a general model of simple and straightforward design for the group mutual exclusion.

Índice general

Índice de figuras

Índice de cuadros

Listado de Publicaciones

[1] K. Cenci, J. Ardenghi. Exclusión Mutua en la Implementación Memoria Compartida Asincrónica. *ICIE Y2K VI Congreso Internacional de Ingeniería Informática*, ISBN 987-461764-7, Facultad de Ingeniería - UBA, 26 al 28 de Abril 2000.

[2] K. Cenci, J. Ardenghi. Sobre Algoritmos Distribuidos de Exclusión Mutua para n procesos. *Congreso Argentino de Ciencias de la Computación CACIC 2000*, ISBN 950-763-033-3, pp. 973–984, UNPSJB, Ushuaia, 2-7 Octubre 2000.

[3] K. Cenci, J. Ardenghi. Exclusión Mutua para Coordinación de Sistemas Distribuidos. *VII Congreso Argentino de Ciencias de la Computación CACIC 2001*, ISBN 987-96 288-6-1, pp. 717–724, UNPA, Calafate, 16-20 Octubre 2001.

[4] K. Cenci, J. Ardenghi. Algoritmo para Coordinar Exclusión Mutua y Concurrencia de Grupos de Procesos. *VIII Congreso Argentino de Ciencias de la Computación CACIC 2002*, pp. 263–271, UBA, Buenos Aires, 15-18 Octubre 2002.

[5] K. Cenci, J. Ardenghi. Exclusión Mutua en Grupos de Procesos a través de Mensajes. *IX Congreso Argentino de Ciencias de la Computación CACIC 2003*, pp. 345–353, UNLP, La PLata, 7-10 Octubre 2003.

[6] K. Cenci, J. Ardenghi. Modelo Asincrónico Adaptativo de Exclusión para Grupos de Procesos. *XI Congreso Argentino de Ciencias de la Computación CACIC 2005*, ISBN 950-698-166-3, pp. 1116–1127, Universidad Nacional de Entre Ríos, 17-21 Octubre 2005.

[7] K. Cenci, J. Ardenghi. Exclusión Mutua para Grupos de Procesos utilizando un actor. *XIII Congreso Argentino de Ciencias de la Computación CACIC 2007*, ISBN 978-950-656-109-3, pp. 1216–1226, Universidad Nacional del Nordeste, Corrientes. 1-5 de Octubre 2007.

Capítulo 1

Introducción

Los nuevos requerimientos en la velocidad de cómputos, información altamente calificada y la confiabilidad de los sistemas requieren el desarrollo de nuevas alternativas. El primer avance para alcanzar las nuevas necesidades comienza con el desarrollo de poderosos microprocesadores accesibles en costo a una gran cantidad de usuarios. El otro elemento es el crecimiento y expansión de las redes; hacen que sea una realidad la computación distribuida (distributed computing).

La computación distribuida brinda acceso transparente a computadoras y datos como el usuario requiere para realizar cualquier tarea, y al mismo tiempo, alcanzando objetivos de alto rendimiento (velocidad) y confiabilidad. La posibilidad de trabajar en un ambiente distribuido, brinda a los usuarios una amplia gama de opciones nuevas para resolver sus problemas, como es el caso del trabajo cooperativo, paralelo y/o concurrente. Es necesario en este nuevo tipo de aplicaciones utilizar protocolos que coordinen y sincronicen la utilización de los recursos que se tienen en el sistema.

1.1. Evolución

El desarrollo de la tecnología de computación se puede caracterizar por las diferentes aproximaciones de acuerdo a cómo eran utilizadas las computadoras. (Cuadro 1.1).

Los sistemas distribuidos pueden contener diferentes arquitecturas físicas: un grupo de computadoras personales (PCs) conectadas a través de una red de comunicaciones; un conjunto de estaciones de trabajo con archivos compartidos, sistemas de base de datos y también ciclos de CPU (en la mayoría de los casos, los procesos locales tienen una prioridad mayor que los procesos remotos, donde un proceso es un programa en ejecución); un pool (equipo) de procesadores donde las terminales no están conectadas directamente a un procesador, y todos los recursos son verdaderamente compartidos sin la distinción de procesos locales y remotos.

Los sistemas distribuidos son *sin suturas*; esto es, las interconexiones entre las unidades funcionales en la red son la mayor parte invisibles para el usuario, y la idea de computación distribuida es aplicada en sistemas de base de datos, sistemas de archivos, sistemas

1950	Las computadoras eran procesadores seriales, ejecutando un trabajo a la vez hasta terminarlo.
1960	Trabajos con necesidades similares eran agrupados en lote y ejecutados en la computadora como un grupo para reducir el tiempo ocioso de la misma. Otras técnicas también fueron introducidas: procesamiento off-line utilizando buffering, spooling y multiprogramming.
1970	Surge el tiempo compartido (time-sharing). Es el primer paso hacia los Sistemas Distribuidos. Los usuarios pueden compartir recursos y acceder a los mismos en diferentes locaciones.
1980	Década de las computadoras para un único usuario, denominadas *estaciones de trabajo* con gran potencia de cómputo. Estas *estaciones de trabajo* funcionan como terminales en un sistema de tiempo compartido. En estos sistemas de tiempo compartido, la mayor parte del procesamiento del trabajo del usuario puede ser realizado en su propia computadora, permitiendo que la computadora central sea simultáneamente compartida por un gran número de usuarios. Decáda de las computadoras personales.
1990	Tecnologías de Redes y los Sistemas Distribuidos.

Cuadro 1.1: Evolución

operativos y ambientes generales.

Otra forma de expresar la misma idea es que el usuario observa el sistema como un *uniprocesador virtual*, no como una colección de diferentes procesadores.

Las principales motivaciones en alcanzar sistemas distribuidos son las siguientes:

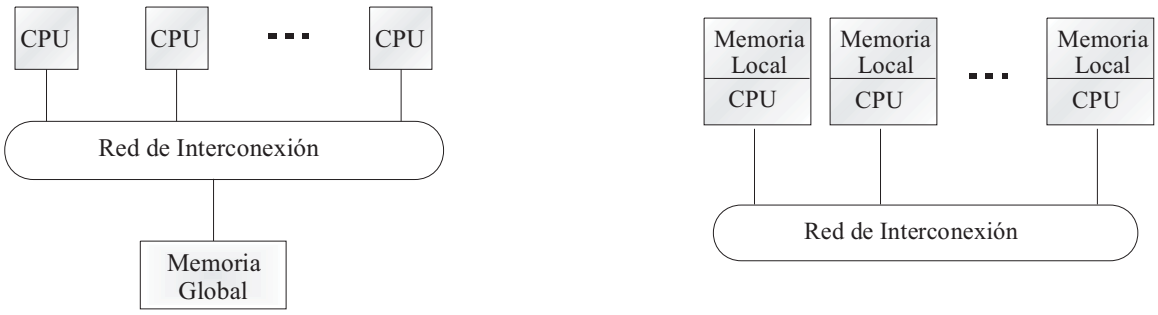
- Aplicaciones inherentemente distribuidas. Un sistema bancario, donde cada una de las sucursales mantiene cuentas, clientes y operaciones, y la mayoría de los movimientos son locales, realizando la casa matriz controles sobre cada una de las sucursales.

- Rendimiento (velocidad) / Costo. El paralelismo de los sistemas distribuidos reduce el *cuello de botella* del procesamiento y provee mejoras en general sobre el rendimiento. Tienen en potencia una proporción precio / desempeño mucho mejor que la de un sistema centralizado equivalente.

- Recursos compartidos. Permite compartir datos (a través de un base de datos común) y dispositivos.

- Extensibilidad y Escalabilidad. Es posible gradualmente extender la potencia y funcionalidad de un sistema distribuido, simplemente mediante la incorporación de recursos (tanto sea hardware como software) al sistema.

- Flexibilidad. En un sistema distribuido puede tener un pool (conjunto) de diferentes tipos de computadoras, en el cual se puede seleccionar la más apropiada para procesar el trabajo de un usuario dependiendo del tipo de tarea.



(a) Estructura de memoria compartida fisicamente

(b) Estructura de memoria distribuida fisicamente

Figura 1.1: Organizaciones de Computadoras

- Disponibilidad y tolerancias a fallas. Con la multiplicidad de unidades de almacenamiento y elementos de procesamiento, los sistemas distribuidos tienen la potencialidad de continuar operando en la presencia de fallas en el sistema (Confiabilidad).

1.2. Organizaciones

Los sistemas distribuidos constan de varias CPUs, por lo que existen diversas formas de organizar el hardware; en particular en la forma de interconectarse y comunicarse entre sí. Surgen dos organizaciones básicas de computadoras.

1. Una estructura de memoria compartida físicamente, tiene un único espacio de direcciones física compartido por todas las CPUs. Denominados comúnmente *sistemas fuertemente acoplados (tightly coupled)*. En un sistema de memoria compartido físico, la comunicación entre las CPUs se realiza a través de la memoria compartida mediante la utilización de las operaciones de lectura y escritura.

2. Estructura de memoria distribuida físicamente, no presenta memoria compartida y cada CPU tiene su propia memoria local. La unión de la CPU y la memoria local se denomina elemento de procesamiento (EP) o simplemente *procesador*. Denominados comúnmente *sistemas débilmente acoplados (loosely coupled)*. En un sistema de memoria distribuido físico, la comunicación entre los procesadores es realizada mediante el pasaje de mensajes sobre la red de comunicación a través de un comando de envío en el procesador emisor y un comando de recepción en el procesador receptor.

En la figura 1.1, la red de interconexión es utilizada para conectar diferentes unidades del sistema y soportar los movimientos de datos e instrucciones.

La elección del modelo no debería estar fijado (ligado) por el sistema físico. Se puede realizar una distinción entre el compartimento físico presentado por el hardware (Cuadro 1.2) y el compartimento lógico presentado por el modelo de programación. Es posible obtener cuatro combinaciones de compartimento.

	Lógicamente Compartidos	*Lógicamente Distribuidos*
Físicamente Compartido	Memoria Compartida	Simulación de Pasaje de Mensajes
Físicamente Distribuido	Memoria Compartida Distribuida	Pasaje de Mensajes

Cuadro 1.2: Combinaciones de Compartimento

Memoria Compartida: representa programación concurrente en un uniprocesador y programación en un multiprocesador de memoria compartida basado en el modelo de memoria compartida.

Pasaje de Mensajes: representa un sistema de memoria distribuida en el cual la comunicación es mediante pasaje de mensajes. Se utilizan comandos explícitos de envío y recepción.

Simulación de Pasaje de Mensajes: puede ser utilizado para alcanzar un modelo de programación lógicamente distribuida en una arquitectura de memoria físicamente compartida. Los mensajes son enviados a través de los buffers de memoria compartida.

Memoria Compartida Distribuida: este modelo trata de capturar la facilidad de la programación en un sistema de memoria compartida en un ambiente de memoria distribuido, haciendo que el sistema de memoria distribuido aparezca a los programadores como si fuera un sistema de memoria compartida.

1.3. Definición de Sistemas Distribuidos

Cuando se trata sobre los sistemas distribuidos se encuentra un número de diferentes tipos de identificadores mediante estos adjetivos: *distribuido, redes, paralelismo, concurrencia* y *descentralizado*. El procesamiento distribuido es un campo relativamente nuevo, y aún no se acordado una definición. En contraposición con la computación secuencial, paralelismo, concurrencia y computación distribuida requieren acciones colectivas y coordinadas de múltiples elementos de procesamiento. Estos términos se superponen en su alcance y en algunos casos se los utiliza indistintamente. Seitz [128] brinda una definición de cada uno para distinguir los diferentes significados:

- *Paralelismo* significa acciones *lockstep* (paso a paso) sobre un conjunto de datos desde un único hilo (*thread*) de control.
- *Concurrencia* significa que ciertas acciones pueden ejecutarse en cualquier orden.
- *Distribuido* significa que el costo o velocidad (rendimiento) de una computación es gobernada mediante la comunicación de datos y control.

Entre los modelos que brindan definiciones específicas de sistemas distribuidos, Enslow [55] propone que los sistemas distribuidos puedan ser examinados utilizando estas tres dimensiones: hardware, control y datos.

Sistema Distribuido = hardware distribuido + control distribuido + datos distribuidos

La definición de Enslow también requiere que la distribución de los recursos sea transparente a los usuarios. Un sistema puede clasificarse como sistema distribuido si en todas las categorías (hardware, control y datos) alcanza un cierto grado de descentralización. Algunos investigadores consideran también a las redes de computadoras y computadoras en paralelo como parte de un sistema distribuido.

Una definición de sistema distribuido es una colección de procesadores conectados en una red que no comparten memoria o un reloj. Estos sistemas proveen al usuario el acceso a diferentes recursos que el sistema mantiene. El acceder a los recursos compartidos mejora la disponibilidad y confiabilidad.

La definición de sistemas distribuidos que se considerará en el texto, es la siguiente.

> *Un sistema distribuido es uno que luce como un sistema común a los usuarios, pero se ejecuta en un conjunto de elementos de procesamiento (EPs) autónomos donde cada EP tiene un espacio separado de direcciones físicas y la demora en la transmisión de mensajes no es despreciable. El sistema debería soportar un número arbitrario de procesos y extensiones dinámicas de EPs.*

Un sistema distribuido debe proveer mecanismos de sincronización y comunicación para manejar los problemas de bloqueo y la variedad de fallas que no se encuentran en un sistema centralizado. A pesar de las dificultades que se presentan en un sistema distribuido, es importante considerar su diseño e implementación ya que hay aplicaciones que son inherentemente distribuidas; por ejemplo un sistema bancario, donde cada una de las sucursales mantiene cuentas, clientes y operaciones, y la mayoría de los movimientos son locales, realizando la casa matriz controles sobre cada una de las sucursales.

En ambientes de este tipo, existen procesos cooperativos que pueden directamente compartir un espacio de dirección lógico, o solamente compartir los datos a través de archivos. En este tipo de entorno es fundamental la *concurrencia*. Con la incorporación de la concurrencia surgen nuevas situaciones, algunas de las cuales producen problemas:

- Los procesos concurrentes (hilos) a menudo necesitan compartir datos y recursos.
- Si no existe un acceso controlado a los datos compartidos, algunos procesos podrían obtener una vista inconsistente de los mismos.
- Las acciones ejecutadas por los procesos concurrentes podrían depender del orden de entrelazado.

Se necesitan mecanismos/protocolos para asegurar un ordenamiento de la ejecución de procesos cooperativos que comparten un espacio de direccionamiento lógico, tal que la consistencia en los datos sea respetada.

El tema de sincronización de procesos es un paradigma fundamental en los sistemas de computación. Más aún en un ambiente distribuido, hay una gran variedad de estudios realizados utilizando la comunicación de procesos por pasaje de mensajes asincrónicos y de memoria compartida distribuida.

Las características para considerar bueno un protocolo es que controle el acceso de un único proceso a un recurso compartido (exclusión mutua), que no se produzca inanición, y que progrese. Además, se debe tener en cuenta la complejidad del protocolo y el costo de implementación.

1.4. Modelos para diseñar Sistemas Distribuidos

Un modelo presenta los elementos esenciales que se necesitan considerar para entender el comportamiento del sistema. El modelo de un sistema debe responder a los siguientes interrogantes:

- ¿Cuáles son las principales entidades en el sistema?
- ¿Cómo interactúan?
- ¿Cuáles son sus características, que afectan el comportamiento individual y colectivo?

El propósito de un modelo es:

- Explicitar todas las hipótesis sobre el sistema que se está modelando.
- Realizar generalizaciones (teoremas) concernientes a lo que es posible o imposible, de acuerdo a las hipótesis.

Se obtienen ventajas conociendo qué realiza o no el diseño especificado. Permite decidir si el mismo funcionará en una implementación particular.

Existen diferentes tipos de modelos para especificar un sistema distribuido:

- *Modelo Arquitectónico*: trata sobre la ubicación de los componentes y las relaciones entre ellas. Por ejemplo: el modelo cliente servidor y el modelo de procesos *"peer-to-peer"* (P2P).
- *Modelo Fundamental*: describe rigurosamente las propiedades que son comunes en todos los modelos arquitectónicos.

En un sistema distribuido no hay un tiempo global, por lo que los relojes en las diferentes computadoras no tienen necesariamente el mismo tiempo. Toda comunicación entre los procesos se realiza por medio de mensajes. La comunicación mediante mensajes sobre una red puede verse afectada por retrasos, puede sufrir variedad de fallos y es vulnerable a los ataques a la seguridad. Estos tópicos son considerados en los siguientes modelos.

- El modelo de interacción: trata de las prestaciones y de la dificultad de poner límites temporales en un sistema distribuido, por ejemplo, en la entrega de los mensajes.
- El modelo de fallos: intenta dar una especificación precisa de los fallos que pueden producirse en los procesos y en los canales de comunicación.
- El modelo de seguridad: discute sobre las posibles amenazas para los procesos y los canales de comunicación. Introduce el concepto de canal seguro, que enfrenta a dichas amenazas.

1.4.1. Modelo Fundamental

Un modelo fundamental para un sistema distribuido debe incluir:

- Procesos.
- Cómo se comunican.
- Hipótesis de Tiempo (consideraciones de tiempo).
- Cómo pueden fallar los procesos, los canales de comunicación y otros componentes.

El concepto de proceso combina actividad y protección de memoria, que son características esenciales de la computadora. El canal de comunicación es una abstracción del medio físico de comunicación.

Los procesos en un sistema distribuido se pueden caracterizar de la siguiente manera:

- Tienen su propio estado, los cuales son datos abstractos que se pueden transformar. En general, pueden también acceder y manipular objetos del mundo real como dispositivos.
- En el modelo de pasaje de mensajes, los procesos interactúan solamente enviando y recibiendo mensajes y no comparten memoria; en el modelo de memoria compartida, comparten una región específica de memoria e interactúan mediante lecturas y escrituras a las variables compartidas.
- Cuando se considera la interacción de procesos o dispositivos, es irrelevante si el proceso es internamente de un único hilo o multihilo (threaded o multi-threaded).

1.4.2. Propiedades de los Modelos

Un sistema distribuido contiene una colección de procesos p_i, $i = 1, \ldots, n$. Los modelos se pueden categorizar de acuerdo a cómo interacúan los procesos: por pasaje de mensajes y por memoria compartida.

Modelo de Pasaje de Mensajes

Cada proceso maneja dos tipos de eventos de comunicación:

$send_i(m) - p_i$ envía el mensaje m

$rcv_j(m) - p_j$ recibe el mensaje m

El mensaje es transmitido a través del canal de comunicación conectando p_i y p_j. Hay un canal (link) entre cada par de procesos que necesitan comunicarse. Cada canal puede ser unidireccional ó bidireccional. El conjunto de procesos y canales forman un grafo dirigido, con los procesos como nodos y los canales como arcos.

Modelo de Memoria Compartida

Los procesos que están conectados por una red no tienen acceso directo a la memoria física de cada uno. En estos casos se utiliza el diseño de *memoria compartida distribuida*, qué es una abstracción de un repositorio de objetos de datos que los procesos comparten y que pueden leer y/o actualizar. Intuitivamente, se puede implementar la memoria compartida distribuida utilizando un sistema que en tiempo de ejecución capture localmente los accesos a memoria compartida de cada proceso y utilice pasaje de mensajes (transparentes al proceso) para obtener los valores requeridos o actualizar otras copias de procesos locales de los datos compartidos.

La vista del programador de la memoria compartida es una colección de variables que cada proceso puede leer y escribir. Las dos operaciones disponibles son:

$R(x)u$ - es una operación de lectura de la variable x, la cual retorna el valor u.

$W(y)v$ - es una operación de escritura que inicializa el valor de la variable y a v.

Ambos modelos pueden ser utilizados, sin que ninguno prevalezca sobre el otro. Dependiendo de la situación, se puede elegir cual es más conveniente.

Procesos, estados y eventos

Un proceso p_i contiene su propio estado s_i. El estado puede consistir de variables en su espacio de direcciones. El estado de un proceso no lo puede observar otro proceso.

Cada proceso ejecuta una serie de pasos. Un paso es una operación (dependiendo del modelo puede ser una operación de send, receive, lectura ó escritura), u operaciones internas en su estado.

Se caracteriza la *historia* del proceso p_i como una serie de eventos correspondientes a estos pasos:

$\text{historia}(p_i) = h_i = e_i^1\ e_i^2\ e_i^3\ \ldots$

Los eventos relacionados con la comunicación entre procesos, son en general los analizados en los modelos distribuidos. La importancia de un paso dependerá del estado asociado al proceso.

La *historia global* del sistema se la puede caracterizar como la únion individual de la historia de los procesos:

$$H = h_1 \cup h_2 \cup \ldots \cup h_n$$

La historia global está especificada como un conjunto, sin la imposición de un ordenamiento en los eventos globales. Una corrida (run) es un ordenamiento total de los eventos en la historia global, la cual es consistente en el ordenamiento de la historia local de cada proceso.

Una linealización o *corrida consistente* es un ordenamiento de los eventos en la historia global que es consistente con la relación *ocurrió antes* (happened-before). El ordenamiento de una corrida y linealización no necesariamente corresponde con el orden cronológico en el cual los eventos ocurren en alguna ejecución del sistema. En algunas ejecuciones dos eventos pueden ocurrir simultáneamente.

La ejecución de un sistema distribuido se puede caracterizar como una serie de transacciones entre estados globales del sistema:

$$S_0, e_1, S_1, e_2, S_2, e_3, \ldots$$

En cada transición, precisamente un evento ocurre, en un único proceso del sistema. El evento puede ser de envío o recepción de un mensaje o un evento interno. Si dos eventos ocurren silmultáneamente, es posible considerar que han ocurrido en un orden definido, teniendo en cuenta el identificador del proceso (eventos que ocurren simultáneamente deben ser concurrentes). En una linealización se puede alterar el orden de los eventos concurrentes y aún se obtiene una ejecución que solamente pasa a través de estados globales consistentes.

Sistemas Sincrónicos y Asincrónicos

La ejecución de un sistema se puede definir (una linealización) solamente como una secuencia de eventos, y no se especifica cuando ocurre cada evento. Si dos procesos ejecutan exactamente la misma serie de eventos, pero uno lo ejecuta más rápido que el otro, por consiguiente se podría decir que ambos tienen la misma historia.

Si se asocia estampillas de tiempo a los eventos, las estampillas de tiempo en la historia de los dos diferentes procesos derivan de relojes cuya sincronización es solamente aproximada; en consecuencia, no siempre es posible compararlos significativamente. Esta razón motiva a formular un tipo de modelo de sistema distribuido del cual se remuevan las rectricciones de tiempo. Hadzilacos y Toug [60] definen como sistema **asincrónico** aquel en el cual no hay límites en:

- la velocidad de ejecución del proceso: un proceso puede tomarse un tiempo arbitrariamente largo entre los pasos.

- la demora en la transmisión de mensajes: un mensaje puede ser recibido en un tiempo arbitrariamente largo desde que fue enviado.
- el desplazamiento de la frecuencia del reloj.

En resumen, para asumir que un sistema es asincrónico no se realizan suposiciones sobre el intervalo de tiempo transcurrido en cualquier ejecución. Un sistema distribuido **sincrónico**, es aquel en el cual existen límites superiores e inferiores en estos casos:

- cada proceso tiene un tiempo limitado entre los pasos,
- cada mensaje es transmitido sobre un canal y es recibido en un tiempo limitado,
- el reloj del proceso local puede desplazarse del tiempo real solamente por una frecuencia limitada.

Nótese que la última restricción solamente limita la frecuencia de desplazamiento del reloj local, no la precisión. La inclusión de esta restricción permite que se utilicen *timeouts* en los protocolos, por ejemplo, para detectar la falla de otro proceso. Si los canales de comunicación son confiables, y no se recibe una respuesta desde un proceso en un tiempo máximo del reloj local, luego se puede concluir en un sistema sincrónico que ese proceso ha fallado (asumiendo, que el proceso está programado para responder al mensaje).

El modelo parcialmente sincrónico (basado en el tiempo) asume algunas restricciones en el tiempo relativo de los eventos, pero la ejecución no es completamente paso a paso (lock-step) como en el modelo sincrónico. En este tipo de modelo, los componentes brindan algún tipo de información sobre el tiempo, pero puede no ser exacta. Estos modelos son los más realistas, pero también los más difíciles de programar.

En la realidad se trabaja con sistemas parcialmente sincrónicos, pero diseñar algoritmos directamente para este tipo de sistemas es una tarea complicada. La tarea es más sencilla pensando en sistemas sincrónicos o asincrónicos. Si existe un algoritmo para el problema en un sistema asincrónico entonces un algoritmo seguramente existirá en un sistema parcialmente sincrónico.

Fallas

En un sistema distribuido pueden fallar tanto los procesos como los canales de comunicación. Algunos de los modos de falla son independientes del tiempo, y se pueden clasificar en dos tipos: fallas *por omisión* y fallas *arbitrarias*. En el cuadro 1.3, se muestran las fallas que pueden ocurrir.

Las fallas por omisión y arbitrarias pueden ocurrir tanto en sistemas sincrónicos como asincrónicos. Para el caso de los sistemas sincrónicos se consideran fallas relacionadas con el tiempo, las cuales pueden ser detectadas a través de *timeouts*.

Falla	Afecta	Descripción
Por omisión		
Failstop	Proceso	Un proceso para y se mantiene parado. Otros procesos pueden detectar el estado.
Crash	Proceso	Un proceso para y se mantiene parado. Otros procesos no están capacitados para detectar el estado.
Omisión en la Transmisión	Proceso	Un proceso genera un mensaje, pero el mensaje no es ubicado en el canal.
Omisión	Canal	Un mensaje insertado en el buffer del canal nunca arriba al otro extremo.
Omisión en la Recepción	Proceso	Un mensaje es entregado en el buffer de entrada del proceso, pero el proceso nunca lo recibe.
Arbitrario (Bizantino)	Proceso ó Canal	Un proceso/canal exhibe un comportamiento arbitrario; puede enviar/transmitir mensajes en un tiempo arbitrario, realizar omisiones; un proceso puede parar o ejecutar pasos incorrectos.

Cuadro 1.3: Clasificación de Fallas

1.4.3. Herramientas para Modelar los Sistemas Distribuidos

Las herramientas para diseñar los modelos son comúnmente utilizadas para especificar las propiedades de un sistema distribuido. El propósito de un modelo es definir precisamente las propiedades específicas o características de un sistema que se ha de construir o analizar para proveer los fundamentos y, de esa manera, poder verificar las propiedades.

Existen diferentes herramientas para especificar las propiedades de los modelos de acuerdo a las características de los mismos. Las herramientas más representativas son:

- Funciones Matemáticas. El mismo consiste en un dominio de entrada, un dominio de salida y reglas para transformar las entradas en las salidas. En general, una función matemática no es un algoritmo. Una función matemática puede ser descompuesta, esto es, especificada por una combinación de funciones lógicas y de bajo nivel. Esta característica brinda una jerarquía de funciones. La ventaja de este modelo consiste en la habilidad de organizar una gran cantidad de datos y de chequear la consistencia de las entradas y salidas entre la función de mayor nivel y sus descomposiciones. La limitación del modelo es que no almacena los datos.

- Máquinas de Estado Finito. Es un conjunto de entradas, un conjunto de salidas, un conjunto de estados, un estado inicial y un par de funciones utilizadas para especificar las salidas y la transición de estados como resultado de las entradas dadas.

- Modelo Gráfico. El modelo gráfico de computación es un grafo direccionado de vértices (nodos) y arcos (links) y es utilizado para especificar tanto el flujo de control como el flujo de datos. Cada nodo del gráfico representa un paso de procesamiento que tiene

uno o más arcos de entradas y uno o más arcos de salida. La limitación es que no incorpora el concepto de *estado*. Un caso especial son las Redes de Petri.

La utilización de alguno de estos modelos permite verificar la correctitud del algoritmo, protocolo o sistema distribuido que se esté modelando.

Autómata de E/S

El modelo de Autómata de E/S (Entrada/Salida) o I/O (Input/Output) se ha definido [97, 98] como una herramienta para modelar sistemas concurrentes y sistemas distribuidos de eventos discretos. Es un modelo formal apropiado para modelar cómputo asincrónico. Este es un modelo general, adecuado para describir la mayoría de los tipos de sistemas concurrentes asincrónicos, como son los tipos de sistema de memoria compartida asincrónica y sistemas de redes/mensajes asincrónicos. Los mismos serán utilizados en los diferentes protocolos que se analizarán en los siguientes capítulos. El modelo de autómata de E/S tiene una estructura pequeña, la cual permite ser utilizada para modelar diferentes tipos de sistemas distribuidos, como por ejemplo: sistemas de bases de datos concurrentes, algoritmos de comunicación. El modelo provee una manera precisa para describir y razonar sobre los componentes del sistema (esto es, procesos o canales de comunicación) que interactúan entre sí y que operan en diferentes velocidades.

Cada componente del sistema es modelado como un autómata de E/S, el cual es esencialmente un autómata en una acción asociada a cada transición. Una propiedad fundamental del modelo es que realiza una clara distinción entre las acciones cuya performance (ejecución) está bajo el control del autómata y aquellas acciones cuya performance (ejecución) está bajo el control de su ambiente.

Un autómata de E/S modela un componente distribuido de un sistema que puede interactuar con otros componentes del sistema. Es un tipo simple de máquina de estados finito en el cual las transiciones son asociadas con el nombre de *acciones*. Las acciones son clasificadas en: *entrada, salida ó internas*. Las acciones de entrada y salida son utilizadas para la comunicación con el ambiente del autómata, mientras que las acciones internas son visibles solamente dentro del autómata. Se asume que las acciones de entrada no se encuentran bajo el control del autómata, sólo arriban desde el exterior, mientras que el autómata especifica que acciones de salida e internas se ejecutan. La distinción entre acciones de entrada y el resto es fundamental, para poder determinar quién realiza una acción, ya que un autómata puede establecer restricciones en las acciones que él ejecuta, esto es, internas o de salida, pero no puede bloquear la ejecución de una acción de entrada.

El modelo no permite que un autómata bloquee su ambiente o elimine entradas indeseables. Si se deseara garantizar que un autómata exhiba algún comportamiento cuando el ambiente observa ciertas restricciones en la producción de entradas, entonces, en vez de permitir que el autómata bloquee las entradas erróneas, se permiten que esas entradas ocurran, pero permitir que el autómata exhiba un comportamiento arbitrario cuando ocurren.

Una característica del modelo es que puede ser no determinístico, y el no determinismo es una parte importante del poder descriptivo del modelo. Describir algortimos tan no determinísticamente como sea posible permiten obtener resultados más bien generales sobre los algoritmos, después de numerosos resultados sobre algoritmos no determinístico aplicados a fortiori a todos los algoritmos obtenidos mediante la restricción de opciones no determinísticas. La utilización ayuda a evitar descripción de algoritmos complejos (cluttering) y pruebas con detalles innecesarios.

Como se mencionó previamente, las acciones de un autómata son particionadas en conjuntos de acciones de entrada, salida e internas. El conjunto de acciones y su partición determina una interface entre el autómata y su ambiente. Referimos a esta interface como la acción de *signature* (identidad) de un autómata. Formalmente lo primero que se especifica en un autómata E/S es su *signature* (firma), que es simplemente una descripción de todas sus acciones de entrada, salida e internas. Se asume un conjunto universal de acciones.

La *signature* S es una partición del conjunto acts(S) de acciones en tres conjuntos disjuntos de acciones:

las acciones de entrada, in(S)
las acciones de salida, out(S)
las acciones internas, int(S)

También se pueden vincular las acciones teniendo en cuenta los tres conjuntos iniciales de la siguiente manera:

las acciones externas, ext(S) como in(S) $\cup$ *out(S)*
las acciones controladas localmente, local(S) como out(S) $\cup$ *int(S)*
acts(S) como todas las acciones de S

La *external signature*, *extsig(S)*, es definida como la *signature* (*in(S), out(S),* $\emptyset$).
Un autómata I/O A consiste de cinco componentes:

- *sig(A)*, una signature.
- *states(A)*, un conjunto de estados.
- *start(A)*, un subconjunto no vacío de *states(A)* conocido como estados de comienzo o estados iniciales.
- *trans(A)*, una relación de transición de estados, donde

  $$trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A);$$

  esto debe tener la propiedad que para todo estado s y para toda acción de entrada π, existe una transición (s, π, s') $\in$ *trans(A)*.
- *tasks(A)*, una partición de tareas.

1.5. Resumen

En este capítulo se presentaron conceptos generales de un sistema distribuido, características y organizaciones de hardware de los mismos. También se presentaron modelos y propiedades de diseño de los sistemas distribuidos. Un concepto importante al modelar un sistema, es el tiempo asociado con la velocidad de ejecución. Cuando no se asocian suposiciones sobre el intervalo de tiempo transcurrido en cualquier ejecución se dice que el sistema es asincrónico; de lo contrario, el sistema es sincrónico. Por último, se presentaron herramientas de modelado, profundizando en las máquinas de estado finito, ya que se utilizan para alcanzar modelos formales que facilitan la verificación de las propiedades.

Capítulo 2

Sincronización y Coordinación

2.1. Sincronización de Procesos

Un proceso cooperativo es aquel que puede afectar (influenciar) o ser afectado por otro/s proceso/s que se encuentran en ejecución en el sistema. Los procesos cooperativos pueden directamente compartir un espacio de dirección lógico (esto es, código y datos), o solamente comparte los datos a través de archivos.

El acceso concurrente a los datos compartidos puede resultar en una inconsistencia de los mismos. Se necesitan mecanismos / protocolos para asegurar un ordenamiento en la ejecución de procesos cooperativos que comparten un espacio de direccionamiento lógico, tal que la consistencia en los datos sea respetada.

Los mecanismos para asegurar el ordenamiento en la ejecución de procesos son: sincronización, exclusión mutua en las secciones críticas y asignación de recursos.

2.2. Coordinación Distribuida

El concepto de tiempo es fundamental en nuestra forma de pensar. Está derivado desde el concepto básico de orden en el cual los eventos ocurren. En los sistemas distribuidos, los procesos que están separados físicamente se comunican a través del intercambio de mensajes. En algunos casos es imposible poder determinar cual de dos eventos ocurre primero. En otros casos es necesario el ordenamiento de los eventos.

"Ocurre antes" ($\rightarrow$) es una relación transitiva entre los eventos relacionados causalmente. Más formalmente, es posible definir la relación ocurre antes ($\rightarrow$) de la siguiente manera:

Definición 2.1 *La relación "$\rightarrow$" en el conjunto de eventos de un sistema es la mínima relación que satisface estas tres condiciones:*

1. Si a y b son eventos en el mismo proceso, y a precede a b entonces $a \rightarrow b$.

2. Si a es el envío de un mensaje por un proceso y b es la recepción de dicho mensaje por otro proceso, entonces $a \rightarrow b$.

3. Si $a \rightarrow b$ y $b \rightarrow c$ entonces $a \rightarrow c$.

Dos eventos distintos a y b se dicen concurrentes si a "no ocurre antes que" b y b "no ocurre antes que" a. Es decir, dos eventos a y b son concurrentes si ninguno de ellos puede afectar causalmente al otro. Se asume que a "no ocurre antes que" a. Esto implica que $\rightarrow$ es un ordenamiento parcial y reflexivo en el conjunto de todos los eventos del sistema.

2.2.1. Control de Concurrencia

Es función del manejador de transacciones de un sistema de base de datos distribuida manejar la ejecución de esas transacciones (o subtransacciones) que acceden a datos almacenados en un sitio local. Cada manejador de transacciones es responsable de mantener una bitácora para propósitos de recuperación, y para participar en un apropiado esquema de control de concurrencia con el fin de coordinar la ejecución concurrente de las transacciones que se están ejecutando en un nodo.

2.2.2. Exclusión Mutua

El estudio de la exclusión mutua es el problema de manejar el acceso a un único e indivisible recurso (como por ejemplo una impresora) que solamente puede soportar a un usuario a la vez, entre n usuarios $\mathrm{U}_1 \ldots \mathrm{U}_n$, o los conflictos resultantes de varios procesos concurrentes compartiendo recursos. Alternativamente, se puede a pensar éste como el problema de asegurar que ciertas secciones de código de programa sean ejecutadas en forma estrictamente exclusiva (sección crítica).

Un usuario con acceso al recurso es modelado estando en la sección crítica, la cual es simplemente un subconjunto de sus estados posibles. Cuando un usuario no está involucrado de ninguna manera con el recurso, se dice que está en la sección *resto*. Para obtener la admisión a la sección crítica, un usuario ejecuta un protocolo de entrada (*trying*), y después que utiliza el recurso, se ejecuta un protocolo de salida (*exit*). Este procedimiento puede repetirse, de modo que cada usuario sigue un ciclo, desplazándose desde la sección resto (R), a la sección de entrada (T), luego a la sección crítica (C) y por último a la sección de salida (E), y luego vuelve a comenzar el ciclo en la sección resto. En la figura 2.1 se puede observar el ciclo de la exclusión mutua.

Características de los Algoritmos

Los algoritmos de exclusión mutua basados en el modelo de memoria compartida se pueden modelar a través de un autómata de E/S. El sistema de memoria compartida contiene n procesos, numerados de $1, \ldots, n$, cada uno correspondiendo a un usuario U_i.

Las entradas al proceso i son:

- La acción try_i, que modela un requerimiento del usuario U_i para acceder al recurso.

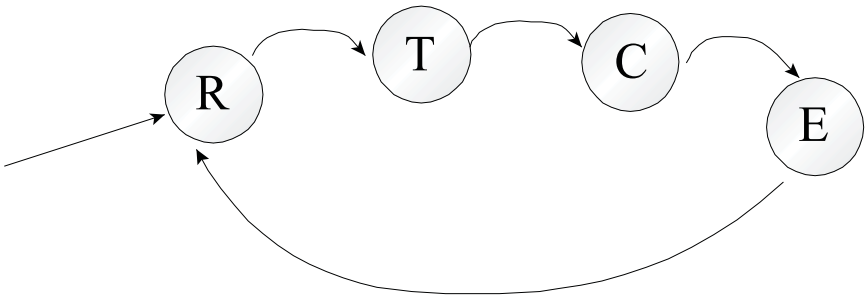


Figura 2.1: Estados en el Ciclo de Exclusión Mutua

- La acción exit_i, la cual modela un anuncio del usuario U_i que ya utilizó el recurso.

Las salidas del proceso i son:

- La acción crit_i, que modela la autorización de acceso al recurso por parte de U_i.
- La acción rem_i, la cual avisa a U_i que puede continuar con el resto del trabajo.

Las acciones *try, crit, exit y rem* son las únicas acciones externas en un sistema de memoria compartida. Cada proceso U_i, $1 \leq i \leq n$, es modelado como una máquina de estado (formalmente un autómata de E/S) que se comunica con el proceso utilizando las acciones externas.

Cada usuario U_i puede estar ejecutando cualquier programa de aplicación. Lo único que se asume sobre U_i es que obedece el ciclo de la región del protocolo, esto es, mantiene el orden del ciclo de las acciones try_i, $crit_i$, $exit_i$, ... (comenzando con try_i) entre el mismo y el proceso.

Formalmente, una secuencia de acciones try_i, $crit_i$, $exit_i$ y rem_i está bien formada para el usuario i si es un prefijo de la secuencia cíclica ordenada try_i, $crit_i$, $exit_i$, rem_i, try_i, ... Se requiere que U_i preserve la propiedad de traza definida por el conjunto de secuencias que son bien formadas para el usuario i.

Condiciones de un Algoritmo Correcto

Para determinada colección de usuarios y para un sistema de memoria compartida A, resolver el problema de exclusión mutua significa satisfacer las siguientes condiciones:

- Buena Formación: en cualquier ejecución, para cualquier i, la subsecuencia que describe la interacción entre U_i y A está bien formada para i.
- Exclusión Mutua: no se alcanza un estado del sistema (una combinación de un estado del autómata de A y estados para todos los U_i) en el cual más de un usuario se encuentra en la sección crítica C.

- Progreso: en cualquier punto de una ejecución imparcial:
  1. Progreso para la sección de entrada: si al menos un usuario está en T y ningún usuario en C, en un punto posterior en el tiempo algún usuario entra a C.
  2. Progreso para la sección de salida: si al menos un usuario está en E, en un punto posterior en el tiempo algún usuario entra a R.

Con la condición de progreso se asume que la ejecución del sistema es imparcial, y se considera que todos los procesos (y los usuarios) continúan ejecutando pasos. Si no se asume esta condición, entonces no sería razonable requerir que las acciones de salida (*crit* o *rem*) eventualmente se ejecuten (se realicen). En contrapartida, no es necesario asumir imparcialidad para requerir que el sistema garantice buena formación o exclusión mutua. La diferencia es que las condiciones de buena formación y exclusión mutua son propiedades de seguridad, mientras que la condición de progreso es una propiedad de *vivacidad*.

Un algoritmo regular/bueno de exclusión mutua debería satisfacer las siguientes condiciones:

1. Libre de Interbloqueo: cuando la sección crítica (región crítica) está disponible, los procesos no deben esperar indefinidamente, alguno de estos puede entrar.
2. Libre de Inanición: cada requerimiento a la sección crítica debería ser eventualmente garantizado.
3. Imparcialidad: los requerimientos serán otorgados basados en ciertas reglas de imparcialidad. Típicamente, está basado en los requerimientos de tiempo determinados por relojes lógicos. Inanición e imparcialidad están relacionados con lo que se denomina una condición fuerte (prioridad).

En la realidad, inanición e imparcialidad pueden no ser críticas, en muchas situaciones en que se utiliza exclusión mutua, la contención entre los usuarios es poco frecuente y un usuario puede permitirse esperar hasta que todos los usuarios en conflicto obtengan un turno. La importancia de estas condiciones depende del grado de contención del recurso, como así también de la criticidad individual del programa usuario.

Tipos de Algoritmos de Exclusión Mutua

Los algoritmos de exclusión mutua se pueden clasificar de acuerdo al modelo sobre el cual van a estar implementados, esto es, basados en memoria compartida o en pasaje de mensajes. Los algoritmos de exclusión mutua basados en el modelo de pasaje de mensajes se los puede clasificar en: *token* ó *permiso*.

Para los algoritmos que se basen en el modelo de memoria compartida existen diferentes tipos de algoritmos de acuerdo a las propiedades que presentan: algoritmos de exclusión mutua rápidos, algoritmos *adaptive* (adaptivos), algoritmos de exclusión mutua basados en el tiempo, algoritmos *nonatomic*.

En los algoritmos de exclusión mutua *rápidos*, existe una gran diferencia en la complejidad de tiempo entre casos que están libres de contención y en los que se presenta contención. Esto significa que la sección crítica está disponible, el recurso no está siendo utilizado, y la complejidad de tiempo será constante; en los otros casos, la complejidad podrá o no tener puntos de contención.

En los algoritmos de exclusión mutua *adaptive*, el incremento en la complejidad de tiempo en la contención es más gradual, está en función del número de procesos en contención. En un algoritmo adaptivo la complejidad del paso de una operación depende solamente del número de procesos que realmente ejecutan el paso concurrentemente con esa operación, esto es, la complejidad de una operación está en función de la contención real que encuentra y no del número total de procesos. Las propiedades consideradas para la implementación de un algoritmo adaptivo están relacionadas con el nivel de contención y de adaptabilidad.

Los niveles de adaptabilidad tienen en cuenta cuando un proceso es considerado activo para la evaluación de contención. Estos pueden ser por contenido y completa. En una implementación completa, los procesos son solamente considerados activos durante la ejecución de las operaciones. En cambio en una implementación por contenido un proceso también será considerado activo en algún estado del sistema, aún si no está en el medio de alguna operación.

Las nociones de contención para el caso de los algoritmos adaptivos que se han considerado en la literatura son las siguientes:

- intervalo de contención,
- punto de contención,
- contención total.

Estas nociones están definidas con respecto a la historia H. El *intervalo de contención* sobre H es el número de procesos que están activos en H, *i.e*, que se ejecuta fuera de su sección resto (también denominada sección *no-crítica*) en H. El *punto de contención* sobre H es el número máximo de procesos que están activos en el mismo estado en contención. La *contención total* es el número total de procesos que participan de la ejecución.

En algunos sistemas, es beneficioso considerar como información la disponibilidad de tiempo para diseñar algoritmos de exclusión mutua mucho más eficientes. Estos algoritmos de exclusión mutua se los denomina *basados en tiempo* y sirven para mantener un grado de sincronización. Se utilizan en modelos parcialmente sincrónicos. No necesariamente todos los algoritmos de exclusión mutua que utilizan el modelo de memoria compartida pertenecen a las categorías mencionadas.

Otra propiedad que se puede aplicar en los algoritmos de exclusión mutua es la *self-stabilization*. Dijkstra [54] introduce el concepto de *self-stabilization* en el contexto de sistemas distribuidos. Un sistema es *self-stabilization* (auto-estabilización) con respecto a un conjunto de estados legítimos, si comenzando desde cualquier estado inicial arbitrario, el sistema garantiza arribar a un estado legítimo y mantenerse en esos estados. Esto es, un sistema self-stabilization no necesita ser inicializado y puede recuperarse de fallas perecederas (*transient*).

Gouda y Multani [59] demuestran que un algoritmo self-stabilization en un modelo de pasaje de mensajes debe tener *timeouts* y también que debe tener un número infinito de estados (propiedad de *liveness*).

Medidas en la Complejidad de un Algoritmo

Las consideraciones para las medidas en la complejidad de un algoritmo pueden ser varias:

- Complejidad en un paso remoto (referencias de memoria remota) de un algoritmo es el número máximo de operaciones de memoria compartida requeridas por un proceso para ingresar y salir de su sección crítica, asumiendo que cada sentencia *await* es contabilizada como una operación indivisible.
- El tiempo de espera de un proceso es el intervalo de tiempo entre la solicitud para ingresar a la sección crítica y el acceso efectivo a la misma.
- El tiempo de respuesta del sistema es el intervalo de tiempo entre entradas a la sección crítica.
- La cantidad de tráfico de interconexión que el algoritmo genera.
- Demora en la sincronización es el tiempo transcurrido entre el momento en que un proceso libera la sección crítica y el momento en que otro proceso pueda ingresar.

Dependiendo de la situación en la cuál se aplique un algoritmo de exclusión mutua, por ejemplo se puede seleccionar un algoritmo rápido, o sea que obtenga la menor cantidad de operaciones remotas en un caso óptimo, o un algoritmo que presente una cota inferior ó superior en la cantidad máxima de operaciones remotas (cantidad de mensajes).

Extensiones de Exclusión Mutua

El problema de la exclusión mutua tradicional se lo puede considerar como el caso 1-exclusión, en el cual un sólo proceso puede acceder a la sección crítica en un determinado instante de tiempo. El concepto de exclusión mutua se puede utilizar para sincronizar tareas en las cuales varios procesos pueden compartir un recurso, o un espacio. Por este motivo se comienza con el estudio de extensiones del concepto de exclusión mutua a κ-exclusión y exclusión mutua para grupos.

κ-exclusión

El problema de la κ-exclusión fue impulsado por Fischer, como una generalización al problema de exclusión mutua en el cual hasta κ procesos pueden estar al mismo tiempo en la sección crítica.

La propiedad de libre inanición es modificada para garantizar progreso en la fase incremento hasta $k-1$ ("face up to $k-1$") fallas indetectables de parada de procesos. Esto significa que hasta $k-1$ procesos fallan por parada (*halting*), y que cualquier proceso que no falla en su sección de entrada eventualmente ingresa en la sección crítica. En el caso que un proceso se para no inicializa ninguna de sus variables a los valores por defecto.

En la κ-exclusión mutua se puede considerar que existen hasta κ recursos que se utilizan en el mismo instante y cada proceso requiere una unidad de éstos. Una generalización de este problema, donde haya κ unidades de recursos compartidos y cada proceso requiere h unidades ($1 \leq h \leq \kappa$) al mismo tiempo, es denominado h-requerimientos de-κ.

Exclusión Mutua para Grupos

Es otra generalización del problema de la exclusión que permite que múltiples procesos ejecuten su sección crítica simultáneamente. En este problema, los procesos se asocian para ejecutarse con una o más *sesiones*. Múltiples procesos pueden simultáneamente ejecutarse dentro de la misma sesión, pero diferentes sesiones no pueden estar activas en el mismo instante de tiempo.

2.2.3. Herramientas de Modelado

En esta sección, se muestran un modelo de memoria compartida asincrónico y otro de mensaje asincrónico, extraído de [95]. El modelo asincrónico asume que los componentes separados toman/ejecutan sus pasos en un orden, y a una velocidad arbitrarios. Los algoritmos diseñados con modelos asincrónicos son generales y portables, garantizando la ejecución correcta sobre redes con garantía arbitraria de tiempo.

Modelo de memoria compartida asincrónica

El sistema es modelado como una colección de procesos y variables compartidas con interacciones, como se observa en la figura 2.2. Cada proceso i es una clase de máquina de estado finito de entrada/salida, con un conjunto de estado$_i$ y un subconjunto de estados de inicio$_i$ indicando el estado inicial. El proceso i también tiene acciones etiquetadas, que describen las actividades en las cuales participa. Estas son clasificadas en acciones de entrada, salida o internas. Los componentes se comunican entre sí a través de las acciones de entrada y salida.

Las flechas que entran y salen de los procesos (círculos) representan las acciones de entrada y salida de los diferentes procesos. Dentro del conjunto de acciones internas se las puede dividir en dos grupos: las acciones que acceden a variables compartidas, y aquellas que involucran estrictamente computación local. Si se accede a variables compartidas, se asume que involucra solamente a una variable compartida.

En el modelo hay una relación de transición *trans* para el sistema entero, la cual es un conjunto de ternas (s,π, s'), donde s y s' son estados de autómata, esto es, combinaciones de estado para todos los procesos y valores para todas las variables compartidas, y donde π es

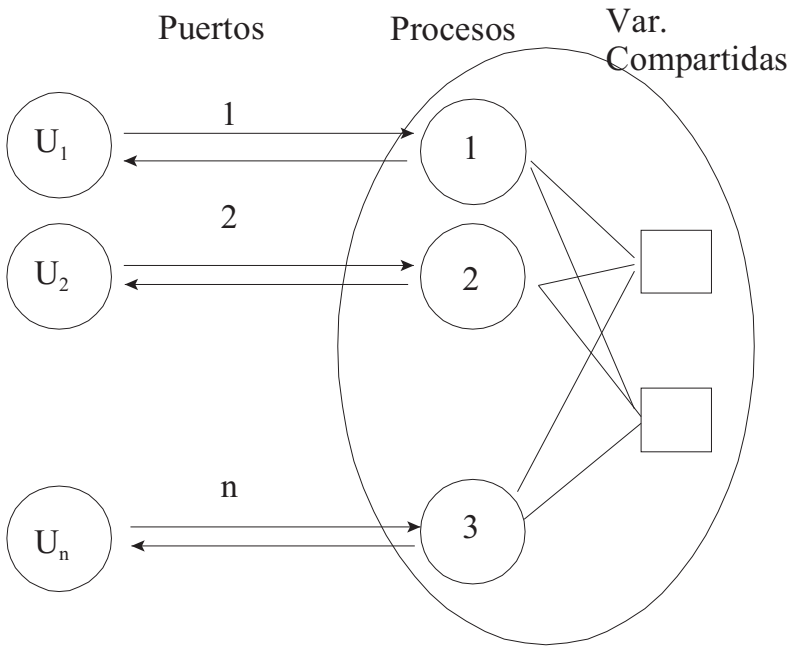


Figura 2.2: Modelo de memoria compartida asincrónica

la etiqueta de una acción de entrada, salida ó interna. La relación que (s, π, s')$\in$ *trans* indica que desde el estado de autómata s es posible ir al estado de autómata s' como resultado de ejecutar la acción π. Se asume que las acciones de entrada arriban desde el exterior, o sea, el sistema es de entrada permitida (*input-enabled*). Esto significa, que para cada estado autómata s y una acción de entrada π, existe s' tal que (s, π, s')$\in$*trans*. En contraposición, los pasos internos y de salida van a estar habilitados solamente en un subconjunto de los estados.

En un sistema de memoria compartida asincrónica se asume que los procesos se ejecutan en un orden arbitrario. Una ejecución es formalizada como una secuencia alternativa so, π1, s1, ..., la cual consiste de estados de autómata alternados con acciones (cada acción corresponde a un proceso en particular), donde sucesivas ternas (estado, acción, estado) satisfacen la relación de transición.

Ejemplo : Sistema de Memoria Compartida

Sea V un conjunto de valores fijos. Consideremos un sistema de memoria compartida A consistente de n procesos, numerados de $1, \ldots, n$ y una única variable compartida x con valores en V $\cup$ {unkown}, inicialmente en unkown.

Signature

Entrada :
init(v)_i, donde v ∈ V
Salida :
$\text{decide(v)}_\text{i}$
Interna :
access_i

Estados

status ∈ {idle, access, decide, done}, inicialmente idle
input ∈ V ∪ {unknown}, inicialmente unknown
output ∈ V ∪ {unknown}, inicialmente unknown

Transiciones

init(v)$_i$
efecto:
input := v
if status = idle then
status = access

access$_i$
precondición:
status = access

efecto:
if x = unknown then
output := x
status := decide

decide(v)$_i$
precondición:
status = decide
output = v

efecto:
status := done

No es difícil de observar que cada ejecución imparcial α de A, y cualquier proceso que recibe como entrada un *init* eventualmente realiza una salida *decide*. Más aún, toda ejecución (imparcial o no, y con indistintos números de eventos *init* ocurriendo en diferentes lugares) satisfece la propiedad de concordancia (*agreement*), no existen dos procesos que deciden en diferentes valores, y la propiedad de validación que todo valor de decisión es el valor inicial de algún proceso.

Modelo de Red Asincrónico

Una red asincrónica consiste de una colección de procesos comunicados a través de un subsistema de comunicación. En el modelo la comunicación puede ser *punto a punto*, utilizando las acciones de *send* y *receive*. Otra versión del modelo permite acciones *broadcast*, a través de esta un proceso puede enviar un mensaje a todos los procesos en la red. Un caso especial en la comunicación es el *multicast*, una combinación de punto a punto y broadcast.

Sistemas Send/Receive

Para definir formalmente un sistema de red asincrónico, se comienza un grafo dirigido G = (V,E). Donde n representa |V|, esto es, el número de nodos en la red del grafo dirigido. Para cada nodo i de G, se utiliza la siguiente notación:

- *out-nbrs*$_i$ representa los *vecinos de salida* de i, esto es, aquellos nodos para los cuales hay conectores desde i en el grafo dirigido G.
- *in-nbrs*$_i$ representa a los *vecinos de entrada* de i, aquellos nodos desde los cuales hay conectores a i en G.
- *distance(i,j)* representa la longitud del camino directo más corto desde i hasta j en G, si existe alguno; sino distance$(i,j) = \infty$.
- *diam* (diámetro) es la máxima distance(i,j) entre un nodo y otro.

Se supone que se tiene algún alfabeto fijo de mensaje M, donde se considera *null* como ausencia de mensaje. Cada *proceso* se asocia con los nodos de G y se habilita la comunicación entre ellos a través de *canales* asociados con conectores dirigidos. Los *procesos* y *canales* se modelan como autómatas de E/S.

Procesos

Los procesos asociados con cada nodo i son modelados como un autómata de E/S P_i. P_i usualmente tiene acciones de entrada y salida mediante las cuales se comunica con usuarios externos; esto permite expresar problemas a ser resueltos a través de redes asincrónicas en términos de traces (trazas) en la interfaz del usuario. P_i tiene salidas de la forma *send(m)*$_{ij}$, donde j es un vecino de salida i y m es el mensaje; y las entradas de la forma *receive(m)*$_{ji}$, donde j es un vecino de entrada (llegada) de i.

Canales Send/Receive

El canal asociado con cada conector dirigido (i, j) de G es modelado como un autómata de E/S C_{ij}. Su interface externa consiste de entradas de la forma *send(m)*$_{ij}$ y salidas de la forma *receive(m)*$_{ij}$, donde $m \in M$. En general, excepto por esta restricción de la especificación de la interface externa, el canal puede ser un autómata de E/S arbitrario.

Canales Confiables FIFO

El comportamiento permitido para este tipo de canal es fácilmente especificado como las trazas imparciales (*fair*) de un autómata de E/S con la apropiada interface externa, cuyo estado es una cola de mensajes. La acción *send(m)*$_{ij}$ agrega m al final de la cola. La acción *receive(m)*$_{ij}$ está habilitada si m está primero en la cola, y su efecto es remover el primer mensaje de la cola. La tarea de partición coloca todas las acciones controladas localmente en una única clase.

Sistemas Asincrónicos Send/Receive

Un sistema de red asincrónico send/receive para un grafo dirigido G es obtenido mediante la composición de proceso y canal, utilizando la operación de composición de autómata de E/S. La definición del autómata de E/S para el canal de comunicación es la siguiente:

Signature

Entrada :

send(m)$_{ij}$, m ∈ M

Salida :

receive(m)$_{ij}$, m ∈ M

Estados

queue, una cola FIFO[1] de elementos de M, inicialmente vacía.

Transiciones

send(m)$_{ij}$

efecto:

agregar a queue {se considera que la cola no tiene límites}

receive(m)$_{ij}$

precondición:

m es el primero en queue

efecto:

remover primer elemento de queue

Tareas

{receive(m)$_{ij}$: m ∈ M}

A continuación se presenta la definición del autómata de E/S para el proceso.

Signature

Entrada :

init(v)$_i$, v ∈ V

receive(v)$_{ji}$, v ∈ V, 1≤ j ≤ n, j≠i

Salida :

decide(v)$_i$, v ∈ V

send(v)$_{ij}$, v ∈ V 1≤ j ≤ n, j≠i

[1]FIFO : First In First Out - Primero en Llegar Primero en Salir

Estados
val, un vector indexado por {1, ..., n} de elementos en V ∪ {null}, todos inicializados en null

Transiciones
init(v)$_i$, $v \in V$
efecto:
val(i) := v

send(v)$_{ij}$, $v \in V$
precondición:
val(i) := v
efecto:
ninguno

receive(v)$_{ji}$, $v \in V$
efecto:
val(j) := v

decide(v)$_i$, $v \in V$
precondición:
para todo j, $1 \leq j \leq n$
val(j) $\neq$ null
v = f(val(1), ..., val(n))
efecto:
ninguno

Tareas
para cada $j \neq i$:
{send(v)$_{ij}$: v ∈ V}
{decide(v)$_i$: v ∈ V}

En la figura 2.3 se observa la arquitectura de la composición de los autómatas Canal y Procesos.

La definición de la composición permite las interacciones correctas entre componentes, por ejemplo, cuando el proceso P_i realiza la acción de salida *send(m)*$_{ij}$, simultáneamente la acción de entrada *send(m)*$_{ij}$ es realizada a través del canal C_{ij}. Apropiados cambios de estado ocurren en ambos componentes.

2.2.4. Asignación de Recursos

El problema de exclusión mutua, es una abstracción del problema de asignación de recursos conteniendo accesos de usuarios concurrentes a un único recurso no compartido. La generalización del problema incluye varios recursos en vez de uno. Existen problemas más generales de tipo de asignación de recursos de los cuales se consideran en esta sección. Por ejemplo:

1. No se considera la posibilidad que un usuario pueda ganar aceptando combinaciones alternativas de recursos, por ejemplo, un usuario debe requerir “alguna impresora” en vez de una impresora específica.

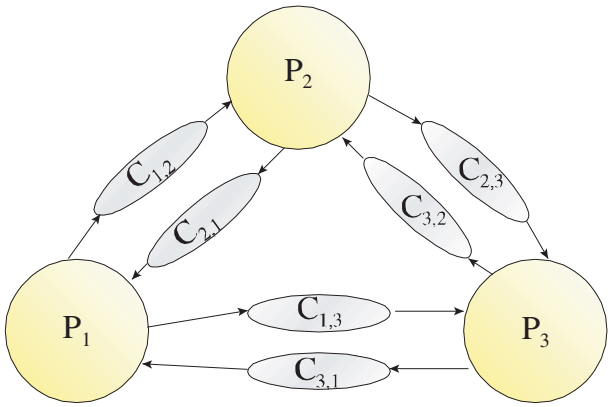


Figura 2.3: Arquitectura de la Composición de C_i's y P_i's

2. No se considera la posibilidad que un recurso pueda ser compartido. Por ejemplo, un conjunto de transacciones que requieren leer un dato de la base de datos, pueden realizar concurrentemente dicha operación.

Especificaciones Explícitas de Recursos y Exclusión

Se puede observar el problema desde dos puntos diferentes: como el problema de una asignación explícita de un recurso; o como el problema de asegurar que solamente un usuario a la vez está en la sección crítica. La *especificación de recursos explícitos* y la *especificación de exclusión* son diferentes alternativas para describir relaciones conflictivas entre usuarios.

Una especificación explícita de recursos R para n usuarios consiste de:

1. Un conjunto universal finito R de objetos denominado Recursos

2. Para cada i, $1 \leq i \leq n$, un conjunto $R_i \subseteq R$

La idea es que los recursos en R_i sean aquellos que el usuario U_i necesita para realizar su trabajo. Dos usuarios U_i y U_j están en *conflicto* con respecto a una dada especificación de recursos explícitos si requieren algún recurso en común, esto es, si $R_i \cap R_j \neq 0$

Por otra parte, la *especificación de exclusión* no hace ninguna mención explícita sobre los recursos. En realidad, la especificación está dada en términos de una colección E de "Conjuntos Inconsistentes" de índices de usuarios. Un "conjunto inconsistente" es un conjunto de índices de usuarios que no pueden ejecutarse simultáneamente. Hay una restricción en la especificación de exclusión: la colección de conjuntos inconsistentes requiere que sea *cerrada* bajo el superconjunto. Esto es, si un particular conjunto inconsistente E de usuarios pertenece a una especificación de exclusión ε, luego cualquier subconjunto de E también pertenece a ε. Por ejemplo:

1. La condición de exclusión mutua puede ser descripta mediante la especificación de exclusión:
$$\varepsilon = \{E \subseteq \{1, ..., n\} :| \ E \ | > 1\} \tag{2.1}$$

2. La condición de k-exclusión (en el cual el número de usuarios en la región crítica en cualquier instante de tiempo está restringido a lo sumo a k) puede ser descripta mediante la especificación de exclusión:
$$\varepsilon = \{E \subseteq \{1, ..., n\} :| \ E \ | > k\} \tag{2.2}$$

Note que la especificación explicíta de recursos genera la especificación de exclusión que es equivalente en el sentido de que permite la misma combinación de usuarios para ejecutarse simultáneamente. La especificación de exclusión consiste de exactamente aquellos conjuntos de usuarios con intersección vacía entre los requerimientos de sus recursos.

Problema de Asignación de Recursos

Se describe cómo incorporar especificaciones explícitas de recursos y especificaciones de exclusión en el problema de asignación de recursos a ser resuelto utilizando sistemas de memoria compartida. Para ser específico, se considera una especificación de exclusión fija ε (la cual puede ser derivada a partir de una especificación explícita de recursos).

La arquitectura es exactamente igual que la utilizada para el problema de exclusión mutua, una combinación de un autómata usuario y un sistema autómata de memoria compartida. El ciclo del usuario atraviesa las regiones *resto* (R), *entrada* (T), *crítica* (C) y *salida* (E).

La condición de bien formado en la composición del sistema es igual que antes. *Bien-formado*: en cualquier ejecución, y para cualquier i, la subsecuencia que describe la interacción entre U_i y A está bien-formada para i.

La condición de exclusión mutua es reemplazada por una condición más general en la condición. *Exclusión*: no se alcanza un estado del sistema en el cual el conjunto de usuarios en su sección crítica es un conjunto ε.

La condición de progreso es como antes. *Progreso*: en cualquier punto de una "ejecución imparcial":

1. Progreso para entrar a la región. Si al menos un usuario está en T (entrada) y no hay usuarios en C, entonces en un tiempo próximo algún usuario ingresará a C.

2. Progreso para región de salida. Si al menos un usuario está en E, entonces en un tiempo próximo algún usuario ingresará a R.

Se dice que un sistema de memoria compartida A resuelve el problema general de asignación de recursos para determinada colección de usuarios que, en combinación con aquellos usuarios, satisfacen el bien-formado, exclusión y condiciones de progreso. Se dice que A resuelve el problema general de asignación de recursos si lo resuelve para cada colección de usuarios.

La condición de progreso para la región de entrada es más débil que la presentada. Para el problema general de asignación de recursos, además se considera que usuarios que no están en conflicto no se impidan entre ellos ingresar a la región crítica, aunque tengan los recursos para siempre.

Progreso Independiente: en cualquier punto de una "ejecución imparcial".

1.) Progreso independiente para la región de entrada. Si U_i está en T y todos los usuarios conflictivos están en R, entonces en algún tiempo próximo tanto U_i ingresa a C o algún usuario conflictivo ingresa a T.

2.) Progreso independiente para la región de salida. Si U_i está en E y todos los usuarios conflictivos están en R, luego en algún tiempo próximo tanto U_i ingresa a R o algún usuario conflictivo ingresa a T.

Para las condiciones de alto nivel imparcialidad (*fairness*), las condiciones de libre de interbloqueo y libre de inanición son las mismas que se definen para el problema de exclusión mutua.

2.3. Resumen

En los sistemas distribuidos, hay procesos que compiten por los recursos y otros procesos que comparten los recursos para alcanzar un fin. En este capítulo se presentaron los mecanismos/protocolos para asegurar un ordenamiento en la ejecución de los procesos: sincronización, exclusión mutua y asignación de recursos.

Para cada uno de los mecanismos se presentaron las propiedades para alcanzarlo, y como es el ordenamiento de los eventos a través de la relación "ocurre antes"para la sincronización. Se presentaron las condiciones de un buen algoritmo de exclusión mutua: exclusión, progreso y equitatividad (libre de inanición). Además del problema tradicional se presentaron extensiones k-exclusión mutua y exclusión mutua para grupos de procesos. Y por último se presentaron las condiciones para un buen algoritmo de asignación de recursos.

Capítulo 3

Exclusión Mutua utilizando Memoria Compartida

3.1. Introducción

El problema de manejar el acceso a un único e indivisible recurso (como por ejemplo una impresora) que solamente puede soportar a un usuario a la vez, entre n usuarios $U_1..U_n$, o los conflictos resultantes de varios procesos concurrentes compartiendo recursos, es el estudio de exclusión mutua. Alternativamente, se puede pensar éste como el problema de asegurar que ciertas secciones de código de programa sean ejecutadas en forma estrictamente exclusiva (sección crítica).

Un usuario con acceso al recurso es modelado estando en la sección crítica, la cual es simplemente un subconjunto de sus estados posibles. Cuando un usuario no está involucrado de ninguna manera con el recurso, se dice que está en la sección *resto*. Para obtener la admisión a la sección crítica, un usuario ejecuta un protocolo de entrada (*trying*), y después que utiliza el recurso, se ejecuta un protocolo de salida (*exit*). Este procedimiento puede repetirse, de modo que cada usuario sigue un ciclo, desplazándose desde la sección resto (R), a la sección de entrada (T), luego a la sección crítica (C) y por último a la sección de salida (E), y luego vuelve a comenzar el ciclo en la sección resto.

3.2. Características de los Algoritmos

Los algoritmos de exclusión mutua tratados en este capítulo se basan en el modelo de memoria compartida descripto anteriormente. El sistema de memoria compartida contiene n procesos, numerados de $1, \ldots, n$, cada uno correspondiendo a un usuario U_i.

Cada proceso U_i, $1 \leq i \leq n$, es modelado como una máquina de estado (formalmente un autómata de E/S) que tiene como únicas acciones externas try_i, $exit_i$, $crit_i$ y rem_i.

En la figura 3.1 se muestran las interacciones entre los diferentes U_i y A. Como se observa

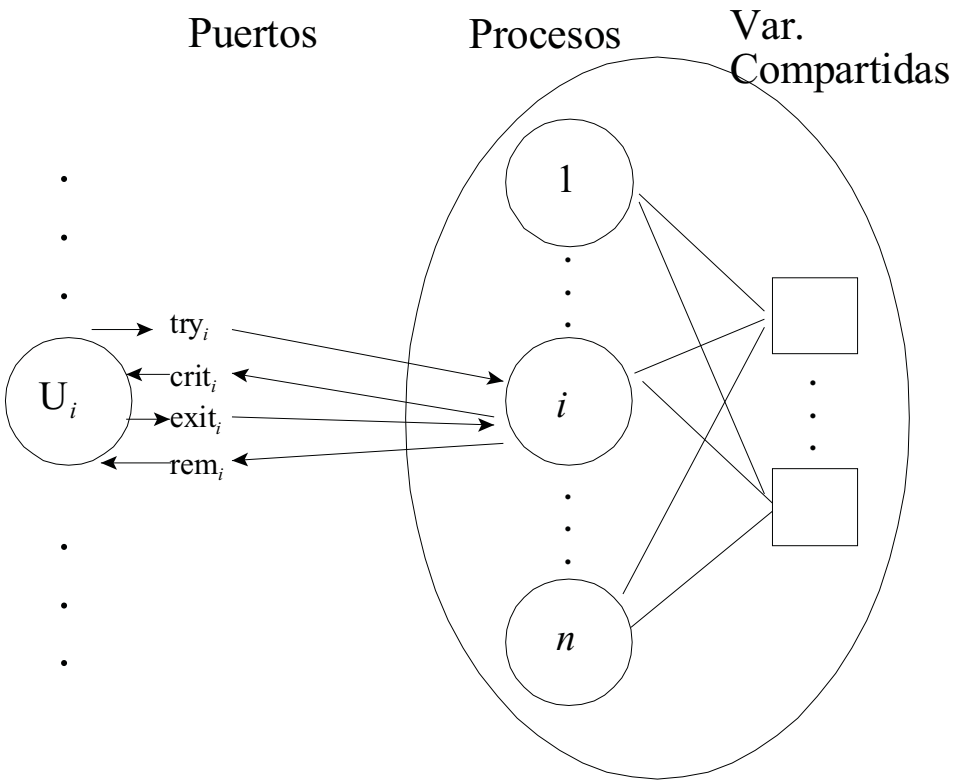


Figura 3.1: Interacción entre U_i y A

en la misma las acciones de entrada son try_i (requerimiento del usuario para acceder al recurso) y $exit_i$ (aviso de liberación del recurso) y las de salida $crit_i$ (autorización de acceso al recurso) y rem_i (puede continuar con el resto del trabajo).

Cada usuario U_i puede estar ejecutando cualquier programa de aplicación. Lo único que se asume sobre U_i es que que obedece el ciclo de la sección del protocolo, esto es, mantiene el orden del ciclo de las acciones try_i, $crit_i$, $exit_i$, ... (comenzando con try_i) entre el mismo y el proceso.

En la ejecución de U_i que observa el orden en el ciclo de acciones, se dice que U_i:

- Está inicialmente en la sección resto entre cualquier evento rem_i y el siguiente evento try_i.
- Está en la sección de entrada entre cualquier evento try_i y el siguiente evento $crit_i$.
- Está en la sección crítica entre cualquier evento $crit_i$ y el siguiente evento $exit_i$. Durante este lapso de tiempo tiene la posibilidad de utilizar el recurso.

- Está en la sección de salida entre cualquier evento $exit_i$ y el siguiente evento rem_i.

3.3. Revisión de Algoritmos de Exclusión Mutua

3.3.1. Algoritmo de Dijkstra

Este algoritmo es uno de los primeros diseñados para satisfacer exclusión mutua para un modelo de memoria asincrónica. Este algoritmo no es el más elegante o eficiente de los algoritmos disponibles, ya que no satisface las denominadas *condiciones fuertes* (inanición e imparcialidad). Su estudio es de interés, ya que es uno de los primeros algoritmos que se lo pueden categorizar como *distribuido* El código tradicional se encuentra en la figura 3.2, utiliza variables de múltiple lectura y de múltiple escritura.

Este algoritmo garantiza las propiedades de exclusión mutua y progreso, aunque existen además otras condiciones que son deseables pero que no las garantiza. No garantiza que la sección crítica sea imparcialmente accedida a los diferentes usuarios, por ejemplo, permite que un usuario repetidamente tenga acceso a la sección crítica mientras que otros usuarios tratan de ganar acceso y siempre son prevenidos de hacerlo. Esta es una situación de *inanición*. Este algoritmo no pertenece a ninguno de los tipos mencionados en el capítulo 2.

3.3.2. Algoritmo de Peterson 2P

El algoritmo que se presenta en esta sección es una solución para resolver el problema de exclusión mutua para dos procesos, en la cual se garantiza la exclusión mutua, el progreso, y además las condiciones fuertes de imparcialidad e inanición. Usualmente se denomina a los dos procesos 1 y 2, pero en este caso por conveniencia se denominará al proceso 2 con 0. Si i $\in \{0, 1\}$, se escribe $\bar{\text{i}}$ para indicar i-1, el índice del otro proceso. El código de la forma tradicional se observa en la figura 3.3.

En el algoritmo, el proceso i comienza con la inicialización de su *Flag* a 1, lo cual es lo mismo que el proceso realiza en el algoritmo de Dijkstra. En este caso, inmediatamente después procede a inicializar la variable *Turn* igual a i. Luego espera hasta que descubre que su *Flag* se encuentra en 0 ó que *Turn* $\neq$ i.

Para probar que el algoritmo es correcto se lo debe traducir a un modelo formal, el mismo fue traducido en [95] y demostrado que es correcto y que garantiza exclusión mutua, progreso, imparcialidad e inanición. La desventaja que presenta este algoritmo es que sólo controla 2 procesos.

3.3.3. Algoritmo de Peterson NP

Para n procesos, se utiliza la idea del algoritmo de Peterson 2P iterativamente, en series de $n-1$ competiciones en los niveles $1, 2, \ldots, n-1$. En cada sucesiva competencia, el

Variables Compartidas

Turn ∈ {1,..., n}, inicialmente arbitrario, leída y escrita por todos los procesos

Para cada i, 1 ≤ i ≤ n

Flag(i) ∈ {0,1,2}, inicialmente 0, escrito por i y leído por todo todos los procesos

Proceso i

** Sección Resto **

try i

L : Flag(i) := 1
Mientras Turn ≠ i hacer
if Flag(Turn) = 0 then
Turn := i
Flag(i) := 2
Para j ≠ i hacer
if Flag(j) = 2 then Goto L

crit i

** Sección Crítica **

exit i

Flag(i):= 0

rem i

.....

Figura 3.2: Algoritmo de Dijkstra

Variables Compartidas

Turn ∈ {0, 1}, inicialmente arbitrario, leída y escrita por todos los procesos

Para cada i ∈ {0, 1}
  Flag(i) ∈ {0, 1}, inicialmente 0, escrito por i y leído por $\bar{i}$

Proceso i

** Sección Resto **

try i
  Flag(i) := 1
  Turn := i
  waitfor Flag($\bar{i}$) = 0 or Turn ≠ i

crit i

** Sección Crítica **

exit i

  Flag(i):= 0

rem i

.....

Figura 3.3: Algoritmo de Peterson para 2P

Variables Compartidas

Para cada k ∈ {1, ..., n-1}
  Turn(k) ∈ {0,.., n}, inicialmente arbitrario, leída y escrita por todos los procesos

Para cada i, 1 ≤ i ≤ n:
  Flag(i) ∈ {0, .., n-1}, inicialmente 0, escrito por i y leído por j ≠ i

Proceso i

** Sección Resto **

try i

  para k = 1 hasta n-1 hacer
    Flag(i) := 1
    Turn(k) := i
    waitfor [∀ j ≠ i: Flag(j) < k] or [Turn(k) ≠ i]

crit i

** Sección Crítica **

exit i

  Flag(i):= 0

rem i

.....

Figura 3.4: Algoritmo de Peterson para NP

algoritmo asegura que hay al menos un perdedor. Todos los procesos pueden competir en el primer nivel, pero a lo sumo $n-1$ procesos pueden ganar. En general, a los sumo $n-k$ procesos pueden ganar en el nivel k. En el nivel $n-1$ a lo sumo puede ganar un proceso, el cual logra la condición de exclusión mutua. El código de algoritmo en la forma clásica se observa en la figura 3.4.

Cada nivel k tiene su propia variable *Turn*, *Turn(k)*. En cada nivel, el proceso i se comporta de manera similar al modo que un proceso se comporta en el algoritmo de Peterson 2P; inicializa *Turn(k)* $= i$, luego espera hasta descubrir si todos los procesos en sus respectivas variables *Flag* son estrictamente menores que k, ó que *Turn(k)* $\neq i$.

Este algoritmo de n procesos resuelve el problema de exclusión mutua y libre de interbloqueo según está demostrado en [95]. Con la última propiedad especificada para el algoritmo se puede concluir que garantiza el progreso, ya que una posibilidad para demostrar el progreso es que el algoritmo presente la condición de libre de bloqueos entre los procesos.

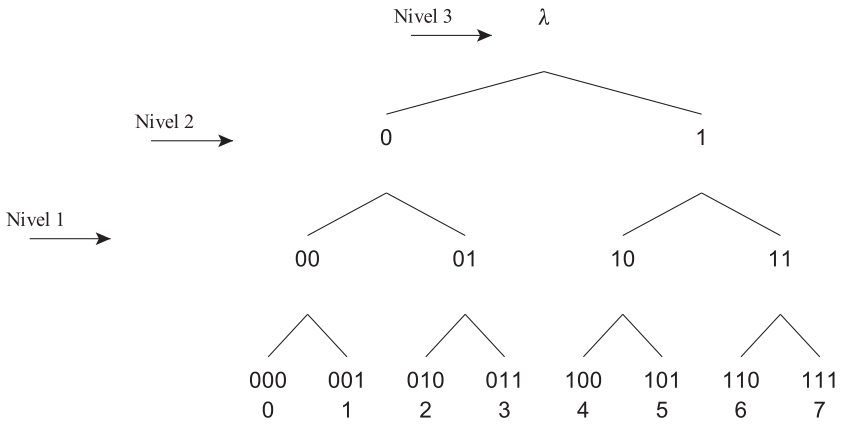


Figura 3.5: Nombres de Competición en el Algoritmo de Tournament

Los algoritmos de Peterson para 2 procesos y para N procesos son distribuidos, pero no se puede considerar que pertenecen a algunos de los tipos de algoritmos especificados en el Capítulo 2.

3.3.4. Algoritmo de Tournament

El algoritmo de Tournament permite resolver el problema de exclusión mutua para n procesos; está basado en el algoritmo de exclusión mutua de Peterson para dos procesos, explicado en secciones anteriores. Para simplificar, se asume que n es el número de procesos, y es una potencia de 2. La numeración de los procesos comienza en 0 y termina en $n-1$.

Cada proceso está ocupado en una serie de competiciones de $O(\log n)$ para obtener el recurso. Puede pensarse que la competición está dispuesta en un árbol de competencia binario de n hojas,y que éstas corresponden de izquierda a derecha a los procesos $0 \ldots (n-1)$. En la figura 3.5 se muestra como quedaría el árbol para 8 procesos.

La notación utilizada para los diferentes roles es la siguiente:

- comp(i, k), el nivel de competencia k para el proceso i, es la cadena que consiste

Variables Compartidas

Para cada cadena binaria x de a lo sumo longitud log n - 1
Turn(x) ∈ {0,1}, inicialmente arbitrario, leída y escrita por exactamente aquellos procesos i para los cuales x es un prefijo de la representación binaria de i

Para cada i, 0 ≤ i ≤ n-1
Flag(i) ∈ {0, ... log n}, inicialmente 0, escrito por i y leído por todo j ≠ i

Proceso i

** Sección Resto **
Try i
For k = 1 to log n do
Flag(i) := k
Turn(comp(i,k)) := role(i,k)
Waitfor (∀ j ∈ opponents(i,k) : Flag(j) < k) or (Turn(comp(i,k))≠ role(i,k))
Crit i
** Sección Crítica **
exit i
Flag(i):= 0
rem i
.....

Figura 3.6: Algoritmo de Tournament

de $\log(n-k)$ bits de orden superior de la representación binaria de i. En términos del árbol de competencia, $\text{comp}(i,k)$ puede ser utilizado como el nombre para el nodo interno que es el ancestro en el nivel k de la hoja. En particular, la raíz es denominada λ, la cadena vacía.

- $\text{role}(i,k)$, es el rol del proceso i en el nivel de competencia k para el proceso i, es la cadena que consiste del $(\log(n-k+1))$ bit de la representación binaria de i. En términos del árbol de competencia, $\text{role}(i,k)$ indica si la hoja i es un descendiente de la rama izquierda o derecha del nodo para la competencia $\text{comp}(i,k)$.

- $\text{opponents}(i,k)$, son los oponentes del proceso i en el nivel de competición k, es el conjunto de procesos con los mismos bits de orden superior de $(\log(n-k))$ bits que i y el opuesto en el $(\log(n-k+1))$ ésimo bit. En términos del árbol de competencia los procesos en $\text{opponents}(i,k)$ son aquellos cuyas hojas son descendientes de la rama opuesta del nodo $\text{comp}(i,k)$, esto es, de la rama que no es ancestro de la hoja i.

El código del algoritmo en un formato clásico se muestra en la figura 3.6. El algoritmo de Tournament satisface exclusión mutua junto con las propiedades de progreso y libre de interbloqueo.

Try i

```
Si truncar(log(n)) = log (n) entonces
   etapas = truncar(log(n))
Sino
   etapas = truncar(log(n)) + 1
Fin si
{etapas contiene la cantidad de niveles}
Para k=1 hasta etapas hacer
   flag(i) := k {representa a los diferentes niveles}
   turn(comp(i,k)) := role(i,k)
   si role(i,k) ≠ 0 ó (i ≤ n-k) entonces
      waitfor [∀ j∈ oponentes(i,k) : flag(j) < k] o [turn(comp(i,k)) ≠ role (i,k)]
```

Figura 3.7: Algoritmo Modificado

Una propiedad poco atractiva del algoritmo de Tournament es que utiliza como variable/dato compartido (*Turn*) de múltiple escritura/múltiple lectura. Este tipo de variables son difíciles y costosas de implementar en muchas clases de sistemas de multiprocesador como ocurre en sistemas de pasaje de mensajes. Es mejor diseñar algoritmos que utilicen solamente simple escritura y múltiple lectura en sus variables/datos, o aún mejor, simple escritura y simple lectura en sus variables datos (pero reduce la concurrencia).

¿Qué sucedería si n no fuera potencia de 2?

El árbol binario no sería completo, lo cual implica que para los procesos denominados i con valor cercano a n en algunos niveles no tendría oponentes. En la figura 3.5 se observa un árbol binario completo para 8 procesos. En el caso que hubiera sólo 6 procesos, para los procesos denominados 5 y 6 en el segundo nivel de competencia no tienen oponentes y pueden pasar directamente al próximo nivel.

Si el proceso desciende de la rama derecha siempre tendrá oponentes; de lo contrario, sino dependerá del nivel en que se encuentre. Si se extiende el algoritmo para que n no sea potencia de 2, se debe calcular los niveles que presenta el algoritmo, ya que $\log n$ no será un número entero sino que en la mayoría de los casos será un número real.

Se define una nueva variable denominada *etapas*, ésta representa la cantidad de bits necesarios para almacenar hasta el valor $(n-1)$. A cada una de las funciones utilizadas en el algoritmo, se les reemplaza $\log n$ por *etapas*. En la figura 3.7 se observan las modificaciones aplicadas al algoritmo.

3.3.5. Algoritmo TSM

El algoritmo TSM fue presentado en [40]. La utilización de variables de control de múltiple escritura para el obtener el acceso a la sección crítica son muy costosas e ineficientes de implementar. La utilización de variables de control de simple escritura para el acceso a la sección crítica facilita el diseño e implementación de los protocolos.

En el algoritmo TSM se realiza una adaptación del algoritmo de Tournament transformando sus variables compartidas a simple escritura y múltiple lectura. En esta adaptación no es necesario utilizar la variable de control $flag$ para acceder a la sección crítica.

La figura 3.8 muestra un ejemplo de árbol con 4 niveles con un total de 16 procesos, considerando que se tienen solamente 13 procesos en el sistema. Se puede observar que el proceso 13 no tendrá oponentes en los niveles 1 y 2 de competencia. En el algoritmo se considera que tenga oponentes reales. La figura 3.9 muestra el algoritmo modificado.

Como es conveniente modelar formalmente el algoritmo, se lo transforma en un autómata de entrada/salida, que requiere solamente el testeo de los oponentes activos en cada uno de los niveles. Se agrega la definición de $\mathrm{OpAct}(i, k)$.

- $\mathrm{OpAct}(i, k) \rightarrow$ los oponentes del proceso i en el nivel k de competición del proceso i, es el conjunto de índices de procesos con el mismo orden de bits en (etapas - k) y el opuesto en (nivel - $k + 1$) y que $\mathrm{Turn}(x, j) \neq \mathrm{Null}$

La figura 3.10 muestra las definiciones de las variables compartidas utilizadas, las acciones correspondientes para cada uno de los procesos y los estados de los mismos y la figura 3.11 muestra las transiciones del algoritmo TSM.

La figura 3.12 muestra que los procesos 5 y 7 son los ganadores en el primer nivel de competencia y compiten en el segundo nivel en donde son oponentes respectivamente. El estado de las variables, después de pasar por la transición *set-turn*, es el siguiente:

Proceso 5

$Turn(\mathrm{comp}(5,2)) = \mathrm{role}(5,2) = 0$

Proceso 7

$Turn(\mathrm{comp}(7,2)) = \mathrm{role}(7,2) = 1$

Cuando un proceso es el ganador de un nivel (en este caso el primero) vuelve al estado de *set-turn*. Se inicializa el valor de la variable de control *Turn* correspondiente al nivel que se encuentra. En este estado se controla si el proceso tiene oponentes en el nivel.

- Si no hay oponentes en el nivel se lo considera el ganador y pasa al siguiente.
- Si tiene oponentes, entonces pasa al estado *check-oponentes*. En este estado se verifican los oponentes que se encuentran activos, y luego se pasa al estado *check-turn* hasta que sea el ganador del nivel o pueda acceder a la sección crítica. En el estado que controla *turn* sólo espera por los procesos que son oponentes activos del nivel.

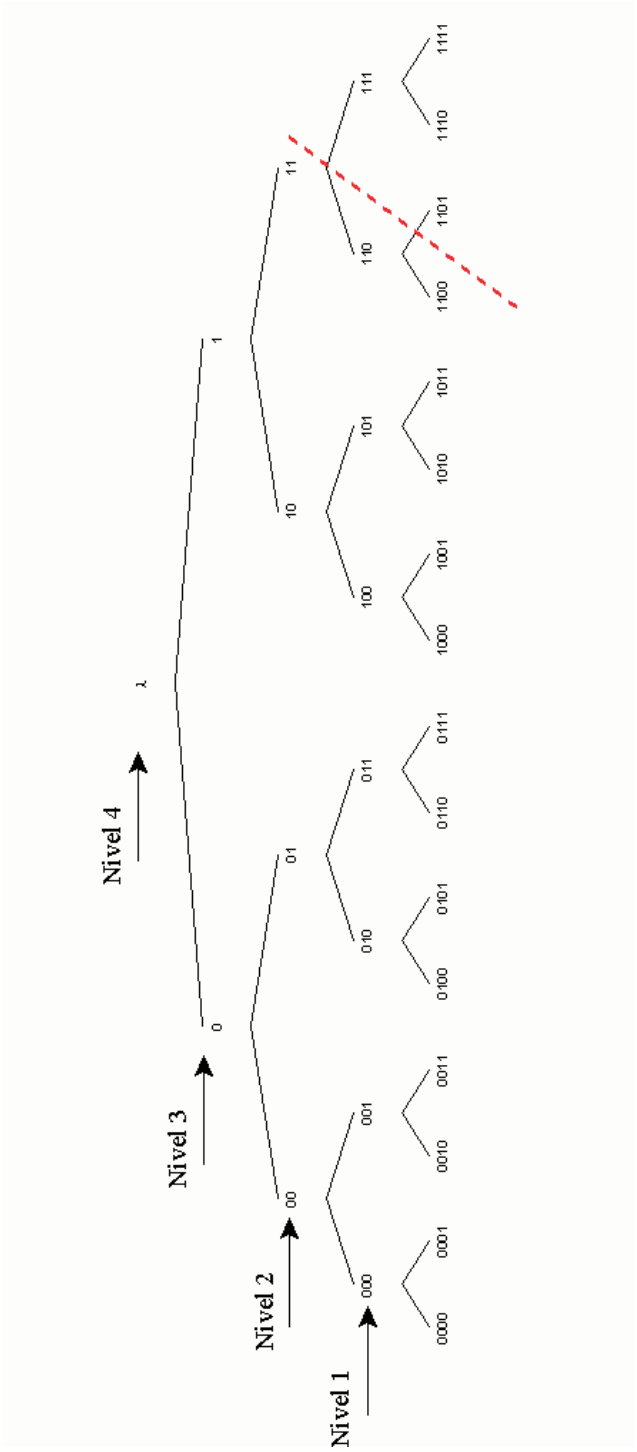


Figura 3.8: Árbol completo para 16 procesos, en este caso N=13

Variables compartidas

Para cada cadena binaria x de a lo sumo longitud etapas -1 y
Para cada i, $0 \leq i \leq n-1$
Turn(x,i) $\in Z \cup$ Null, escrita por i y leída por todo $j \neq i$

Proceso i

*** Sección Resto ***

try i

si truncado(log(n)) = log(n) entonces
etapas := truncado(log(n)) {etapas contiene la cantidad de niveles}
sino
etapas := truncado(log(n)) + 1
fin si
para k=1 hasta etapas hacer
turn(comp(i,k),i) := role(i,k)
si role(i,k) $\neq$ 0 ó (i $\leq$n-k) ó (i<2^{etapas}/2) entonces
($\forall$ j $\in$ oponentes(i,k)
si turn(comp(i,k),j) $\neq$ Null entonces
turn(comp(i,k),i) := turn(comp(i,k),i) + turn(comp(i,k),j)))
waitfor ($\forall$ j $\in$ opact(i,k): (turn(comp(i,k),j)=Null) ó
(turn(comp(i,k),i)<turn(comp(i,k),j)) ó
turn(comp(i,k),i)= turn (comp(i,k),j) y
(i > j)))

*** Sección Crítica ***

Exit i

turn(comp(1, 1..etapas), i):= Null

Rem i

Figura 3.9: Algoritmo de TSM

Variables compartidas

Para cada cadena binaria x de longitud a lo sumo etapas -1 y
Para cada i, $0 \leq i \leq n-1$:
Turn(x,i) $\in$ 0,1 $\cup$ Null, inicialmente Null

Acciones de i

Entrada	Internas
Try_i	$set - turn_i$
$Exit_i$	$check - oponentes_i$, $\forall$ j $\in$ oponentes(i,nivel)
Salida	$check - turn(j)_i$, $\forall$ j $\in$ opact(i,nivel)
$Crit_i$	$reset_i$
Rem_i	

Estados de i

estado $\in$ {rem, set-turn, check-oponentes, check-turn, leave-try, crit, leave-exit} inicialmente rem

Nivel $\in$ {1 ... etapas}, inicialmente 1

S, un conjunto de índices, inicialmente 0

O, un conjunto de índices, inicialmente oponentes(i, nivel)

A, un conjunto de índices, inicialmente 0

Figura 3.10: Variables, Acciones y Estados

Si es el ganador del nivel, pasa nuevamente al estado *set-turn* y repite el proceso. En el caso que sea el ganador del último nivel pasa al estado *leave-try* que le permite acceder a la sección crítica cambiando el estado a *crit*.

Para el ejemplo de la figura 3.12, los procesos oponentes para el proceso 5 en el nivel 2 son los procesos 6,7. Para el proceso 7 en el nivel 2 son los procesos 4 y 5. Para el proceso 5, en el estado *check-oponentes* se obtiene que sólo el proceso 7 es el oponente activo para su nivel de competencia y ocurre lo mismo para el proceso 7. Al entrar ambos procesos en el estado *check-turn*, habrá un ganador (el proceso 5 ó el proceso 7).

En el caso que sea el proceso 5 el ganador se darían las siguientes condiciones:

$Turn(\text{comp}(5,2)) = 0$

$Turn(\text{comp}(7,2)) = 1$

Al entrar en el estado $check - turn_5(7)$ se verificaría que:

$turn(\text{comp}(5,2)) < turn(\text{comp}(7,2))$

y, por lo tanto, se convierte en el ganador del nivel.

En el caso que sea el proceso 7 el ganador se darían las siguientes condiciones:

$Turn(\text{comp}(5,2)) = 1$

$Turn(\text{comp}(7,2)) = 1$

Transiciones de i

Try_i
  efecto:
    estado := set-turn

$set - turn_i$
  precondición:
    estado = set-turn
  efecto:
    Turn(comp(i,Nivel),i) := role(i,Nivel)
    If (role(i,Nivel) $\neq$ 0) or (i $\leq$ n-nivel) then
      //tiene oponentes en el nivel
      S:= $\emptyset$, A:= $\emptyset$
      estado := check-oponentes
    Else // no tiene oponentes en el nivel
      If (Nivel < etapas) then
        Nivel := Nivel + 1
        O := oponentes(i,Nivel)
      Else
        estado := leave-try

$check - oponentes(j)_i$
  precondición:
    estado = check-oponentes
    j $\in$ O
    j $\notin$ S
  efecto:
    S = S $\cup$ j
    If Turn(comp(i,Nivel),j) $\neq$ Null then
      Turn(comp(i,Nivel),i):= Turn(comp(i,Nivel),i) +
      Turn(comp(i,Nivel),j)
    else
      A = A $\cup$ j
      // Son los procesos que no compiten en el nivel
    If |S| =|O| then
      If |A| = |O| then
        S:=$\emptyset$
        If Nivel < etapas then
          Nivel := Nivel + 1
          O := oponentes(i,Nivel)
          estado := set-turn
        Else
          estado := leave-try
      Else
      S:= $\emptyset$ $\cup$ A
      estado := check-turn

$check - turn(j)_i$
  precondición:
    estado = check-turn
    j $\in$ O
    j $\notin$ S
  efecto:
    if ((Turn(comp(i,Nivel), j) = Null) or
    ((Turn(comp(i,Nivel), i)<(Turn(comp(i,Nivel),j))
    or ((Turn(comp(i,Nivel),i)=(Turn(comp(i,Nivel),j)
    and (i >j))) then
      S := S $\cup$ j
      If |S| = |O| then
        S:= $\emptyset$
        If Nivel < etapas then
          Nivel := Nivel + 1
          O := oponentes(i,Nivel)
          estado := set-turn
        Else
          estado := leave-try
    else
      S:= $\emptyset$ $\cup$ A

$Crit_i$
  precondición:
    estado = leave-try
  efecto:
    estado := crit

$Exit_i$
  efecto:
    estado := reset

$Reset_i$
  precondición:
    estado = reset
  efecto:
    For k :=1 to etapas do
      Turn(comp(i,k), i) := Null
    Nivel := 1
    estado := leave-exit

Rem_i
  precondición:
    estado := leave-exit
  efecto:
    estado := rem

Figura 3.11: Transiciones de TSM

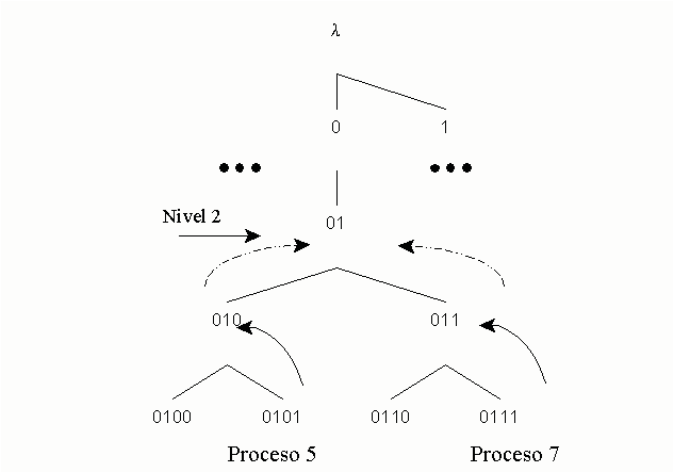


Figura 3.12: Subárbol

Al entrar en el estado $check - turn_7(5)$ se verificaría que:

$turn(\text{comp}(7,2)) = turn(\text{comp}(5,2))$ y $(7 > 5)$,

por lo tanto, se convierte en el ganador del nivel.

Teorema 3.1 *El algoritmo TSM está bien formado.*

La propiedad presentada en el teorema 3.1 se verifica por construcción, ya que fue reescrito formalmente utilizando el modelo de autómata de E/S.

Teorema 3.2 *El algoritmo TSM garantiza propiedad de exclusión mutua.*

Para la exclusión mutua, la idea clave está en que el nivel k de competición solamente permita un proceso ganador desde cualquier raíz del subárbol en el nivel k.

En cualquier estado del sistema del algoritmo planteado, un proceso i es un ganador en el nivel k si se mantiene:

$nivel_i > k$ ó

$Turn(comp(i, nivel), i) < Turn(comp(i, nivel), j)$ ó

$Turn(comp(i, nivel), i) = Turn(comp(i, nivel), j)$ y $(i > j)$ ó

$nivel_i = k$ y $estado_i \in \{$leave-try, crit, exit$\}$

La última condición sucederá cuando $nivel_i =$ etapas. Un proceso i es un competidor en el nivel k, si es un oponente en el nivel k y el $nivel_i = k$ con $estado_i \in \{$*check-oponentes, check-turn*$\}$.

La clave del algoritmo para mantener exclusión mutua es que exista un solo ganador de cada subárbol por nivel. Supongamos que se verifica hasta el nivel $(k-1)$ y en el nivel k hay dos ganadores entonces:

$Nivel_i = Nivel_j = k$, en este caso hay un oponente que intenta obtener el acceso. En la verificación de la variable Turn puede suceder lo siguiente:

$Turn(comp(i,k),i) < Turn(comp(i,k),j)$, el proceso i pasa al siguiente nivel y el proceso j debe esperar.

$Turn(comp(i,k),i) = Turn(comp(i,k),j)$, como $i \neq j$, ó $i > j$ ó $j > i$. Por ende uno de los dos procesos avanza al siguiente nivel y el otro debe esperar.

En los diferentes casos se contradice la suposición inicial de que hay más de un ganador por nivel en cada subárbol. Por lo tanto, el algoritmo satisface exclusión mutua.

Teorema 3.3 *El algoritmo TSM garantiza propiedad de progreso.*

Si en una ejecución a, hay por lo menos un proceso en la sección de entrada y no hay ningún proceso en la sección crítica, supongamos que no puede entrar ningún proceso. Si el proceso, denominado i, está en la sección de entrada entonces está esperando en el estado de $check-turn$, pero al no haber ningún proceso en la sección crítica, entonces $nivel_j < nivel$ (se obtiene del chequeo de $turn$(comp(i,k),j) =Null), para $i \neq j$ podrá avanzar de nivel y en el último nivel tendrá la posibilidad de acceder a la sección crítica. Esto contradice la suposición inicial.

Si un algoritmo está bien formado y es libre de bloqueo (para todos los procesos) entonces garantiza la propiedad de progreso. Si mantiene la propiedad de vivacidad entonces no puede ocurrir inanición. ¿Es posible que un proceso espere indefinidamente para acceder a la sección crítica? Cuando un proceso i ingresa en la sección de entrada, pasa por el estado *set-turn*, en el cual inicializa la variable *turn* para ese nivel y en el estado *check-oponentes* controla cuales procesos son oponentes activos, y actualiza el valor de *turn* de acuerdo a los oponentes activos y al orden de llegada. Por lo tanto, un proceso j que accede después a la sección de entrada, deberá esperar que el proceso i tenga su acceso a la sección crítica o al próximo nivel antes de poder avanzar.

3.3.6. Algoritmos Rápidos / Adaptables de Exclusión Mutua

Aquellos algoritmos en los cuales los procesos ejecutan en un tiempo constante un camino rápido en la ausencia de contención se los denomina algoritmos rápidos de exclusión mutua. Esta denominación se aplica a los algoritmos que utilizan lecturas y escrituras.

Algoritmo de Lamport

Lamport diseñó un algoritmo para exclusión mutua que requiere 7 accesos a memoria en el caso que no exista contención. El algoritmo está libre de bloqueos pero no garantiza

la propiedad de inanición. La idea del algoritmo es que cada proceso debe ejecutar una operación de escritura sobre una variable x seguida de una lectura sobre otra variable y. No tiene sentido realizar una lectura sobre una variable que no será escrita o escribir una variable que no será leída, por lo que el algoritmo requiere que se lea la variable x y que se escriba sobre la variable y. La última operación antes de ingresar a la región crítica va a ser una lectura sobre la variable x. En la región de entrada el proceso debe realizar las siguientes operaciones: *write x, read y, write y, read x*. Lamport [91] inicialmente presenta un algoritmo que requiere solo 5 accesos a memoria, pero tiene el problema que requiere de un tiempo limitado para ejecutar la región crítica.

Otro algoritmo de Lamport [91] se muestra en la figura 3.13, el cual requiere 7 accesos a memoria para el caso que no exista contención, pero tiene la ventaja de no restringir el tiempo en la región crítica. Las operaciones utilizadas en el camino rápido, en el caso de no contención, aparecen resaltadas.

En el caso que haya contención para el ingreso a la sección crítica el algoritmo no presenta un buen comportamiento, ya que está diseñado utilizando *espera ocupada* y no existe una cota en la cantidad de accesos a la memoria compartida, generando así un impacto negativo en las comunicaciones.

El elemento splitter

En el algoritmo rápido de exclusión mutua Lamport [91] utiliza un conjunto de sentencias para definir un elemento *caja negra* denominado *splitter*. Este término no corresponde a Lamport, inicialmente fue abstraído como *caja negra* por Moir y Anderson en [107] y denominado como "splitter"por Attiya y Fouren en [21].

Cada proceso que invoca un código splitter puede tanto parar (*stop*), moverse hacia abajo (*down*) o moverse a la derecha (*right*). Las propiedades del splitter son:

- si varios procesos invocan el splitter, entonces a lo sumo uno de ellos puede parar en el mismo.
- si n procesos invocan el splitter, entonces a lo sumo $n-1$ pueden desplazarse hacia la derecha, y a lo sumo $n-1$ pueden desplazarse hacia abajo.

En la figura 3.14 se muestra el elemento splitter y el código correspondiente. Dadas las propiedades que presenta este elemento se utiliza en algoritmos libre de espera (*wait-free*) para la búsqueda de nombre.

Por las propiedades mencionadas, el elemento splitter y elementos relacionadas se han utilizado en los algoritmos de renombre (buscar temporalmente un nombre de identificación) libres de espera. Los algoritmos de renombre son utilizados para acotar el espacio de nombres de la cual los identificadores de los procesos son tomados. Estos algoritmos son utilizados para acelerar la computación concurrente con ciclos que iteran sobre los identificadores de procesos. Algunos algoritmos de exclusión mutua con la característica de adaptables, inicialmente utilizan un algoritmo de renombre y luego continúan con la competencia por el ingreso a la sección crítica.

Variables Compartidas

b: array[1..N] of boolean; inicialmente en false
x : 1..N;
y: 0..N; inicialmente en 0

Process i

** Sección Resto **

Try i

Comienzo: **b[i] := true;**
x :=i;
Si **y** $\neq$ **0** entonces
b[i]:= false;
await y = 0
goto comienzo
finsi
y := i;
si **x** $\neq$ **i** entonces
b[i] := false;
for j := 1 to N do
await $\neg$ b[j]
od;
si y $\neq$ i entonces
await y = 0;
goto comienzo
finsi
finsi

Crit i

** Sección Crítica **

Exit i

y := 0;
b[i] := false

Rem i

Figura 3.13: Algoritmo de Lamport

shared variable X: {⊥} ∪ {0 .. N-1} initially {⊥}
Y : boolean initially false
private variable dir: {stop, right, down}
1. X := p
2. if Y then dir := right
else
3. Y := true
4. if X = p then dir := stop
else dir := down

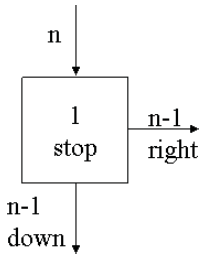


Figura 3.14: El código del elemento splitter - El elemento splitter

Procedure reflector(entrance r:0,1)
R_r = true;
if ($R_1 - r$ = false) then
return ($down_r$)
else
return(up_r)

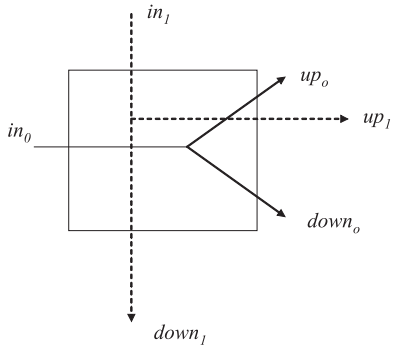


Figura 3.15: El elemento reflector

El elemento reflector

Un *reflector* es similar a un splitter. Un proceso ingresando en el reflector cambia la dirección de sus movimientos si otro proceso pasa a través del reflector. La diferencia es que un reflector tiene dos entradas distinguibles y la dirección del nuevo movimiento depende de la entrada a través de la cual el proceso ingresó en el reflector.

Un reflector tiene dos entradas, in_0 e in_1, dos salidas inferiores (*lower exits*), $down_0$ y $down_1$, y dos salidas superiores (*upper exits*), up_0 y up_1. Un proceso que ingresa al reflector en la entrada in_i deja el reflector solamente en salidas up_i o $down_i$. Si un único proceso ingresa en el reflector entonces debe abandonar en una salida inferior (*lower exit*), y a lo sumo un proceso deja sobre una salida inferior (*lower exit*); y es posible que dos procesos puedan dejar el reflector sobre una salida superior (*upper exit*). Un reflector puede ser implementado con registros booleanos, la figura 3.15 muestra este elemento y su correspondiente código.

El *sieve* básico

Un *sieve* básico [22] permite a los procesos obtener una vista de los procesos que acceden concurrentemente al *sieve*, o no hay vista, esto es una vista vacía. Hay un único conjunto no vacío de candidatos que son vistos por todos los procesos que obtienen una vista. El *sieve* garantiza un acuerdo en la información anunciada por los candidatos, sincronizando el acceso de los procesos al *sieve* y permitiendo intercambiar información en una manera

adaptiva.

Un *sieve* tiene un número indefinido de copias. En cada punto de la ejecución, los procesos están solamente dentro de una única copia del *sieve*. El número de copias corrientes es monótonamente creciente en uno.

Los *sieve* soportan las siguientes operaciones:

- read(s.count): obtiene el número corriente de copia del *sieve* s.
- openFor(s,c): retorna **todos**, si todas las operaciones pueden ingresar copia (s,c), y $\emptyset$ si no hay proceso que pueda ingresar copia (s,c).
- enter(s,c,info): ingresa copia (s,c), anunciando *info*, retorna el conjunto de candidatos, junto con la información anunciada por ellos.
- exit(s,c): deja el *sieve* y activa la próxima copia, (s, c+1), si es posible.

Para acceder al *sieve* s, un proceso debe primero obtener el número de copia corriente desde *s.count*. El proceso ingresa al *sieve* utilizando la función **enter** solamente si la función **openFor** retorna **todos**, y deja el *sieve* utilizando la función **exit**. Un proceso accede al *sieve* s de la siguiente manera:

1. c = read(s.count)
2. if (openFor(s,c) == todos) then
3. enter(s,c,info)

   ...
4. exit(s,c)

Los *sieve* son implementados utilizando las ideas de la simulación de Borowsky-Gafni [28], [29]. En cada punto de la ejecución, los candidatos están dentro de una única copia; estos procesos acceden al sieve simultáneamente.

Para ingresar a un *sieve* un proceso verifica si es el primer proceso que accede a la copia actual de ese *sieve*. Solamente tiene éxito si la copia está libre, y no hay un candidato dentro de la copia previa. El *sieve* captura al menos un proceso: uno de los procesos que acceden a la copia actual ingresan en el mismo. Esta propiedad permite que se lo utilice para el punto de contención en algoritmos adaptivos.

Para cada *sieve* hay una variable entera, *count*, que indica la copia corriente del *sieve*; está inicializada en 1. Hay un número infinito de copias para cada *sieve*, numeradas 1,2, ... Cada copia tiene la siguiente estructura de datos:

- Un arreglo R[1,...,N] de vistas, todas inicializadas vacías. La entrada R[id_i] contiene la vista obtenida por el proceso id_i en esa copia.

- Un arreglo done[1,...,N] de variables boolean, todas las entradas inicializadas en *false*. La entrada done[id_i] indica si el proceso id_i ha finalizado con esta copia.
- Una variable *boolean alldone*, inicializada en *false*. Indica si todos los procesos que podrían estar dentro de la copia han finalizado.
- Una variable *boolean inside*, inicializada en *false*. Indica si algún proceso está aún dentro de la copia.

Los elementos splitter, reflector y *sieve*, por las propiedades que presentan, se los puede utilizar para la construcción de algoritmos adaptivos para resolver el problema de renombre (*renaming*). Éstos se aplican en los algoritmos de coordinación distribuida.

Algoritmo del Panadero

El algoritmo del *panadero* utiliza un concepto muy simple para resolver el problema del acceso a la exclusión mutua, basándose en el modelo de atención de clientes a un comercio, como puede ser una panadería, donde cada cliente que ingresa saca un número y espera a ser atendido, respetando el orden de entrada.

Cada uno de los procesos en la sección de entrada obtiene el número de ingreso, y luego espera hasta que tenga el menor número de ingreso para acceder a la sección crítica. En el caso que 2 o más procesos obtengan el mismo número de ingreso, el ganador es el proceso que tenga el menor número de identificación. El algoritmo del *panadero* se puede convertir en un algoritmo adaptivo. En la figura 3.16 se muestra el código del algoritmo con la modificación especificada.

La adaptación del algoritmo utiliza el concepto de conjunto activo. Un objeto del conjunto activo soporta las siguientes operaciones:

- *join*(): se utiliza para unirse al conjunto
- *leave*(): se utiliza para abandonar el conjunto
- *get_set*(): se utiliza para retornar el conjunto de procesos activos. Debe retornar todos los procesos que han terminado el *join*() antes que *get_set*() haya comenzado y no han comenzado *leave*() durante el *get_set*(). Tampoco debe retornar cualquier proceso que ha finalizado *leave*() antes que haya comenzado el *get_set*() y no ha comenzado el *join*() durante el *get_set*().

El algoritmo observado en la figura 3.16 garantiza la exclusión mutua, el progreso y la inanición; además presenta la característica de que solamente se tienen en cuenta en la sección de entrada los procesos que forman parte del conjunto activo. La demostración aparece en [4].

inicialmente Number[i] = 0
Choosing[i] = false, para i, 1≤i≤N

Proceso i

** Sección Resto **

Try i

join();
Choosing [i] : = true ;
S:= get_set();
Number[i] := 1 + max $_{j \in S}$ (Number[j]);
Choosing [i] := false;
S:= get_set();
para cada j ∈ S hacer
  waitfor (Choosing[j] = false);
  waitfor (Number[j] = 0) or ((Number[j],j) > (Number[i],i));

Crit i

** Sección Crítica **

Exit i

Number[i] := 0;
leave();

Rem i

Figura 3.16: Algoritmo Adaptivo del Panadero

3.4. k-Exclusión Mutua

El problema de k-Exclusión mutua es garantizar que el sistema no ingrese en un estado global en el cuál más de k procesos estén en su sección crítica. Puede ocurrir esta situación, si en el sistema existen k unidades de un recurso y los procesos pueden utilizarlos en forma indistinta y el uso del mismo es excluyente.

La solución más simple para esta situación es distinguir cada instancia del recurso y resolverlo utilizando un algoritmo 1-exclusión mutua. Esta simple propuesta requiere que cada proceso deba especificar cuál de las instancias solicita, generando que varios procesos seleccionen la misma instancia, y permitiendo la existencia de pedidos pendientes y recursos disponibles. Una primera solución propuesta para el problema extendido fue presentada por Fisher y otros [56].

Los algoritmos de k-Exclusión Mutua se pueden extender para resolver el problema de k-asignación. Para realizar esta extensión se requiere introducir la utilización de un algoritmo de *long-lived renaming*, esto es, cada proceso adquiere un único nombre antes de ingresar a la sección crítica.

3.4.1. Una solución limitada First-In, First-Enabled para k-Exclusión Mutua

Afek y otros [3] presentan una solución basada en el modelo *first-come, first-server* propuesto por Lamport. Para coordinar su entrada en la sección crítica este algoritmo se basa en la siguiente idea:

Proceso$_i$

Repetir

Sección Resto$_i$
doorway$_i$ (este paso siempre tiene un número limitado de pasos)
waiting_room$_i$ (puede tener un número no limitado de pasos)
entrada$_i$
Sección Crítica$_i$
exit$_i$

Fin Repetir

Se pueden considerar las siguientes propiedades.

k-lockout avoidance: si menos que k procesos están fallando, cualquier proceso que no ha fallado y abandona la sección resto, eventualmente en un futuro reingresará en ella.

First-Come, First-Served: si el proceso i abandona el *doorway* antes que el proceso j ingresa al *doorway* entonces i ingresa a la sección crítica antes que j.

x_1, ..., x_n : variables compartidas
y_1, ..., y_n, S_l, L_l, $Test_l$: variables locales

0. sección $resto_l$

1. x_l := true; /* Comienzo del doorway */
2. $Label_l$;
3. $S_l := \emptyset$;
4. para toda j $\in$ {1, ... n} hacer
   si x_j = true entonces $S_l := S_l \cup$ {j}; /* Fin del doorway */
5. repetir
6. para toda j $\in S_l$ hacer $y_j := x_j$;
7. $L_l := Scan_l$;
8. $Test_l := \{j \in S_l \mid y_l = true \wedge L_l[j] < L_l[i]\}$
9. hasta ($|Test_l| < k$);

10. sección crítica

11. x_l := false;
12. $Label_l$
Fin Repetir

Figura 3.17: Algoritmo First-in, first-enabled k-exclusion mutua - Código para el proceso l

Las propiedades enunciadas no se pueden satisfacer conjuntamente cuando se considera la extensión al problema de la exclusión mutua, cuando $k > 1$. Para resolver esta situación hay que debilitar la propiedad *First-Come, First-Served*. En vez de considerar atender el proceso que ha llegado primero, se considera el primer proceso habilitado.

> *First-in, First-Enabled*: si i es el último que abandona el *doorway* antes que el último ingresado j, i está en *waiting-room*, y j está en la sección crítica, entonces i está habilitado.

El algoritmo aparece en la figura 3.17. La propiedad de first-in, first-enabled ser puede definir también de la siguiente manera: si i es el último deja el *doorway* antes que j sea el último que ingresa en él, i y j están en el *waiting-room*, y j está habilitado, entonces i está habilitado.

3.5. Resumen

El paradigma de memoria compartida presenta la ventaja de abstraerse de la ubicación de los recursos en el sistema distribuido, propiedad de transparencia que facilita el trabajo a los diseñadores y programadores al momento de desarollar una aplicación, pero presenta un alto costo en la implementación.

En este capítulo se muestran un conjunto de algoritmos que resuelven el problema de exclusión mutua para 1 proceso utilizando memoria compartida y una breve presentación para resolver el problema de la k-exclusión mutua. En estos algoritmos, el mayor costo

en tiempo es el acceso del procesador a memoria. Para poder reducir este costo, surgen algunos algoritmos de exclusión mutua denominados rápidos que utilizan una cantidad fija de accesos para ingresar a la sección crítica pero solamente garantizan esta propiedad cuando no existen oponentes activos. Otra forma de reducir el costo es considerando que los accesos a memoria se pueden clasificar en locales o remotos, haciendo que las esperas ocupadas se pueden realizar sobre variables compartidas ubicadas localmente. De esta manera, se reduce el tráfico en la red y se puede realizar un mejor escalamiento.

Capítulo 4

Algoritmos de Exclusión Mutua basados en Mensajes

En este capítulo, se consideran soluciones basadas en mensajes que satisfagan las propiedades de un buen algoritmo para resolver el problema de exclusión mutua. Se considera que se tienen n usuarios U_1, ..., U_n (definidos como E/S autómata garantizando la buena formación). Se asume que se tiene un sistema A que resuelve el problema en una red asincrónica, donde cada proceso P_i corresponde a un usuario U_i, como se observa en la figura 4.1. La comunicación entre los componentes se realiza sobre un medio confiable.

En los algoritmos se analiza la comunicación y complejidad en el tiempo para realizar un requerimiento. En algunos casos se pueden simular los algoritmos de memoria compartida en el Modelo de Mensajes, un algoritmo que se puede implementar razonablemente es el del *Panadero*.

4.1. Modelos Básicos

Los algoritmos de exclusión mutua basados en mensajes se los puede clasificar en centralizado o descentralizado. Una aproximación centralizada designa un nodo N_m, donde $0 \leq m < n$, para que realice la tarea de coordinar los accesos al recurso (objeto) compartido. Si el nodo N_i quiere acceder en forma exclusiva a un recurso entonces envía un mensaje de solicitud, REQUEST(N_i), al nodo N_m. Espera hasta que recibe el mensaje de permiso, GRANT, desde N_m para ejecutar la sección crítica. Cuando finaliza, envía un mensaje de liberación, RELEASE a N_m.

Los algoritmos descentralizados se los puede clasificar según como se implementa el protocolo: basados en *tokens* o en *quorum*, y la selección del tipo de algoritmo se verá afectada de acuerdo a la topología de la red.

El comportamiento general de un algoritmo basado en quorum es el siguiente: un nodo (sitio) debe obtener permiso (voto) desde todos los nodos del quorum para ingresar a la sección crítica. La exclusión mutua se garantiza si se asegura que cualquier par de quorums tiene al menos un nodo en común.

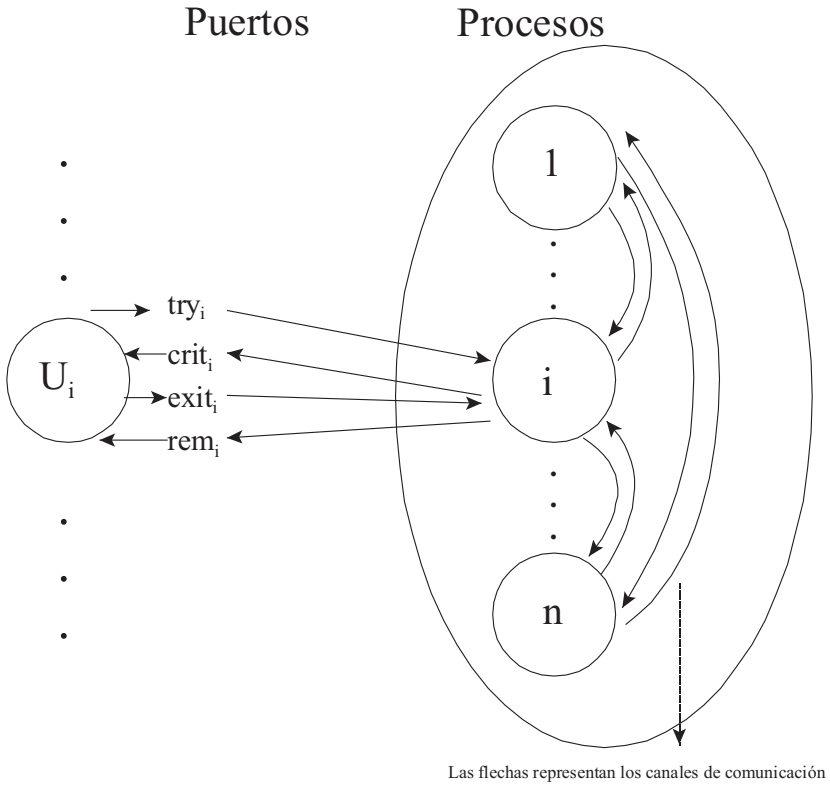


Figura 4.1: Interacciones entre los componentes

4.1.1. Algoritmo de Circulating Token

El algoritmo de Circulating Token, presentado en [92], es el más simple para exclusión mutua en una red asincrónica send/receive y funciona cuando la red es un anillo unidireccional. El algoritmo se comporta de la siguiente manera: un *token* representando el control del recurso circula continuamente a través del anillo. Cuando un proceso P_i recibe el *token*, chequea si hay pendiente un requerimiento desde un usuario U_i. Si no hay tal requerimiento, P_i pasa el *token* a P_{i+1}. En el caso que hubiera un requerimiento pendiente, entonces P_i otorga el recurso a U_i y mantiene el *token* hasta que U_i devuelve el recurso. Sucedido ésto, P_i pasa el *token* a P_{i+1}.

4.1.2. Algoritmo Basado en el Tiempo Lógico

Este algoritmo genera tiempos lógicos para eventos utilizando la estrategia de tiempo de Lamport [88], basado en valores de relojes enteros no negativos. Un tiempo lógico es un par (c, i), donde $c \in \mathrm{N}$ e i es un índice que indica proceso; pares de tiempo lógicos son ordenados lexicográficamente.

El algoritmo utiliza tanto comunicación broadcast como send/receive, donde la comunicación send/receive es permitida a todos los pares de procesos diferentes. Cada proceso P_i mantiene una historia de requerimientos H_i en la cual registra todos los mensajes recibidos con un número no negativo c, el cual es el valor del reloj asociado con un evento de un mensaje *broadcast* o *send*. Los requerimientos de *try* y *exit* son *broadcast*. En vez de tener mensajes dummy *broadcast*, cada proceso reconoce cada mensaje *try* con un mensaje *ack*.

El comportamiento del agoritmo se define a través de las siguientes reglas.

1. Para acceder a la sección crítica (requerir el recurso), el proceso P_i envía una solicitud a través de un mensaje *broadcast* $M(P_i, T_m)$, donde (T_m) es la estampilla de tiempo. El proceso P_i agrega la solicitud a su historia de requerimientos.

2. Cuando el proceso P_j recibe el mensaje de requerimiento $M(P_i, T_m)$, lo agrega a la historia de requerimientos y envía un mensaje de *ack* incluyendo la estampilla de tiempo a P_i.

3. Para salir de la sección crítica (liberar el recurso), el proceso P_i elimina de su historia de requerimientos el mensaje de solicitud $M(P_i, T_m)$ y envía un mensaje *broadcast* de liberación del recurso $Ml(P_i, T_m)$, donde (T_m) es la estampilla de tiempo.

4. Cuando el proceso P_j recibe el mensaje $Ml(P_i, T_m)$, remueve de su historia de requerimientos el mensaje $M(P_i, T_m)$.

5. El proceso P_i puede acceder a la sección crítica (utilizar el recurso) cuando las siguientes condiciones se satisfacen:

   a) Hay un mensaje $M(P_i, T_m)$ en la historia de requerimientos y todos los otros mensajes de requerimientos el valor de la estampilla de tiempo es superior.

b) P_i ha recibido un mensaje desde cada uno de los otros procesos con estampillas de tiempo superior que T_m.

Este algoritmo requiere $3n - 1$ mensajes por requerimiento.

4.1.3. Algoritmo de Ricart Agrawala

El algoritmo que presentan Ricart y Agrawala [125], es una mejora del algoritmo presentado en la sección 4.1.2, necesitando $2n-1$ mensajes por requerimiento. El algoritmo utiliza comunicación *broadcast* y *send/receive*, donde la comunicación *send/receive* es permitida a todos los pares de procesos distintos.

Los tiempos lógicos para eventos son generados de la misma manera que en el algoritmo Basado en el Tiempo (BT). El único mensaje que es *broadcast* es *try*, y el único mensaje que es enviado en un canal *send/receive* es *ok*. Cada mensaje transporta el valor del reloj de su evento *bcast* o *send*.

Después de una entrada try_i, P_i envía un mensaje *broadcast try* como en BT y puede ir a la sección crítica (*Crit*) después que recibe la subsecuencia de mensajes *ok* desde todos los otros procesos. La parte interesante del algoritmo es la regla para cuando un proceso P_i puede enviar un mensaje *ok* a otro proceso P_j. La idea es utilizar un esquema de prioridades. En respuesta a un mensaje *try* desde P_j, P_i realiza lo siguiente:

1. Si P_i está en *Salida*, *Resto*, ó *Entrada* antes de enviar su mensaje de requerimiento broadcast *try*, entonces P_i responde con *ok*.

2. Si P_i está en *Crítica*, entonces P_i demora su respuesta hasta que alcanza *Salida*, y entonces envía inmediatamente todos los *ok* demorados.

3. Si P_i está en *Entrada* y su requerimiento actual ha sido enviado (broadcast), entonces P_i compara el tiempo lógico t_i del evento *bcast* de su propio requerimiento con el tiempo lógico t_j asociado con el mensaje entrante de P_j. Si $t_i > t_j$, entonces el requerimiento propio de P_i se le ha dado una menor prioridad y P_i responde con un mensaje *ok*. Por el contrario, si el requerimiento propio de P_i tiene una prioridad mayor, entonces demora la respuesta hasta que haya finalizado su próxima sección crítica. En ese momento, envía inmediatamente todos los *ok* demorados. P_i puede ejecutar rem_i en cualquier momento después de recibir un $exit_i$. Cuando existe un conflicto el algoritmo resuelve en favor del *primer* requerimiento, como lo determinan los relojes lógicos.

Este algoritmo garantiza las propiedades de un buen algoritmo: exclusión mutua, libre de interbloqueo y libre de inanición.

4.2. Algoritmos de Exclusión Mutua

4.2.1. Modelos Basados en Quorum

Definiciones Quorum, Coteries, k-Coteries, k-arbiters

Un quorum Q es un subconjunto no vacío de procesos/nodos correspondientes a U, donde U es el conjunto total de procesos/nodos. El concepto de *coterie* fue introducido por García-Molina y Barbara [58], como motivación para diseñar un algoritmo de exclusión mutua tanto que sea eficiente en la complejidad de mensajes como robusto con respecto a fallas de red. Un coterie es un conjunto de grupos que están bien formados, considerando a un grupo como un conjunto de procesos/nodos.

Un conjunto C es denominado coterie bajo U si se cumplen las siguientes condiciones.

- No vacío: $\forall Q_i \in C : Q_i \neq \emptyset$
- Intersección: $\forall Q_i, \forall Q_j : Q_i, Q_j \in C : Q_i \cap Q_j \neq \emptyset$
- Minimalidad (Minimality): $\forall Q_i, \forall Q_j : Q_i, Q_j \in C : Q_i \not\subseteq Q_j$

Por ejemplo, C = { {u1, u2}, {u1, u3}, {u2, u3} } es un coterie en U = {u1, u2, u3} porque cada par de conjuntos en C tiene una intersección no nula, y no hay conjunto en C que incluya a otro conjunto en C. Ejemplos de coterie son: coterie por mayoría [135], coterie de Maekawa [99].

Dados C y D dos coteries diferentes, García-Molina y Barbara [58], definen: C domina D si $\forall R \in D,\ \exists S \in C,\ S \subseteq R$.

Por ejemplo, coterie C = {{u_1, u_2}, {u_1, u_3}, {u_1, u_4}, {u_2, u_3, u_4}} domina al coterie D = {{u_1, u_2, u_3}, {u_1, u_2, u_4}, {u_1, u_3, u_4}, {u_2, u_3, u_4}} porque para cada quorum R perteneciente a D se puede encontrar un quorum S perteneciente a C tal que S es un subconjunto de R.

Un coterie dominante, como C, es más adaptable a una falla de nodo y/o link (vínculo) que un coterie dominado, como D, ya que si un quorum se puede formar en un coterie dominado también puede formarse en un coterie dominador. Por ejemplo, si falla un link y el conjunto de nodos se divide en {u_1,u_3} y {u_2,u_4}, en el caso de utilizar el coterie C se puede obtener un quorum {u_1,u_3} y con el coterie D no se puede alcanzar ningún quorum. Un coterie se dice que es *no dominado* (*ND*) si no hay coterie que lo pueda dominar.

El siguiente terorema, presentado y demostrado en [58], permite chequear si un coterie es *no dominado* (ND).

Teorema 4.1 *Sea C un coterie bajo U. Entonces, C es dominado si solo si existe un conjunto $S \in U$ tal que:*

L1. S no es un superconjunto para cualquier grupo en C. Esto es, para cualquier quorum $R \in C$, $R \not\subseteq S$

L2. S presenta la propiedad de intersección. Esto es, para cualquier quorum $R \in C$, $S \bigcap R \neq \emptyset$

Si no se puede encontrar un subconjunto de U que satisfaga L1 y L2 para un coterie, entonces es no dominado. Existen distintos tipos de ND-coteries desarrollados para resolver el problema de exclusión mutua como por ejemplo coterie de mayoría [135], coterie de árbol [6], coterie de composición [111], coterie de nivel [129], coterie de Lovasz [114].

En sistemas verdaderamente distribuidos es deseable que el impacto de una falla de un proceso (en términos de quorums no disponibles) sea la misma para todos los procesos. Un coterie C bajo U es simétrico si las siguientes propiedades se mantienen:

- Propiedad de igualdad de trabajo (esfuerzo): para cada $p_i \in U$ vale que $|\{j|i \in Q_j\}| = \beta$, esto es, todos los procesos están contenidos en el mismo número de quorums β.
- Propiedad de igualdad de tamaño: para cada quorum Q_i vale que $|Q_i| = \gamma$, esto es, todos los quorums son del mismo tamaño γ.

Un k-coterie C es una familia de subconjuntos de U, que mantiene las siguientes propiedades [77].

- No intersección. Para cualquier h ($< k$) quorums, $Q_1, Q_2, ...Q_h \in C$ tal que $Q_i \bigcap Q_j = \emptyset$ (para $1 \leq i \neq j \leq h$), existe $Q \in C$ tal que $Q \bigcap Q_i = \emptyset$ para i ($1 \leq i \leq h$)
- Intersección. Para cualquier $k+1$ quorums $Q_1, Q_2, ...Q_{k+1} \in C$, existe un par Q_i,Q_j ($1 \leq j \neq i \leq k+1$) tal que $Q_i \bigcap Q_j \neq \emptyset$
- Minimalidad (*Minimality*). Para cualquier par de diferentes quorums $Q_i, Q_j \in C$ vale que $Q_i \not\subseteq Q_j$

Por ejemplo {{1,3}, {1,4}, {2,3}, {2,4}} es un 2-coterie porque se pueden encontrar 2 quorums mutuamente disjuntos, tales como {1,3}, {2,4} ó {1,4}, {2,3}, pero no más. La propiedad de intersección se la puede utilizar para desarrollar algoritmos que garanticen k-entradas en la sección crítica, y éstos se pueden aplicar en la resolución del problema de k-exclusión mutua.

Para ingresar a la sección crítica, un nodo requiere obtener permisos desde todos los nodos de algún quorum. Por la propiedad de intersección, no más de k nodos pueden formar quorums simultáneamente, y entonces no más que k nodos pueden acceder a la sección crítica al mismo tiempo. La propiedad de no intersección asegura que si existe una entrada de sección crítica no ocupada, entonces algún nodo que espera para ingresar a la sección crítica puede proceder. La propiedad de minimalidad (*minimality*) no está relacionada con la correctitud del algoritmo de k-exclusión mutua; está solamente para mantener la eficiencia. Los algoritmos de k-exclusión mutua que utilizan k-coteries son tolerantes a fallas en el sentido que cuando un nodo esté inaccesible en el sistema, quorums que no lo incluyan pueden encontrarse.

Los k-coteries son una extensión de los coteries y también se pueden obtener estos con la características de ND k-coteries.

Hay definiciones similares de ND k-coteries,como las que aparecen en [115], aunque aquí se presenta la definición de [71]. Un k-coterie se dice que domina otro k-coterie si y solo si cada quorum en el dominado es un superconjunto de algún quorum del que domina. Sea C y D dos k-coteries, D domina C si y solo si, $\{C \neq D$ y $(\forall\ R \in C)\ [\exists\ S \in D, S \subseteq R]\}$ Por ejemplo, considerando los siguientes 2-coteries:

A = {{1,2}, {3,4}, {1,3}, {2,4} }

B = {{1,2}, {3,4}, {1,3}, {2,4}, {1,4}, {2,3}}

C = {{1,2}, {1,3}, {2,3}, {4}}

En este caso A está dominado por B y C, y que B es dominado por C. Un k-coterie dominante es superior a un k-coterie dominado, ya que si se puede formar un quorum en el último también se podrá en el primero. Como los ND k-coteries, presentan una mejor adaptación a las fallas de un nodo ó link, es importante poder determinar si un k-coterie presenta esta propiedad. En [71] se presenta y demuestra el siguiente teorema.

Teorema 4.2 *Sea C un k-coterie bajo U. Entonces, C es dominado si solo si existe un conjunto $S \in U$ tal que:*

L1. Para cualquier quorum $R \in C$, $R \not\subseteq S$.

L2. Para cualquier k par de quorums disjuntos R_1, ...,$R_k \in C$, R_1, ..., R_k y S no son pares disjuntos.

Si no se puede encontrar un subconjunto de U que satisfaga L1 y L2 para un k-coterie, entonces es no dominado. Sin embargo, la existencia de un conjunto que satisfaga L1 y L2 para un k-coterie C no implica que sea dominado.

Baldoni y otros [25] presentan el concepto de k-arbiter para resolver el problema de h-requeridos de-k (h-*out of*-k) exclusión mutua. Un conjunto de quorums CQ = {Q_1, Q_2, ..., Q_m} es un k-arbiter bajo U si las siguientes propiedades se mantienen.

- Intersección: para cualquier (k+1) quorum, Q_{i_1}, Q_{i_2}, ..., $Q_{i_{k+1}} \in$ CQ tal que $\bigcap_{1 \leq j \leq k+1} Q_{i_j} \neq \emptyset$.
- Minimalidad: para cualquier par de distintos quorums, Q_i, $Q_j \in$ CQ tal que $Q_i \not\subseteq Q_j$

Ejemplos de k-arbiter:

- Un conjunto C de quorums tal que C = {{p_i}} es denominado un *singleton* (único de un tipo) y es un k-arbiter.

- Un conjunto C de quorums tal que C = {Q ⊂ U $||Q| = \lfloor k.n/(k+1) \rfloor + 1$} es denominado uniforme y es un k-arbiter.

Sea C y D dos k-arbiters bajo U. C domina a D si C ≠ D y para cada $Q_i \in D$ hay un $Q_j \in C$ tal que $Q_j \subseteq Q_i$. Un k-arbiter C es dominado si existe un k-arbiter que domina C. En caso contrario C es un ND k-arbiter. El siguiente teorema presentado en [25], permite chequear si un k-arbiter es ND.

Teorema 4.3 *Sea C un k-arbiter bajo U. Entonces, C es dominado si y solo si existe un quorum H tal que:*

L1. Para cualquier $Q \in C$, $Q \not\subseteq H$

L2. Para cualquier k quorums Q_1, Q_2, ..., $Q_k \in C$, $H \cap (\bigcap_{j=1}^{k} Q_j) \neq \emptyset$

Al igual que con coterie, se define k-arbiter simétrico [25]. Un k-arbiter C bajo U es simétrico si se mantienen las siguientes propiedades:

- Propiedad de igualdad de trabajo (esfuerzo): para cada $p_i \in U$ vale que $|\{j|i \in Q_j\}| = \beta$, esto es, todos los procesos están contenidos en el mismo número de quorums β.
- Propiedad de igualdad de tamaño: para cada quorum Q_i vale que $|Q_i| = \gamma$, esto es, todos los quorums son del mismo tamaño γ.

Algoritmo de Maekawa

El algoritmo de Maekawa [99] resuelve el problema tradicional de exclusión mutua utilizando mensajes para la comunicación entre los procesos, con la característica de que el acceso al recurso se alcanza a través de quorum (subconjunto del total de procesos/nodos). En el modelo se pueden tener un conjunto N de quorums, denominados S_1 ... S_N.

Los requerimientos para la resolución, teniendo en cuenta que el algoritmo solicita *lock* a su quorum, son los siguientes:

(a) para cualquier combinación de i y j, $1 \leq i, j \leq N$, $S_i \cap S_j \neq \emptyset$

Esta condición especifica que la intersección entre cualquier par de quorums es no nula, garantizando que no se le otorgue al mismo tiempo todos los *locks* a dos procesos para ingresar a la exclusión mutua. Esta es una condición necesaria para que funcione el algoritmo.

Para considerar un algoritmo verdaderamente distribuido, se requieren de las siguientes propiedades:

(b) S_i, $1 \leq i \leq N$, siempre contiene a i

(c) El tamaño de S_i, $|S_i|$, es k para cualquier i. Esto es, $|S_1|=|S_2|=|S_3|= ... =|S_N|=k$

(d) Cualquier j, $1 \leq j \leq N$, está contenido en el D S_i's, $1 \leq i \leq N$

La elección de los S_i's es una de las claves para que el algoritmo resuelva el problema de la exclusión mutua y además sea distribuido, con igual cantidad de responsabilidad para controlar la exclusión mutua, realizando la misma cantidad de trabajo. Maekawa demostró que para un valor fijo k, el máximo valor posible de N en el cual todas las propiedades se satisfacen es igual a $k(k-1)+1$, asumiento que cualquier par de quorums tienen solamente un nodo de intersección. Teóricamente el límite inferior es aproximadamente igual a $\sqrt{N}$. Para encontrar la posible solución para $N = k(k-1)+1$ es equivalente a encontrar finitos planos proyectivos de orden n, donde $n = k-1$. Sin embargo, no existen todos los finitos planos proyectivos y sólo se conoce como construir aquellos con potencia prima de orden $n = p^i$, donde p es un número primo e i es un entero [65].

El algoritmo se basa en el hecho de que, si un nodo i obtiene los *locks* de todos los miembros de S_i, ningún otro nodo puede obtener los *locks* de todos los miembros por la propiedad (a). Cuando se invoca exclusión mutua, el nodo i trata de obtener el *lock* (*bloquear*) todos los miembros de S_i. Si tiene éxito, entonces puede ingresar a la sección crítica. Si falla, espera a que todos los nodos miembros del quorum liberen su *lock* y se lo otorguen para que luego pueda ingresar a la sección crítica.

En el caso que más de un nodo quiera ingresar al mismo tiempo a la sección crítica, todos estos nodos mandan mensajes de solicitud de *lock* a sus respectivos quorums. Si cada nodo receptor del mensaje tiene disponible el *lock* lo otorga al nodo correspondiente; en el caso que los mensajes arriben en diferente orden a los nodos podría suceder que todos los nodos que quieren acceder a la sección crítica estén esperando por un *lock* que lo tiene otro y así sucesivamente generando una espera indefinida denominada interbloqueo. Para evitar el interbloqueo (*deadlock*) cuando más de un nodo inicia simultáneamente requerimientos de exclusión mutua, cada requerimiento tiene asignada una prioridad. Un nodo va a ganar a los otros si su requerimiento tiene una prioridad mayor que el resto de los requerimientos conflictivos. La prioridad del requerimiento es determinada por un número de secuencia (estampilla de tiempo) y el nodo correspondiente.

El algoritmo garantiza la exclusión mutua y además cumple las condiciones de un buen algoritmo, esto es, el progreso, libre de interbloqueo y libre de inanición. La complejidad de mensajes es de $c\sqrt{n}$ donde c es una constante entre 3 y 5.

Algoritmo de Sanders

Sanders en [126] introduce el concepto de estructura de la información (*information structure*) aplicado a un algoritmo. En este caso, la estructura de la información describe cuáles procesos mantienen información de estado sobre otros procesos, y para cada proceso, el conjunto de procesos desde el cual se requerirán información o permiso antes de que ingrese a la sección crítica.

Formalmente, la estructura de la información puede ser descripta por pares de subconjuntos de procesos ID asociados con cada proceso. El *conjunto informe* para el proceso i es denominado I_i y el *conjunto requerimiento* para i es denominado R_i. En general, un proceso envía mensajes a cada proceso de su *conjunto informe* cada vez que cambia el estado desde SECCIÓN CRÍTICA a SALIDA o desde RESTO a ENTRADA.

Se define también para cada proceso, el *conjunto estado* S_i. Los conjuntos estado están determinados por los conjuntos informe donde $j \in S_i$ si $i \in I_j$. El conjunto estado S_i indica el conjunto de procesos sobre los cuales el proceso i mantiene información.

Estos conceptos se pueden aplicar a los algoritmos que ya se han presentado, como por ejemplo:

1. Algoritmo Centralizado.

   Nodo 0: $I_0 = \{1\}$, $R_0 = \{1\}$

   Nodo 1: $I_1 = \{1\}$, $R_1 = \{1\}$, $S_1 = \{0, 1, 2, 3\}$

   Nodo 2: $I_2 = \{1\}$, $R_2 = \{1\}$

   Nodo 3: $I_3 = \{1\}$, $R_3 = \{1\}$

2. Algoritmo Ricart-Agrawala - Completamente Distribuido, esto es, solicita permiso a todos los nodos.

   Nodo 0: $I_0 = \{0\}$, $R_0 = \{0, 1, 2, 3\}$

   Nodo 1: $I_1 = \{1\}$, $R_1 = \{0, 1, 2, 3\}$

   Nodo 2: $I_2 = \{2\}$, $R_2 = \{0, 1, 2, 3\}$

   Nodo 3: $I_3 = \{3\}$, $R_3 = \{0, 1, 2, 3\}$

3. Algoritmo Maekawa - Distribuido, esto es, solicita permiso a un subconjunto de todos los nodos.

   Nodo 0: $I_0 = \{0, 1, 2\}$, $R_0 = \{0, 1, 2\}$

   Nodo 1: $I_1 = \{0, 1, 3\}$, $R_1 = \{0, 1, 3\}$

   Nodo 2: $I_2 = \{0, 2, 3\}$, $R_2 = \{0, 2, 3\}$

   Nodo 3: $I_3 = \{1, 2, 3\}$, $R_3 = \{1, 2, 3\}$

Se pueden construir algoritmos a partir de diferentes elecciones en los conjuntos informe y requerimiento. Esta elección no puede ser arbitraria, ya que el resultado no necesariamente sería un algoritmo correcto.

El siguiente teorema, demostrado en [126] presenta las condiciones necesarias y suficientes.

Teorema 4.4 *Dado* $\forall i,\ i \in I_i$, *las propiedades a y b juntas son necesarias y suficientes para realizar un algoritmo que garantice exclusión mutua.*

a. I_i *es un elemento de* R_i

b. $\forall i, j,\ I_i \cap I_j \neq \emptyset$ *o bien* $j \in R_i$ *e* $i \in R_j$

La propiedad (a) requiere que un proceso solicite permiso a cada miembro de su conjunto informe antes de ingresar a la sección crítica. La propiedad (b) garantiza que un proceso i tenga información sobre los otros procesos antes de ingresar a la sección crítica, tanto por comunicación directa con ellos o mediante la comunicación con un proceso que mantiene información actualizada sobre éstos.

Se muestran a continuación las acciones que realiza en la sección de entrada y de salida.

Sección Entrada

1. El proceso selecciona la estampilla de tiempo que adjuntará en el mensaje de REQUEST que enviará a cada proceso perteneciente al conjunto R_i.
2. El proceso espera por mensajes GRANT de cada proceso en R_i. Cuando tiene todos los permisos que fueron recibidos a través de los mensajes GRANT, entonces puede ingresar a la *sección crítica*

Sección Salida

1. Envía un mensaje RELEASE a todos los procesos en el conjunto infome I_i.

En el momento que un proceso recibe un mensaje debe atenderlo y de acuerdo al tipo del mismo y de su estado actual será el conjunto de acciones que llevará a cabo. Cada uno de los procesos tiene variables locales que se verán afectadas con la llegada de un mensaje. Las variables locales son:

- *coladeprioridades*: contiene la lista de procesos desde los cuales se ha recibido un mensaje REQUEST pero no se ha recibido aún un mensaje GRANT.
- *csstat* indica al proceso el estado o quien conoce mejor el estado de la sección crítica. Puede tener el valor libre (FREE) o la identidad del proceso al cual se le ha enviado un mensaje GRANT y que aún no ha enviado un mensaje RELEASE.

A continuación se muestra el comportamiento del algoritmo cuando recibe los mensajes y como garantiza que el mismo este libre de interbloqueo.

Cuando se recibe un mensaje REQUEST:

1. El ID del proceso es ubicado en la cola de prioridades.
2. Si *csstat* indica que la sección crítica no está libre, entonces es comparada la estampilla de tiempo del proceso en la sección crítica con la del proceso solicitante. Si la estampilla del proceso en la sección crítica es menor, entonces un mensaje FAIL es enviado al proceso solicitante. En el otro caso, un mensaje INQUIRE es enviado al proceso indicado por *csstat*, a menos que ya se le haya enviado uno. Un mensaje FAIL es enviado a cualquier proceso en la cola de prioridades con una estampilla de tiempo mayor que aún no se le haya enviado un mensaje FAIL.

3. Si *csstat* indica que la sección crítica está libre, entonces un mensaje GRANT es enviado al proceso ubicado al principio de la cola y el proceso es removido de la misma. Si el receptor del mensaje GRANT está en S_i, entonces *csstat* se le asigna ese proceso para indicar que está en la sección crítica.

Cuando se recibe un mensaje RELEASE:

1. *csstat* toma el valor FREE.
2. Un mensaje GRANT es enviado al proceso ubicado al principio de la cola, si este existe, y el proceso es removido de la cola. Si el receptor del mensaje GRANT está en S_i, entonces a *csstat* se le asigna ese proceso para indicar que está en la sección crítica.
3. El paso 2 se repite hasta *csstat* indique un proceso que está en la sección crítica o hasta que la cola esté vacía.

Cuando se recibe un mensaje INQUIRE:

1. El proceso chequea los mensajes recibidos desde los procesos en su conjunto de requerimientos.
2. Si ha recibido un mensaje FAIL desde cualquier proceso, o ha enviado un mensaje YIELD a cualquier proceso y no ha recibido un nuevo mensaje GRANT, suspende (cancela) el mensaje GRANT correspondiente al proceso emisor del mensaje INQUIRE. Un mensaje YIELD es enviado al proceso emisor del mensaje INQUIRE.

Cuando se recibe un mensaje YIELD:

1. *csstat* es marcado FREE, y el proceso emisor del mensaje YIELD es retornado a la cola de prioridades en la ubicación que corresponde.
2. El proceso receptor procede como si hubiera recibido un mensaje RELEASE.

Dependiendo de la elección, la estructura de la información puede ser similar al algoritmo presentado por Maekawa [99]. La diferencia que presenta el algoritmo de Maekawa, es que no envía un mensaje FAIL al proceso involucrado en un intercambio INQUIRE/YIELD cuando arriba un nuevo REQUEST con una menor estampilla de tiempo pudiendo esto producir una situación de interbloqueo.

La performance del algoritmo está relacionada con la cantidad de mensajes requeridos y el tiempo de demora para acceder a la sección crítica. Para este algoritmo, la cantidad de mensajes requeridos va a estar relacionado con la cantidad de potenciales interbloqueos. El límite inferior se puede obtener de la siguiente manera:

$$|I_i - \{i\}| + 2(|R_i - \{i\}| \tag{4.1}$$

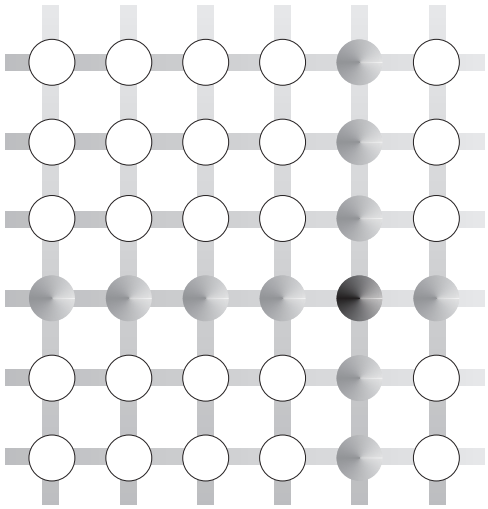


Figura 4.2: Grilla Cuadrada

Si se elige la estructura de información apropiadamente y se asemeja a [99], haciendo los reemplazos en la ecuación 4.1 la cantidad de mensajes sería $3\sqrt{N}$.

En el límite superior se deben incluir los mensajes adicionales necesitados para recuperarse de un posible interbloqueo (INQUIRE, YIELD y el nuevo GRANT).

$$|I_i - \{i\}| + 2(|R_i - \{i\}| + DM$$

$$donde\ DM = \sum_{j \in R_i} r_j\ y\ r_j \begin{cases} 1 & \text{si } S_j = \{j\}\ ó\ \{i,j\} \\ 4 & \text{en otro caso} \end{cases}$$

El intercambio de mensajes podría ocurrir a lo sumo con un miembro del conjunto estado para cada proceso j que sea miembro de R_i. Si S_j contiene solamente a j ó a i y j, entonces los mensajes extras no serán generados. De lo contrario, a lo sumo cero ó tres mensajes (INQUIRE, YIELD y el nuevo GRANT) son generados por cada proceso en R_i. El mensaje extra en cada término DM es de un posible mensaje FAIL.

Algoritmos Basados en Grillas

El modelo de grilla fue presentado por Maekawa [99] como otro método para obtener quorum para cada uno de los nodos. El mismo está organizado en un esquema de grilla con la forma de un cuadrado, como se muestra en la figura 4.2.

Un quorum para un requerimiento de un nodo incluye la unión de la fila y la columna a las cuales el nodo solicitante pertenece. De esta forma, el tamaño del quorum es el doble del límite inferior teórico, esto es, $k = 2\sqrt{N} - 1$.

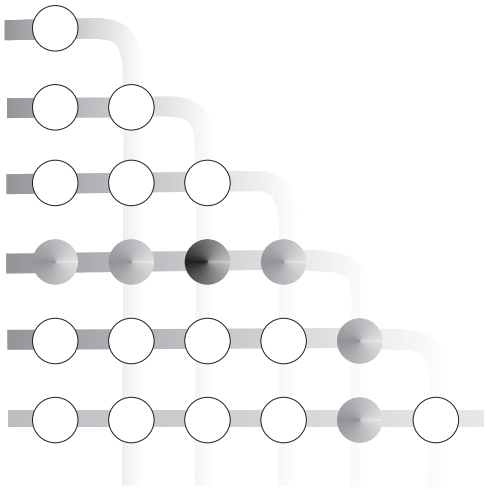


Figura 4.3: Grilla Triangular con fila seleccionada

La ventaja de este algoritmo es que es simple y geométricamente evidente, esto significa que las 4 propiedades, presentadas en la sección del algoritmo de Maekawa, se pueden verificar gráficamente. Esta solución no está bien optimizada en el sentido que S_i (el quorum del nodo P_i) intersecta con S_j (el quorum del nodo P_j) en 2 nodos para todo $i \neq j$.

Luk y Wong [94] consideran que cada quorum debería estar formado o por una fila o una columna. Reorganizan los nodos en forma de triángulo y, en la figura 4.3, se muestra el esquema de la grilla seleccionando una fila. Un quorum de *fila* es construido de la siguiente forma. Se comienza dibujando una línea desde el nodo más a la izquierda de la primera fila y se desplaza horizontalmente hacia la derecha. Cualquier nodo unido por esa línea está incluido en el quorum. Cuando no hay más nodos para unir a la derecha, la línea gira 90 grados y se desplaza hacia abajo. Finaliza cuando se posiciona en el nodo ubicado en el fondo de la grilla. Todos los nodos unidos por esta línea son incluidos en un quorum denominado F. De forma similar, se puede construir otro quorum comenzando la línea horizontal desde otra fila. El requerimiento del nodo P_i será enviado con un mensaje al quorum F_i, el cual corresponde con la línea horizontal que pasa por P_i.

Otra forma de armar los quorums es basándose en una *columna*. En la figura 4.4 se muestra el esquema y es construido de forma similar. Se comienza la línea desde el nodo inferior de la primera columna y se desplaza verticalmente hacia arriba. Gira 90 grados a la izquierda cuando no hay más nodos en el camino y finaliza cuando alcanza el nodo ubicado en el extremo izquierdo. Todos los nodos unidos por esa línea están incluidos en un quorum denominado C. En este caso, P_i enviará el requerimiento al quorum C_i, el cual corresponde a la línea que verticalmente pasa a través de él.

Cualquier par de líneas diferentes se intersectan en un único nodo. Por esto, cualquier par de quorums tiene exactamente un nodo de intersección. Más aún, todos los quorums tienen

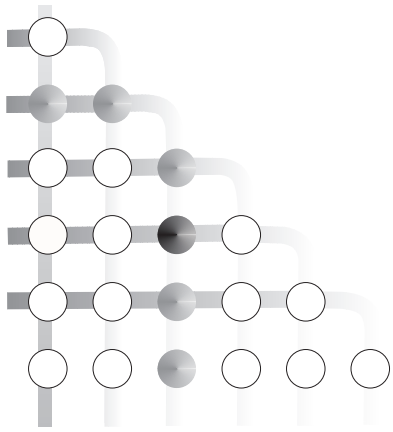


Figura 4.4: Grilla Triangular con columna seleccionada

el mismo tamaño. Pero la distribución de la responsabilidad no es equitativa, ya que algunos nodos formarán parte de mayor cantidad de quorums que otros, y esto está relacionado con el esquema elegido para la construcción de los mismos.

Para resolver el problema de la irregularidad de la responsabilidad, proponen que se alterne la utilización del esquema basado en columna y en fila para mantener la regularidad. Esto significa que cada nodo solicitante tendrá 2 quorums.

Por ejemplo, en el caso que sea $N = 10$ con la siguiente organización de los nodos:

P_1
P_2 P_3
P_4 P_5 P_6
P_7 P_8 P_9 P_{10}

Denominando F_i y C_i el quorum fila y el quorum columna para P_i respectivamente. La configuración del esquema será la siguiente:

$$
\begin{array}{ll}
F_1 = \{P_1, P_3, P_5, P_8\}, & C_1 = \{P_1, P_2, P_4, P_7\}, \\
F_2 = \{P_2, P_3, P_6, P_9\}, & C_2 = \{P_1, P_2, P_4, P_7\}, \\
F_3 = \{P_2, P_3, P_6, P_9\}, & C_3 = \{P_1, P_3, P_5, P_8\}, \\
F_4 = \{P_4, P_5, P_6, P_{10}\}, & C_4 = \{P_1, P_2, P_4, P_7\}, \\
F_5 = \{P_4, P_5, P_6, P_{10}\}, & C_5 = \{P_1, P_3, P_5, P_8\}, \\
F_6 = \{P_4, P_5, P_6, P_{10}\}, & C_6 = \{P_2, P_3, P_6, P_9\}, \\
F_7 = \{P_7, P_8, P_9, P_{10}\}, & C_7 = \{P_1, P_2, P_4, P_7\}, \\
F_8 = \{P_7, P_8, P_9, P_{10}\}, & C_8 = \{P_1, P_3, P_5, P_8\}, \\
F_9 = \{P_7, P_8, P_9, P_{10}\}, & C_9 = \{P_2, P_3, P_6, P_9\}, \\
F_{10} = \{P_7, P_8, P_9, P_{10}\}, & C_{10} = \{P_4, P_5, P_6, P_{10}\}.
\end{array}
$$

Cada nodo está contenido en 8 quorums (esto es, $2k$) y, de esta manera, se alcanza la

propiedad de equitatividad en la responsabilidad. El nodo solicitante no necesita notificar a los otros nodos cuál quorum está utilizando y por eso no se introduce sobrecarga en la comunicación. Para un número fijo de quorum k, el número máximo de nodos que este modelo puede manejar es $k(k+1)/2$. El quorum es aproximadamente $\sqrt{2N}$.

Algoritmo de Agrawal

Agrawal y Jalote [8] proponen un algoritmo basado en quorum que sea completamente distribuido, esto es, que garantice las condiciones propuestas por Maekawa. Asume que el número de nodos es $n(n-1)/2$. Del total de nodos se construyen n grupos de cardinalidad $(n-1)$, cada uno construido de la siguiente manera:

- Se generan todas las combinaciones de 2 números desde el conjunto $(1 \ldots n)$, y se obtendrán $n(n-1)/2$ combinaciones.
- Se hace un mapeo uno a uno desde el conjunto de nodos al conjunto de combinaciones generado. Si un nodo es mapeado en la combinación (i, j), está incluido en los grupos i-ésimo y j-ésimo y en ningún otro grupo.

Por ejemplo, en un sistema con 10 nodos, se forman 5 grupos con una cardinalidad de 4 miembros en cada uno. Después de generar las combinaciones y el mapeo, los grupos son los siguientes:

Grupo1 (P_1, P_2, P_3, P_4)

Grupo2 (P_1, P_5, P_6, P_7)

Grupo3 (P_2, P_5, P_8, P_9)

Grupo4 (P_3, P_6, P_8, P_{10})

Grupo5 (P_4, P_7, P_9, P_{10})

En los casos en que se cumpla la condición de que la cantidad de nodos L es equivalente a $n(n-1)/2$ entonces se obtendrá un conjunto de quorums que garanticen la completa distribución del algoritmo. Si el número de nodos L no cumple con la forma especificada, entonces se puede generar a partir de una aproximación, eligiendo el menor n tal que $L \leq n(n-1)/2$.

Hay 2 consideraciones a tener en cuenta para la evaluación de un algoritmo:

- El costo de la comunicación, que corresponde a la complejidad de los mensajes.
- La disponibilidad ofrecida por el protocolo, que relacionada con la capacidad de tolerar una falla y en esos casos considerar un quorum alternativo.

Con esta forma de construcción del quorum, si un nodo P_i pertenece a 2 grupos G_j y G_k, entonces debe elegir un grupo como *inicial* para solicitar quorum, en el caso que no pueda obtener una respuesta de todos los miembros del quorum (por falla de un nodo ó link), y puede utilizar el otro grupo para solicitar quorum. De esta forma, el algoritmo puede soportar la caída de un nodo ó link. Por ejemplo, si ocurriera la situación que solamente cae el nodo P_4, el resto de los nodos puede obtener el permiso para acceder en forma exclusiva a través de uno de los grupos al que pertenece, y la falla es soportada degradando en algunos casos las performance.

Algoritmo utilizando Cohorts

En los algoritmos que utilizan quorums para acceder a la sección crítica, un parámetro importante en el momento de elegir un algoritmo es la complejidad de mensajes, esto es, la cantidad de mensajes que son requeridos para acceder a la sección crítica y esto está directamente relacionado con el tamaño del quorum.

Jiang [66] presenta un algoritmo para resolver el problema de exclusión mutua a través de coteries y estos son generados a partir de una estructura lógica denominada *cohort*. Una estructura *Cohorts*, Coh(k) $\equiv \{C_1, ..., C_k\}$, es una familia de subconjuntos de U. Cada miembro C_i es denominado un *cohort* y debe observar las siguientes propiedades:

1. $|C_1| = 1$.
2. $\forall i,\ i \neq 1,\ |C_i| \geq 2$.
3. $\forall i, j,\ j \neq i, C_i \not\subset C_j$.

En resumen, el primer *cohort* en la estructura *Cohorts* debe tener un solo miembro y el resto al menos tener dos miembros y cada *cohort* por lo menos debe tener un miembro que no aparezca en ningún otro. Por ejemplo, $\{\{u_1\}\}$, $\{\{u_1\}, \{u_2, u_3, u_4\}\}$ y $\{\{u_1\}, \{u_2, u_3\}, \{u_3, u_4\}\}$ son estructuras *Cohorts*, *Coh*(1), *Coh*(2) y *Coh*(3) respectivamente.

Para una estructura *Cohorts Coh(k)* $= \{C_1, ..., C_k\}$, un conjunto Q se dice que es un quorum bajo *Coh(k)* si Q satisface tanto D1 y D2.

D1. Q contiene todos los miembros de algún *cohort* C_i, $1 \leq i \leq k$ (se dice que Q cubre completamente C_i o que C_i es un *cohort primario* de Q).

D2. Q contiene al menos un miembro de cada *cohort* C_j, $i \leq j \leq k$ (se dice que Q cubre C_j o que C_j es un *cohort soporte* de Q).

En la figura 4.5 se muestra la función *Get_Quorum* y en [66] está demostrado que es mínima y que la familia de quorum mínimo bajo *Coh*(k) constituye un ND coterie. Los quorums que se pueden generar a partir del *Coh*(3) $= \{\{u_1\}, \{u_2, u_3\}, \{u_3, u_4\}\}$ utilizando la función *Get_Quorum (Coh(3))* son:

Función *Get_Quorum*(Coh(k) = {C_1, ..., C_k}: Estructura Cohort): Set;
var R,S,T: Set;
R = ∅; // R: el conjunto de todos los nodos que han alcanzado el permiso
para (i = k, ..., 1) hacer
    S = C_i - R; // S: el conjunto de los nodos cuyos permisos son necesarios para hacer que C_i sea el cohort primario
    T = Obtain(S); // Obtain(S) tratará de obtener permiso desde los nodos de S y retornará el conjunto de nodos que pueden otorgar el permiso
    si T = S entonces retorna (Min(R ∪ T, C_i, ..., C_k)); C_i puede ser el cohort primario, y un quorum mínimo es retornado
    si (R ∪ T) ∩ C_i = ∅ entonces Exit(failure); // Ningún miembro de C_i otorga su permiso. Solicitud falla.
    si (C_i ∩ R) = ∅ entonces R = R ∪ {t}, donde t ∈ T. // R no cubre C_i entonces se agrega t de T para hacer que R cubra C_i.
finpara
exit(failure); // No se puede formar un quorum. Solicitud falla.
fin *Get_Quorum*

función *Min* (R, C_i, ..., C_k: Set): Set;
para (r ∈ R) hacer
    si cover(R - {r}, C_i, ..., C_k) entonces R = R - {r}
finpara
// Si r no es esencial en el cubrimiento de C_i, ..., C_k, se remueve r de R. Se asume Cover(R, C_i, ..., C_k) es un predicado que retorna verdadero si R cubre C_{i+1}, ..., C_k y completamente cubre C_i para algún i, sino retorna false
Retorna (R);
fin *Min*

Figura 4.5: Función para generar quorums mínimos bajo *Coh*(k)

```
Función GetQuorum (Árbol: RedJerárquica): QuorumSet;
  var
    izq, der: QuorumSet

  Si vacío(Árbol) Entonces
    Retornar({})
  Sino
    Si GrantsPermiso(Árbol.Nodo) Entonces /* Pueda dar el permiso. Formar parte del quorum */
      Retornar ({Árbol.Nodo} ∪ GetQuorum(Árbol.HijoIzq))
      ó
      Retornar ({Árbol.Nodo} ∪ GetQuorum(Árbol.HijoDer))
    Sino
      izq = GetQuorum(Árbol.HijoIzq)
      der = GetQuorum(Árbol.HijoDer)
      Si (izq = ∅ ∨ der = ∅) Entonces
        Exit(-1) /* no se puede establecer un quorum */
      Sino
        Retornar(izq ∪ der)
      FinSi
    FinSi
  FinSi
```

Figura 4.6: Función GetQuorum

$Q_1 = \{u_1, u_2, u_4\}$, $Q_2 = \{u_1, u_3\}$, $Q_3 = \{u_2, u_3\}$, $Q_4 = \{u_3, u_4\}$

Para un quorum bajo *Coh(k)* = $\{C_1, ..., C_k\}$, cuanto mayor es el índice del cohort primario, menor es el tamaño del quorum. En un caso extremo, si C_k es el cohort primario, entonces no es necesario un cohort soporte. En tal caso, el tamaño del quorum es una constante $|C_k|$. En otro caso extremo, si C_1 es el cohort primario con los otros cohorts como soporte, entonces el quorum puede ser de tamaño $O(N)$. En el mejor de los casos, un quorum bajo el *Coh*(k) es de tamaño constante, y en el peor de los casos de $O(N)$.

Algoritmo del Árbol

Agrawal y otros [5, 6] proponen una solución basada en quorum, que utiliza una estructura de árbol binario. El algoritmo para construir quorums puede utilizarse en cualquier árbol, pero por simplicidad, se lo considera completo, esto es, si tiene k niveles entonces hay $2^{k+1} - 1$ nodos. Un camino en el árbol es una secuencia de nodos $s_1, s_2, ..., s_i, s_{i+1}, ...s_n$, tal que s_{i+1} es un hijo de s_i. Para obtener un quorum válido, se utiliza la función GetQuorum que trata de construir el quorum seleccionando cualquier camino comenzando desde la raíz y finalizando en cualquier hoja. En la figura 4.6 se muestra el comportamiento de la función para obtener el quorum.

En el caso que un nodo falle, obtiene el quorum a través de sus 2 caminos alternativos. Con este modelo, cada nodo no tiene asignado un único conjunto quorum, sino cada vez

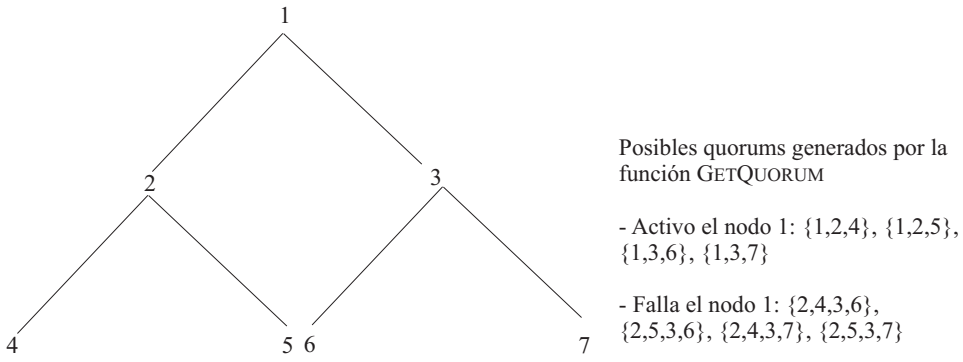


Figura 4.7: Ejemplo de un árbol binario y quorums generados

que quiere acceder a la sección crítica obtiene el quorum. En la figura 4.7 se muestra un ejemplo de 7 nodos y algunos quorums que se pueden generar con la función GETQUORUM.

Cada vez que un proceso quiere ingresar a la sección crítica, obtiene el quorum a través de la función GETQUORUM, luego envía un mensaje de requerimiento a todos los nodos del quorum. Cada requerimiento tiene una estampilla de tiempo local única. Cada nodo tiene una cola de requerimientos que están ordenados por la estampilla de tiempo. Cuando un requerimiento avanza al primer lugar de la cola, el nodo envía un mensaje de permiso.

Cuando recibe un requerimiento con una estampilla de tiempo menor que el requirimiento que se encuentra primero en la cola entonces envía un mensaje de retorno al nodo que se encuentra primero en la cola de requerimientos y espera por su respuesta. La respuesta puede ser de retorno del permiso o de liberación del permiso; en el primer caso, vuelve a poner en la cola de requerimientos el pedido; en ambos casos envía el mensaje de permiso al nodo solicitante.

Cuando un proceso recibe todos los mensajes de permiso de los nodos pertenecientes a su actual quorum puede ingresar a la sección crítica. Cuando sale de la sección crítica les avisa con un mensaje de liberación para que lo puedan remover del primer lugar de la cola.

El número de nodos que son requeridos para formar un quorum es $O(\log(n))$ en la ausencia de fallas, pero no es completamente distribuido, ya que el nodo que está en la raíz formará parte de todos los quorums. El protocolo también puede soportar la falla de $n - \log(n)$ nodos, y aún obtener un quorum para la exclusión mutua. En el peor de los casos requerirá $\left\lceil \frac{(n+1)}{2} \right\rceil$ nodos.

Algoritmo Distribuido con Self Stabilization

Nesterenko y Mizuno [112] proponen una solución al problema de la exclusión mutua con la propiedad de self-stabilization, basándose en el algoritmo de Maekawa. Utiliza un modelo de pasaje de mensajes asincrónico. El mismo está diseñado para soportar la pérdida de mensajes. En estados legítimos, el algoritmo presenta la misma complejidad de mensajes

y demora en la sincronización que el propuesto por Maekawa.

El algoritmo está compuesto por dos módulos: el *solicitante* y el *árbitro*. Estos módulos realizan funciones independientes, el *solicitante* obtiene permisos desde los procesos en el quorum, y el *árbitro* otorga los permisos a los otros procesos.

Las estampillas de tiempo son utilizadas para ordenar los requerimientos de acceso a la sección crítica de los diferentes procesos. Las estampillas de tiempo son únicas, esto significa que la estampilla de tiempo de un requerimiento de un proceso no puede ser igual a la estampilla de tiempo del requerimiento de otro proceso aún en el caso de un estado ilegítimo. Los números de vuelta son utilizados para evitar interbloqueos.

En la figura 4.8 se muestra el comportamiento del módulo *solicitante*. Las variables que utiliza el mismo para un proceso P_i son las siguientes:

- K: constante, mantiene el quorum del proceso P_i.
- L: mantiene el conjunto de procesos que otorgaron el permiso a P_i para ingresar a la sección crítica.
- *needcs*: indica si P_i quiere ingresar a la sección crítica. Es *verdadero* si P_i está en la sección de entrada o en la sección crítica y *falso* en otros casos (sección salida y sección resto).
- ts: almacena la estampilla de tiempo del requerimiento de P_i para acceder a la sección crítica.
- mts, mrd: almacenan respectivamente la estampilla de tiempo y la vuelta del mensaje recibido.

La función *newts()* retorna la mayor estampilla de tiempo cada vez que es invocada.

En la figura 4.9 se muestra el comportamiento del módulo *árbitro*. Las variables y funciones que utiliza para el módulo *árbitro* son:

- AK: constante, el conjunto anti-quorum de P_i [27]. Esto es, $AK \equiv \{P_j | P_i \in K_j\}$.
- *round*: mantiene el número de vueltas para la revocación (recall) del permiso.
- mts, mrd: almacena la estampilla de tiempo y la vuelta del mensaje recibido.

El *árbitro* utiliza la función *newrd*(). Esta función retorna un mayor número de vuelta cada vez que es invocada. El *árbitro* de P_i mantiene una cola Q de requerimientos para la sección crítica de los procesos pertenecientes al anti-quorum de P_i. Los requerimientos de los procesos que no están en AK no se almacenan en Q. Los requerimientos en Q están almacenados según el orden de su estampilla de tiempo. En cualquier estado solo un requerimiento de la cola está siempre marcado como *locked*. Este es el requerimiento del proceso que ha obtenido el permiso de P_i.

Las funciones que utiliza tienen el siguiente comportamiento:

```
proceso P_i

*[
(r1)    external →
            si ¬needcs entonces
                needcs := TRUE
                L := ∅
                ts := newts()
                (P_k ∈ K) enviar REQUEST(ts) a P_k

[]
(r2)    (P_j ∈ K) recibir GRANT(mts) desde P_j →
            si needcs ∧ ts = mts entonces
                L := L ∪{P_j}
                si L = K entonces
                    sección crítica
                    needcs := FALSE
                    (P_k ∈ K) enviar RELEASE a P_k

[]
(r3)    (P_j ∈ K) recibir INQUIRE(mts, mrd) desde P_j →
            si needcs ∧ ts = mts entonces
                L := L \ {P_j}
                enviar YIELD(ts, mrd) a P_j

[]
(r4)    (P_j ∈ K) recibir FAILED(mts) desde P_j →
            si needcs ∧ ts = mts entonces
                L := L \ {P_j}

[]
(r5)    (P_j ∈ K) TIMEOUT(P_j) →
            si needcs entonces
                enviar REQUEST(ts) a P_j
            sino
                enviar RELEASE a P_j
```

Figura 4.8: Módulo Solicitante del proceso P_i

```
[]
(a1)   (P_j ∈ AK) recibir REQUEST(mts) desde P_j →
           si ¬empty() ∧ firstid() = lockid() ∧ locksts() > mts entonces
               round := newrd()
           update(P_j, mts)
           si firstid() = P_j entonces
               si lockid() = P_j entonces
                   enviar GRANT (lockts()) a lockid()
               sino
                   enviar INQUIRE(lockts(), round) a lockid()
           sino
               enviar FAILED(mts) a P_j

[]
(a2)   (P_j ∈ AK) recibir YIELD(mts, mrd) desde P_j →
           update(P_j, mts)
           si lockid () = P_j ∧ round = mrd entonces
               lockfirst()
               enviar GRANT(lockts()) a lockid()

[]
(a3)   (P_j ∈ AK) recibir RELEASE desde P_j →
           si lockid() = P_j entonces
               delete(P_j)
               si ¬empty() entonces
                   enviar GRANT(lockts()) a lockid()
               sino
                   delete(P_j)
```

Figura 4.9: Módulo del árbitro de P_i

- delete(P_j): elimina el requerimiento de P_j de Q, si tal requerimiento está presente en la misma. Si el requerimiento P_j está marcado como *locked* y hay otros requerimientos en la cola, el primero de tales requerimientos es marcado como *locked* cuando el requerimiento de P_j sea borrado.
- firstid(): retorna el identificador del proceso cuyo requerimiento está en la cabeza de Q. Si Q está vacía, retorna un valor diferente a cualquier identificador de proceso de AK.
- lockfirst(): marca el primer requerimiento de la cola como *locked*.
- lockid(): retorna el identificador del proceso cuyo requerimiento está *locked*. Si Q está vacía, retorna un valor diferente a cualquier identificador de proceso de AK.
- empty(): TRUE si está vacío, FALSE en caso contrario.
- lockts(): retorna la estampilla de tiempo del proceso cuyo requerimiento está *locked*. Si Q está vacía, el valor no está especificado.
- update(P_j,ts_j): agrega el requerimiento en Q según el orden de la estampilla de tiempo. Si es el único requerimiento entonces se marca como *locked*. Si existe un requerimiento de P_j en Q con una estampilla de tiempo diferente, el requerimiento más antiguo es eliminado y el nuevo es agregado. Si el antiguo requerimiento está marcado como *locked*, el primer requerimiento de la cola es marcado como *locked*.

Como está basado en Maekawa [99], el comportamiento básico es el mismo. Presenta diferencias en la forma de evitar interbloqueos. Como la comunicación no es directa con todos los procesos para ingresar a la sección crítica, el permiso puede ser otorgado aunque no respete el orden de las estampillas de tiempo aún en estados legítimos. La posibilidad de interbloqueo puede ocurrir cuando los requerimientos no llegan en el orden de las estampillas de tiempo y, para evitarlo, se requiere que los permisos se revoquen.

Supongamos que el *árbitro* de P_j otorga el permiso P_i con la estampilla de tiempo ts_i y recibe un REQUEST desde el proceso P_k con estampilla de tiempo ts_k tal que $\mathrm{ts}_k < \mathrm{ts}_i$. El proceso P_j obtiene un nuevo número de vuelta y envía INQUIRE a P_i. Este mensaje lleva el nuevo número de vuelta. Cuando P_i recibe INQUIRE, remueve P_j de L_i y envía de respuesta YIELD a P_j. Cuando P_j recibe YIELD, envía un GRANT al proceso ubicado al principio de Q_j. Esto es, un requerimiento que quiere ingresar en la sección crítica con una estampilla de tiempo menor no espera por un proceso con una estampilla de tiempo superior. El *árbitro* puede otorgar permiso a un proceso y luego revocarlo varias veces antes que el proceso ingrese a la sección crítica.

El algoritmo garantiza la propiedad de estabilización. Para mostrar la estabilización se definen predicados y estados legítimos. Se considera Ch_{ij} como una cola de mensajes enviados desde P_i a P_j. El Lema 1 y Lema 2 presentan los predicados a los cuales estabiliza el algoritmo y que se utilizan para definir la invariante I_{SS}.

Lema 4.1 *El algoritmo se estabiliza al siguiente predicado:*

(R1)

Los mensajes INQUIERE y YIELD en Ch_{ij} y Ch_{ji} tienen un orden no decreciente en el número de vuelta y el número de vuelta de cualquier mensaje no es mayor que $round_j$.

Lema 4.2 *El algoritmo se estabiliza al siguiente predicado:*

(R2)

Las estampillas de tiempo de los mensajes en Ch_{ij} y Ch_{ji} tienen un orden no decreciente, y las estampillas de tiempo de cualquier mensaje no es mayor que ts_i; y
si P_j tiene un requerimiento de P_i entonces
ts_j^i no es mayor que ts_i, y
la estampilla de tiempo de un mensaje en Ch_{ij} no es menor que ts_j^i, y
la estampilla de tiempo de un mensaje en Ch_{ji} no es mayor que ts_j^i.

Se definen el siguiente conjunto de estados legítimos con respecto al par (P_i, P_j):

- A: el conjunto de estados donde P_i no desea ingresar en la Sección Crítica.
- B: un requerimiento para ingresar a la Sección Crítica es enviado pero no recibido por P_j.
- C: P_j otorgó el permiso a P_i para ingresar en la sección crítica y ts_j^i es la menor estampilla en Q.
- D: P_j otorgó el permiso a P_i pero P_j tiene un requerimiento con una estampilla de tiempo menor que ts_i.
- E: el requerimiento de P_i es agregado en Q de P_j pero no se le ha otorgado el permiso para acceder a la sección crítica.

Los conjuntos se definen formalmente a través de los siguientes predicados.

- (A) $\neg needcs$
- (B) $needcs \wedge (P_i \notin Q \vee (P_i \in Q \wedge ts_i \neq ts_j^i)) \wedge (P_j \notin L) \wedge$
  no hay un mensaje RELEASE siguiendo un mensaje REQUEST en Ch_{ij} y
  no hay mensaje en Ch_{ji}
- (C) $needcs \wedge (P_i \in Q) \wedge (ts_i = ts_j^i) \wedge (lockid = P_i) \wedge (firstid = P_i) \wedge$
  no hay mensaje RELEASE en Ch_{ij}
- (D) $needcs \wedge (P_i \in Q) \wedge (ts_i = ts_j^i) \wedge (lockid = P_i) \wedge (firstid \neq P_i) \wedge$
  no hay mensaje RELEASE en Ch_{ij} y
  no hay mensaje GRANT siguiendo un mensaje INQUIRE(mrd) en Ch_{ji} donde ($mrd = round$) y
  si YIELD(mrd),($mrd = round$) en Ch_{ij} entonces
  $P_j \notin L$ y no hay GRANT en Ch_{ji}
- (E) $needcs \wedge (P_j \in Q_i) \wedge (ts_i = ts_j^i) \wedge (P_j \notin L) \wedge (lockid \neq P_i) \wedge$
  no hay mensaje RELEASE en Ch_{ij} y
  no hay mensaje GRANT en Ch_{ji}

Se define la invariante I_{SS} de la siguiente forma. Sean P_i y P_j dos procesos cualesquiera tal que $P_j \in K_i$ se mantiene: $R_1 \wedge R_2 \wedge (A \vee B \vee C \vee D \vee E)$. La ejecución de cualquier acción mantiene el sistema en el mismo conjunto o se mueve a otro conjunto dentro del conjunto de estados legítimos.

La performance de este algoritmo va a estar relacionada con la carga en el sistema, esto es, dependiendo de los procesos que compitan por ingresar a la sección crítica. Si la carga es baja, pueden necesitarse 3 mensajes para entrar a la sección crítica por cada miembro del quorum (REQUEST, GRANT, RELEASE). Si el tamaño del quorum es proporcional a $\sqrt{N}$, entonces se necesitarán $3\sqrt{N}$ mensajes. Si la carga es alta el *árbitro* envía un mensaje FAILED al *solicitante* cuando tiene un requerimiento con una estampilla de tiempo menor. La complejidad de mensajes será proporcional a $4\sqrt{N}$. En el caso que los mensajes lleguen fuera de orden y se tengan que revocar permisos (INQUIRE, YIELD) la complejidad de mensajes será proporcional a $6\sqrt{N}$.

4.2.2. Modelos basados en token

Algoritmo utilizando un coordinador central

En la sección 4.1 se clasifica a los algoritmos en centralizados y distribuidos. En el caso de los algoritmos centralizados se cuenta con un nodo que realiza las funciones de coordinación para la utilización de un objeto compartido en modo exclusivo y en 4.1 se lo cita brevemente. La ventaja del algoritmo centralizado está en el número de mensajes requeridos para cada requerimiento; la mayor desventaja está en la demora de sincronización.

Wu y Shu [142], teniendo en cuenta el modelo centralizado, presentan un algoritmo distribuido basado en *tokens* utilizando un nodo como coordinador central. Cuando un nodo envía un mensaje REQUEST al coordinador, éste propaga el requerimiento al nodo que realizará el RELEASE, esto es, el último que tiene el *token*. El coordinador registra el número de nodo para saber a cual nodo debe reenviar la próxima propagación.

En el algoritmo se utililizan los siguientes tipos de mensajes:

- REQUEST se utiliza para solicitar el acceso a la sección crítica. Lo envía el nodo solicitante al nodo coordinador.
- FORWARD notifica al último nodo que realizó un requerimiento de acceso a la sección crítica la identificación del nodo que accederá después que él, esto es, envía al nodo con REQUEST (seq−1) la identificación del nodo con REQUEST(seq). Este mensaje es enviado por el coordinador.
- GRANT lleva el *token* al nodo solicitante. Puede ser enviado por el coordinador o el nodo que ha finalizado el acceso a la sección crítica.
- RELEASE lleva el *token* al coordinador.

En la figura 4.10 se muestra el comportamiento de los mensajes. En la figura 4.10(a), se muestran los mensajes requeridos en el caso que el *token* lo tiene el coordinador, cuando

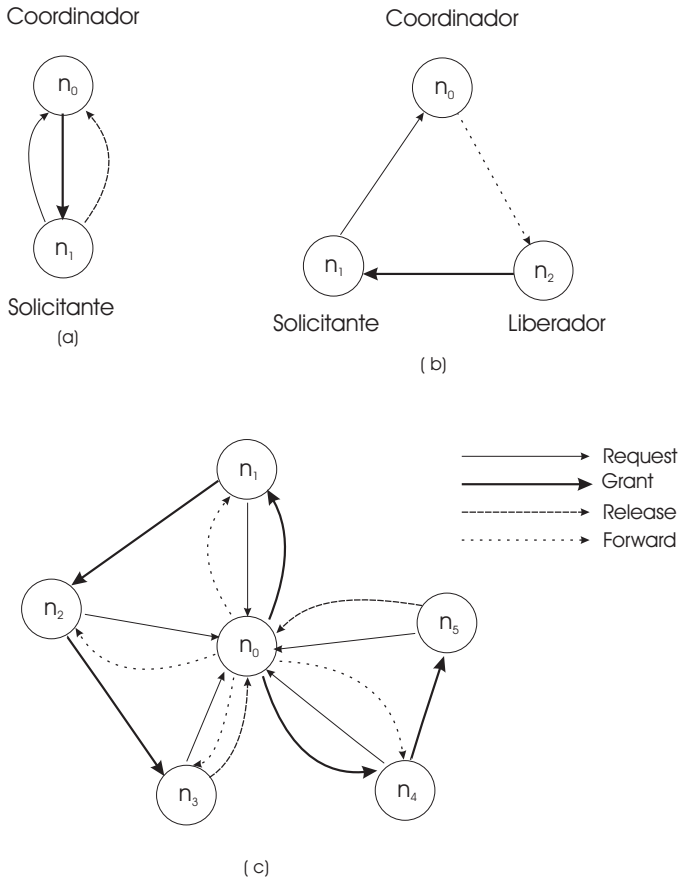


Figura 4.10: Coordinador Central con *Token*

las condiciones de carga son bajas. El primer mensaje es REQUEST que lo envía el nodo solicitante (n_1) al coordinador (n_0). Si éste tiene el *token* entonces envía un mensaje GRANT permitiendo el acceso a la sección crítica y al finalizar le envía un mensaje RELEASE liberando el *token*. En la figura 4.10*(b)* se muestra el caso que el *token* no lo tenga el coordinador. El nodo solicitante (n_1) envía un mensaje REQUEST al coordinador (n_0), y si éste no tiene el *token* entonces envía un mensaje FORWARD al nodo que está accediendo en la sección crítica (n_2). Cuando el nodo liberador finaliza el acceso a la sección crítica entonces envía un mensaje GRANT al nodo solicitante (n_1) y por último cuando finaliza el acceso envía un mensaje RELEASE al coordinador. En el caso que las condiciones de carga sean altas, cuando un nodo finaliza la sección crítica pueden ocurrir dos situaciones.

- Un mensaje FORWARD ha arribado. El nodo enviará el *token* al próximo nodo mediante el envío de un mensaje GRANT.
- Un mensaje FORWARD no ha arribado. El nodo enviará el *token* al nodo coordinador mediante el envío de un mensaje RELEASE.

En la figura 4.10(c) se muestra la siguiente situación, el nodo n_0 es el coordinador y los nodos n_1, n_2, n_3, n_4 y n_5 realizan requerimientos para acceder a la sección crítica. El primer requerimiento lo realiza n_1 y recibe un mensaje GRANT desde n_0. Cuando n_2 realiza un requerimiento, n_0 propaga el requerimiento a n_1. Cuando n_1 finaliza su acceso a la sección crítica, envía un mensaje GRANT a n_2. Cuando n_3 realiza un requerimiento, el nodo n_0 lo propaga a n_2. Cuando finaliza n_2 envía un mensaje GRANT a n_3. Cuando n_4 realiza un requerimiento, n_0 propaga el requerimiento a n_3. Cuando n_3 finaliza su sección crítica controla si ha recibido algún mensaje FORWARD y, como no es el caso, entonces envía un mensaje RELEASE al nodo n_0, y después recibe el mensaje de propagación, el cual es descartado. Cuando n_0 recibe el mensaje de RELEASE del nodo n_3, actualiza la información que tiene sobre los requerimientos de acceso y envía en mensaje GRANT al nodo n_4. Cuando n_5 realiza un requerimiento, el nodo n_0 lo propaga a n_4. Y cuando n_5 finaliza la sección crítica envía un mensaje RELEASE a n_0.

El algoritmo garantiza exclusión mutua, ya que el permiso se obtiene a través del *token* que es único, está libre de interbloqueo ya que el nodo que tiene el *token* nunca espera por otro nodo. Además está libre de inanición ya que los requerimientos son atendidos en orden, esto es el primero en llegar es el primero en atenderse (first-come, first-served). La cantidad de mensajes requeridos para acceder a la sección crítica son 3 en el caso de baja carga y en el caso de alta carga pueden ser 3 ó 4 mensajes. El tiempo de espera es 2T + E, en el caso que la carga sea baja, donde T es el tiempo promedio de demora de un mensaje y E el tiempo promedio de ejecución en la sección crítica. En el caso que las condiciones de alta carga, el tiempo de respuesta depende de la demora de la sincronización, sería k(T + E).

Algoritmo con lista ordenada

Banerjee y Chrysanthis [26] proponen una solución de exclusión mutua utilizando *token*. Éste contiene una lista ordenada o cola (*Q-list*) de todos los nodos que han planificado

acceder a la sección crítica. El *token* es pasado de nodo en nodo en el orden especificado por la *Q-list*. El nodo que está accediendo en un determinado instante de tiempo se encuentra en ese momento primero en la *Q-list*.

La *Q-list* es creada por un nodo que es designado como el árbitro del sistema. Inicialmente, un nodo específico es asignado para ser el árbitro del sistema. La responsabilidad del nodo árbitro es compartida por los otros nodos del sistema. Un nodo árbitro ejecuta dos fases: *fase de recolección de requerimientos* y *fase de propagación de requerimientos*. En la *fase de recolección de requerimientos*, el nodo árbitro recolecta todos los requerimientos desde los nodos que buscan acceder a la sección crítica y crea la *Q-list* ordenada. Al final de esta fase, cuando el árbitro obtiene la posesión del *token*, el *token* es actualizado con la nueva *Q-list* construida y lo transmite al nodo que se encuentra al principio de la *Q-list* junto con el contenido de la misma. Y por último, el actual árbitro declara como nuevo árbitro al nodo ubicado en el último lugar de la *Q-list* y envía un mensaje broadcast comunicándolo a todos los nodos del sistema.

El *token* es pasado de nodo en nodo en orden de acuerdo al *Q-list* hasta que finaliza en el nodo árbitro. Los mensajes que utiliza el algoritmo son los siguientes:

- Mensaje PRIVILEGIO, el cual tiene la forma PRIVILEGIO(Q), donde Q es una lista ordenada de nodos para obtener el permiso de acceso a la sección crítica.
- Mensaje REQUEST, es de la forma REQUEST(j), donde el nodo j es el que realiza un requerimiento de acceso a la sección crítica.
- Mensaje NUEVO-ARBITRO, es de la forma NUEVO-ARBITRO(j), donde j es el nuevo árbitro. La *Q-list* es enviada como parte del mensaje NUEVO-ARBITRO.

Los nodos que quieren ingresar en la sección crítica envían sus requerimientos al árbitro. Es posible que un nodo envíe su requerimiento al previo árbitro antes de recibir el mensaje que le informe del nuevo árbitro. En el caso que un previo árbitro reciba un mensaje de requerimiento éste es propagado al nuevo árbitro. La *fase de propagación de requerimientos* ocurre después de la *fase de recolección de requerimientos* para ordenar los mensajes que no arribaron durante la fase de colección, pero fueron transmitidos antes que el nodo emisor reciba la identidad del nuevo árbitro. Cualquier requerimiento recibido fuera de estas fases es descartado.

El algoritmo puede producir inanición si un requerimiento es continuamente propagado sin que el requerimiento sea registrado en el árbitro, o que sea descartado porque arriba fuera de las fases del árbitro. Esta situación tiene mayores probabilidades de suceder en el caso que se tenga una carga baja. Para solucionar esta situación, al esquema básico del algoritmo se le incorpora un *nodo monitor*, conocido por todos, el cual recibe los requerimientos que son propagados o descartados una cierta cantidad de veces. Periódicamente se le pasa el *token* al monitor y estos requerimientos los agrega a la *Q-list*.

El algoritmo, en promedio requiere 3 mensajes para acceder a la sección crítica, cuando tiene una alta carga de solicitudes. En el caso que el sistema tenga baja carga de solicitudes, el algoritmo tiende a requerir N mensajes para acceder a la sección crítica.

4.2.3. Algoritmo de Thambu-Wong

Thambu y Wong [134] basan su propuesta en la idea de finitos planos proyectivos, similar al algoritmo propuesto por Maekawa, pero en vez de basarse en consenso, la solución se basa en tokens. Las propiedades que presentan los finitos planos proyectivos fueron presentadas en la sección 4.2.1 junto con el algoritmo de Maekawa. Este algoritmo requiere enviar requerimientos de token a un conjunto de $\sqrt{N}$ nodos como máximo.

En cada uno de los nodos se tienen las siguientes estructuras de datos:

SUPERIORS$_i$: contiene el conjunto de nodos distintos j, $j \neq i$. El tamaño de este conjunto es el mismo para todos los nodos. Un nodo puede tener más de un superior y por ende ser un inferior de más de un nodo. Un nodo i tiene tantos nodos superiores como inferiores.
$\forall j \in$ SUPERIORS$_i$, $i \in$ INFERIORS$_j$

> Cada nodo j en SUPERIORS$_i$ podrá saber si i tiene el token, porque i estará en INFERIORS$_j$.
>
> Si cualquier $j \in$ SUPERIORS$_i$ recibe un requerimiento desde i, j sabrá a donde redireccionar el requerimiento.

INFERIORS$_i$: contiene un conjunto de nodos distintos j, $j \neq i$. Al igual que SUPERIORS$_i$, el tamaño de este conjunto es el mismo para todos los nodos. Si i tiene el token, cada $j \in$ SUPERIORS$_i$ tendrán esta información. Por otra parte, i no podrá saber si cualquiera de sus superiores tiene el token; en cambio si sabrá si lo tiene cualquiera de sus inferiores.

REQ_QUEUE$_i$: esta es una cola con una lista de nodos (posiblemente vacía), cada uno de los cuales ha enviado un requerimiento para el *token* y desea ingresar en la sección crítica, pero aún no ha obtenido el permiso. El requerimiento pudo haber sido enviado directamente al nodo i o haber sido ruteado al nodo i desde otros nodos.

CONTROL_NODE$_i$: tiene la identidad de un nodo, del nodo que tiene o tendrá el token en el futuro. Antes que el *token* sea enviado desde el nodo i, CONTROL_NODE$_i$ es asignado el último nodo del REQ_QUEUE$_i$.

IN_CS$_i$ indica si el nodo i está en la sección crítica.

ENTRY_REQUIRED$_i$ indica si el nodo i desea ingresar en la sección crítica.

TOKEN_PRESENT$_i$ indica si el nodo i ha recibido el *token*.

Los mensajes utilizados son:

REQUEST_TOKEN: este mensaje lo envía un nodo que requiere el *token* para ingresar a la sección crítica. Un nodo que recibe este mensaje puede agregar el requerimiento a su REQ_QUEUE, enviar el *token* al nodo solicitante, o rutear este requerimiento a otro nodo.

I_HAVE_TOKEN: con este mensaje un nodo informa a cada uno de sus superiores que él tiene el *token*. Cada uno de los superiores asigna a su variable CONTROL_NODE el nodo que envió el mensaje.

RELINQUISH_TOKEN: con este mensaje un nodo informa a sus superiores que ya no tiene más el *token*. Los superiores asignan NULL a su variable CONTROL_NODE cuando reciben este mensaje.

[TOKEN, QUEUE] un nodo envía el token al primer nodo de su REQ_QUEUE, junto con el resto de la cola. Si la cola está vacía entonces no se envía el mensaje.

[QUEUE] un nodo puede recibir requerimientos para el *token* después que lo liberó. El nodo encola estos requerimientos y los rutea al nodo que tiene asignado el CONTROL_NODE.

Cuando un nodo i desea ingresar en la sección crítica, envía un mensaje REQUEST_TOKEN(i) a:

- un inferior j, si tiene asignado en CONTROL_NODE$_i$ a j.
- a todos los superiores de i, si no conoce quien mantiene el token.

Si un nodo solicitante j no conoce si cualquiera de sus inferiores mantiene el *token*, entonces solicitando a todos sus superiores llegará hasta el nodo que mantiene el *token*. Siempre existirá un camino entre el nodo solicitante y el nodo que mantiene el *token*, y este camino se encuentra a través de la variable CONTROL_NODE.

Cuando un nodo i sale de su sección crítica, si la cola REQ_QUEUE$_i$ no está vacía envía el *token* al primer nodo de la misma junto con el resto de la cola. De lo contrario, mantiene el *token* hasta que recibe una solicitud. Si el *token* es enviado a otro nodo, envía un mensaje RELINQUISH_TOKEN(i) a cada uno de sus superiores.

Este algoritmo envía requerimientos de *token* a lo sumo a un conjunto $\sqrt{N}$ de nodos. En el mejor de los casos este algoritmo no requiere ningún mensaje, ya que puede mantener el *token* el nodo que lo requiere y en el peor de los casos requerirá $3\sqrt{N} + 2$ mensajes.

Algoritmo Basado en Árbol

Raymond [121], propone una estructura de árbol sin raíz, donde cada nodo se comunica solamente con sus nodos vecinos del alcance del árbol (spanning tree), y mantiene información relacionada solamente con estos vecinos.

Un árbol puede ser tanto un alcance de árbol mínimo de la actual topología de red, o meramente una estructura lógica impuesta sobre una completa red. No hay necesidad que cada nodo conozca el árbol completo. Es suficiente que cada nodo conozca la existencia de sus vecinos en el árbol. En la figura 4.11 se muestra una estructura de árbol. El nodo A conoce y está conectado con 3 nodos B, C y D pero puede no conocer la ubicación o exitencia de otros nodos en el árbol, como puede ser E ó F.

El nodo que tiene el *token* es el que tiene permiso para ingresar a la sección crítica. Si no hay requerimientos para acceder a la sección crítica, entonces el *token* lo mantiene el último nodo que lo requirió. Cada nodo tiene una variable HOLDER, la cual indica la locación del *token* relativo a su nodo.

Considerando que el *token* lo tiene el nodo E, de acuerdo al árbol de la figura 4.11, se tiene el siguiente estado para la variable HOLDER en cada nodo.

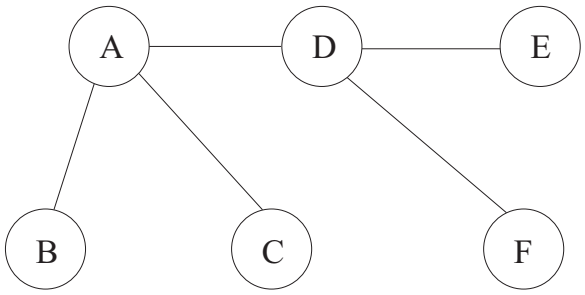


Figura 4.11: Estructura del árbol

Representando HOLDER_X = Y como un arco dirigido de X a Y se puede observar en la figura 4.12 que todos los nodos se dirigen hacia el nodo que tiene el token.

HOLDER_A = D

HOLDER_B = A

HOLDER_C = A

HOLDER_D = E

HOLDER_E = *self*

HOLDER_F = D

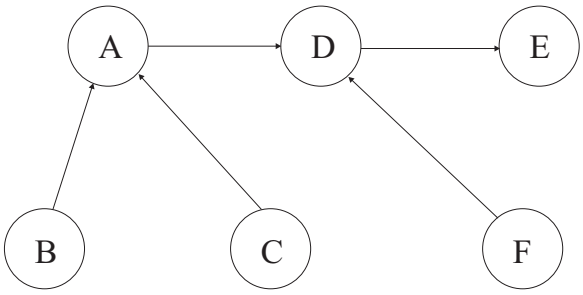


Figura 4.12: Árbol dirigido

El algoritmo requiere dos tipos de mensajes de comunicación: mensaje REQUEST y mensaje PRIVILEGE (o de *token*). Cada nodo X mantiene una cola REQUEST_Q de tipo primero en entrar primero en salir (FIFO), que mantiene el nombre de aquellos vecinos que han enviado un mensaje REQUEST a X, y que todavía no han recibido el mensaje de PRIVILEGE (el *token*). El número máximo de elementos es el número de vecinos más uno (por él mismo).

El comportamiento del algoritmo es el siguiente:

Cuando un nodo Y que no tiene el *token* desea ingresar en la sección crítica, envía un mensaje REQUEST al nodo que contiene HOLDER_Y.

Cuando un nodo Y recibe un mensaje de REQUEST, chequea si tiene el *token* y si está disponible. En el caso que se cumplan todas estas condiciones entonces le envía el *token* al nodo que lo solicitó y cambia el valor asignado de HOLDER_Y al de ese nodo. Sino se cumplen estas condiciones pueden suceder los siguientes casos:

- El nodo tenga el *token* y esté ejecutando la sección crítica en cuyo caso solo encola en requerimiento.

- El nodo no tenga el *token* y no tenga requerimientos pendientes en cuyo caso envía un mensaje REQUEST al nodo que tiene asignado el HOLDER_Y, encola el requerimiento y registra que ha enviado un mensaje REQUEST.
- El nodo no tenga el *token* y tenga requerimientos pendientes en cuyo caso solo encola el requerimiento.

El buen comportamiento del algoritmo no se ve afectado porque los mensajes no lleguen ordenados. El algoritmo garantiza exclusión mutua, está libre de interbloqueo y libre de inanición. El límite superior para el número de mensajes por sección crítica es 2D, donde D es la mayor distancia del árbol.

Si la estructura del árbol es lineal, con N nodos entonces el número de mensajes requeridos es 2(N-1), donde (N-1) mensajes son de requerimiento y (N-1) mensajes de *token* (Privilege), con D = N-1. Si la estuctura del árbol es una formación de estrella radial, entonces el peor caso para esta topología es $O(\log_{k-1}N)$, donde k es la valencia de cada nodo no hoja. El diámetro de un arbitrario árbol es típicamente O(log N) entonces el número de mensajes requeridos está en el O(log N). Cuando el sistema está cargado, el número de mensajes por entrada en la sección crítica es aproximadamente 4.

Algoritmo de Chang con O(log N)

Chang y otros [46] proponen un algoritmo de exclusión mutua, basado en *token* para resolver el problema de exclusión mutua, mejorando el O(log N) mensajes para ingresar a la sección crítica.

Cada proceso p mantiene dos variables $Root_p$ y $Next_p$. Las variables *Root* mantienen un árbol lógico en el sistema y, a través de este los mensajes de requerimiento pueden ser propagados al *token holder*. El *token* es transferido mediante un mensaje *token*, el cual también lleva información sobre el proceso que puede llegar a ser el último *token holder* en el futuro. Los procesos utilizan esta información para actualizar las variables *Root* como un intento para minimizar la altura del árbol lógico. Minimizar la altura del árbol lógico reduce la demora promedio para ingresar a la sección crítica. La variable $Next$ mantiene pista de los procesos que tienen requerimientos pendientes a la sección crítica en una manera distribuida y, en cada proceso la variable $Next$ mantiene, si es que existe, cuál es el próximo proceso que quiere ingresar en la sección crítica. Cuando un proceso p sale de la sección crítica, envía el *token* al proceso $Next_p$, si $Next_p$ no está apuntando al valor null ($\perp$).

En la figura 4.13, se muestra en (a) el estado inicial, donde el token lo tiene el proceso R, la variable *Root* de cada proceso tiene asignado el valor de R; en la figura se muestra este valor como un arco dirigido. Cuando el proceso Q quiere ingresar a la sección crítica, le solicita a R el token y el mismo es enviado a Q. En la figura 4.13 en (b) se muestra la situación donde se actualiza el valor de la variable *Root* en R y Q. Cuando el proceso P quiere ingresar a la sección crítica, le solicita a R el token, y si no lo tiene, propaga el requerimiento a Q y actualiza el valor de su variable *Root* a P. Esta situación se muestra en la figura 4.13 en (c). Cuando el proceso Q recibe la solicitud por el token actualiza el

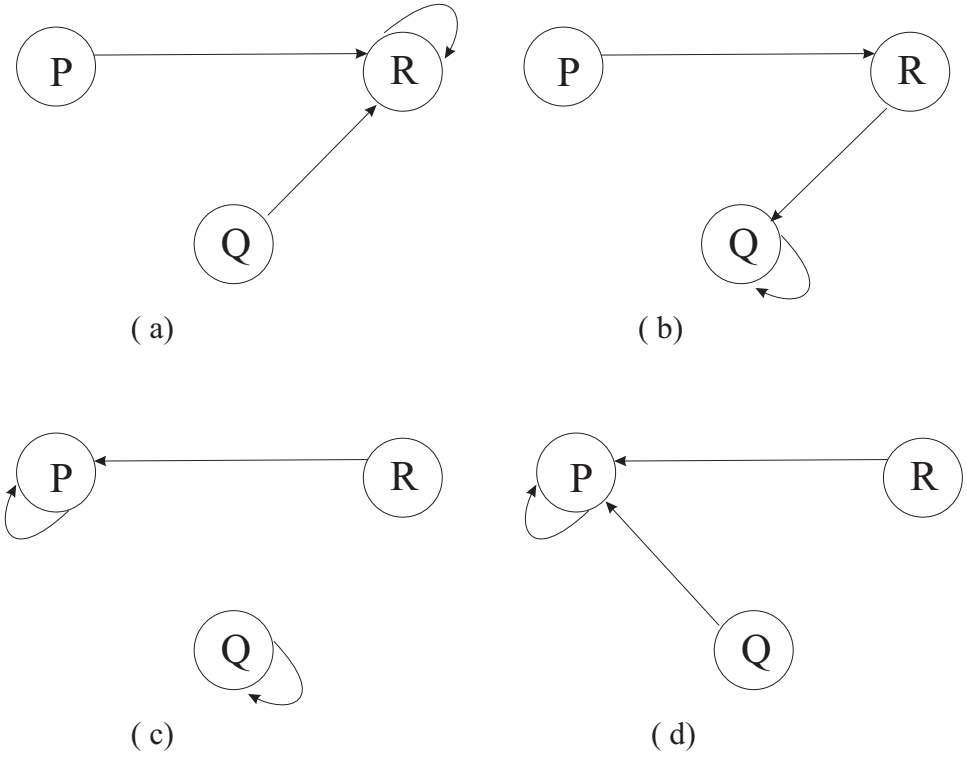


Figura 4.13: Algoritmo de Chang y otros. Evolución

valor de *Root* a P, tal como se muestra en la figura 4.13 en (d). El algoritmo requiere un promedio de 3 mensajes por sección crítica y está libre de interbloqueo y libre de inanición. El algoritmo no tolera fallas, un caso es la pérdida de *token*.

El algoritmo de Choy [50], es una adaptación del algoritmo de Chang y otros [46], que mediante la utilización de timeouts permite manejar fallas de nodos y comunicación, la detección tardía de *token* y, en el peor caso, la pérdida del *token*. Estas fallas, se soportan mediante la regeneración del *token* en el proceso que se supone recibirá el token en último lugar.

4.2.4. Comparación de los Algoritmos

Las métricas a tener en cuenta para la comparación de los algoritmos son las siguientes:

- La complejidad de los mensajes que es el número de mensajes necesitados para un acceso a la sección crítica.
- La demora en la sincronización que es el número de intercambios causalmente relacionados de mensajes requeridos para una entrada en la sección crítica.

La complejidad de mensajes, para los algoritmos mostrados en las secciones anteriores, considerando el límite inferior y superior es:

- El algoritmo de Maekawa requerirá en el límite inferior y superior $\mathrm{O}(\sqrt{N})$, la cantidad de mensajes para ingresar y salir serán 3 $\sqrt{N}$ en el mejor de los casos y 5 $\sqrt{N}$ en el peor de los casos, en el caso teórico, generado el quorum con finitos planos proyectivos. En el caso de que se genere el quorum con la grilla, el tamaño del mismo es $(2\sqrt{N}-1)$, la cantidad de mensajes para ingresar y salir serán $(6\sqrt{N}-3)$ en el mejor de los casos y $(10\sqrt{N}-5)$ en el peor de los casos.
- El algoritmo de Sanders dependerá de la elección de los conjuntos I_i y R_i para determinar el orden de los límites superiores e inferiores y la cantidad de mensajes requeridos.
- El algoritmo de la grilla triangular ([94]) requerirá en el límite inferior y superior $\mathrm{O}(\sqrt{N})$, la cantidad de elementos en el quorum aproximadamente de $\sqrt{2N}$, la cantidad de mensajes requeridos para ingresar y salir de la sección crítica es aproximadamente 3 $\sqrt{2N}$ en el mejor de los casos y 5 $\sqrt{2N}$ en el peor de los casos.
- El algoritmo del Árbol requerirá en el límite inferior $O(\log(N))$ mensajes y en el límite superior $\mathrm{O}(N/2)$, la cantidad de mensajes para ingresar y salir serán 3 $\log(N)$ en el límite inferior y 3 $(\frac{n+1}{2})$ en el límite superior.
- El algoritmo de Agrawal requerirá en el límite inferior y superior $\mathrm{O}(\sqrt{N})$, con un tamaño de quorum de $\sqrt{2N}$.

Algoritmo	Cantidad Quorums	Orden	Cantidad Mensajes Mejor Caso	Cantidad Mensajes Peor Caso
Maekaewa [99]	1	O($\sqrt{N}$)	3 $\sqrt{N}$	5 $\sqrt{N}$
Sanders [126]	dependiendo de I_i y R_i	O($\|I_i\| + 2\,\|R_i\|$)	$\|I_i - \{i\}\| +$ $2(\|I_i - \{i\}\|)$	$\|I_i - \{i\}\| +$ $2(\|I_i - \{i\}\|) + DM$
Grilla Rectangular [99]	1	O($2\sqrt{N}$)	$6\sqrt{N} - 3$	$10\sqrt{N} - 5$
Grilla Triangular [94]	2	O($\sqrt{2N}$)	aprox. $3\sqrt{2N}$	aprox. $5\sqrt{2N}$
Agrawal [8]	2	O($\sqrt{N}$)	$3\sqrt{2N}$	
Cohorts [66]	varios	O($\|Q\|$), lím. inf. $\|Q\| = \|C_k\|$ O(N) lím. sup.		
Árbol [5]	varios	$O(\log(N))$ lín. inf. O(N/2) lím. sup.	3 $\log(N)$ lím. inf. $3(\frac{N+1}{2})$ lím. sup.	

Cuadro 4.1: Algoritmos Basados en Quorum

- El algoritmo utilizando Cohorts requerirá en el límite inferior O($|Q|$), donde $|Q| = |C_k|$, y O(N) en el límite superior, para un Coh(k) = {C_1, ..., C_k}.

En el cuadro 4.1 se muestra una comparación entre los algoritmos basados en quorum presentados. La diferencia que se puede observar entre el algoritmo de Maekawa y el de Agrawal es que el primero tiene asignado un único quorum para cada nodo y en el segundo caso se tienen 2 quorums asignados a cada nodo y se adapta de mejor manera a fallos de nodos. En los otros casos, sucede que hay gran diferencia entre el límite superior e inferior debido a la posibilidad de soportar de mejor forma la falla de red o de nodos. El algoritmo basado en Cohorts puede generar un quorum mínimo, pero a costa de no mantener la propiedad de completamente distribuido, presentada por Maekawa, la cual si fue considerada en otros casos como el de Agrawal, o la Grilla.

Una propiedad a tener en cuenta en los algoritmos es la imparcialidad (*fairness*). Un algoritmo de exclusión mutua presenta esta característica si $\Pr(R_i) > \Pr(R_j) \Longleftrightarrow P_j$ ejecuta la sección crítica después que P_i finaliza la sección crítica, donde $\Pr(R_i)$ es la prioridad de P_i y $\Pr(R_j)$ es la prioridad de P_j. Esta definición está en términos de satisfacer los requerimientos para acceder a la sección crítica en orden decreciente de prioridades, que están definidas por las estampillas de tiempo de Lamport. Esta propiedad es garantizada por el algoritmo de Lamport (Tiempo Lógico), Ricart Agrawala y Lodha [93]. Este último algoritmo es una propuesta que está basada en el algoritmo de Ricart Agrawala, que requiere entre $(N-1)$ y $2(N-1)$ mensajes por acceso a la sección crítica. El número exacto de mensajes para cualquier acceso a la sección crítica es $2(N-1) - x$, donde x es el número de otros requerimientos que son realizados concurrentemente con este requerimiento. Estos algoritmos (Lamport, Agrawala y Lodha) no soportan fallas de nodos o de vínculos.

Algoritmo	Cantidad de Mensajes
Nodo Coordinador [142]	3 mensajes con Carga Baja 3 ó 4 mensajes con Carga Alta
Banerjee [26]	N con Carga Baja en promedio 3 mensajes con Carga Alta
Thambu y Wong [134]	0 mensajes en el mejor caso 3 $\sqrt{N}$ + 2 en el peor caso
Árbol - Raymond [121]	2D en el peor caso, D: es la mayor distancia del árbol Si el árbol es lineal, requerirá 2(N-1) mensajes En general, con un diámetro de $O(\log N)$, requerirá $O(\log N)$ mensajes y con alta carga requerirá 4 mensajes.
Chang [46]	en promedio 3 mensajes

Cuadro 4.2: Algoritmos Basados en Token

En el cuadro 4.2 se muestra una comparación de los algoritmos presentados basados en token, indicando la cantidad de mensajes requeridos para ingresar y salir de la sección crítica.

En resumen, el análisis comparativo se basó en la cantidad de mensajes que son requeridos para acceder a la sección crítica y, ésta depende de la carga del sistema en cuánto a la cantidad de requerimientos que son realizados en un determinado instante. En los algoritmos basados en *token*, el caso que el nodo sea el poseedor del mismo, no se tendrá costo de comunicación, esta situación es ideal y no es posible en los algoritmos basados en quorum. Generalmente, los algoritmos basados en *token* presentan una buena performance sobre el número de mensajes intercambiados para obtener el recurso compartido, pero pueden sufrir de pobre performance ante fallas resilientes. Los algoritmos basados en quorum toleran de mejor forma fallas de nodos y particiones de la red.

4.3. Algoritmos de k-Exclusión Mutua

Un extensión del problema de la exclusión mutua es tener k unidades de un recurso, donde cada proceso requiere acceder en forma exclusiva a una unidad. En esta sección se presentan algoritmos que garantizan la k-exclusión mutua.

4.3.1. Algoritmo distribuido utilizando k-coteries

Kakugawa y otros [78] resuelven el problema de la k-exclusión mutua distribuida basada en el modelo de quorum utilizando el concepto de k-coteries. En cada requerimiento se adjunta el par (t,p), donde t es el tiempo lógico en el cual un proceso p inicia el requerimiento y se considera que p es único.

Esta propuesta presenta similitudes con respecto al algoritmo de Maekawa. La diferencia está en que cada proceso p no tiene asignado un quorum Q estático, ya que un quorum puede negarle el permiso para acceder a la sección crítica pero puede existir algún otro quorum que le otorgue el mismo, ya que hay $(k-1)$ quorums que no intersectan con Q.

El algoritmo utiliza las siguientes variables.

- C un k-coterie.
- Cada proceso p tiene:
  - Yes mantiene el conjunto de procesos que han otorgado el permiso a p para que ingrese a la sección crítica,
  - $NotNow$ mantiene el conjunto de procesos que aún no han otorgado el permiso a p para que ingrese en la sección crítica,
  - $Perm$ mantiene el proceso al que p ha otorgado el permiso para ingresar a la sección crítica pero aún no ha recibido el mensaje $RELEASE$ indicando que ha liberado la sección crítica, si existe dicho proceso. Ya que p nunca da permiso a 2 procesos a la vez, $Perm$ está vacío o es un conjunto simple.
  - $Queue$ es la cola de prioridades para mantener los requerimientos en el orden de sus estampillas de tiempo.

Inicialmente están vacíos los conjuntos Yes, $NotNow$ y $Perm$. Un proceso p puede recibir un mensaje OK desde procesos $NotNow$. En tal caso, los procesos son movidos desde $NotNow$ a Yes.

El algoritmo presenta el siguiente comportamiento:

- Cuando p desea ingresar a la sección crítica. Selecciona un quorum Q de C, y envía $Request(t,p)$ a cada miembro q de Q (incluyéndose el mismo) y espera por una respuesta (OK ó $WAIT$) desde q, donde (t,p) es la estampilla de tiempo. Si cada q responde OK, p puede ingresar en la sección crítica.

  Si algún proceso responde $WAIT$, p agrega los procesos que respondieron OK a Yes y los que respondieron $WAIT$ a $NotNow$, selecciona otro quorum Q' el cual minimiza $|Q \cap Yes|$ desde quorums en C que la intersección con $NotNow$ sea vacía, y repite el procedimiento desde el principio. En este caso, p envía $Request(t,p)$ solamente a los miembros en (Q' - Yes). Cada proceso recibe a lo sumo un mensaje de $Request$ desde p. Si p no puede encontrar un quorum que satisfaga las condiciones, entonces p espera por la recepción de los mensajes OK.

  Durante el proceso de búsqueda de un nuevo quorum, p puede recibir un mensaje OK desde un proceso q ubicado en $NotNow$. Entonces, p testea si un quorum está incluido en Yes después de mover q desde $NotNow$ a Yes, y p puede ingresar en la sección crítica si el test es exitoso.

- Cuando p libera la sección crítica. Envía un mensaje $Relesase$ a cada proceso en $Yes \cup NotNow$.

- Cuando p recibe $Request(t, q)$ desde el proceso q. El proceso p responde OK, si $Perm$ está vacío, y agrega $Request(t, q)$ a $Perm$.

  Si $Perm$ no está vacío, esto es tiene $\{Request(u, s)\}$, entonces realiza lo siguiente. El proceso p inserta $Request(t, q)$ en $Queue$. Sea $Request(v, r)$ el requerimiento que tiene la menor estampilla de tiempo (esto es, el que tiene la mayor prioridad) entre aquellos que están en $Queue$. Si $(t,q) > \min\{(u,s), (v,r)\}$, entonces p responde $WAIT$ a q. De lo contrario, $\{Request(u, s)\}$ tiene la mayor prioridad, p envía un mensaje $Query$ para rescindir el permiso a s, a menos que s se encuentre en la sección crítica, y espera por una respuesta ($Relinquish$ ó $Release$) desde s. Si p ya ha enviado un mensaje de $Queue$ a s y está esperando por una respuesta, entonces no envía nuevamente el pedido. Si p recibe un $Relinquish$, entonces intercambia $\{Request(u, s)\}$ y $\{Request(t, q)\}$ desde $Queue$ a $Perm$, envía un mensaje $WAIT$ a todos los procesos en $Queue$ a los cuales p no ha enviado una respuesta desde la última realización de $Query$, y finalmente envía un mensaje OK a q.

- Cuando p recibe un mensaje $Release$ desde q. El proceso p remueve el requerimiento realizado por q de $Perm$. Si $Queue$ no está vacía, $\{Request(v, r)\}$ es el requerimiento que tiene la mayor prioridad en $Queue$, entonces, p lo mueve desde $Queue$ a $Perm$, envía un mensaje OK a r, y envía un mensaje $WAIT$ a todos los procesos en $Queue$ a los cuales p no les haya enviado una respuesta desde la última realización de $Query$.

- Cuando p recibe un mensaje $Query$ desde q. Si p no está en la sección crítica y q está en Yes, entonces p mueve desde Yes a $NotNow$ y envía un mensaje de $Relinquish$ a q. Si tanto p está en la sección crítica o q no está en Yes, entonces p no hace nada.

El algoritmo garantiza k-exclusión mutua, está libre de interbloqueo y libre de inanición, está demostrado en [78].

Para construir los k-coteries utiliza el método propuesto en [57], cuyo tamaño de quorum es $O(\sqrt{n}\ \log(n))$, donde n representa la cantidad de nodos/procesos en el sistema. La complejidad de mensajes es del $O(\sqrt{n}\ \log(n))$ para el mejor de los casos.

El número de mensajes requeridos para ingresar en la sección crítica require $3|Q|$ en el mejor de los casos, donde Q es un quorum en C. Cuando un proceso falla en la obtención de los permisos utilizando el quorum Q, entonces busca otro quorum para tratar de obtener los permisos, considerando el caso que busque en los k-ésimos quorums disjuntos, se requerirá $6n$ mensajes para ingresar en la sección crítica, donde n es el número de procesos.

4.3.2. Análisis de Quorums para k-exclusión

Para resolver el problema tradicional de la exclusión mutua, una simple solución es utilizar quorums por mayoría [135] de tamaño $\lfloor \frac{n}{2} \rfloor + 1$. Para el problema de 3-exclusión mutua se puede reducir el tamaño de obtención de permisos a $\lfloor \frac{n}{3} \rfloor + 1$. Para $(k + 1)$-exclusión se puede seleccionar un tamaño de quorum $\lfloor \frac{n}{k+1} \rfloor + 1$, mayoría para el problema de la $(k + 1)$-exclusión basada en la propuesta de MAJ [57].

Otra aproximación para alcanzar $(k+1)$-exclusión es considerar k instancias de cualquier solución de exclusión mutua. Un proceso que desea $(k+1)$ accesos exclusivos a los recursos adquiere permisos desde cualquiera de las k instancias. Este modelo de construcción de quorums está referido a DIV [57].

A partir de estas dos aproximaciones, MAJ y DIV, Agrawal y otros [7] proponen un espectro de posibilidades de generar quorums. Se basa en explorar las posibilidades de imponer $(k+1)$-exclusión variando el número r de clases desde 1 hasta k, y definen un método de generación de quorum MAJ_r para cualquier r dividiendo a k. Consideremos un conjunto de n nodos donde $n = kN$ para algún $N \geq 1$, para simplificar la presentación. En MAJ_r, el problema de $(k+1)$ exclusión es resuelto particionando los nodos en r clases disjuntas donde $r = k/i$ para algún entero i. Dentro de cada clase, se eligen los quorums de tamaño q_r, que garantizan que al menos 2 conjuntos desde cualquier colección de $i+1$ conjuntos intersectan dentro de la clase. Se utiliza el siguiente método para construir MAJ_r:

1. Los nodos $1, 2, \ldots, n$ son particionados en r clases de tamaño cada una $\frac{n}{r} = iN$, donde n es el número de nodos.

2. Desde cada clase, todos los subconjuntos de tamaño $q_r = \lfloor \frac{iN}{i+1} \rfloor + 1$ son tomados como quorums, donde $r = \frac{k}{i}$.

Los casos especiales en la construcción de quorums se producen en los extremos, cuando $r = 1$ $\text{MAJ}_1 = \text{MAJ}$, $r = k$ $\text{MAJ}_k = \text{DIV}$.

El costo de comunicación es una de las medidas que se evalúan y está directamente relacionada con el tamaño del quorum. También se considera la disponibilidad del quorum y la cantidad de quorums que se pueden asociar con cada proceso (nodo). Teniendo en cuenta sólo el costo de comunicación es preferible el que genere el menor tamaño de quorum, DIV. En general la disponibilidad MAJ tiene mejor rendimiento, excepto en los sistemas con altas probabilidades de fallos. En el caso de una probabilidad de fallo de nodo baja, el incremento en la disponibilidad mantenido por MAJ sobre cualquier miembro de la familia MAJ_r, $r > 1$, decrementa rápidamente si el número de nodos es grande.

4.3.3. ND k-coteries para EM

Nielsen y Mizuno [115] proponen construir k-coteries que cumplan con la propiedad de ND para utilizarlos en los algoritmos de exclusión mutua basados en quorums. Consideran como definición de k-coterie, a una colección no vacía de conjuntos, Q bajo U si satisface todas las propiedades definidas en 4.2.1 y la siguiente propiedad.

> Propiedad de No-Intersección. Para cualquier m, tal que $1 \leq m \leq k-1$, y para cualquier colección de m pares de quorums disjuntos, $\{G_1, ...,G_m\} \subseteq$ Q, existe un quorum $G \in Q$ tal que $G \cap G_i = \emptyset$ $\forall i, 1 \leq i \leq m$.

Para construir k-coteries presentan un modelo basado en votación por peso y otro en composición.

1. Votación por peso.

   A cada nodo se le asigna un número específico de votos. Un quorum es formado mediante la obtención de al menos la mayoría de los votos. En el caso del consenso por mayoría, cada nodo tiene asignado un único voto, y un quorum está formado obteniendo la mayoría de los votos.

   Sea U un conjunto de nodos no vacío; $q = \lceil (N+1)/(k+1) \rceil$. Sea $\text{Maj}_k = \{\ G \subseteq U \mid |G| = q\ \}$, esto es, Maj_k es una colección de todos los subconjuntos de U conteniendo exactamente q elementos. Por la definición dada en 4.2.1 es un k-coterie.

   Mediante la restricción de los valores permitidos de q y N, se puede utilizar el mismo método para construir ND k-coteries. Eligiendo N tal que $N+1 = (k+1)q$ para algún valor positivo q.

   Sea $Q = \text{Maj}_k$, entonces Q es un ND k-coterie. Está probado en [115] mediante el siguiente teorema.

   Teorema 4.5 *Dado un fijo número positivo entero k, sea* $U = \{x_1, x_2, ..., x_n\}$ *y dado un conjunto de nodos tal que* $N + 1 = (k + 1)q$ *para algún número entero positivo q. Sea* $Q = \{G \subseteq U |\ |G| = q\}$*, entonces q es un ND k -coterie.*

2. Composición.

   La composición se puede utilizar para construir ND coteries y en [111] la proponen para construir ND k-coteries. El siguiente teorema presenta un método general para obtener ND k-coteries.

   Teorema 4.6 *Dado un número entero fijo positivo* $k \geq 2$*, seleccionando números enteros positivos,* k_1 *y* k_2*, tal que* $k_1 + k_2 = k$*. Sea* U_1 *y* U_2 *conjuntos de nodos no vacíos tal que* $U_1 \cap U_2 = 0$*. Suponiendo que* Q_1 *es un ND* k_1*-coterie bajo* U_1*, y* Q_2 *es un ND* k_2*-coterie bajo* U_2*. Sea* $Q = Q_1 \cup Q_2$ *y* $U = U_1 \cup U_2$*. Entonces, Q es un ND k-coterie bajo U.*

   Como ND coterie es lo mismo que ND 1-coterie, entonces es posible construir ND k-coteries a utilizando ND coteries.

Estos métodos para la construcción de quorums, son más resistentes a las fallas de comunicación y caídas de nodo, ya que se construyen utilizando el concepto de ND-coterie para generar ND k-coteries.

4.3.4. k-Exclusión Mutua utilizando Cohorts

Jiang y Huang [72] proponen una solución basada en quorum para el problema de k-exclusión mutua distribuido y se construyen quorums de un k-coterie. La solución utiliza una estructura lógica denominada *Cohorts* para construir los quorums de tamaño constante

función Get_Quorum($Coh(k,l)$ = (C_1, ..., C_l): Estructura Cohort): Set;
var S: Set;
si l < 1 entonces exit(falla);
S = Obtain(C_l);
si $|S|$ = $|C_l|$ - (k-1) entonces return(S); // C_l puede ser el cohort primario
si $|S|$ = 1 entonces return ($S \cup$ Get_Quorum($Coh(k, l-1)$ = (C_1, ..., C_{l-1}))) // C_l puede ser un cohort soporte pero no el primario
si $S = \emptyset$ entonces exit(falla); // No puede formar un quorum

Figura 4.14: Función que genera quorums bajo $Coh(k,l)$

en el mejor de los casos. En el caso que haya nodos que se encuentren inaccesibles el tamaño del quorum puede llegar a ser del $O(n)$, donde n es el número de nodos.

La estructura Cohorts $Coh(k,l)$ = (C_1, ..., C_l) es una lista de conjuntos de pares disjuntos, donde cada conjunto C_i es denominado un *cohort*. La estructura cohort debe observar las siguientes propiedades.

P1. $|C_1| = k$.

P2. $\forall i$, 1< i $\leq l$, $|C_i| >$ max(2k-2, k). (max(2k-2,k) = 2k-2 cuando $k > 1$; max(2k-2,k) = k cuando $k = 1$)

Una estructura Cohorts $Coh(k,l)$ tiene l pares disjuntos de cohorts con el primer cohort teniendo k miembros y los otros cohorts tiene más que 2k-2 miembros. Por ejemplo, {{u_1, u_2}, {u_3, u_4, u_5}, {u_6, u_7, u_8, u_9, u_{10}}} es un Coh (2,3) ya que tiene 3 pares de cohorts disjuntos con el primer cohort y los otros cohort teniendo 2 (=k) y más que 2 (=2k-2) miembros respectivamente.

Un conjunto Q se dice que es un quorum bajo Coh(k,l) si algún cohort C_i en $Coh(k,l)$ es un cohort primario de Q, y cada cohort C_j, j > i, es un cohort soporte de Q, donde:

D1. un cohort C es un cohort primario de Q si $|Q \cap C| = |C|$ - (k-1) (esto es, Q contiene todos excepto k-1 miembros de C).

D2. un cohort C es un cohort soporte de Q si $|Q \cap C| = 1$ (esto es, exactamente tiene un miembro de C).

Por ejemplo, los siguientes conjuntos son quorums bajo Coh(2,2) = {{u_1, u_2}, {u_3, u_4, u_5}}:
Q_1 = {u_3, u_4}, Q_2 = {u_3, u_5}, Q_3 = {u_4, u_5}, Q_4 = {u_1, u_3},
Q_5 = {u_1, u_4}, Q_6 = {u_1, u_5}, Q_7 = {u_2, u_3}, Q_8 = {u_2, u_4}, Q_9 = {u_2, u_5}
Los quorums Q_1 a Q_3 tiene a {u_3, u_4, u_5} como su cohort primario y no necesitan un cohort soporte y los quorums Q_4 a Q_9 tiene como cohort primario a {u_1, u_2} y como cohort soporte a {u_3, u_4, u_5}. Este conjunto de quorums constituye un 2-coterie.

En la figura 4.14 se muestra la función *Get_Quorum*, la cual produce quorums bajo $Coh(k,l)$. Considera que la función *Obtain(C_l)*, la cual es invocada por la función

Get_Quorum, trata de obtener los permisos desde los nodos C_l y retorna el conjunto de nodos de C_l que pueden otorgar el permiso. Los posibles resultados se pueden dividir en 3 casos:

1. Un conjunto de $|C_l|$-k+1 nodos de C_l si más que $|C_l|$-k+1 pueden otorgar su permiso.
2. Un único conjunto de un arbitrario nodo si más que un nodo puede otorgar su permiso.
3. Un conjunto vacío, en los otros casos.

El comportamiento del algoritmo es similar al presentado en sección 4.3.1. El tamaño del quorum bajo el $Coh(k, l)$, con $l > 1$ será:

- En el mejor caso (límite inferior): si $k = 1$ entonces 2, sino $(k > 1)$ k.
- En el peor caso (límite superior): $O(n)$.

4.3.5. Algoritmo de Raymond

Raymond [120], propone una solución al problema extendiendo la propuesta presentada por Ricart y Agrawala [125] para 1-exclusión mutua.

Cada proceso (nodo) tiene asociado un reloj lógico de Lamport [88]. Cuando un proceso (nodo) desea ingresar en la sección crítica, envía un mensaje *broadcast* REQUEST a los otros $(N - 1)$ procesos. En cada requerimiento se le adjunta la estampilla de tiempo de acuerdo al reloj lógico y la identidad del proceso (es única en el sistema).

Cuando un proceso p recibe un mensaje REQUEST de q:

- Si él no requiere una unidad del recurso compartido, inmediatamente otorga el permiso a p enviando un mensaje REPLY.
- Si p está en su sección crítica, demora el envío del mensake REPLY hasta que finaliza la misma.
- Si p está solicitando el acceso a la unidad compartida, compara las estampillas de tiempo de ambos requerimientos. Si son iguales, define la prioridad por la identidad de los procesos. Si la prioridad de q es mayor, entonces le envía un mensaje REPLY; de lo contrario, demora en envío hasta que él finalice su sección crítica.

Cuando p ha recibido $(N - k)$ mensajes REPLY puede ingresar a la sección crítica, ya que está seguro que a lo sumo existen hasta $(k-1)$ procesos ejecutándose concurrentemente en la sección crítica, garantizando la propiedad de seguridad. La utilización de estampillas de tiempo en los requerimientos garantizan la propiedad de vivacidad ya que definen un ordenamiento de los mismos.

Este algoritmo tolera la falla de hasta $(k - 1)$ procesos y aún garantiza la propiedad de seguridad. Con la caída de un proceso se degrada en 1 el grado de concurrencia en la sección crítica. Para ingresar a la sección crítica requiere $2N - k - 1$ mensajes como cota inferior y $2(N - 1)$ como cota superior.

4.3.6. Algoritmo de Makki

Makki y otros [100] proponen una solución basada en *token*. En el sistema hay un único *token* y tiene asociada la siguiente información:

- $cola - token$ que mantiene la lista de nodos (procesos) que quieren ingresar en la sección crítica. Uno de los nodos pertenecientes está etiquetado como el nodo $token - req - nodo$.
- $sem - token$ es un semáforo de propósito general que indica el número de secciones críticas que están disponibles.

Cada nodo (proceso) contiene los siguientes elementos de información:

- $cola - nodo$ inicialmente está vacía, cuando el nodo es designado como el encargado de recibir los requerimientos de token entonces almacena estos requerimientos.
- $token - req - nodo$ mantiene la identificación del nodo actual al cuál se le envían los requerimientos.

En la inicialización del sistema, el token debe ser creado con la $cola - token$ vacía y el $sem - token$ con el máximo valor. Cada nodo debe conocer la identificación del nodo que inicialmente recibirá los requerimientos.

La idea básica del algoritmo es que el token tiene asociada una cola, $cola - token$, con requerimientos que debe atender. Cuando un nodo N_i quiere ingresar en la sección crítica, y conoce el nodo que el $token - req - nodo$ es N_j, le envía un mensaje de requerimiento, TOKEN-REQUEST. Como el nodo N_j está esperando el arribo del token, éste almacena el requerimiento de N_i en su cola local, $cola - nodo$. Cuando arriba el token a N_j, el requerimiento realizado por N_i es ubicado en la cola del token, $cola - token$. Todos los nodos que no tengan requerimientos en la $cola - token$, se les envía un mensaje UPDATE-TOKEN-REQ, informándoles de la actualización del $token - req - nodo$. Para que esta actualización funcione, el nodo que la realiza debe esperar dos períodos de tiempo antes de enviar el token. Se define un período de tiempo como el tiempo máximo requerido para enviar un mensaje de un nodo a otro.

Cuando un nodo recibe el mensaje TOKEN, elimina su requerimiento de la $cola - token$, verifica el valor del semáforo. Si el valor es positivo, entonces decrementa el semáforo y puede ingresar a la sección crítica; sino no modifica el valor del semáforo y espera hasta que recibe un mensaje RELEASE para acceder. Un mensaje TOKEN es enviado al siguiente nodo de la cola.

Cuando un nodo finaliza su sección crítica, un mensaje RELEASE es enviado al nodo ubicado en la k-ésima posición de la $cola - token$, después de haber eliminado su propio requerimiento. De esta manera, se garantiza que a lo sumo k nodos en un instante de tiempo se encuentran en la sección crítica. En el caso que el nodo que recibe los requerimientos, $token - req - nodo$, se encuentre antes de la k-ésima posición entonces el mensaje RELEASE

es enviado a ese nodo. Esto produce que el nodo $token - req - nodo$ reciba los k mensajes RELEASE e incremente el semáforo al valor máximo.

Una de las medidas de rendimiento es el promedio de mensajes requeridos para ingresar en la sección. Para esta propuesta, se define el concepto de ciclo. Un ciclo comienza cuando el nodo $token - req - nodo$ elecciona un nuevo nodo que cumpla esta función, envía los mensajes de actualización y espera los 2 período de tiempo. El ciclo finaliza cuando el nodo $token - req - nodo$ recibe el token e ingresa a su sección crítica. Durante el ciclo, todo los nodos de la $cola - nodo$ hasta el nodo $token - req - nodo$ reciben el token e ingresan en sus respectivas secciones críticas. Al finalizar un ciclo comienza uno nuevo. En un ambiente donde la carga de requerimientos es baja se requieren $N + 2$ mensajes. En un ambiente donde la carga de requerimientos es alta se requieren 3 mensajes por sección crítica y en general $(N/m) + 2$ mensajes por sección crítica, donde m es la cantidad de nodos de la $cola - token$.

4.3.7. Algoritmo de Bulgannawar/Vaidya

El algoritmo propuesto por Bulgannawar y Vaidya [32] utiliza *tokens* para resolver el problema de la k-exclusión mutua. En el sistema hay k *tokens* e inicialmente el *token* t lo posee el nodo t, $1 \leq t \leq k$.

Cada nodo tiene como estructura de datos asociadas:

- Un arreglo *puntero* con una entrada para cada *token*. Estos punteros definen k bosques (conjunto de árboles) correspondientes a los k tokens. El t-bosque se refiere al bosque correspondiente al token t formado por el *puntero[t]* de cada nodo. *puntero[t]* del nodo j contiene el indentificador del padre del nodo j en el t-bosque (correspondiente al *token* t). Si el *puntero[t]* en el nodo j comienza igual a j significa que el nodo j está en la raíz de un árbol para el *token* t. Inicialmente, *puntero[t]* para cada nodo es asignado igual a t, $1 \leq t \leq k$.

- Una cola FIFO *cola-nodo*, que mantiene los identificadores de nodos que han requerido un token t si el nodo también está esperando por el *token* t, esto es, si el nodo A está esperando por el *token* t y recibe un requerimiento del nodo B para el *token* t, entonces el identificador del nodo B es guardado en la *cola-nodo* del nodo A.

- $token - id$, si existe, mantiene el identificador del token que está presente en el nodo.

- $token - esperado$ mantiene la identificación del token por el cuál está esperando.

Cada *token* tiene asociada una estructura de datos que se envía con el mismo.

- Una cola *cola-token* que guarda la identificación de los nodos a los cuales el *token* sea propagado.

- $request - modifier - tags$, este tag es adjuntado en cada entrada de la $cola - token$.

Cuando el nodo (proceso) I quiere acceder a la sección crítica, chequea si tiene disponible un *token*. Si es así entonces toma el *token* y puede acceder a la sección crítica. En caso contrario elige un *token* t utilizando alguna heurística (esto puede ser en forma aleatoria) y luego envía un mensaje REQUEST(I,t) y espera hasta que recibe un *token*.

Cuando el nodo I sale de la sección crítica libera el *token*, si hay requerimientos por el *token*, entonces envía un mensaje TOKEN(t) al primer nodo de la *cola-token*; sino envía un mensaje INFORM(I,t) a v nodos avisándoles que dispone de un *token*. Esta última acción la introduce para reducir la distancia de un nodo al *token*.

Cuando se recibe un mensaje INFORM(Y, t) se asigna a *puntero*[t] el valor de Y. Con este mensaje se puede reducir la distancia de un nodo al *token*. Cuando se recibe un mensaje TOKEN(t) chequea si el *token* t es el mismo *token* que ha requerido, sino es el mismo modifica su requerimiento y luego accede a la sección crítica. Cuando se recibe un mensaje REQUEST(Y,t) significa que el nodo Y requiere el *token* t. Si el nodo I posee un *token* u (puede o no ser igual a t), agrega el nodo Y en la *cola-token* del *token* u. En el caso que $t \neq u$ registra esta información. Si el nodo I no está en la sección crítica, entonces actualiza *puntero[t]* con Y y envía el *token* al nodo Y. Si el nodo I no posee un *token* y ha requerido el *token* t entonces Y es guardado en la *cola-nodo*. Si el nodo I no posee un *token* y no ha requerido el *token* t, entonces propaga el requerimiento al *puntero[t]* y asigna *puntero[t]* = Y.

La performance del algoritmo depende de las elecciones en los mecanismos de decisión.

- El *token* requerido para acceder a la sección crítica (la heurística de elección).
- El número (v) de mensajes INFORM enviados en la sección crítica.
- Los nodos destinos para los mensajes INFORM.

4.3.8. Métricas para k-exclusión

En el momento de la elección es importante conocer los requerimientos de la aplicación y los atributos que presenta el algoritmo a seleccionar. En el caso de los algoritmos basados en quorum se consideran las siguientes tópicos:

- Costo de comunicación: directamente relacionado con el tamaño del quorum.
- Disponibilidad y tolerancia a fallas: son determinadas por el número de formas en el cual un quorum puede ser construido a partir de un conjunto de nodos de la red.

Cuando MAJ y DIV son evaluados puramente en términos del costo de comunicación incurrido en alcanzar $(k + 1)$-exclusión, DIV es preferible a causa del tamaño de quorum más pequeño.

MAJ tiene una mejor disponibilidad que DIV como también todos los otros MAJ_r, excepto para sistemas con alta probabilidad de fallas en los nodos. Los quorums generados por MAJ

	MAJ_r	MAJ	DIV	COHORTS
Nro. total de Conjunto	$r\binom{iN}{q_r}$	$\binom{n}{q_{\text{MAJ}}}$	$k\binom{\frac{n}{k}}{q_{\text{DIV}}}$	
Tamaño del quorum	$q_r = \lfloor \frac{iN}{i+1} \rfloor + 1$	$q_{\text{MAJ}} = \lfloor \frac{n}{k+1} \rfloor + 1$	$q_{\text{DIV}} = \lfloor \frac{n}{2k} \rfloor + 1$	Lím. inferior constante Lím Sup O(n)

Cuadro 4.3: Algoritmos basados en quorum

tienen el doble de tamaño que los generados por DIV, y son siempre mayores que los quorums de MAJ_r para $r > 1$.

El modelo de k-exclusión mutua utilizando la estructura lógica cohorts, genera un quorum cada vez que se require acceder. En el mejor de los casos, puede generar un quorum con un número constante de nodos, si k=1 con 2 nodos, en el resto de los casos con k nodos. En el caso que haya fallas en los nodos, genera un quorum con mayor cantidad de nodos, el límite superior es de $O(n)$. En el cuadro 4.3 se muestra una tabla comparativa de los algoritmos basados en quorum, detallando la cantidad de quorums y el tamaño de los mismos.

En los algoritmos basados en *token* los parámetros de consideración para la elección son: *tiempo promedio* para acceder a la sección crítica, *número de mensajes promedio* para la entrada a la sección crítica, *información promedio por mensaje*. También se puede considerar si se particiona la red en grupos, si está sobrecargado de requerimientos el sistema o no.

Raymond [120] (utilizando permisos) presenta un límite inferior a $2N - k - 1$ en el número de mensajes requeridos por entrada en la sección crítica y un límite superior de $2(N - 1)$ mensajes. El algoritmo soporta hasta $k - 1$ caídas de nodos. Makki y otros [100] (utilizando un token) presentan como límite inferior 3 mensajes cuando la carga del sistema es alta, un límite superior de $N+2$ mensajes cuando la carga es baja y en general $(N/m)+2$ mensajes por entrada en la sección crítica. Requiere tener el tiempo máximo para enviar un mensaje de un nodo a otro. Bulgannawar y Vaidya [32], proponen una solución basada en token, cuyo rendimiento depende de las elecciones realizadas en los tres mescanismos de decisión.

4.4. Problema de h-requeridos de-k Exclusión Mutua

El problema de la k-exclusión mutua es una extensión del problema original, en donde existen hasta k recursos que se pueden utilizar en el mismo instante y cada proceso requiere una unidad de los k recursos. Este problema es una generalización del problema de k-exclusión mutua, donde hay k unidades de recursos compartidos y cada proceso requiere h unidades ($1 \leq h \leq k$) al mismo tiempo. Un ejemplo de este tipo es la utilización del ancho de banda en la comunicación. Una línea de comunicación con un ancho de banda fijo k es compartido por varios procesos. Cada proceso puede comunicar audio y video por esa línea. Como el ancho de banda requerido para la transferencia de audio y video difiere

significativamente, el ancho de banda necesitado difiere entre los requerimientos.

Raynal [123] define el problema h-requeridos de-k exclusión mutua de la siguiente manera. Hay k unidades idénticas de recurso que son compartidas por los procesos en U. Cada unidad no puede ser asignada a más de un proceso en el mismo instante de tiempo. Un proceso requiere h ($1 \leq h \leq k$) unidades de recursos todas juntas, y para evitar interbloqueo, el proceso es bloqueado hasta que obtiene todas las unidades requeridas. El proceso puede comenzar a utilizar las unidades y, cuando finaliza, las libera todas juntas.

Si $h = 1$ para cada requerimiento, el problema h-requerido de-k exclusión mutua corresponde con el problema de k-exclusión mutua; más aún, si k=1 obtenemos el problema tradicional. Un algoritmo de h-requerido de-k exclusión mutua debe satisfacer las siguientes propiedades:

- seguridad: cada unidad de recurso puede ser utilizada a lo sumo por un proceso en todo instante de tiempo.
- vivacidad: todos los requerimientos eventualmente deben ser satisfechos.

La seguridad es obtenida garantizando que la siguiente inecuación siempre se cumpla (donde h_j es el número de unidades actualmente asignadas al proceso p_j).

$$\sum_{j \in U} h_j \leq k \tag{4.2}$$

Esta es una invariante asociada con el problema de h-requerido de-k exclusión mutua. Un punto clave para alcanzar seguridad es detectar todos los conflictos que pueden surgir en la adquisición de recursos. La vivacidad requiere que el sistema siempre progrese a través de la sección crítica (libre de interbloqueo) y los conflictos no se resuelven siempre beneficiando un subconjunto de procesos (libre de inanición).

Una forma de resolver este problema es utilizar un algoritmo de k-exclusión mutua y ejecutar h requerimientos cuando un proceso necesita h unidades de recursos compartidos. Esta solución presenta algunos inconvenientes de eficiencia y puede ocasionar interbloqueo. Por ejemplo, si se considera que p_i requiere h_i unidades y p_j requiere h_j tal que $h_i + h_j > k$, puede ocurrir la siguiente situación: p_i ha obtenido $h_x < h_i$ unidades, p_j ha obtenido $h_w < h_j$ unidades y $h_x + h_w = k$. Este estado genera un interbloqueo.

Manabe y otros [101] han demostrado que los k-arbiters, una solución basada en quorum, se pueden utilizar para resolver este problema. Otra solución es propuesta en [103], y está basada en quorum que utiliza un *(h,k)-arbiter*. *(h,k)-arbiter* es un conjunto quorum para cada h ($1 \leq h \leq k$), $\{Q_{h,k} | 1 \leq h \leq k\}$, donde $Q_{h,k}$ es un conjunto de quorums. Un proceso que utiliza un quorum en $Q_{h,k}$ es que desea utilizar h unidades de recurso compartido.

4.4.1. Algoritmo utilizando coteries

Baldoni y otros [25], proponen una solución al problema de h-requeridos de k-exclusión mutua es solicitando en un único requerimiento las h unidades del recurso y esperando hasta

que estén todas disponibles como en 1 y k-exclusión mutua. Se puede obtener un algoritmo extendiendo el algoritmo de Maekawa [99]. Utilizando tal aproximación, cada proceso puede actuar tanto como un proceso requiriente y como un proceso árbitro para resolver conflictos en la adquisición de unidades del recurso. La solución consiste en dividir el código en dos partes principales: la primera parte es ejecutada cuando un proceso, p_i, el cual requiere algunas unidades del recurso; está formado por el protocolo de entrada y de salida de su sección crítica. La segunda parte está compuesta por dos manejadores de mensajes: uno para los mensajes requirientes y otro para mensajes de liberación (*release*). El manejador de mensajes se ejecuta cada vez que un mensaje es recibido.

sección de entrada

el proceso p_i selecciona un quorum Q y envía un mensaje *request* de h unidades del recurso a cada miembro de Q y espera por la recepción de un mensaje de *permiso* desde cada proceso en Q.

sección crítica

sección de salida

p_i envía un mensaje *release* de h unidades del recurso a cada miembro de Q.

recepción mensaje request desde p_k

Si

$$\sum_{j \in U} c_j + h \leq k \tag{4.3}$$

donde h es el número de unidades requeridas desde p_k y c_j es el número de unidades del recurso actualmente utilizadas desde p_j desde el punto de vista del proceso p_i (esto es, procesos a los cuales el proceso p_i ha enviado un mensaje de *permiso* y no ha recibido aún un mensaje de *release*)
entonces p_i envía un mensaje de *permiso* a p_k y c_k es inicializado en h
sino el mensaje de *request* de p_k es insertado en una lista de espera.

recepción de mensaje release desde p_k

c_k es inicializado en 0 y p_i trata de enviar un mensaje de *permiso* a algún requerimiento de la lista de espera utilizando la relación de 4.3

Se puede garantizar seguridad ya que se considera la relación 4.3 para brindar el mensaje de *permiso* y además se mantiene la propiedad de intersección para k-coterie, si no se mantiene esta propiedad entonces no se puede garantizar seguridad. La vivacidad está relacionada con la no ocurrencia de inanición e interbloqueo. En este caso podría ocurrir

inanición si las reglas para resolver conflictos benefician a un subconjunto de proceso. Interbloqueo puede ocurrir ya que la comunicación es asincrónica y se podría otogar permisos que deriven en un bloqueo entre procesos.

4.4.2. Algoritmo utilizando k-arbiters

Manabe y Tajima [103], presentan un algoritmo distribuido basado en quorum utilizando (h, k)-arbiter. Cada proceso utiliza un diferente arbiter para cada h $(1 \leq h \leq k)$.

Conceptos de (h, k)-arbiter

Un (h, k)-arbiter, $\vartheta_{h,k}$, es un conjunto de arbiters $\{Q_{h,k} | 1 \leq h \leq k\}$, donde $Q_{h,k}$ es un conjunto de quorums. Si un proceso p desea utilizar h unidades de los recursos compartidos, p selecciona un quorum $q_{h,k} \in Q_{h,k}$ y el comporatmiento es el mismo que utilizando k-arbiter. La diferencia que presenta con respecto al k-arbiter es en la propiedad de intersección.

Un conjunto no tiene elementos repetidos, en cambio una bolsa (bag) puede contener elementos repetidos.

Definición 4.1 *Un patrón de requerimiento para la exclusión mutua de h-requeridos de-k, r_k es una bolsa de valores enteros positivos hasta k.*

Definición 4.2 *Un patrón de requerimiento r_k es conflictivo si solo si $\sum_{h \in r_k} h \geq k+1$. Un patrón de requerimiento conflictivo r_k es crítico si solo si $\forall h \in r_k$, $r_k - \{h\}$ no es conflictivo.*

Definición 4.3 *Para un patrón de requerimiento r_k, un bolso $\{q \in Q_{h,k} | h \in r_k\}$ es denominado un quorum asignado para r_k. Sea $QA(r_k)$ el conjunto de quorums asignados a r_k.*

La propiedad de intersección está definida a través del siguiente teorema, que está demostrado en [103].

Teorema 4.7 *La propiedad de seguridad es garantizada si solo si*
$$\forall \Gamma \in QA(cr_k),\ \bigcap_{q \in \Gamma} q \neq \emptyset$$
se satisface para cualquier patrón de requerimiento conflictivo crítico cr_k.

Algoritmo

El algoritmo resuelve el problema de h-requeridos de-k exclusión mutua utilizando (h, k)-arbiter, con las propiedades de seguridad y vivacidad. El comportamiento es el siguiente. Cada proceso mantiene un reloj lógico de Lamport [88]. El valor del reloj lógico del proceso p sea c_p. Cuando envía un mensaje m, el valor actual de c_p es enviado en el mensaje. Sea c_m el valor del reloj enviado en m cuando p recibe el mensaje, entonces p actualiza $c_p = \max(c_m, c_p) + 1$.

Cada requerimiento es una tupla (h, p, c), donde:

- h es un número de unidades.
- p es el proceso requiriente.
- c el valor del reloj lógico cuando p realiza el requerimiento.

Una prioridad es asignada a cada requerimiento (h, p, c), esto es $(h, p, c) > (h', p', c')$ si $c < c'$ ó $(c = c'$ y $p < p')$. De esta manera, se obtiene un ordenamiento total de los requerimientos. Cuando p inicia un requerimiento (h, p, c), selecciona el quorum $q \in Q_{h,k}$ y envía un mensaje de *request* (h, p, c) a cada proceso en q. Si p recibe una respuesta OK de todos los procesos en q, entonces puede utilizar h unidades de los recursos compartidos. Cuando p finaliza la utilización de las unidades, envía un mensaje *release* a cada proceso en q. El proceso p puede recibir un mensaje de *cancel* mientras está esperando por los mensajes de aceptación (OK).

Cada proceso tiene inicialmente k permisos. Sea x_p los permisos que actualmente tiene p. También cada proceso mantiene una cola de prioridades de requerimientos, que inicialmente está vacía. Se registra el estado de cada uno de los requerimientos: *wait*, OK ó *cancel*. Cuando p recibe un mensaje *request* (h', p', c'), lo inserta en la cola de prioridades e inicializa el estado a *wait*.

Si el mensaje de *request* (h', p', c') satisface que el total de unidades requeridas en los requerimientos de mayores prioridades en la cola no es mayor que $k - h'$, y $x_p \geq h'$, p envía un mensaje OK a p', cambia el estado del requerimiento a OK, y actualiza $x_p := x_p - h'$.

Puede haber un *request* (h'', p'', c'') en la cola de prioridades cuyo estado sea OK y el número total de requerimientos desde las solicitudes de alta prioridad sea ahora más que $k - h''$ como un resultado de insertar nuevos requerimientos. Múltiples requerimientos pueden satisfacer esta condición como resultado de insertar uno nuevo. Los OK de estos requerimientos deben ser cancelados, para evitar un posible interbloqueo. Esto es, p envía un mensaje *cancel* a p'' para cancelar el OK y cambiar el estado del requerimiento a *cancel*.

Cuando p'' recibe un mensaje *cancel* desde p, le envía un mensaje de respuesta *cancelled* a p si es que aún no ha recibido un OK de cada uno de los procesos en q. Entonces espera por otro mensaje OK desde p.

Cuando p recibe un mensaje *cancelled* desde p'', cambia el estado del requerimiento (h'', p'', c'') a *wait*. p ejecuta $x_p := x_p + h''$ y trata de enviar un mensaje OK a un requerimiento de mayor prioridad basado en la condición propuesta para enviar un mensaje de OK.

Cuando p recibe un mensaje *release* desde p'', elimina el requerimiento (h'', p'', c'') desde la cola de prioridad, ejecuta $x_p := x_p + h''$, y trata de enviar un mensaje OK al otro requerimiento basado en la condición propuesta para enviar un mensaje OK.

Complejidad de Mensajes

El número de mensajes enviados por cada requerimiento, en el caso que no existan conflictos, es el siguiente.

1. p envía un mensaje de *request* a cada proceso en algún $q \in Q_{h,k}$.
2. Cada proceso en q envía un mensaje OK a p.
3. p ingresa en la sección crítica y finaliza. p envía un mensaje *release* a cada proceso en q.

Por lo tanto, la complejidad de mensajes es $3|q_{h,k}|$ por requerimiento, donde $|q_{h,k}|$ es el tamaño del mayor quorum en $Q_{h,k}$.

En el caso que exista conflicto, los requerimientos de baja prioridad deben ser cancelados, en cuyo caso el detalle de mensajes enviados es el siguiente.

1. p envía un mensaje de *request* a cada proceso en algún $q \in Q_{h,k}$.
2. Cada proceso $u \in q$ ha enviado k OK a otros requerimientos cuya unidad requerida sea 1. El requerimiento desde p tiene una prioridad mayor que estos requerimientos. Esto es, u envía un mensaje *cancel* a h procesos requirientes para cancelar el mensaje OK.
3. Cada proceso envía un mensaje de respuesta *cancelled* a u. Entonces, u envía un mensaje OK a p.
4. p ingresa en la sección crítica y finaliza. p envía un mensaje *release* a cada proceso en q.
5. u recibe un mensaje *release* y envía un mensaje OK otra vez a los procesos cancelados.

Note que después que p recibe un OK, puede ser cancelado por otro requerimiento de un proceso r. Los mensajes para este procedimiento son contados en el procedimiento de r. La complejidad de mensajes para este caso es $(3h + 3)|q_{h,k}|$ por requerimiento.

4.5. Resumen

El modelo de pasaje de mensajes se utiliza en la implementación de otros modelos de comunicación entre procesos, como es el caso del paradigma de memoria compartida distribuida. Por este motivo una de las ventajas que presenta es el costo de implementación y como desventaja la falta de transparencia. Para algunas extensiones del problema tradicional de exclusión, h-requeridos de-k exclusión, se adapta mejor para alcanzar una solución.

En este capítulo, se presentaron diferentes modelos para resolver el problema de exclusión mutua y algunas extensiones como k-exclusión, h-requeridos de-k exclusión, utilizando mensajes. Los algoritmos basados en mensajes se los puede clasificar en: basados en *token* o basados en *quorum* (permiso).

En las secciones 4.2, 4.3 y 4.4 se mostraron algunas propuestas presentadas en la literatura para resolver el problema de la exclusión mutua, k-exclusión mutua y h-requeridos de-k exclusión, respectivamente. En todas las propuestas que se han presentado, además de

garantizar las propiedades de un buen algoritmo de exclusión mutua, se mide la complejidad del mismo a través de la cantidad de mensajes requeridos, el tiempo de espera, la cantidad de información necesaria y, si soporta fallas de nodos o de comunicación.

Capítulo 5

Exclusión Mutua para Grupos

5.1. Presentación del Problema

Consideremos un conjunto de n procesos $p_0, p_1, .., p_{n-1}$ los cuales trabajan en forma independiente o en forma cooperativa en un grupo. Los procesos pueden participar de cualquiera de los diferentes m grupos G_0, G_1, ... , G_{m-1}. Cada uno de los grupos, cuando se encuentra activo utiliza el/los recurso/s por los cuales compite en el sistema. En un determinado instante, sólo un grupo puede estar *activo*. En el grupo, participan todos los procesos que están interesados en el trabajo.

Inicialmente cada uno de los procesos está trabajando individualmente. Cuando desea trabajar en equipo, elige el *grupo*. Se considera que cada proceso trabaja en equipo por un tiempo finito, y que puede participar en cualquiera de los diferentes grupos. En la figura 5.1, se observa que los procesos P_2, P_3 y P_4 están actualmente vinculados al grupo G_2; éste se encuentra activo y con permiso para utilizar el recurso, significando que los procesos que lo integran están concurrentemente utilizando el mismo. Los procesos P_1 y P_6 están integrando el grupo G_3 que está compitiendo por alcanzar el permiso de utilizar el recurso.

En el caso en que se considerara que cada grupo estuviera formado por un solo proceso, el problema se reduciría al modelo convencional de exclusión mutua para n procesos, donde solamente un proceso a la vez puede estar en la sección crítica. Esta situación es una extensión del problema tradicional de la exclusión mutua, donde k procesos pueden compartir la utilización del recurso en un instante de tiempo. Para resolver este problema se requiere un algoritmo que satisfaga los siguientes requerimientos:

- *Exclusión Mutua*: si algún proceso está trabajando en un grupo, no puede haber otro proceso trabajando en un grupo diferente simultáneamente.
- *Demora Limitada (libre de inanición)*: un proceso que desea participar de un grupo eventualmente tendrá éxito.
- *Entrada Concurrente*: si algunos procesos están interesados en un grupo y no hay un

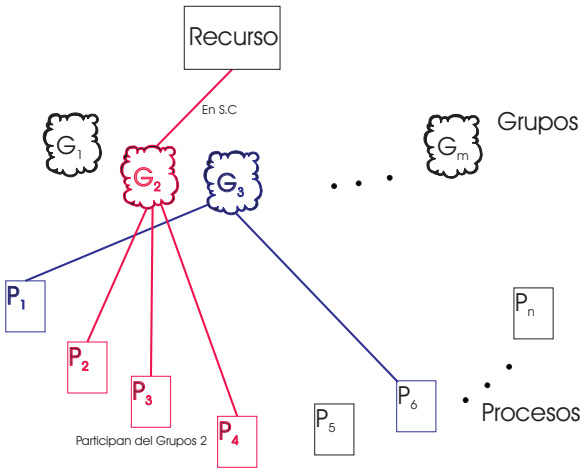


Figura 5.1: Relación entre grupo y procesos

proceso interesado en otro grupo, entonces los procesos pueden participar concurrentemente del grupo.

- *Libre de interbloqueo*: cuando la sección crítica está disponible, los grupos no deberían esperar indefinidamente y alguno debería obtener el permiso para acceder.

En este capítulo se presenta un modelo basado en dos actores para resolver el problema de Exclusión Mutua para Grupos de Procesos, que se puede aplicar a los algoritmos que solucionan el problema original de exclusión mutua, presentados en los capítulos 3 y 4, para extenderlos y aplicarlos a esta situación diferente. Se proponen implementaciones de este modelo aplicadas a algoritmos basados en memoria compartida y en pasaje de mensajes. Además se presentan algunas soluciones para el modelo basado en un actor bajo el paradigma de memoria compartida y se comparan las diferentes implementaciones. Se brindan ejemplos de aplicación que se resuelven utilizando exclusión mutua para grupos de procesos.

5.2. Modelo Basado en un Actor

El modelo basado en un actor tiene un único tipo de componente que es el actor *proceso* que utiliza el recurso en forma compartida con otros actores *procesos* que realizan su trabajo en forma conjunta, esto es participan del mismo grupo de interés. El actor *proceso* es el que compite por acceder al recurso participando de un determinado *grupo*. El actor *proceso* puede participar de diferentes grupos durante el transcurso de su trabajo, pero en un determinado instante de tiempo participa de un único grupo.

En la figura 5.2, se muestra un ejemplo de competición y concurrencia con este modelo. Se observa que los procesos P_1, P_2 y P_7 están asociados al grupo G_3 y utilizando el recurso.

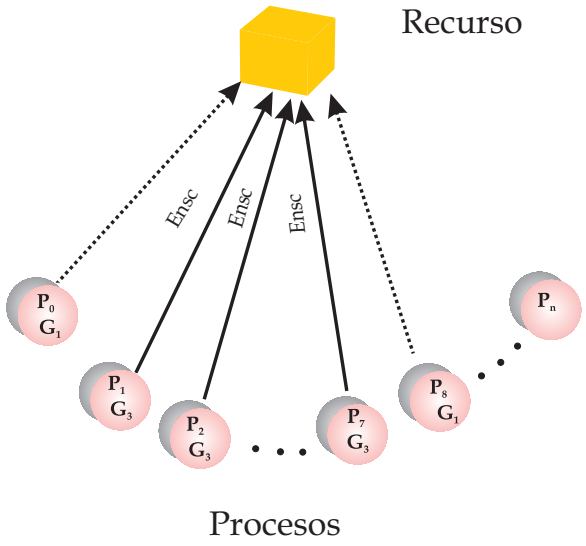


Figura 5.2: Ejemplo de Competición y Concurrencia - Un Actor

En esta sección se presentan un par de algoritmos que resuelven el problema de exclusión mutua para grupos de procesos utilizando el modelo de memoria compartida.

5.2.1. Algoritmo CTP-*m*

Yuh-Jzer Young [74] presenta el problema a partir de un modelo (paradigma). Existen n filósofos p_0, p_1, ..., p_{n-1} que utilizan su tiempo en pensar y dialogar en un foro. Los filósofos tienen m diferentes foros de discusión para seleccionar, F_0, F_1, ..., F_{m-1}, por la capacidad de la sala de reunión y, solamente un foro se puede realizar por vez. Más de un filósofo puede estar en un foro simultáneamente.

El tiempo y la concurrencia son dos criterios para evaluar las soluciones del problema. El tiempo está relacionado con cuánto un filósofo debe esperar hasta que pueda utilizar o participar en un foro (desde el tiempo que requiere el foro). Dos diferentes medidas pueden utilizarse: *paso (passage)* y *vuelta de paso (round of passage)*.

Un *paso* se refiere al intervalo durante el cual un filósofo pasa a través de la sala de reunión (sección crítica). Cada *paso* tiene un atributo $< p, F >$ que denota que el *paso* está hecho por p para utilizar el foro F. Una *vuelta de paso* a través del foro F (o simplemente una vuelta de F) es definida como el conjunto máximo de *pasos* consecutivos tal que (1) todos los *pasos* en el conjunto pueden ser atendidos en el foro F, y (2) no hay otro *paso* pretendiendo un foro diferente de F, que se intercepte con el conjunto de pasos atendidos en el foro F. Esto es, no hay otro *paso* pretendiendo un foro diferente de F que ocurra entre el tiempo del *paso* más temprano en el conjunto de comienzo hasta el tiempo del último *paso* en el conjunto finalización.

La solución al problema en forma generalizada, esto es, considerando m foros se muestra

/* asumiendo p_i está utilizando F_k */
1.- flag(i) = <request, F_k >
2.- successor(i) = $\perp$
3.- Repetir
4.- flag(i) =<request, F_k >
5.- Mientras successor(i) $\neq F_k \wedge$ next_op(turn) $\neq F_k$ Hacer skip
6.- flag(i) = <in_cs, F_k >
7.- Hasta successor(i) $= F_k \vee$ (none_in_cs$(\overline{F_k})$ $\overrightarrow{\wedge}$ no_successor$(\overline{F_k})$ $\overrightarrow{\wedge}$ (turn $= F_k \vee$ all_passive(turn)))
/* comienzo de la sección crítica */
8.- Si successor(i) $\neq F_k$ Entonces {
9.- turn = next_op(F_{k+1})
10.- para j = 0 hasta n-1, j $\neq$ i, hacer /* comienza a capturar filósofos */
11.- Si flag(j) $\in$ {<request, F_k>, <in_cs, F_k>} Entonces successor(j) = F_k
12.- « attend foro F_k »
13.- flag(i) = < passive, $\perp$ >
/* Finalización de la sección crítica */

all_passive(F_g) $\equiv \forall$ j, j $\neq$ i, 0 $\leq$ j $\leq$ n-1 : flag(j) = <state,op> $\Rightarrow$ op $\neq F_g$

none_in_cs$(\overline{F_k})$ $\equiv \forall$ j, j $\neq$ i, 0 $\leq$ j $\leq$ n-1 : flag(j) = <state,op> $\Rightarrow$ (state $\neq$ in_cs $\vee$ op $= F_k$)

no_successor$(\overline{F_k})$ $\equiv \forall$ j, j $\neq$ i, 0 $\leq$ j $\leq$ n-1 : flag(j) = <state,op> $\Rightarrow$
$\neg(\exists\, l,\ l \neq$ k : successor(j) $= F_l \wedge$ op $= F_l)$

/* La función *next_op*(F_g) retorna el primer foro F_h en la secuencia F_g, F_{g+1}, ..., F_{g+m-1} tal que algún filósofo ha requerido F_h pero ningún filósofo ha requerido F_g, F_{g+1}, ..., F_{h-1}. */
1.- *next_op* (F_g) :: {
2.- next = g + m
3.- para j=0 hasta n-1 hacer {
4.- let flag(j) = <state,op>
5.- Si op $\neq \perp$ Entonces
6.- let op = F_l
7.- Si $l <$ g Entonces $l = l +$ m
8.- Si $l <$ next Entonces next = l }}
9.- $F_{next(mod\ m)}$
10.- }

Figura 5.3: Algoritmo CTP-m

en la figura 5.3. El algoritmo utiliza las siguientes variables:

- turn. Inicialmente arbitraria.
- flag: array [0..n-1] of <state,op>, donde flag(i) registra el estado de p_i's y el foro en el cual desea participar. Hay varios posibles estados: *passive, request, in_cs*. El estado *passive* significa que el filósofo no intenta participar de algún foro; *request* significa que el filósofo desea participar en algún foro; e *in_cs* significa que el filósofo ha obtenido un permiso temporario para su requerimiento. El acceso a flag(i) es atómico.
- successor: array [0..n-1] of (F_0, F_1, ..., F_{m-1}, $\perp$), donde successor(i) indica para cual foro p_i es capturado para participar, o successor(i) = $\perp$

El algoritmo garantiza exclusión mutua y espera limitada. Si un filósofo está esperando por un foro, entonces eventualmente algún filósofo podrá ingresar en la sección crítica para participar en el foro.

El algoritmo de Joung [74] utiliza un arreglo de *flag(i)* el cual debe ser leído por todos los procesos y más de una vez. Esta solución presenta una complejidad alta en tiempo, aún en el caso que no exista contención. Otra desventaja del algoritmo es la forma en que efectúa la espera, ya que usa una espera ocupada sobre las variables compartidas.

5.2.2. Algoritmo EM Grupo sobre Peterson

El algoritmo de exclusión mutua para grupos, presentado por Vidyasankar [136], se basa en el algoritmo de exclusión mutua para n procesos de Peterson. El algoritmo de Peterson es inicialmente reescrito de la siguiente manera:

Proceso$_i$
1.- para j = 1 hasta n-1 hacer
2.- flag(i) = j
3.- turn(j) = i
4.- wait until (findcountp$_i$(j) =1) $\vee$ (turn(j) $\neq$i)
5.- sección crítica
6.- flag(i) = 0

Se considera un tiempo particular de referencia tal que para $1 \leq j \leq n-1$:

$$countp(j) =\mid \{k : flag(k) \geq j\} \mid \tag{5.1}$$

que representa el número de procesos en el nivel j o superior.

La función *findcountp$_i$(j)* es el cálculo de buscar el valor de *countp(j)*. Con esta interpretación probó que el algoritmo cumple con las condiciones de un buen algoritmo para exclusión mutua con n procesos.

Para que el algoritmo soporte la exclusión mutua para grupos, se asume que cada proceso P_i que compite, primero declara el foro en que desea participar en una variable atómica

$forum(i)$ de simple escritura y múltiple lectura; incorpora la función *countf(j)* como el número de foros requeridos por los procesos en el nivel j o mayor. Define para $1 \leq j \leq n-1$:

$$countf(j) =\mid \{forum(k) : flag(k) \geq j\} \mid \tag{5.2}$$

Se utiliza *countf(n)* para denotar el número de foros requeridos por los procesos en la sección crítica. Se desea utilizar *countf(j)* = 1 como la condición *nonpushing*. Esto es, si no hay un proceso en el nivel j o superiores que requieran un foro diferente del *forum(i)*, entonces P_i puede avanzar al nivel j. El algoritmo que obtiene es el siguiente:

Proceso$_i$
1.- Seleccionar forum(i)
2.- para j = 1 hasta n-1 hacer
3.- flag(i) = j
4.- turn(j) = i
5.- wait until (findcountf$_i$(j) =1) $\vee$ (turn(j) $\neq$i)
6.- sección crítica
7.- flag(i) = 0
8.- clear forum(i)

Esta solución utiliza variables compartidas, *flag* leída por todos los procesos y *turn* leída y escrita por todos los procesos. Además, la forma en que efectúa la espera es una espera ocupada sobre las variables compartidas.

5.2.3. Algoritmo de Un Actor

Este algoritmo fue presentado en [44], se basa en el algoritmo Tournament (en la sección 3.3.4). El proceso inicialmente está en la sección resto y cuando ingresa en la sección de entrada, selecciona el grupo en el cual va a participar y comienza la competencia por acceder al recurso. La competencia se realiza por niveles y, cuando supera el último nivel, puede acceder a la sección crítica. Si en un determinado nivel se encuentra con un proceso que quiere acceder por el mismo grupo entonces compiten conjuntamente para ingresar a la sección crítica. En la figura 5.4 se muestra la situación en la cual los procesos P_5 y P_7 compiten en el nivel 2 y comparten el mismo grupo G_3 y luego avanzan juntos.

En el modelo presentado cuando un proceso P_i compite con el proceso P_j en el nivel k e identifica que es un compañero entonces espera hasta que el proceso P_j acceda al recurso y luego participa en forma cooperativa. El algoritmo presentado se basa en el paradigma de memoria compartida distribuida sobre las variables de control utilizadas. En la figura 5.5 se muestran las variables compartidas.

Las variables *flag(i)* y *grupo(i)* son escritas por el proceso P_i y leídas por el resto de los procesos; la primer variable indica el nivel de competencia del proceso y la segunda en cuál grupo está vinculada. Las variables *pganador* y *gganador* pueden ser leídas y escritas por todos los procesos; la variable *pganador* contiene la información de cuál es el primer proceso que ingresó en la sección crítica de todos los que trabajan cooperativamente y *gganador* contiene la información de cuál es el grupo al que pertenecen los procesos que están

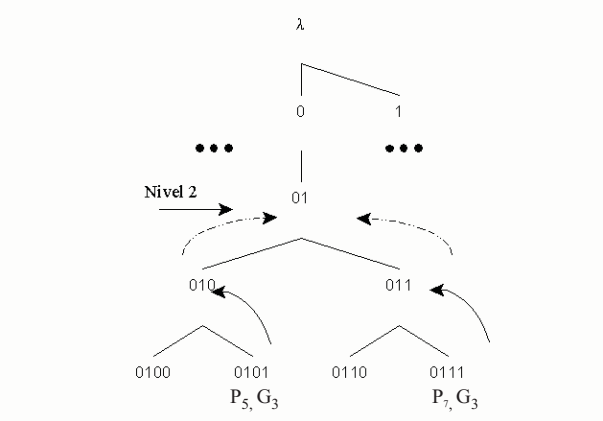


Figura 5.4: Procesos Concurrentes

∀ flag(i): 0 ≤ i ≤ (n-1), flag(i) = 0 inicialmente
∀ grupo(i): 0 ≤ i ≤ (n-1), grupo(i) = -1 inicialmente
∀ oponente_compa(i,i): 0 ≤ i ≤ (n-1), oponente_compa(i,i) = 0 inicialmente
para cada cadena binaria x de a lo sumo longitud etapas-1

> turn(x) inicialmente arbitraria, escrito y leído por exactamente aquellos grupos i para los cuales x es un prefijo de la representación binaria de i.

etapas = (Si truncar(log(n)) = log(n) entonces = truncar(log(n)) sino = truncar(log(n)) + 1 fin si)
pganador = -1
gganador = -1

Figura 5.5: Variables Compartidas

en la sección crítica. *Etapas* contiene el número de niveles de competencia. Las variables *oponente_compa*$(0..n-1, i)$ son escritas por el proceso P_i y leídas por el proceso P_j.

En la figura 5.6, se muestra el comportamiento del protocolo de entrada y salida en formato tradicional. Se considera que los accesos a las variables compartidas son atómicos. Un buen algoritmo de exclusión mutua garantiza las condiciones de buena formación, exclusión mutua y progreso.

Teorema 5.1 *El algoritmo de un actor garantiza las propiedades de exclusión mutua, progreso y demora limitada.*

El modelo está basado en [40] que garantiza las condiciones de un buen algoritmo, y se le incorporó el avance de procesos oponentes que son compañeros para acceder conjuntamente a sección crítica. En el caso que todos los oponentes sean competidores el algoritmo garantiza todas las condiciones. ¿Qué sucede si un proceso P_i en el nivel k obtiene que un oponente P_j es compañero? El estado del proceso P_i sería el siguiente:

- $flag(i) = k$
- $oponente\_compa(j, i) = 1$
- $grupo(i) = g$

El proceso P_j podría estar en las siguientes situaciones:

Nivel k	Nivel Superior	En Sección Crítica
$flag(j) = k$	$flag(j) > k$ y $<$ etapas + 1	$flag(j) =$ etapas + 1
$grupo(j) = g$	$grupo(j) = g$	$grupo(j) = g$

Si está en el mismo nivel entonces avanza al próximo nivel de competición. Si está en un nivel superior o en la sección crítica, cuando se encuentre en la sección de salida, debe esperar hasta que el proceso P_i tome una decisión, acceda a la sección crítica o haya perdido el permiso de acceder directamente y tenga que continuar compitiendo.

Para garantizar la equidad, una vez que el primer proceso que ingresó a la sección crítica en un grupo G_l finaliza su sección crítica, no se permite que ingresen nuevos procesos vinculados al grupo G_l.

Medidas

Para estimar la complejidad del algoritmo se consideran la cantidad de referencias remotas y el tiempo de espera para ingresar en la sección crítica. En este algoritmo, cada proceso P_i compite en diferentes niveles y la cantidad máxima de niveles es del $O(log(n))$. En el caso que los n procesos quieran acceder a la sección crítica y no compartan trabajo, un proceso debe esperar como máximo $(n-1)$ entradas diferentes en la sección crítica.

Para poder estimar la cantidad de referencias a memoria remota, se considera que cada proceso P_i accede a las variables *flag(i)*, *grupo(i)* y *oponente_compa(i, 1...n)* en forma local

Proceso$_i$

... *Sección Resto*

Entrada$_i$
g = seleccionar grupo
grupo(i) = g
k = 1
mientras (k $\leq$ etapas) hacer {
flag(i) = k
turn(comp(i,k)) = role(i,k)
si (role(i,k) $\neq$ 0) ó (i $\leq$ n-k) ó (i $< 2^{etapas}$ / 2) entonces
waitfor [Hay_Compa(i,k)] ó [$\forall j \in$ Oponentes(i,k) : flag(j) $<$ k] ó [turn(comp(i,k) $\neq$ role(i,k)]
Si Hay_Compa(i,k) entonces
oponente_compa(j,i) = 1
waitfor [ganar(i)] ó [$\nexists$ Oponente(i,k) : (flag(j) $\geq$ k) $\wedge$ (grupo(j) == grupo(i))]
Si ganar(i) entonces k = etapas, flag(i) = etapas + 1
oponente_compa(j,i) = 0
k = k + 1}
flag(i) = etapas + 1
Si (pganador == -1) entonces
pganador = i
gganador = grupo(i)

... *Sección Crítica*

Salida$_i$
Si (pganador == i) entonces
pganador = -1
gganador = -1
waitfor ($\forall$ j, 0 $\leq$ j $\leq$ (n-1) oponente_compa(i,j) == 0)
grupo(i) = -1
flag(i) = 0

Hay_Compa(i,k) $\equiv$ *Si*($\exists j \in$ Oponentes(i,k) : flag(j) $\geq$ k $\wedge$ grupo(j) == grupo(i)) Entonces Verdadero Sino Falso

ganar(i) $\equiv$ Si (gganador == grupo(i) $\wedge$ pganador $\neq$ -1) Entonces Verdadero Sino Falso

Figura 5.6: Algoritmo de un actor

Casos	Cantidad de Accesos
P_i quiere acceder cuando hay un compañero y es el primero	8 accesos para ingresar 1 acceso para salir
P_i es el primer proceso y no tiene oponentes	$3 + log(n) + (n-1)$ accesos para ingresar 3 accesos para salir

Cuadro 5.1: Accesos Requeridos

y el resto de los accesos en forma remota (NUMA). En cuadro 5.1 se muestra la cantidad de accesos requeridos a memoria remota para 2 casos especiales; el primer caso se puede asociar con el concepto de algoritmos de exclusión mutua rápidos.

En las figuras 5.7 y 5.8, se muestran algunos gráficos relacionando la cantidad de procesos, la cantidad de accesos y la proporción que hay entre ellos, para 32 y 64 procesos respectivamente.

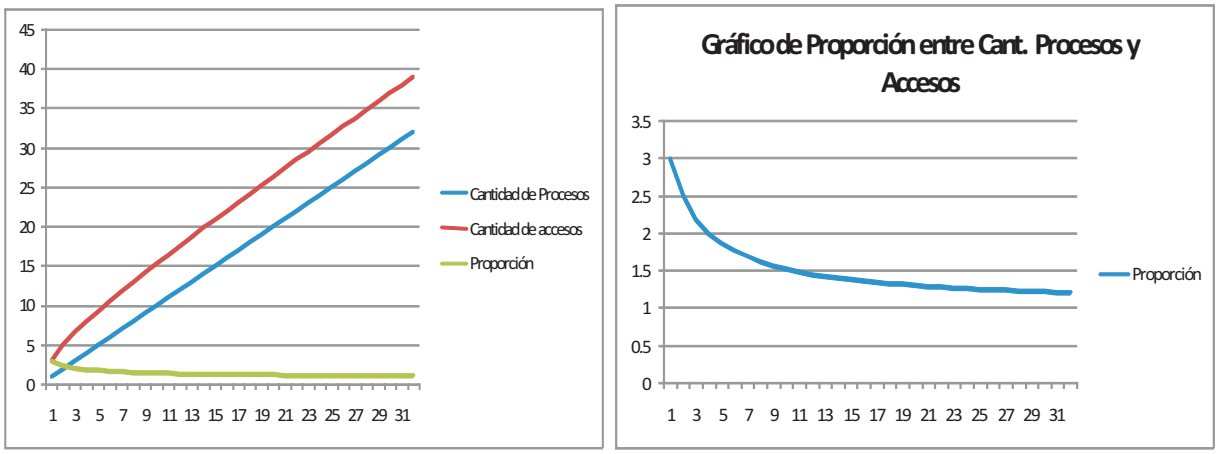


Figura 5.7: Con 32 procesos

En otros casos, no se puede estimar la cantidad de accesos, ya que se tienen esperas ocupadas sobre variables compartidas. Para poder estimar el peor caso, se debería adaptar el algoritmo para que todas las esperas ocupadas sean locales. La adaptación introduce mayor complejidad en el algoritmo e incluye nuevas variables.

5.3. Modelo Basado en dos Actores

La propuesta de este modelo está compuesta por dos tipos de actores: los *procesos* y los *grupos*, que se integran para competir por la utilización de un recurso. El *actor proceso* selecciona un grupo de trabajo, y el *actor grupo* compite para acceder a la sección crítica.

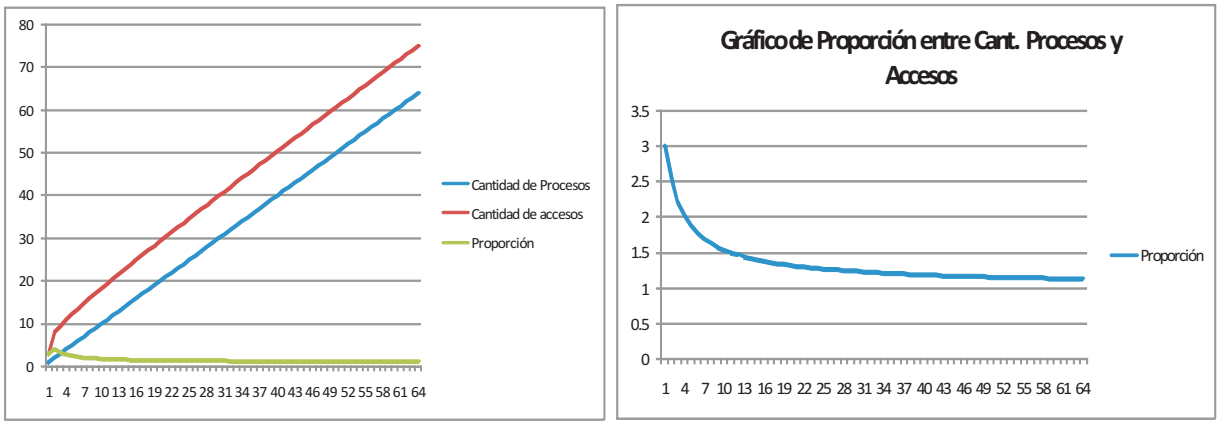


Figura 5.8: Con 64 procesos

Figura 5.9: Comunicación entre los dos Actores

En la figura 5.9 se muestra la relación entre los dos componentes y en la figura 5.10 un ejemplo de competición y concurrencia entre los diferentes actores.

Cuando un actor no está involucrado de ninguna manera con el recurso, se dice que está en la sección *resto*. Para obtener la admisión a la sección crítica, el actor ejecuta un protocolo de entrada (*trying*), después que utiliza el recurso, se ejecuta un protocolo de salida (*exit*). Este procedimiento puede repetirse, de modo que cada actor sigue un ciclo, desplazándose desde la *sección resto (R)*, a la *sección de entrada (T)*, luego a la *sección crítica (C)* y por último a la *sección de salida (E)*, y luego vuelve a comenzar el ciclo en la *sección resto*.

Como se muestra en la figura 5.11, el primer paso que realiza el *actor proceso*, en la sección de entrada, es seleccionar el grupo en el cual desea participar del conjunto de m grupos. El segundo paso es esperar hasta que el grupo seleccionado entre en la sección crítica para que pueda acceder a la misma. Cuando finaliza su actividad, sale de la sección crítica y se desvincula del grupo (sección de salida). En la figura 5.12 se muestran los estados del mismo.

El *actor grupo* está inicialmente inactivo, en la sección resto. Esto representa que ningún proceso lo ha seleccionado para participar en el mismo. El primer proceso que lo selecciona

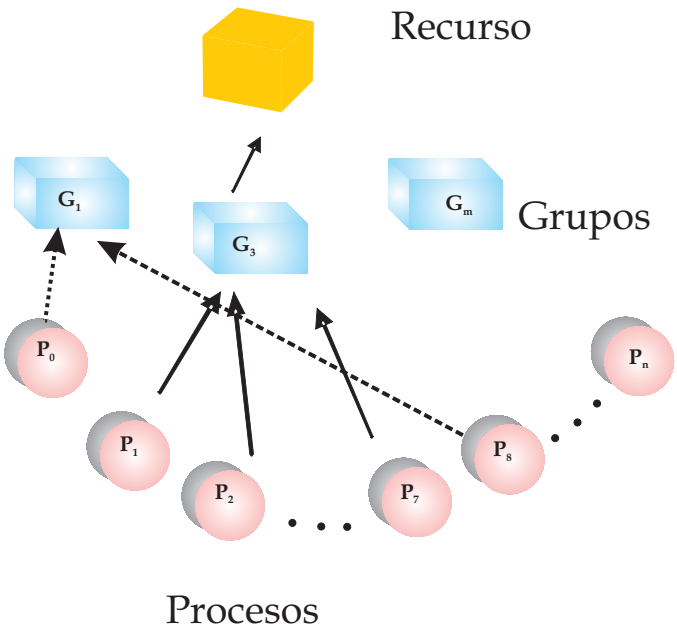


Figura 5.10: Ejemplo de Competición y Concurrencia - Dos Actores

Proceso$_i$

1. ..... {Sección Resto}
2. El proceso selecciona el grupo de trabajo {Grupo$_k$}
3. Espera hasta que entra a la sección crítica
   {Sección de Entrada}
4. ..... { Sección Crítica}
5. Sale de la sección crítica y se desvincula del grupo.
6. ..... {Sección Resto}

Figura 5.11: *Esquema del Actor Proceso*

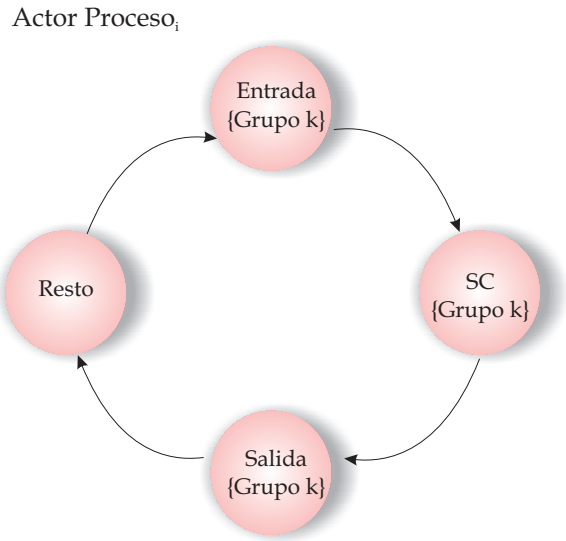


Figura 5.12: Estados del Actor Proceso

para participar hace que comience la competencia por entrar en la sección crítica, se lo identifica como el primer proceso que pertenece al grupo, y pasa a la sección de entrada. Todos los procesos que lo seleccionen mientras se encuentra en competición por entrar a la sección crítica se agregan a los procesos ya existentes. En el caso que el grupo esté en la sección crítica, si el proceso que activó al grupo está trabajando en la misma, entonces el proceso se incorpora; de lo contrario, se pone en cola de espera hasta que termine la actual vuelta (todos los procesos que están trabajando finalicen su tarea y el grupo salga de la sección crítica). En la próxima vuelta, cuando se reinicie el ciclo, esto es, compita nuevamente el grupo por el ingreso en la sección crítica, el proceso pasa a la lista de espera actual. En la figura 5.13 se observa el comportamiento del actor grupo mientras se encuentra activo y en la figura 5.14 se muestran los estados del mismo.

Los *actores grupos* compiten por alcanzar el permiso para acceder al recurso (acceso a la sección crítica), y sólo un único grupo tiene derecho de utilizar el recurso en un determinado instante de tiempo.

5.3.1. Algoritmo EMG Basado en Tournament

Esta implementación del modelo de dos Actores fue presentada en [41]. El esquema base para la competencia de los grupos, está basado en el algoritmo de Tournament, con la extensión a m elementos, donde m no es necesariamente una potencia de 2 (en la sección 3.3.4).

Las variables compartidas utilizadas se muestran en la figura 5.15. La variable *flag* indica si el grupo está compitiendo, esto es cuando flag(k) $\neq 0$, y además en qué nivel se encuentra.

Grupo$_k$

1. ..... {Sección Resto (Inactivo) }
2. Un proceso lo selecciona para trabajar en él.
   {Sección de Entrada (Compitiendo)}
3. Si es el primer proceso en el grupo entonces
   Comienza a competir en el ingreso a la S.C.
4. sino
   Si no está en la S.C. entonces
   Agrega el proceso a la lista de procesos
   sino
   Si está el primero del Grupo entonces
   entra en la S.C. el proceso
   sino
   El proceso lo coloca en la lista de espera hasta que termine la sección crítica actual y compita por ingresar nuevamente
5. Si no hay ningún proceso en la Sección Crítica libera el recurso.
6. ..... {Sección Resto (Inactivo)}

Figura 5.13: *Esquema del Actor Grupo*

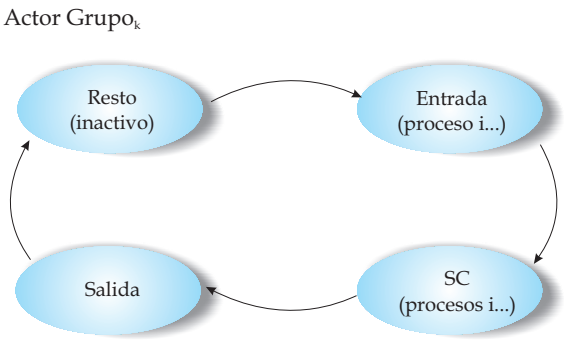


Figura 5.14: Estados del Actor Grupo

∀ flag(i): 0 ≤ i ≤ (m-1), flag(i) = 0 inicialmente
para cada cadena binaria x de a lo sumo longitud etapas-1
turn(x) inicialmente arbitraria, escrito y leído por exactamente aquellos grupos i
para los cuales x es un prefijo de la representación binaria de i.
etapas = (Si truncar(log(m)) = log(m) entonces etapas = truncar(log(m)) sino = truncar(log(m)) + 1)

Figura 5.15: Variables Compartidas

Grupo$_k$

Entrada$_k$
para l= 1 hasta etapas hacer
flag(k) = l {representa a los diferentes niveles}
turn(comp(k,l)) = role(k,l)
Si role(k,l) ≠ 0 ó (k ≤ n-l) ó (i < $2^{etapas}/2$) entonces
Waitfor [∀ j ∈ oponentes(k,l) : flag(j) < l] o [turn(comp(k,l)) ≠ role(k,l)]

S.C. {en la sección crítica}

Salida$_k$
flag(k) = 0

Figura 5.16: Componente Grupo

Cada grupo está ocupado en una serie de competencias de $O(\log m)$ para obtener el recurso. Puede considerarse que la competición está dispuesta en un árbol de competencia binario. Las hojas corresponden a los m grupos. En la sección 3.3.4 se definieron las funciones que se utilizan en el algoritmo. La característica que presenta es que en algunos niveles un actor grupo puede que no tenga oponentes en el mismo. En tal situación, el actor grupo avanza directamente al próximo nivel. Esto puede ocurrir en el caso que m no sea una potencia de 2 y, por ende, el árbol de competición no estaría completo. En la figura 5.16 se muestra el comportamiento del actor grupo.

El esquema base para cada uno de los actores *procesos* se asemeja al comportamiento que tiene un proceso que quiere acceder a la utilización de recurso o sección de código. En la figura 5.17 se muestran el conjunto de pasos que realiza.

En el algoritmo se utiliza la variable *lista*, donde el primer índice especifica el grupo y el segundo índice el proceso. En la sección de entrada, se selecciona el grupo en el cual va a participar. Luego verifica si el grupo está inactivo. Si la respuesta es afirmativa, es el primero que ingresa en el mismo, inicializa el nivel de la variable lista(k, i) en 2, de lo contrario lo hace en 1. Cuando se cumplen las condiciones indicadas en (1) significa que el grupo k está en la sección crítica y que el proceso i pueda acceder a ella.

La idea es que el algoritmo cumpla las siguientes condiciones:

- Un único grupo está activo utilizando el recurso compartido (exclusión mutua).
- Si el recurso está disponible y un grupo quiere utilizarlo (está en espera) puede acceder

```
Proceso_i
Entrada_i
    Seleccionar el grupo (g_k)
    Si inactivo(g_k) entonces
        lista[k,i] = <2, espera>
    sino
        lista[k,i] = <1, espera>
    fin Si
    Waitfor (flag(k) = etapas) y (∃ j:1..n, lista[k,j]=<2, en_cs>) (1)
    lista[k,i] = <..,en_cs>
Sección Crítica
.....
Salida_i
    lista[k,i] = <0, resto>
```

Figura 5.17: Componente Proceso

al mismo sin tener más demora.

- Si un grupo está activo y el primer proceso también, todo proceso que quiera trabajar en el mismo que lo pueda hacer. De esa manera, se logra un mayor nivel de concurrencia.

Como se muestra en la figura 5.18, el grupo g se encuentra en la sección crítica y el proceso i es el primer proceso del grupo. Luego le sigue el proceso k que se agrega al grupo, entran a la sección crítica en forma concurrente y, cuando el proceso j selecciona el grupo g, como éste ya se encuentra en la sección crítica y el primer proceso (proceso i) sigue trabajando en la misma, entonces puede acceder también sin tener que esperar y trabajar concurrentemente.

Implementación del Algoritmo

Este algoritmo utiliza el paradigma de memoria compartida distribuida sobre las variables de control utilizadas con múltiples accesos de lectura y escritura a las mismas. Las variables compartidas utilizadas en el mismo se encuentran definidas en la figura 5.19.

La variable *flag(i)* es escrita por el $grupo_i$, es leída por el resto de los grupos y los procesos que se asocian al mismo, y especifica el nivel de competencia. La variable *lista* contiene los procesos que están asociados en un instante de tiempo a los grupos, *lista(i,j)* es escrita por el $proceso_j$ que se asocia al $grupo_i$ y leída por el $grupo_i$ y todos los procesos que se asocian al mismo grupo. *Etapas* contiene el número de niveles de competencia.

En el cuadro 5.2, se observa que cada grupo está esperando que algún proceso lo seleccione, inicializando la variable *lista(i,j).estado* a "espera". Cuando ocurre este evento, el grupo se activa y comienza su competencia por ingresar en la sección crítica. Se indica a los procesos asociados al mismo que se alcanzó el objetivo inicializando *flag(i)* a $etapas + 1$. Se

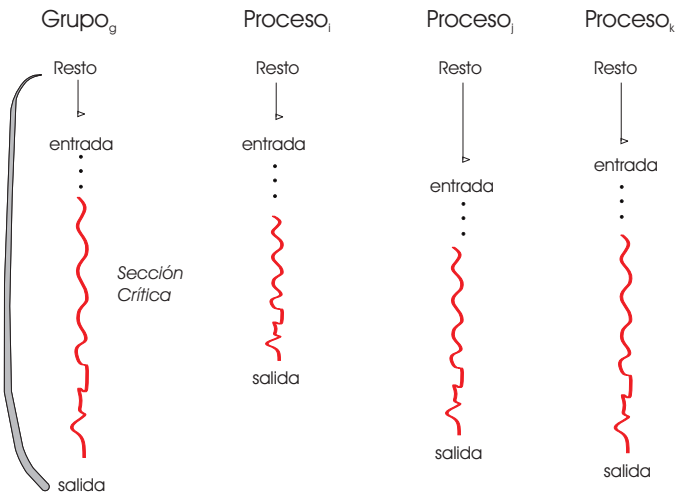


Figura 5.18: Concurrencia en el grupo g

$\forall$ flag(i): 0 $\leq$ i $\leq$ (m-1), flag(i) = 0 inicialmente
para cada cadena binaria x de a lo sumo longitud etapas-1

turn(x) inicialmente arbitraria, escrito y leído por exactamente aquellos grupos i para los cuales x es un prefijo de la representación binaria de i.

$\forall$ lista(i,j): 0 $\leq$ i $\leq$ (m-1), 0 $\leq$ j $\leq$ (n-1),
lista(i,j) = <0, resto> inicialmente
etapas = (Si truncar(log(m)) = log(m) entonces = truncar(log(m)) sino = truncar(log(m)) + 1)

Figura 5.19: Variables Compartidas

Grupo$_i$
Entrada$_i$
waitfor [∃ j: 1..n, lista[i,j] = < .., espera>]
Bucar_lider(lista,i)
para k = 1 hasta etapas hacer

flag(i) = k {representa los diferentes niveles}
turn(comp(i,k)) = role(i,k)
Si (role(i,k)≠0) ó (i≤m-k) ó (i < 2^{etapas}/2) entonces

Waitfor [∀ j ∈ oponentes(i,k) : flag(j) < k] ó [turn(comp(i,k))≠role(i,k)]

flag(i)= etapas + 1 {para que el proceso sepa que está en la S.C.}
... *Sección Crítica*

Waitfor [∀ j: 1..n, lista(i,j)≠<..,en_cs> ∧ lista(i,j)≠<2,..>]
flag(i) = 0

Cuadro 5.2: Algoritmo Componente Actor Grupo$_i$

permanece en la sección crítica mientras haya procesos activos en el grupo. Cuando todos los procesos salen de la sección crítica, el grupo libera el recurso, inicializando el *flag(i)* a 0 y comienza otra vez el ciclo, esperando por un nuevo proceso que lo seleccione.

Teorema 5.2 *El algoritmo EMG garantiza las propiedades de exclusión mutua, progreso y espera limitada.*

Las condiciones de *buena formación, exclusión mutua y progreso* las garantiza porque está basado en el algoritmo presentado en la sección 3.3.4; y además satisface los requisitos de un buen algoritmo de exclusión mutua, esto es, *Libre de Interbloqueo*, *Libre de inanición* e *Imparcialidad*.

Los procesos trabajan en forma individual y cuando desean trabajar en equipo seleccionan el grupo en el cual quieren participar. En el cuadro 5.3, se muestra el ciclo que realizan cada uno de los procesos, verifican si el grupo está inactivo al momento de la selección y, si es así, se considera que es el primer proceso del grupo.

Para garantizar que un grupo no permanezca indefinidamente en la sección crítica y se alcance un alto grado de concurrencia entre los procesos que trabajan cooperativamente, el proceso de verificar el nivel de competencia del grupo, también controla si el primer proceso (el líder) del grupo está actualmente trabajando en la sección crítica. Si el proceso líder no está activo entonces espera la otra vuelta.

¿Qué sucedería si los procesos p_i y p_j seleccionan el mismo grupo g_k y este se encuentra inactivo?

Para p_i

Proceso$_i$
... *Sección Resto*
Entrada$_i$
Selección del grupo en g
Si inactivo(g) entonces

lista(g,i) = <2, espera> {Es el primer proceso en el grupo, habilita mientras está en la sección crítica que otro procesos puedan participar concurrentemente en la misma}

sino

lista(g,i) = <1, espera> {Por lo menos hay otro proceso que estaba en el grupo}

fin si
Waitfor ((flag(g) = etapas + 1) $\wedge$ (lista(g,i)=<2, espera>)) $\vee$ ($\exists$ j: 1..n, lista(g,j)=<2, en_cs>)
lista(g,i)=<.., en_cs>
... *Sección Crítica*
Salida$_i$
lista(g,i) = <0, resto>

inactivo(g) $\equiv$ (flag(g) = 0) {Indica que el grupo está en la sección resto}

Cuadro 5.3: Algoritmo Componente Actor Proceso$_i$

flag(k) = 0 $\Longrightarrow$ el grupo está inactivo
lista(k, i) = <2, espera>

Para p_j
flag(k) = 0 $\Longrightarrow$ el grupo está inactivo
lista(k, j) = <2, espera>

En este caso, ocurre que dos procesos se consideran el primero del grupo. En el protocolo que controla el acceso del grupo al recurso, lo que se tiene en cuenta es que un proceso lo seleccione. El paso de *Buscar_líder* es necesario, ya que puede haber procesos en espera por participar del grupo, que surgen mientras el mismo se encuentra en la sección crítica. Estos procesos no están habilitados para ingresar a la sección crítica. El hecho de que haya 2 procesos líderes del grupo no afecta el protocolo de entrada del mismo y, por lo tanto, se mantienen las condiciones de exclusión mutua y progreso. Lo que ocurre es que mientras alguno de estos 2 procesos se encuentre activo en el grupo, todos los procesos que quieren participar en el mismo pueden ingresar. Como se consideró que todos los procesos trabajan un tiempo finito, entonces el grupo no queda indefinidamente en la sección crítica, permitiendo que evolucionen otros grupos, evitando así la inanición.

Si un proceso selecciona un grupo que tiene el acceso a la sección crítica pueden suceder los siguientes casos:

1. Que el primer proceso del grupo se encuentre activo: entonces habilita al nuevo proceso a que trabaje en el grupo cooperativamente, sin tener que esperar.

2. Que el primer proceso del grupo no se encuentra activo: entonces el proceso se agrega a la lista de procesos en espera. Debe esperar que termine el grupo la utilización de la sección crítica y vuelva a competir por el recurso. En el caso que todos los grupos estén compitiendo por acceder a la sección, deberá esperar en el $O(m)$ ciclos para acceder nuevamente a la misma, esto es, esperar que entren los grupos restantes.

Mediciones

Los algoritmos de exclusión mutua deben garantizar las condiciones para un buen algoritmo, y en este caso particular con la extensión para grupos de procesos es conveniente también que maximice la concurrencia de los mismos. Estos requerimientos son condiciones necesarias para seleccionar un algoritmo, pero no son las únicas que se tienen en cuenta al momento de la implementación. Es importante poder definir funciones que limiten la cantidad de pasos que realizan y en este caso la cantidad de accesos a la memoria compartida, ya que estos consumen recursos y decrementan la performance del mismo. En el capítulo 3 se presentaron diferentes tipos de algoritmos de exclusión mutua que buscan: minimizar la cantidad de accesos en el caso que no exista contención o que se consideren sólo los actores que realmente están en competición.

Analizando el algoritmo presentado en la sección anterior se llegan a las siguientes consideraciones:

- Los pasos requeridos por cada actor para ingresar en la sección crítica está limitado por la cantidad de niveles. La espera limitada está garantizada porque cumple las condiciones de un buen algoritmo y como máximo deberá esperar un proceso para ingresar m ciclos (vueltas) esto es, esta en $O(m)$.

- Algunos de los pasos corresponden a una espera ocupada sobre variables compartidas. No es posible obtener una función límite de $O(n)$ sobre la cantidad de accesos a memoria compartida (remota).

- Para acceder el actor siempre compite contra todos los otros actores aunque éstos no se encuentren activos.

- El algoritmo se puede implementar para que chequee primero a los oponentes y luego a la variable compartida *turn*, o modificar el orden de control obteniendo en algunos casos menor cantidad de accesos a la memoria compartida.

Se considera que un acceso a memoria compartida es equivalente a un mensaje.

5.3.2. Algoritmo Adaptivo de Grupos

El algoritmo presentado en la sección 5.3.1 tiene como desventaja que controla todos los oponentes que tiene un *actor grupo* en cada uno de los niveles. Una mejora es que chequee sólo los actores que están compitiendo en la sección de entrada y esta propuesta fue presentada en [43].

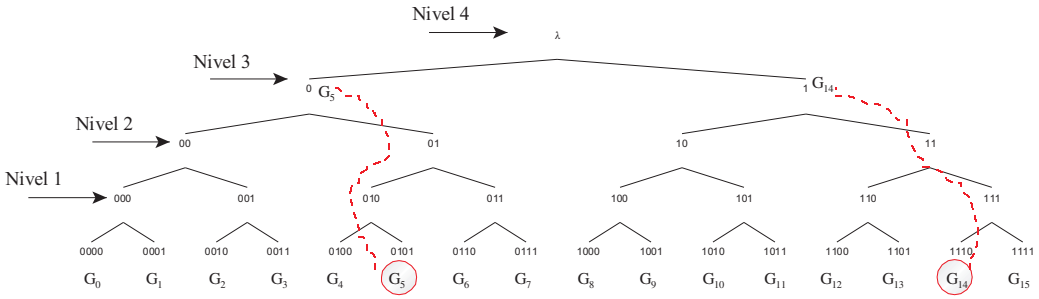


Figura 5.20: Ejemplo de Competición

Como la construcción del algoritmo se basa en un árbol binario, en donde los actores grupo se encuentran en las hojas y los ganadores de cada uno de los niveles en la raíz de cada subárbol, se puede considerar que en cada nivel de competencia cada uno de los actores tiene un sólo oponente activo. En la figura 5.20 se muestra un ejemplo teniendo 16 actores donde se observa que en el nivel 4 están compitiendo los actores grupo g_5 y g_{14}. El grupo g_5 está en el nivel 4 de compentencia y en el modelo de la sección 5.3.1 se chequearían todos los oponentes. Éstos serían $\{g_8, g_9, g_{10}, g_{11}, g_{12}, g_{13}, g_{14}, g_{15}\}$, que en realidad todos no pueden ser oponentes reales en ese nivel, sino solamente el ganador del nivel 3 que en este caso es g_{14}.

Para resolver el problema de adaptabilidad se debe conocer el ganador de cada uno de los niveles. Para esto es necesario contar con una estructura que mantenga esta información. Se incorpora una nueva variable compartida denominada *ganad* y en la figura 5.21 se muestran las modificaciones. Esta variable es escrita por aquellos grupos i para los cuales x es un prefijo de la representación binaria de i y leída por todos los grupos, ya que se utiliza para conocer los ganadores de cada uno de los niveles.

La sección de entrada ($Entrada_i$) se debe modificar para que se controle el oponente real, esto es, el ganador del nivel anterior y además se debe considerar que el nivel 1 es un caso especial, ya que tiene siempre un único oponente y es el mismo en todos los casos. Las modificaciones realizadas se muestran en el cuadro 5.4. Aquí se realiza la llamada a

∀ flag(i): 0 ≤ i ≤ (m-1), flag(i) = 0 inicialmente
para cada cadena binaria x de a lo sumo longitud etapas-1

turn(x) inicialmente arbitraria, escrito y leído por exactamente aquellos grupos i para los cuales x es un prefijo de la representación binaria de i.

ganad(x), ganad(x) = -1 inicialmente

....

Figura 5.21: Variables Compartidas

una función denominada *ChequearGanad* la cual controlará si está activo alguno de los oponentes correspondientes a ese nivel y si ya ha ganado en el nivel. La función $\overline{role}(i,k)$ devuelve como resultado el negativo con respecto a la función role. Esto es, si role devuelve 1 entonces $\overline{role}$ devuelve 0 y viceversa. El resultado de esta función es concatenado con el resultado de aplicar comp(k,i) que permite obtener el ganador oponente correspondiente.

La sección de salida también se debe adaptar para mantener los nuevos requerimientos. Para cada uno de los niveles se libera al ganador del mismo para que los competidores puedan avanzar. Este algoritmo cumple con las condiciones de un buen algoritmo de exclusión mutua para grupos de procesos y además es adaptivo, esto es, sólo intervienen en la sección de entrada los grupos que están en competición. No es posible obtener una función límite de $O(n)$ sobre la cantidad de accesos a memoria compartida (remota), ya que está diseñado a través de una espera ocupada sobre las variables compartidas *ganad*, *flag* y *turn*.

Para eliminar las esperas ocupadas sobre variables compartidas (esto significa acceder a las mismas en forma remota) se puede testear inicialmente el estado de dichas variables y luego esperar hasta que las mismas hayan cambiado su estado avisándole al actor. Considerando que el mayor costo es de comunicación, se puede incorporar una espera ocupada en forma local (*local spin*).

Se requiere incorporar variables para que cada uno de los actores grupo accedan localmente mientras esperan que cambie el estado respectivo permitiendo que avance en el nivel de competencia. Para eliminar la espera ocupada sobre las variables compartidadas *ganad* y *flag*, se agrega una variable denominada *oponente_nivel*.

∀ oponente_nivel(i): 0 ≤ i ≤ (m-1), inicialmente oponente_nivel (i) = 0, oponente_nivel (i) ∈ {0, 1, 2}, leída por i y escrita por todos.

Para cada uno de los *actores grupo* el valor de *oponente_nivel* está inicializado en 0. Esto representa que es la primera vez que se va a chequear la variable compartida *ganad* o *flag*. Si en el primer control encuentra que debe esperar asigna, el valor 1 a *oponente_nivel* y en los siguientes controles espera hasta que cambie este valor. El mismo es modificado por el *actor grupo* oponente ganador cuando se encuentra en la sección de salida.

Para eliminar la espera ocupada sobre la variable *turn* se agrega una variable denominada *turno*.

Grupo$_i$
Entrada$_i$
waitfor [∃ j: 1..n, lista[k,j] = < .., espera>]
Bucar_lider(lista,i)
para k = 1 hasta etapas hacer

flag(i) = k {representa los diferentes niveles}
turn(comp(i,k)) = role(i,k)
Si (role(i,k)≠0) ó (i≤m-k) ó (i < 2^{etapas}/2)) entonces

Si (k = 1) entonces
Waitfor [∀ j ∈ oponentes(i,k) : flag(j) < k]
ó [turn(comp(i,k))≠role(i,k)]
sino
Waitfor [ChequearGanad(i,k)] ó [turn(comp(i,k))≠role(i,k)]
ganad(comp(i,k)) = i;

flag(i)= etapas + 1 {para que el proceso sepa que está en la S.C.}
... *Sección Crítica*
Salida$_i$
Waitfor [∀ j: 1..n, lista(i,j)≠<..,en_cs> ∧ lista(i,j)≠<2,..>]
flag(i) = 0
para k = 1 hasta etapas hacer

ganad(comp(i,k)) = -1

ChequearGanad(i,k) = Si (ganad(comp(i,k)) ≠ -1) ó (ganad(comp(i,k)$\overline{role(i,k)}$) ≠ -1)
Entonces Falso
Sino Verdadero

Cuadro 5.4: Algoritmo Adaptivo Componente Actor Grupo$_i$

∀ flag(i): 0 ≤ i ≤ (m-1), flag(i) = 0 inicialmente
para cada cadena binaria x de a lo sumo longitud etapas-1

turn(x) inicialmente arbitraria, escrito y leído por exactamente aquellos grupos i para los cuales x es un prefijo de la representación binaria de i.

ganad(x), ganad(x) = -1 inicialmente

∀ turno(i): 0 ≤ i ≤ (m-1), inicialmente vacío, turno(i) ∈ {0,1}, leído por i y escrita por todos
∀ oponente_nivel(i): 0 ≤ i ≤ (m-1), inicialmente oponente_nivel(i) = 0, oponente_nivel(i) ∈ {0,1,2}, leído por i y escrita por todos
∀ lista(i,j): 0 ≤ i ≤ (m-1), 0 ≤ j ≤ (n-1), lista(i,j) = <0, resto> inicialmente
∀ opcion(j): 0 ≤ j ≤ (n-1), opcion(j) = resto inicialmente
etapas = (Si truncar(log(m)) = log(m)
entonces
= truncar(log(m))
sino
= truncar(log(m)) + 1 fin si)

Figura 5.22: Variables Compartidas

∀ turno(i): 0 ≤ i ≤ (m-1), inicialmente vacío, turno (i) ∈ {0, 1}, leída por i y escrita por todos.

En cada iteración de la espera se controla por el estado de la variable *turno(i)* que se encuentra localmente y las modificaciones se deben realizar en la sección de entrada para que actualice el valor de la variable *turno(j)* correspondiente con el competidor del nivel.

En el algoritmo se incorporan las siguientes funciones:

- ActualizarTurno(grupo,k): esta función tiene como parámetros el grupo y el nivel correspondiente. Controla si existe un oponente activo. Si es así entonces actualiza el valor de la variable *turno* local con el valor de la variable *turn(comp(grupo,k))*.
- ChequearFlag(grupo,k): esta función tiene como parámetros el grupo y el nivel correspondiente. La primera vez que es invocada en el primer nivel, controla el estado de la variable *flag* del oponente. Si está activo entonces en las próximas iteraciones controlará el estado de la variable *oponente_nivel(grupo)* que se encuentra localmente. Si no es el primer nivel entonces la primera vez que es invocada controla el estado de la variable *ganad(comp(grupo,k))* para poder determinar si ya hay un grupo ganador en el nivel, o si hay un oponente activo en el mismo. Si está activo el oponente en las próximas iteraciones controlará el estado de la variable *oponente_nivel(grupo)*

En la figura 5.22 se muestran todas las variables compartidas utilizadas en el algoritmo, en el cuadro 5.5 se muestra el comportamiento del componente grupo en un algoritmo con formato tradicional, y en el cuadro 5.6 el comportamiento del componente proceso con el mismo formato.

Grupo$_i$
Entrada$_i$
waitfor [∃ j: 1..n, lista[k,j] = < .., espera>]
Bucar_lider(lista,i)
para k = 1 hasta etapas hacer
oponente_nivel(i) = 0
flag(i) = k {representa los diferentes niveles}
turn(comp(i,k)) = role(i,k)
turno(i) = turn(comp(i,k))
ActualizarTurno(i,k)
Si (role(i,k)≠0) ó (i≤m-k) ó (i < (2^{etapas}/2)) entonces
Waitfor [ChequearFlag(i,k)] ó [turno(i)≠role(i,k)]
ganad(comp(i,k)) = i
flag(i)= etapas + 1 {para que el proceso sepa que está en la S.C.}
para cada j = 0 hasta n-1 hacer
si lista(i,j) = <..,espera> entonces opcion(i) = en_cs

... *Sección Crítica*

Salida$_i$

Waitfor [∀ j: 1..n, lista(i,j)≠<..,en_cs> ∧ lista(i,j)≠<2,..>]
flag(i) = 0
para k = etapas hasta 1 hacer
ganad(comp(i,k)) = -1
Si ((j=OponenteActivo(i,k)) ≠ -1) entonces
oponente_nivel(j) = 2

ChequearFlag(i,k) = Si (oponente_nivel(i) = 0)
Entonces (Si (k=1)
Entonces (Si (para el grupo oponente flag(j) ≥ k)
Entonces oponente_nivel(i) = 1, Falso
Sino Verdadero)
Sino (Si (ganad(comp(i,k)) ≠ -1) ó (ganad(comp(i,k)$\overline{role(i,k)}$) ≠ -1)
Entonces Falso
Sino Verdadero))
Sino (Si (oponente_nivel(i) =1)
Entonces Falso
Sino Verdadero)

ActualizarTurno(i,k) = Si ((j= OponenteActivo(i,k)) ≠ -1) Entonces
Si (flag(j) = k) Entonces turno(j) = turn(comp(i,k))

OponenteActivo(i,k) = Si (k=1) Entonces j=Oponentes(i,k)
Sino (Si (ganad(comp(i,k)) ≠ -1) ó (ganad(comp(i,k)$\overline{role(i,k)}$) ≠ -1)
Entonces j = ganad(comp(i,k)$\overline{role(i,k)}$)
Sino j = -1)

Cuadro 5.5: Algoritmo Adaptivo Local del Componente Grupo$_i$

Proceso$_i$
Entrada$_i$
  Selección del grupo en g
  Si inactivo(g) entonces
    lista(g,i) = <2,espera>
  sino
    lista(g,i) = <1,espera>
  opcion(i) = entrar
  Waitfor (chequear_entrada(g))
  lista(g,i) = <..,en_cs>

  ... *Sección Crítica*

Salida$_i$

  lista(g,i) = <0,resto>
  opcion(i) = resto

inactivo(g) = (flag(g) = 0)

chequear_entrada(g) = Si (opcion(i)= entrar) entonces
  (si (flag(g) = etapas + 1) $\wedge$ ((lista(g,i) = <2,espera>) $\vee$
    ($\exists$ j:0..n-1, lista(g,j)= <2,en_cs>))
      entonces verdadero
      sino opcion(i) = espera, falso)
  sino si opcion(i) = espera
      entonces falso
      sino verdadero

Cuadro 5.6: Algoritmo Adaptivo Local del Componente Proceso$_i$

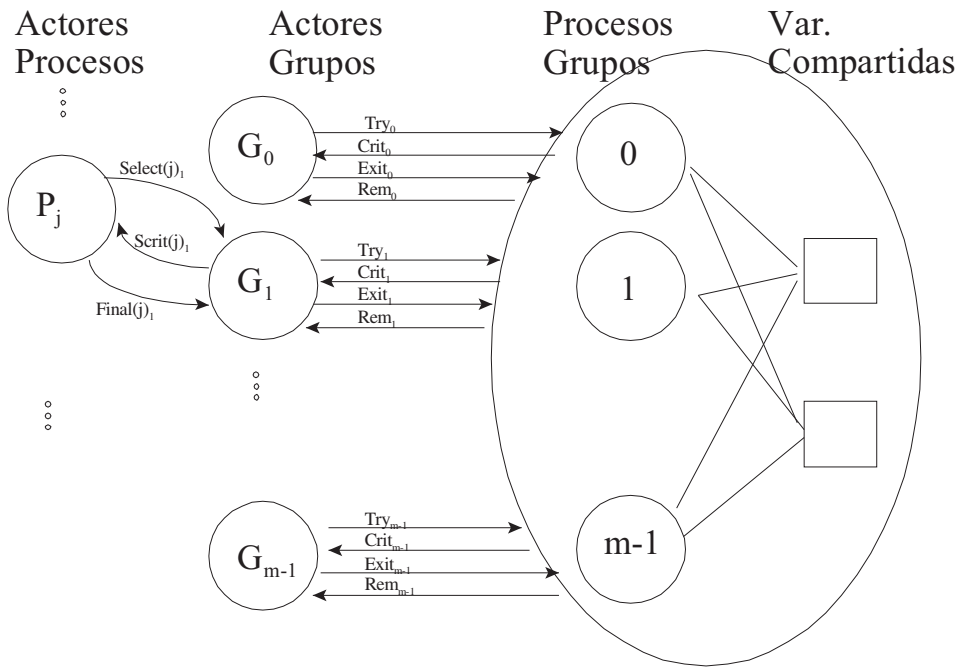


Figura 5.23: Componentes del Modelo

Para considerarlo buen algoritmo debe estar bien formado, garantizar las condiciones de exclusión mutua, progreso (libre de bloque) para ingresar en la sección crítica y maximice la concurrencia. Para alcanzar estas condiciones es conveniente reescribir el algoritmo en un modelo formal, y para esto se lo tranformará en un autómata de entrada/salida. En la figura 5.23 se muestra cómo interactúan los componentes del autómata de entrada/salida que se construye. El actor proceso j se asocia en un instante, a través de la acción externa *select(j)*$_i$, con un actor grupo i y éste luego inicia el protocolo a través de la acción externa *try*$_i$.

En la figura 5.24 se muestran las variables compartidas utilizadas, las acciones correspondientes para cada uno de los grupos y los estados de los mismos. En la figura 5.25 se muestran las transiciones que realiza el *procesogrupo*$_i$, donde i compite por el recurso.

En el modelo realizado, que se muestra en la figura 5.23, aparecen el *actor grupo* y el *actor proceso*, en la figura 5.26 se muestran las variables y acciones que utiliza el actor grupo, y en la figura 5.28 las variables y acciones que utiliza el actor proceso. Se define una variable denominada *opcion* que se utiliza para avisarle al proceso que puede ingresar a la sección crítica. Las transiciones de cada uno de los actores se muestran en las figuras 5.27 y 5.29.

¿El algoritmo presentado garantiza la exclusión para los actores grupo? Consideremos

Variables Compartidas
∀ flag(i): 0 ≤ i ≤ (m-1), flag(i) = 0 inicialmente
para cada cadena binaria x de a lo sumo longitud etapas-1
turn(x) ∈ {0,1}, inicialmente arbitrario
ganad(x) ∈ {-1, 0, 1, ..., m-1 }, ganad(x) = -1 inicialmente
∀ turno(i): 0 ≤ i ≤ (m-1), inicialmente vacío, turno(i) ∈ {0,1}
∀ oponente_nivel(i): 0 ≤ i ≤ (m-1), inicialmente oponente_nivel(i) = 0, oponente_nivel(i) ∈ {0,1,2}
∀ lista(i,j): 0 ≤ i ≤ (m-1), 0 ≤ j ≤ (n-1), lista(i,j) = <0, resto> inicialmente
etapas = (Si truncar(log(m)) = log(m) entonces = truncar(log(m))
sino = truncar(log(m)) + 1 fin si)

Acciones de i

Entrada	Internas
Try_i	$set\text{-}turn_i$
$Exit_i$	$actualizar\text{-}turno_i$
Salida	$chequear\text{-}flag_i$
$Crit_i$	$chequear\text{-}turno_i$
Rem_i	$chequear\text{-}oponente_i$
	$ganador_i$
	$set\text{-}flag_i$
	$reset_i$

Estados de i
estado ∈ {rem, set-turn, actualizar-turno, check-flag, set-flag, check-oponentes, check-turno, leave-try, ganador, crit, leave-exit }, inicialmente rem
nivel ∈ {1 ... etapas}, inicialmente 1

Figura 5.24: Variables, acciones y estados ProcesoGrupo$_i$

tener m = 16 cantidad de *actores grupo*. Supongamos que los grupos G_4 y G_{12} ingresan simultáneamente a la sección crítica. Se tendría un total de 4 niveles de competencia. Como cada uno de los grupos pertenece a una rama diferente del árbol construido, ya que la representación binaria para G_4 es 0100 y para el G_{12} es 1100. Estos grupos van a ser competidores en el último nivel, esto es en el nivel 4. Ambos actores grupo tendrían valor 4 en sus correspondientes *flag*, y considerando que primero actualiza la variable *turn* el grupo G_4, entonces se tendría la siguiente situación:

1. (G_4) turn(comp(4,4)) = role(4,4) = 0.
2. (G_4) turno(4) = turn(comp(4,4)) = 0.
3. (G_{12}) turn(comp(12,4))= role(12,4) = 1.
4. (G_4) ActualizarTurno(4,4), como está activo el oponente correspondiente al nivel, turno(12) = turn(comp(4,4)) = 1.
5. (G_{12}) turno(12) = turn(comp(12,4)) = 1.
6. (G_{12}) ActualizarTurno(12,4), como está activo el oponente correspondiente al nivel, turno(4) = turn(comp(12,4)) = 1.
7. (G_4) Waitfor [ChequearFlag(4,4)] ó [turno(4) ≠ role(4,4)].

Transiciones de i

Try_i
  efecto:
    estado := set-flag

$set - flag_i$
  precondición:
    estado = set-flag
  efecto:
    flag(i) = nivel
    estado = set-turn

$set - turn_i$
  precondición:
    estado = set-turn
  efecto:
    turn(comp(i,Nivel)) = role(i, nivel)
    turno(i) = turn(comp(i,nivel))
    estado = Actualizar-Turno

$actualizar - turno_i$
  precondición:
    estado = Actualizar-Turno
  efecto:
    Si (j= OponenteActivo(i,nivel)) Entonces
      Si flag(j) = nivel Entonces
        turno(j) = turn(comp(i,nivel))
    estado = check-oponente

$chequear - oponente_i$
  precondición:
    estado = check-oponente
  efecto:
    Si (role(i, nivel) $\neq$ 0) ó (i $\leq$ m-nivel) ó (i$< 2^{etapas}/2$)
    entonces
      estado = check-flag
    sino
      estado = ganador

$ganador_i$
  precondición:
    estado = ganador
  efecto:
    ganad(comp(i, nivel)) = i
    oponente_nivel(i) = 0
    si nivel = etapas entonces
      estado = leave-try
    sino
      nivel = nivel + 1
      estado = set-flag

$chequear - turno_i$
  precondición:
    estado = check-turno
  efecto:
    si turno(i) $\neq$ role(i,nivel)
    entonces
      estado = ganador
    sino
      estado = check-flag

$chequear - flag_i$
  precondición:
    estado = check-flag
  efecto:
    si oponentenivel(i) = 0 entonces
      si nivel = 1 entonces
        j = Oponentes(i, nivel)
        si flag(j) $\geq$ nivel entonces
          oponente_nivel(i) = 1
          estado = check-turno
        sino estado = ganador
      sino
        si (j = OponenteActivo(i,nivel)) $\neq$ -1 entonces
          oponente_nivel(i) = 1
          estado = check-turno
        sino estado = ganador
    sino
      si oponente_nivel(i) = 1
      entonces estdo = check-turno
      sino estado = ganador

$Crit_i$
  precondición:
    estado = leave-try
  efecto:
    flag(i) := etapas + 1
    estado := crit

$Exit_i$
  efecto:
    estado := reset

$Reset_i$
  precondición:
    estado = reset
  efecto:
    flag(i) = 0
    For k :=1 to etapas do
      ganad(comp(i,k)) = -1
      si ((j = OponenteActivo(i,k)) $\neq$ -1 entonces
        oponente_nivel(j) = 2
    oponente_nivel(i) = 0
    nivel = 1
    estado := leave-exit

Rem_i
  precondición:
    estado := leave-exit
  efecto:
    estado := rem

Figura 5.25: Algoritmo Adaptivo de Grupos

Variables
∀ flag(i): 0 ≤ i ≤ (m-1), flag(i) = 0 inicialmente
∀ lista(i,j): 0 ≤ i ≤ (m-1), 0 ≤ j ≤ (n-1), lista(i,j) = <0, resto> inicialmente
∀ opcion(j): 0 ≤ j ≤ (n-1), opcion(j) = resto inicialmente

Acciones de i

Entrada	Salida
$Select(j)_i$, $j \in (0..n\text{-}1)$	$Scrit_i$
$Crit_i$	Try_i
Rem_i	$Exit_i$
$Final(j)_i$, $j \in (0..n\text{-}1)$	
Internas	
$Chequear\_Estado_i$	
$Chequear\_Proceso_i$	
$Entrada_i$	

Estados de i
estadog ∈ {rem-g, entrada, chequear-proceso, chequear-estado, crit-g, try, enter-try, leave-try-g, leave-exit-g }, inicialmente rem-g
situación ∈ {inactivo, espera, en_cs, saliendo}, inicialmente inactivo

Figura 5.26: Variables, Acciones y Estados

8. (G_{12}) Waitfor [ChequearFlag(12,4)] ó [turno(12) ≠ role(12,4)].

Cuando el grupo G_4 se encuentre en el Waitfor (paso 7) el control sobre el *ChequearFlag* será falso ya que el oponente también está en el mismo nivel. Lo mismo le ocurrirá al grupo G_{12} en el Waitfor (paso 8). Para que ambos ingresen entonces tendrá que ser verdadera la otra condición, esto es, el valor de la variable *turno* deberá ser diferente al *role*. Pero esta variable está relacionada con la variable *turn* que en un instante de tiempo tiene el valor 0 ó 1, por lo que uno de los 2 grupos tendrá falso en la condición. En este caso, ingresaría el grupo G_4 a la sección crítica y el otro grupo deberá esperar que el grupo libere la sección crítica. Si el grupo G_{12} hubiera actualizado primero la variable *turn* entonces sería este grupo el que ingresaría a la sección crítica y el otro debería esperar. Por lo tanto, se contradice la suposición inicial en la cual los 2 grupos pueden ingresar al mismo tiempo en la sección crítica.

Teorema 5.3 *El algoritmo Adaptivo para grupos garantiza la exclusión mutua para los actores grupos.*

Consideremos la situación que se observa en la figura 5.30, donde se tienen 2 grupos G_i y G_j que están compitiendo en el mismo nivel k y que los 2 grupos pueden ganar al mismo tiempo en el nivel k. Ambos grupos tendrían en la variable *flag* correspondiente el valor k, que es el nivel de competencia. Considerando que G_j actualiza primero la variable *turn*entonces se tiene la siguiente situación:

1. (G_j) turn(comp(j,k)) = role(j,k) = 1.
2. (G_j) turno(j) = turn(comp(j,k)) = 1.

Transiciones de i

$Select(j)_i$
efecto:
Si situación = inactivo entonces
estadog = entrada

$SCrit(j)_i$
precondición:
estadog = leave-try-g
efecto:
estadog = crit-g

Try_i
precondicón:
estadog = enter-try
efecto:
estadog = try

$Chequear - Proceso_i$
precondición:
estadog = chequear-proceso
efecto:
si ∃ j ∈ (0, .., n-1) tq lista(i,j) = <.., espera> entonces
j = BuscarLider(i)
lista(i,j) = <2, espera>
estadog = entrada
sino
situación = inactivo
estadog = rem-g

Rem_i
efecto:
estadog = chequear-proceso

$Entrada_i$
precondición:
estadog = entrada
efecto:
situación = espera
estadog = enter-try

$Crit_i$
efecto:
Si situación = espera entonces
situación = en_cs
para j = 1 hasta (n-1) hacer
Si lista(i,j) = <.., espera> entonces
lista(i,j)= <.., en_cs>
opcion(j) = en_cs
estadog = leave-try-g

$Final(j)_i$
efecto:
Si ∀ l ∈ {0,..,n-1} ∄ lista(i,l) = <2, ..> entonces
situación = saliendo
estadog = chequear-estado

$Chequear - Estado_i$
precondición:
estadog = chequear-estado
efecto:
si ∄ j ∈ (0, .. n-1) tq lista(i,j) = <.., en_cs> entonces
estadog = leave-exit-g
sino
estadog = crit-g

$Exit_i$
precondición:
estadog = leave-exit-g
efecto:
situación = saliendo
estadog = rem-g

Figura 5.27: Actor Grupo i

Variables
∀ flag(i): 0 ≤ i ≤ (m-1), flag(i) = 0 inicialmente
∀ lista(i,j): 0 ≤ i ≤ (m-1), 0 ≤ j ≤ (n-1), lista(i,j) = <0, resto> inicialmente
∀ opcion(j): 0 ≤ j ≤ (n-1), opcion(j) = resto inicialmente

Acciones de j

Entrada	Internas
$Ingreso_j$	$Set\text{-}opcion_j$
$Salir_j$	$Chequear\text{-}entrada\text{-}p_j$
$Scrit(j)_i$	$Check\text{-}opcion_j$
Salida	
$Final(j)_i$	
$Select(j)_i$	
$Critp_j$	

Estados de j
estadop ∈ {set-opcion, chequear-entrada-p, check-opcion, try-crit, rem-p }, inicialmente rem-p
i ∈ {-1, 0, .. m-1}, inicialmente i = -1

Figura 5.28: Variables, Acciones y Estados - Actor Proceso

Transiciones de j

$Ingreso_j$
  efecto:
    selecciona el grupo i
    si inactivo(i) entonces
      lista(i,j) = <2,espera>
    sino
      lista(i,j) = <1,espera>
    estadop = set-opcion

$Set - opcion_j$
  precondición:
    estadop = set-opcion
  efecto:
    opcion(j) = entrar
    estadop = chequear-entrada-p

$Check - opcion_j$
  precondición:
    estadop = check-opcion
  efecto:
    si opcion(j) = en_cs entonces
      estadop = try-crit

$Chequear - entrada - p_j$
  precondición:
    estadop = chequear-entrada-p
  efecto:
    si (flag(i) = etapas + 1) ∧ ((lista(i,j) = <2,espera>)
      ∨ (∃ j:0, .., n-1, lista(i,j) = <2, en_cs>)) entonces
      estadop = try-crit
    sino
      estadop = check-opcion

$Critp_j$
  precondición:
    estadop = try-crit
  efecto:
    lista(i,j)= <.., en_cs>
    estadop = critp

$Salir_j$
  efecto:
    lista(i,j) = <0, resto>
    i = -1
    opcion(j) = resto
    estadop = remp

Figura 5.29: Actor Proceso j

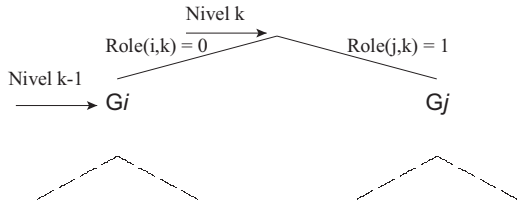


Figura 5.30: Ejemplo de Competición en el Nivel k

3. (G_i) turn(comp(i,k))= role(i,k) = 0.
4. (G_j) ActualizarTurno(j,k), como está activo el oponente correspondiente al nivel, turno(i) = turn(comp(j,k)) = 0.
5. (G_i) turno(i) = turn(comp(i,k)) = 0.
6. (G_i) ActualizarTurno(i,k), como está activo el oponente correspondiente al nivel, turno(j) = turn(comp(i,k)) = 0.
7. (G_j) Waitfor [ChequearFlag(j,k)] ó [turno(j) $\neq$ role(j,k)].
8. (G_i) Waitfor [ChequearFlag(i,k)] ó [turno(i) $\neq$ role(i,k)].

Cuando el grupo G_j se encuentra en el Waitfor (paso 7) el control sobre *ChequearFlag* será falso ya que el oponente también está en el mismo nivel. Lo mismo le ocurrirá al grupo G_i en el Waitfor (paso 8). Para que ambos sean los ganadores entonces tendrá que ser verdadera la otra condición, esto es, el valor de la variable *turno* deberá ser diferente del *role* en ambos casos. Pero esta variable está relacionada con la variable *turn* que en un instante de tiempo tiene el valor 0 ó 1. Cuando se actualiza *turn* se invoca a la función *ActualizarTurno* y se actualiza el valor del oponente si este está activo. Por lo tanto, un sólo grupo va a ser el ganador y en este caso sería G_j, lo cual contradice la suposición inicial. Por lo tanto, en cada uno de los niveles del subárbol hay un único ganador y en el último nivel el ganador accede a la sección crítica.

Teorema 5.4 *El algoritmo Adaptivo para grupos garantiza progreso para los actores grupos.*

Si en una ejecución a, por lo menos existe un grupo en la sección de entrada y no existe ningún grupo en la sección crítica, supongamos que no entra ningún grupo. Si el

grupo, denominado i, está en la sección de entrada entonces está esperando en el estado para chequear el *flag* (*check-flag*) o para chequear el *turno* (*check-turno*). Si no hay nadie en la sección crítica y es el único competidor entonces el grupo en el estado *chequear-flag* verificará que no hay ganador y podrá avanzar de nivel y así hasta el último nivel. Si hay 2 grupos compitiendo en el mismo nivel, por lo mostrado anteriormente uno será el ganador y, por lo tanto, podrá acceder a la sección crítica. Esto contradice la suposición inicial.

Si un algoritmo está bien formado y es libre de bloqueo (para todos los procesos) entonces garantiza la propiedad de progreso. Si mantiene la propiedad de vivacidad entonces no puede ocurrir inanición.

Teorema 5.5 *El algoritmo Adaptivo para grupos garantiza demora limitada para los actores.*

Supongamos que es posible que un proceso/grupo espere indefinidamente para acceder a la sección crítica. Cuando un grupo i ingresa en la sección de entrada, pasa por el estado *set-turn*, en el cual se inicializa la variable *turn* para ese nivel. Por otra parte, ActualizarTurno, controla si hay un grupo oponente activo, y actualiza el valor de *turno*, por lo que un grupo j que accede después a la sección de entrada, deberá esperar que el grupo i tenga su acceso a la sección crítica o al próximo nivel antes de poder avanzar. Por lo tanto, no tiene una espera indefinida.

La idea del algoritmo es poder obtener una función de $O(n)$ que limite la cantidad de accesos a memoria compartida, donde $n \geq m$ ó $n \gg m$. El modelo tiene *etapas* niveles donde $etapas \equiv \log(m)$ ó $etapas \equiv \mid \log(m) \mid +1$, dependiendo si m es una potencia de 2. ¿Cuántos accesos se requieren para acceder a la sección crítica?. Analizando el algoritmo se divide el problema en dos casos: cuando se encuentra en el primer nivel ($k = 1$) o cuando se encuentra en el resto de los niveles ($k \neq 1$), considerando al actor *grupo* i:

- Para $k = 1$
  - 1 acceso a variable *turn* (es la que mantiene el turno, para evitar la espera indefinida).
  - 3 accesos en la función ActualizarTurno, se accede a la variable *flag(j)* y a la variable *turno(j)* asignándole el valor de *turn*, en el caso que el oponente se encuentre activo en el nivel; sino sería un solo acceso.
  - 1 acceso en la función ChequearFlag, se accede a la varibale *flag(j)*.
  - 1 acceso a la variable *ganad*.

  La cantidad de accesos son 6 en el peor de los casos.
- Para $k \neq 1$
  - 1 acceso a variable *turn* (es la que mantiene el turno, para evitar la espera indefinida).

- 4 accesos en la función ActualizarTurno, se accede a la variable *ganad* para buscar el oponente, a la varible *flag(j)* y a la variable *turno(j)* asignándole el valor de *turn*, en el caso en que el oponente se encuentre activo en el nivel. Si no hay oponente activo entonces se tendría un sólo acceso que es a la variable *ganad*. Si hay oponente pero se encuentra en un nivel superior entonces se tendrían sólo los 2 primeros accesos a las variables *ganad* y *flag*.
- 2 accesos en la función ChequearGanad, se accede a la varibale *ganad* para saber si hay un ganador en el nivel o si el oponente se encuentra compitiendo en el nivel.
- 1 acceso a la variable *ganad*.

La cantidad de accesos son 8 en el peor de los casos para cada una de los niveles. Considerando que se tienen $(etapas - 1)$ niveles entonces la cantidad de accesos que se requiere para acceder a la sección son $8 \times (etapas - 1)$.

Considerando los 2 casos y que etapas está definido en función de m, entonces se puede determinar que se requiere

$$6 + 8 \times (\log(m) - 1) \tag{5.3}$$

en el peor de los casos para acceder a la sección crítica, considerando como un único paso las esperas ocupadas localmente.

5.3.3. Comparaciones de los Algoritmos

Los algoritmos presentados tienen características diferentes, y pueden utilizarse como referencias para continuar investigaciones o utilizarlos para su implementación. Se han tenido en cuenta algunos parámetros para su comparación, entre ellos:

- Vueltas de espera: cantidad de grupos que preceden el acceso de un proceso que ha seleccionado un grupo para acceder a la sección crítica.
- Accesos a Memoria Compartida: cantidad de accesos a memoria que no se encuentre localmente.

Teniendo en cuenta estos parámetros se obtienen las siguientes conclusiones:

- Los algoritmos de EMG Basado Tournament, Adaptivo Grupos, Young, EM Grupo Perterson y 1 Actor para el parámetro vueltas de espera tienen un cota superior del $O(m)$, donde m es la cantidad de grupos.
- Para los algoritmos que tienen espera ocupada sobre variables a memoria compartida no se puede estimar la cantidad de accesos que se requiere.

Casos	Alg. dos actores	Adaptivo Grupo	Alg. un actor
P_i quiere acceder cuando hay un compañero y es el primero	4 accesos (entrada) 1 acceso (salida)	4 accesos (entrada) 1 acceso (salida)	8 accesos (entrada) 1 acceso (salida)
P_i es el primer proceso y no tiene oponentes	no se puede determinar	5 accesos (entrada) 1 acceso (salida)	$3 + \log(n) + (n-1)$ accesos (entrada) 3 accesos (salida)

Cuadro 5.7: Comparación entre algoritmos

- El algoritmo Adaptivo Grupos realiza la espera utilizando local spin, considerando que se tienen algunos accesos en forma local y otros en forma remota (NUMA). El *actor grupo* require $6 + 8(log(m) - 1)$ accesos, cada *actor proceso* requiere $(3 + n)$ accesos. Si en un determinado momento se tienen l procesos trabajando cooperativamente en el mismo grupo se requerirán $(3 + n)l + 6 + 8(log(m) - 1)$ accesos.

En el cuadro 5.7 se muestra una comparación entre los algoritmos con dos actores y con un actor, considerando en el *algoritmo con dos actores* la cantidad de accesos del actor proceso.

5.3.4. Algoritmo EMG basado en Mensajes

Esta implementación del modelo de dos actores está basada en pasaje de mensajes y fue presentada en [42]. Se supone que la red es confiable y no necesita reconocimiento. El modelo de competición para el *actor grupo* está basado en el algoritmo de Maekawa (sección 4.2.1). Este algoritmo pertenece a la clasificación de algoritmos de exclusión mutua basados en quorum.

Comportamiento del Actor Proceso

Cada actor proceso i, $1 \leq i \leq m$, realiza los siguientes pasos: selecciona el grupo en el cual va a trabajar, envía un mensaje de solicitud de requerimiento y espera hasta que su requerimiento sea aceptado. Luego ingresa a la sección crítica por un tiempo limitado, y al salir le comunica al grupo que finalizó su trabajo en la sección crítica. Cada proceso está vinculado al grupo mientras utiliza el *recurso* y luego se desvincula del mismo aunque podría vincularse con diferentes grupos a través del tiempo. Se considera que un proceso está vinculado a un grupo un tiempo finito, con el fin de garantizar la propiedad libre de inanición. Se muestra en el cuadro 5.8 los pasos que realiza el *actor proceso*.

Comportamiento del Actor Grupo

El actor grupo está inactivo hasta que recibe un miembro que temporalmente va a trabajar en él. Cuando recibe un mensaje de *Req_Proceso* entonces el grupo inicia su competencia

Proceso P_i

Resto

...

Entrada

G_k = Seleccionar el grupo
Enviar Req_Proceso(G_k, P_i)
Recibir Rep_Proceso(G_k, P_i)

Sección Crítica

...

Salida

Enviar Rep_Proceso_Fin (G_k, P_i)

Mensajes:

- Req_Proceso(G_k, P_i): el proceso P_i envía un mensaje de solicitud para participar en el grupo G_k y utilizar el recurso.
- Rep_Proceso(G_k, P_i): el proceso P_i recibe la respuesta a la solicitud realizada al grupo G_k y puede acceder al recurso.
- Rep_Proceso_Fin(G_k, P_i): el proceso P_i le indica al grupo G_k que ha finalizado su participación del grupo y que se desvincula del mismo.

Cuadro 5.8: Pasos del Actor Proceso

por acceder a la sección crítica. Un único grupo puede utilizar el recurso en un instante de tiempo, pero en ese instante de tiempo pueden compartir el trabajo varios procesos. La competencia la inicia enviando un mensaje *multicast* a todos los miembros de su quorum, y espera que todos le envíen el *lock* para acceder a la exclusión mutua.

El actor *grupo* puede recibir un conjunto de mensajes y de acuerdo al requerimiento su estado actual puede modificarse y además generar nuevos mensajes a otros actores. Teniendo en cuenta el origen del mensaje se lo puede clasificar proveniente de un componente *proceso* o de un actor *grupo*. Los mensajes afectan el estado del *grupo* y, a continuación, se definen para cada tipo de mensaje las acciones que involucran.

- Req_Proceso(G_k, P_i): proviene de un proceso P_i. Si el grupo está esperando que lo seleccione un proceso, esto es, su estado es INACTIVO entonces lo activa y el proceso P_i se convierte en el proceso líder del grupo. Si el grupo está en el estado COMPITIENDO se agrega el proceso a la Lista de Procesos. Si el grupo está en SC (en *Sección Crítica*) y el líder está activo entonces le avisa al proceso que puede comenzar a trabajar, sino lo agrega a la Lista de Procesos (LG).

- Req_Proceso_Fin(G_k, P_i): proviene de un proceso P_i que avisa al grupo que ha finalizado su trabajo en el mismo. Si el proceso es el líder entonces lo deshabilita. Si es el último proceso libera la sección crítica y vuelve al estado INACTIVO y controla si hay procesos que están esperando para ingresar al grupo para iniciar nuevamente la competencia por alcanzar la sección crítica.

- Req_Grupo(G_l, priori): proviene del grupo G_l que requiere el *lock* del grupo G_k. El grupo G_k otorgará el *lock* si lo tiene disponible. Si no tiene disponible el *lock* pueden ocurrir dos casos diferentes: (a) La prioridad del mensaje recibido es menor que la prioridad del mensaje el cual cedió el *lock*, en cuyo caso el requerimiento es demorado. (b) Si la prioridad es mayor entonces reclamará el *lock* al grupo correspondiente para luego cederla al de mayor prioridad.

- Rec_Grupo(G_l, G_k): proviene del grupo G_l, como respuesta afirmativa al mensaje Req_Grupo de requerimiento de *lock*. Si el grupo G_k tiene todos los *locks* entonces cambia su estado a SC y avisa a todos los procesos que están asociados al grupo.

- Rel_Grupo(G_l, G_k): proviene del grupo G_l solicitando que le devuelva el *lock*. Esto será exitoso en el caso que el grupo G_k no esté en la sección crítica.

- Rep_Rel_Grupo(G_l, G_k): proviene del grupo G_l liberando el *lock* que había dado el grupo G_k. El *lock* es cedido al requerimiento con mayor prioridad.

- Lib_Grupo(G_l): proviene del grupo G_l comunicando que terminó su tiempo en la sección crítica. Si hay requerimientos pendientes entonces elige el de mayor prioridad y le otorga el *lock*.

En los cuadros 5.9 y 5.10 se observan los pasos que realiza el actor grupo.

Grupo G_k
♢ Recibir Req_Proceso (G_k, P_i)

```
Si estato = "Inactivo."entonces
    Lider ← P_i
    estado ← "Compitiendo"
    conj ← ∅
    priori ← priori + 1
    AgregarListaProcesos(LP, P_i)
    AgregarListaGrupos(LG, G_k, priori)
    Enviar Multicast Req_Grupo(G_k, priori)
Sino Si estado = "Compitiendo" o
        estado = "Salida" entonces
        AgregarListaProcesos(LP, P_i)
        Sino Si (estado = "SC") y (Lider ≠ -1) entonces
            Enviar Rep_Proceso(G_k, P_i)
        Sino
            AgregarListaProcesos(LP, P_i)
```

♢ Recibir Req_Proceso_Fin (G_k, P_i)

```
    Si Lider = P_i entonces
        Lider ← -1
    EliminarListaProcesos(LP, P_i)
    Si vaciaactivos(LP) entonces
        estado = ← "Salida"
        Enviar Multicast Lib_Grupo(G_k)
        G_l ← SeleccionarGrupo(LG)
        Enviar Rec_Grupo(G_l, G_k)
        estado ← "Inactivo"
        Si HayEnEspera(LP) entonces
            Lider ← Seleccionar(LP)
            estado ← "Compitiendo"
            conj ← ∅
            priori ← priori + 1
            Enviar Multicast Req_Grupo (G_k, priori)
```

♢ Recibir Req_Grupo (G_l, priori)

```
Si vacios(LG) entonces
    AgregarListaGrupos(LG, G_l, priori)
    Enviar Rec_Grupo(G_k, G_l)
Sino
Si MayorPrioridad(LG, G_l, priori)
    entonces
        G_s ← BuscarMayor(LG)
        Enviar Rel_Grupo(G_s, G_k)
        AgregarListaGrupos(LG, G_l, priori)
    Sino
    AgregarListaGrupos(LG, G_l, priori)
```

Cuadro 5.9: Pasos del Actor Grupo

♢ Recibir Rec_Grupo (G_l, G_k)

Si $G_l \notin$ conj entonces
  conj ← conj ∪ {G_l}
  Si | conj |=| S_k | entonces
    estado ← "SC"
    Para cada LP hacer
    {Cada miembro de la lista de procesos}
      EnviarRep_Proceso(G_k,P_i)

♢ Recibir Rel_Grupo (G_l, G_k)

Si estado ≠ "SC" entonces
  conj ← conj - {G_l}
  Enviar Rep_Rel_Grupo(G_k, G_l)

♢ Recibir Rep_Rel_Grupo (G_l, G_k)

G_s ← BuscarMayor(LG)
Enviar Rec_Grupo(G_k, G_s)

♢ Recibir Lib_Grupo (G_l)

EliminarListaGrupo(LG, G_l)
Si no VaciaListaGrupo(LG) entonces
  G_s ← BuscarMayor(LG)
  Enviar Rec_Grupo(G_k, G_s)

Variables:

estado: mantiene el estado actual del grupo, los valores que pueden contener son RESTO, COMPITIENDO, SC y SALIDA.

LP: mantiene información de todos los procesos que quieren utilizar el grupo.

LG: mantiene información de todas las solicitudes de *lock* que están pendientes.

lider: mantiene identificado al proceso que activó el grupo mientras el mismo pertenezca al grupo.

Cuadro 5.10: Pasos del Actor Grupo

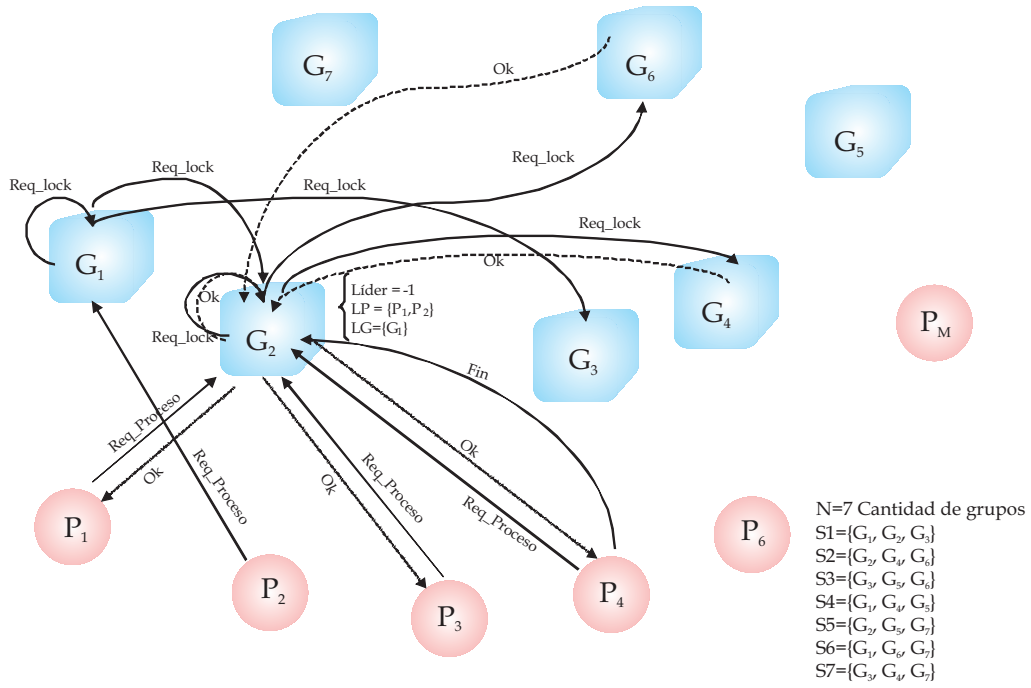


Figura 5.31: Ejemplo

Ejemplo

En la Figura 5.31, se observa el caso que los procesos P_1, P_3 y P_4 seleccionan al grupo G_2 para realizar su trabajo. El grupo comienza su competencia para ingresar a la sección crítica enviando un mensaje *multicast* a los miembros de su quorum, que son los grupos G_2 (siempre está incluido el grupo que realiza la solicitud por la propiedad (b) del algoritmo de Maekawa, sección 4.2.1), G_4 y G_6. Todos los grupos de su quorum le otorgan el *lock* y puede ingresar a la sección crítica. También se observa que el grupo G_1 es activado por el proceso P_2 y envía un mensaje *multicast* a su quorum formado por G_1, G_2 y G_3. Cuando G_2 recibe la solicitud está en SC, y entonces el requerimiento es demorado aunque tenga una prioridad mayor hasta que todos los procesos activos finalicen su tarea en la sección crítica.

En la Figura 5.31, también se observa que el proceso P_4 ha finalizado su trabajo, se ha desvinculado y que era el líder del grupo (Lider=-1). Cuando arribe un nuevo requerimiento de un proceso será incorporado a la Lista de Procesos (LP) pero en estado pendiente. Este condicionamiento es incorporado para garantizar que un grupo permanezca un tiempo finito en la sección crítica y así evitar la inanición.

Condiciones del Algoritmo

Se considera que el algoritmo garantiza exclusión mutua cuando no puede ocurrir el caso en que los procesos P_i y P_j estén utilizando el recurso al mismo tiempo, y donde $P_i \in G_k$ y $P_j \in G_l$ y $G_k \neq G_l$, esto es, hay dos procesos en la sección crítica correspondiente a diferentes grupos.

Teorema 5.6 *El algoritmo EMG basado en mensajes garantiza exclusión mutua.*

Supongamos que los procesos P_i y P_j están en la sección crítica al mismo tiempo, y donde $P_i \in G_k$ y $P_j \in G_l$ y $G_k \neq G_l$. Entonces se cumple que el grupo G_k recibió los *locks* de todo su quorum S_k y también se cumple que el grupo G_l recibió los *locks* de todo su quorum S_l. Por la condición (a) $S_k \cap S_l \neq \emptyset$ entonces ocurrió el caso que el miembro que comparten el quorum cedió el *lock* a 2 requerimientos, y esto es una contradicción. Por lo tanto, el algoritmo garantiza la exclusión mutua.

Teorema 5.7 *El algoritmo EMG basado en mensajes garantiza espera limitada.*

La espera para ingresar a la sección crítica está limitada ya que cada solicitud de *lock* tiene una prioridad y ésta es única, ya que en algún momento en el tiempo cada solicitud será la de mayor prioridad y podrá acceder a la sección crítica. En el peor caso tendrá que esperar que ingresen todos los demás grupos. Además se debe considerar que cada grupo no esté indefinidamente en la sección por el ingreso de nuevos procesos. Para evitar este problema se permite que ingresen nuevos procesos a la sección crítica mientras el primer proceso que activó el grupo está trabajando y, de esta manera, se alcanza un mayor nivel de concurrencia.

Complejidad del Algoritmo

La complejidad de los algoritmos se pueden medir de acuerdo a diferentes consideraciones, como puede ser el número de operaciones requeridas en memoria compartida para ingresar a la sección crítica, ó el intervalo de tiempo entre entradas a la sección crítica, ó la cantidad de tráfico de interconexión que genera. De acuerdo al tipo de algoritmo se puede utilizar una u otra medida.

Para medir la complejidad de este algoritmo, se tiene en cuenta la cantidad de mensajes que son requeridos para acceder a la sección crítica. Entre el proceso y el grupo son requeridos 3 mensajes de comunicación. El primer mensaje lo envía el proceso al grupo para vincularse, el segundo mensaje lo envía el grupo comunicando que tiene acceso a la sección crítica y el tercer mensaje lo envía el proceso para avisar que ha finalizado su trabajo.

Se considera que N es la cantidad de grupos, k es la cantidad de miembros del quorum, con $k = \sqrt{N}$. Las cantidades de mensajes que requiere un grupo para ingresar a la sección crítica, sin tener que devolver ningún *lock* son los siguientes (en el mejor caso):

(a) $k - 1$ mensajes de solicitud de *lock*.

(b) $k-1$ mensajes de otorgamiento de *lock*.

(c) $k-1$ mensajes de liberación de *lock*.

Cuando un proceso quiere trabajar en un grupo y es el líder del mismo, entonces se requieren $3 + 3(k-1)$ mensajes para que pueda acceder a la sección crítica. Si en un grupo están trabajando en forma concurrente l procesos entonces para los $l-1$ procesos se requieren $3(l-1)$ mensajes para acceder a la sección crítica y en total $3l+3(k-1)$ mensajes.

En el caso en el cual se deba devolver todos los *locks* por solicitudes de requerimiento de mayor prioridad, se agregan $k-1$ mensajes de entrega del *lock* y otros $k-1$ mensajes de otorgamiento del *lock*. Si el grupo tuviera l procesos trabajando en forma concurrente entonces requeriría $3l + 5(k-1)$ mensajes.

5.4. Aplicaciones

En esta sección se presentan algunos ejemplos de aplicación donde se requiere la utilización de algoritmos de exclusión mutua para grupos de procesos.

(A) Coordinar las actividades de un congreso.

Caso 1. Un Congreso que está formado por diferentes foros (workshops) y existe una única sala de conferencias para que se realicen los mismos. En cada una de los foros (workshops) disertan expositores que se asocian a los mismos.

Caso 2. Se tiene un Congreso que está formado por diferentes foros (workshops) y existe una única sala de conferencias para que éstos se realicen. En cada foro participan un conjunto de expositores que debaten sobre un tema propuesto. Todos participan activamente del foro.

Cada uno de los foros (workshops) tiene asociado un tiempo máximo que puede permanecer en la sala de conferencias para que lo puedan utilizar otros foros. Además, cada expositor tiene un tiempo para su disertación.

(B) Ambientes Educativos Virtuales.

En los ambientes educativos virtuales, existen diferentes tareas que deben desarrollar los alumnos y los tutores de los mismos, algunas de las cuales son cooperativas. Por ejemplo, la utilización de pizarra compartida, donde en un instante de tiempo sólo la pueden utilizar el conjunto de alumnos asignados a una tarea, la edición de un documento, compartir la sala de chat para resolver la tarea compartida, etc.

5.5. Resumen

La exclusión mutua para grupos de procesos es una extensión del problema tradicional de exclusión mutua, en el cual varios procesos pueden compartir concurrentemente un recurso

para realizar su tarea. En este capítulo se realizó un estudio, análisis y se presentó un modelo de solución para este problema, que se puede aplicar en la resolución de diferentes aplicaciones distribuidas. Algunos ejemplos se presentaron en la sección 5.4.

Los algoritmos se pueden construir utilizando diferentes modelos de diseño, y se los clasifica en dos modelos: un actor y dos actores. Se mostraron algunas soluciones propuestas para este problema diseñadas utilizando el modelo de un sólo actor. El modelo (patrón) general que se presentó para resolver el problema de exclusión mutua de grupos está basado en dos Actores, el *actor proceso* y el *actor grupo*. El mismo se puede aplicar sobre cualquier modelo que resuelva el problema tradicional de la exclusión mutua. Se presentaron soluciones al problema utilizando el modelo de dos actores y diferentes paradigmas de implementación, memoria compartida y pasaje de mensajes. Se estudió la complejidad de los algoritmos teniendo en cuenta el paradigma utilizado para implementarlos.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

Los sistemas centralizados y en especial los sistemas distribuidos requieren de protocolos para sincronizar y coordinar el acceso ó utilización de recursos de uso exclusivo. Con la propagación de las redes de alta velocidad, la mejor calidad en los medios de comunicación, las aplicaciones distribuidas y tareas cooperativas van en crecimiento y popularidad.

Cuando se diseña un sistema es importante analizar los modelos definidos y las propiedades que presentan. En el caso de los sistemas distribuidos un concepto importante es el tiempo asociado con la velocidad de ejecución, esto es, si el sistema es sincrónico o asincrónico. Los modelos asincrónicos son más difíciles de programar que los sincrónicos, pero son más generales y portables, ya que funcionan sobre redes con tiempo de ejecución arbitrarios. En este trabajo se consideran propuestas que soporten las propiedades del modelo asincrónico.

Uno de los problemas es el acceso exclusivo de un proceso/usuario hacia un recurso, y en los ambientes cooperativos varios procesos/usuarios comparten el acceso/utilización del recurso para resolver una tarea. Para estas tareas es necesario contar con protocolos que garanticen exclusión mutua para procesos y para grupos de procesos. De acuerdo al problema que se esté resolviendo, es necesario contar con un protocolo de exclusión mutua, para el caso que exista una única unidad del recurso y el uso sea excluyente; un protocolo de k-exclusión mutua, para el caso que existan varias unidades del recurso y el uso sea excluyente; y un protocolo de exclusión mutua para grupos de procesos, para el caso donde se comparte el recurso. Las condiciones de un buen algoritmo de exclusión mutua son: exclusión, progreso y equitatividad (libre de inanición). Los algoritmos se pueden escribir en un formato tradicional, en pseudo-código (que facilita la comprensión) y luego traducirlo a un modelo formal. En este caso se utilizaron autómatas de E/S para verificar que está bien formado así como su correctitud.

Los diferentes paradigmas para construir los protocolos son: memoria compartida y pasaje de mensajes. La memoria compartida presenta la ventaja de abstraerse de la ubicación de los recursos. Esta propiedad de transparencia facilita el trabajo a los diseñadores y pro-

gramadores al momento de desarollar una aplicación, pero presenta un alto costo en la implementación. El pasaje de mensajes presenta como ventaja el menor costo de implementación y se utiliza en la construcción de otros modelos de comunicación entre procesos, y como desventaja la falta de transparencia.

En el análisis del problema de la exclusión mutua para un proceso, se consideró al sistema compuesto por n procesos, p_0, p_1, ..., p_{n-1}, que compiten por acceder al recurso. Las soluciones a este problema pueden estar basadas en memoria compartida, pasaje de mensajes basados en quorum ó basados en token.

En ambientes distribuidos, algunos compiten por los recursos y otros los *comparten*, cooperando en la realización de su tarea. Esta característica motivó el estudio del problema de exclusión mutua para grupos de procesos. Se considera que el sistema está formado por un conjunto de n procesos, p_0, p_1, ..., p_{n-1}; donde los procesos pueden participar de cualquiera de los diferentes m grupos, G_0, G_1, ..., G_{m-1}. Las soluciones al problema de la exclusión mutua para grupos de procesos se las clasifica de acuerdo a su diseño en: modelos basados en un actor y modelos basados en dos actores. Del trabajo realizado aplicado a la exclusión mutua para grupos de procesos se presenta:

- Un patrón (modelo) general para resolver el problema de exclusión mutua de grupos basados en dos Actores el cual se puede aplicar sobre cualquier modelo que resuelva la exclusión mutua.
- Soluciones para el problema de EMG utilizando el modelo general sobre protocolos basados en memoria compartida y pasaje de mensajes.
- El modelo general de dos Actores presenta las calidades de entendible y fácil de aplicar.
  - El modelo es entendible, ya que un actor es el que juega el rol de proceso que quiere utilizar en forma compartida el recurso y el otro actor juega el rol de grupo que compite por acceder al recurso (ver la figura 5.9).
  - El modelo es fácil de aplicar, ya que el protocolo de exclusión mutua está incluido en el actor grupo y el proceso que quiere acceder se comunica con el actor grupo para competir por el recurso.
- Con respecto a la calidad de performance, dependiendo de las consideraciones que se realicen, como por ejemplo la cantidad de accesos a memoria, alcanza un óptimo desempeño según se observa en el cuadro 5.7.

6.2. Trabajo Futuro

Las aplicaciones distribuidas se pueden utilizar en diferentes ambientes y modelos de redes.

- Ambientes que comparten la memoria o utilizan el paradigma de memoria compartida.

- Ambientes donde la red es cableada.
- Ambientes donde la red es inalámbrica, puede ser una red celular ó *ad hoc*.

Las redes *ad hoc* no tienen una infraestructura fija y todos los nodos/sitios son capaces de moverse, las cuales determinan la conectividad de la red. Los nodos ad hoc pueden comunicarse solo directamente con los nodos que están inmediatamente dentro de su rango de transmisión. Para comunicarse con otros nodos, un nodo intermedio es utilizado para propagar al nodo destino. Los nodos necesitan cooperar en orden de mantener la conectividad y cada nodo actúa como un router.

Las características de un sistema *ad hoc* son la auto-organización, ser verdaderamente decentralizado, y ser altamente dinámicos. Estas características favorecen el desarrollo de aplicaciones de interés como son conferencias, encuentros, comunicación inalámbrica entre vehículos en movimiento, etc. Una de las líneas de proyección para trabajo futuro es la consideración del modelado de soluciones para el problema de la exclusión mutua de grupos aplicadas al modelo de red *ad hoc*.

Otra línea de proyección para el futuro es la consideración de restricciones de tiempo en la utilización del recurso por parte del grupo y de los procesos.

Las aplicaciones distribuidas tienen asociadas requerimientos funcionales, y algunas también incluyen requerimientos temporales, esto es, restricciones de tiempo ó deadlines. Por ejemplo, un Congreso Científico está formado por sesiones de trabajo, dónde cada una de las sesiones tiene una duración máxima de dos horas, y se permite el ingreso de nuevos disertantes en función del tiempo que queda disponible en la sesión y el tiempo que requiera para su disertación. Los sistemas distribuidos multimedia presentan requerimientos funcionales y temporales, y requieren que se controlen las competiciones por el recurso, en especial, el ancho de banda de la red, considerando soluciones de requerimiento dinámico.

Bibliografía

[1] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Adaptive long-lived renaming using bounded memory (extended abstract), 1999. http://www.cs.technion.ac.il/˜ hagit/pubs/AAFST99disc.ps.gz.

[2] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, Mayo 1999.

[3] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A bounded first-in, first-enabled solution to the l-exclusion problem. *ACM Transactions on Programming Language and Systems*, 16(3):939–953, Mayo 1994.

[4] Y. Afek, G. Stupp, and D. Touitou. Long-lived and adaptive collect with applications. *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 262–272, Octubre 1999.

[5] D. Agrawal and A. El Abbadi. Efficient solution to the distributed mutual exclusion problem. In *Annual ACM Symposium on Principles of Distributed Computing*, pages 193–200, 1989.

[6] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, Febrero 1991.

[7] D. Agrawal, Ö. Eğecioğlu, and A. El Abbadi. Analysis of quorum-based protocols for distributed (k+1)-exclusion. *IEEE Transaction on Parallel and Distributed Systems*, 3(3), Abril 1997.

[8] G. Agrawal and P. Jalote. An efficient protocol for voting in distributed systems. *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 640–647, Junio 1992.

[9] E. Ailiomandi, M. J. Fischer, and N. A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the Association for Computing Machinery*, 30(3):449–456, Julio 1983.

[10] J. Anderson and Y. J. Kim. Adaptive mutual exclusion with local spinning. *Proceedings of the 14th International Symposium on Distributed Computing*, Octubre 2000.

[11] J. Anderson and Y. J. Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1):17–29, Enero 2001.

[12] J. Anderson, Y. J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986, 2001. En http://citeseer.ist.psu.edu/anderson01sharedmemory.html.

[13] J. H. Anderson and Y. J. Kim. Fast *and* scalable mutual exclusion. *Proceeginds of the 13th International Symposium on Distributed Computing*, Septiembre 1999.

[14] J. H. Anderson and Y. J. Kim. An improved lower bound for the time complexity of mutual exlcusion. *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 90–99, Agosto 2001.

[15] J. H. Anderson and Y. J. Kim. Nonatomic mutual exclusion with local spinning, (extended abstract). In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC '02)*, pages 3–12, 2002.

[16] J. H. Anderson and M. Moir. Using k-exclusion to implement resilient, scalable shared objects (extended abstract). *Proceedings of the 13th ACM Symposium on Principles of Distributed Computing*, pages 141–150, Agosto 1994.

[17] J. H. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11(1):1–20, Diciembre 1997.

[18] M. Ben Ari. *Principles of Concurrent Programming.* Prentice Hall, Englewood Cliffs, 1982.

[19] M. Ben Ari. *Principles of Concurrent and Distributed Programing.* Englewood Cliffs NJ: Prentice Hall, 1990.

[20] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. *Distributed Computing*, 15(3):177–189, 2002. Proceedings of 19th Annual ACM Symposium on Principles of Distributed Computing, Julio 2000.

[21] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. *Proceedings 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 279–286, Junio 1998. Extended version disponible en Technion Computer Science Department Technical. Report #0931, Abril 1998.

[22] H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report 0956, Faculty of Computer Science, Technion, Haifa. http://www.cs.technion.ac.il/~ hagit/pubs/tr0956.ps.gz, 1999.

[23] H. Attiya and A. Fouren. Polynomial and adaptive long-lived (2k-1) renaming. (Extended Abstract), http://www.cs.technion.ac.il/~ hagit/pubs/af00f.ps.gz, 2000.

[24] H. Attiya, A. Fouren, and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, 15(2):87–96, 2002.

[25] R. Baldoni, Y. Manabe, M. Raynal, and S. Aoyagi. k-arbiter: A safe and general scheme for h-out of-k mutual exclusion problems. Reporte de Investigación no. 2523, INRIA. http://citeseer.comp.nus.edu.sg/baldoni95karbiter.html, Abril 1995.

[26] S. Banerjee and P. K. Chrysanthis. A new token passing distributed mutual exclusion algorithm. *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pages 717–724, 1996.

[27] D. Barbara and H. García-Molina. Mutual exclusion in partitioned distributed systems. *Distributed Computing*, 1:119–132, 1986.

[28] E. Borowsky and E. Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium Theory of Comp*, pages 91–100, 1993.

[29] E. Borowsky, E. Gafni, N. Lynch, and S. Rajsbaum. The bg distributed simulation algorithm. Technical Report MIT/LCS/TM-573, Laboratory for Computer Science, MIT, Diciembre 1997.

[30] V. Bortnik. Adaptive algorithms for mutual exclusion. Master's thesis, Department of Computer Science, The Technion, Diciembre 2000.

[31] J. Brzęziński and D. Wawrzyniak. Consistency requirements of distributed shared memory for dijkstra's mutual exclusion algorithm. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 618–625, 2000. Taipei.

[32] S. Bulgannawar and N. H. Vaidya. Distributed k-mutual exclusion. In *Proceedings of the 15th IEEE International Conference on Distributed Systems*, pages 153–160, Mayo 1995.

[33] J. Burns and N. A. Lynch. Bound on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

[34] J. E. Burns, P. Jackson, N. A. Lynch, M. J. Fischer, and G. L. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1), 1982.

[35] J. E. Burns and N. A. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of the 18th Allerton Conference on Communication, Control and Computing*, pages 833–842. Monticello, IL, Octubre 1980.

[36] S. Cantarell, A. K. Datta, F. Petit, and V. Villain. Token based group mutual exclusion for asynchronous rings. (extended abstract). In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, pages 691–694, 2001.

[37] G. Cao, M. Singhal, Y. Deng, N. Rishe, and W. Sun. A delay-optimal quorum-based mutual exclusion scheme with fault-tolerance capability. *IEEE Transactions on Parallel and Distributed Systems archive*, 12(12):1256–1268, Diciembre 2001.

[38] K. Cenci and J. Ardenghi. Exclusión mutua en la implementación memoria compartida asincrónica. In *ICIE Y2K VI Congreso Internacional de Ingeniería Informática*. Facultad de Ingeniería - UBA. Abril 26-28, 2000. ISBN 987-461764-7.

[39] K. Cenci and J. Ardenghi. Sobre algoritmos distribuidos de exclusión mutua para n procesos. In *VI Congreso Argentino de Ciencias de la Computación CACIC 2000*, pages 973–984. UNPSJB, Ushuaia, 2-7 Octubre, 2000. ISBN 950-763-033-3.

[40] K. Cenci and J. Ardenghi. Exclusión mutua para coordinación de sistemas distribuidos. In *VII Congreso Argentino de Ciencias de la Computación CACIC 2001*, pages 717–724. UNPA, Calafate, 16-20 Octubre, 2001. ISBN 987-96 288-6-1.

[41] K. Cenci and J. Ardenghi. Algoritmo para coordinar exclusión mutua y concurrencia de grupos de procesos. In *VIII Congreso Argentino de Ciencias de la Computación CACIC 2002*, pages 263–271. UBA, Buenos Aires, 15-18 Octubre, 2002.

[42] K. Cenci and J. Ardenghi. Exclusión mutua en grupos de procesos a través de mensajes. In *IX Congreso Argentino de Ciencias de la Computación CACIC 2003*, pages 345–353. UNLP, La PLata, 7-10 Octubre, 2003.

[43] K. Cenci and J. Ardenghi. Modelo asincrónico adaptativo de exclusión para grupos de procesos. In *XI Congreso Argentino de Ciencias de la Computación CACIC 2005*, pages 1116–1127. Universidad Nacional de Entre Ríos, 17-21 Octubre, 2005. ISBN 950-698-166-3.

[44] K. Cenci and J. Ardenghi. Exclusión mutua para grupos de procesos utilizando un actor. In *XIII Congreso Argentino de Ciencias de la Computación CACIC 2007*, pages 1216–1226. Universidad Nacional del Nordeste, Corrientes. 1-5 de Octubre, 2007. ISBN 978-950-656-109-3.

[45] Y. Chang, M. Singhal, and M. Liu. A fault tolerant algorithm for distributed mutual exclusion. In *Proceedings of 9th IEEE Symposium on Realiable Distributed Systems*, pages 146–154, Octubre 1990.

[46] Y. Chang, M. Singhal, and M. Liu. An improved O($\log n$) mutual exclusion algorithm for distributed systems. In *Proceedings of the International Conference on Parallel Processing*, volume 3, pages 295–302, Agosto 1990.

[47] Y. Chang, M. Singhal, and M. Liu. A dynamic token-based distributed mutual exclusion algorithm. In *Proceedings of 10th International Phoenix Conference on Computers and Communications*, pages 240–246, Marzo 1991.

[48] S. Y. Cheug, M. Ahamad, and M. H. Ammar. The grid protocol: A high performance scheme for maintaining replicated data. In *Proceedings of 6th IEEE Conference on Data Engg.*, pages 438–445, Enero 1990.

[49] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01)*, pages 11–20, 2001.

[50] M. Choy. Robust distributed mutual exclusion. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems (ICDCS '96)*, pages 760–767, 1996.

[51] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. In *Proceedings of the Twenty Fourth Annual ACM Symposium on Theory of Computing*, pages 593–602. Victoria, British Columbia, Canada, 4-6 Mayo, 1992.

[52] R. Cypher. The communication requirements of mutual exclusion. In *ACM Symposium on Parallel Algorithms and Architectures archive Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures Santa Barbara, California, United States*, pages 147–156, 1995.

[53] D. M. Dhamdhere and S. S. Kulkami. A token based k-resilient mutual exclusion algorithm for distributed systems. *Information Processing Letters*, 50:151–157, 1994.

[54] E. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, Noviembre 1974.

[55] P. H. Enslow. What is a 'distributed' data processing systems? *IEEE Computers*, pages 12–27, Diciembre 1981.

[56] M. Fisher, N. Lynch, J. Burns, and A. Borodin. Distributed fifo allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, 1989.

[57] S. Fujita, M. Yamashita, and T. Ae. Distributed k-mutual exclusion problem and k-coteries. *2nd International Symposium on Algorithms, Lecture Notes in Computer Science*, 557:22–31, 1991.

[58] H. García-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the Association for Computing Machinery*, 32(4):841–860, Octubre 1985.

[59] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, Abril 1991.

[60] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerante broadcasts and related problems. Technical report, Technical Report, Dept. of Computer Science, University of Toronto, 1994.

[61] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Enero 1991.

[62] A. Housni and M. Trehel. A new distributed mutual exclusion algorithm for two groups. In *Proceedings of the 2001 ACM, Symposium on Applied Computing*, pages 531–538, Marzo 2001.

[63] S. T. Huang, J. R. Jiang, and Y. C. Kuo. K-coteries for fault-tolerant k entries to a critical section. In *Proceedings 13th IEEE Internatational Conference on Distributed Computing Systems*, pages 74–81, Mayo 1993.

[64] T. L. Huang. Fast and fair mutual exclusion for shared memory systems. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, pages 224–231, 1999.

[65] D. R. Hughes and F. C. Piper. *Proyective planes*. Springer Verlag, 1973.

[66] J. R. Jiang. Fault-tolerant distributed mutual exclusion with O(1) message overhead. In *Proceedings of the 13th International Conference on Applied Informatics*, pages 228–231. Austria, Febrero 1995.

[67] J. R. Jiang. A distributed h-out of-k mutual exclusion algorithm for ad hoc mobile networks. In *Proceedings of 2002 International Parallel and Distributed Processing Symposium (IPDPS '02). 2nd International Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, pages 196–202, Abril 2002.

[68] J. R. Jiang. Distributed h-out of-k mutual exclusion using k-coteries. In *Proceedings of the 3rd International Conference on Parallel and Distributed Computing, Application and Technologies (PDCAT'02)*, pages 218–226, 2002.

[69] J. R. Jiang. A group mutual exclusion algorithm for ad hoc mobile networks. In *Proceedings of the 6th International Conference on Computer Science and Informatics*, pages 266–270, Marzo 2002.

[70] J. R. Jiang. A prioritized h-out of-k mutual exclusion algorithms with maximum degree of concurrency for mobile ad hoc networks and distributed systems. In *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Application and Technologies (PDCAT'03)*, pages 329–334, 2003.

[71] J. R. Jiang and S. T. Huang. Obtaining nondominated k-coteries for fault-tolerant distributed k-mutual exclusion. Technical report, Technical Report, Department of Computer science, Tsing Hua University, Taiwan, 1994.

[72] J. R. Jiang and S. T. Huang. Fault-tolerant distributed k-mutual exclusion with constant expected message cost. In *Workshop on Distributed System Technologies and Applications*, pages 104–110, Julio 1995.

[73] J. R. Jiang, S. T. Huang, and Y. C. Kuo. Cohorts structures for fault-tolerant k entries to a critical section. *IEEE Transactions on Computers*, 46(2):222–228, Febrero 1997.

[74] Y. J. Joung. Asynchronous group mutual exclusion (extended abstract). In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 51–60, Junio 1998.

[75] Y. J. Joung. The congenial talking philosophers problem in computer networks (extended abstract). In *Proceedings 13th International Symposium on Distributed Computing*, pages 195–209, 1999.

[76] Y. J. Joung. Quorum-based algorithms for group mutual exclusion. *IEEE Transactions on Parallel and Distributed Systems*, pages 463–476, Mayo 2003.

[77] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. Availability of k-coterie. *IEEE Transactions on Computers*, 42(5):553–558, Mayo 1993.

[78] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. A distributed k-mutual exclusion algorithm using k-coteries. *Information Processing Letters*, 49:213–218, Marzo 1994.

[79] H. Kakugawa and M. Yamashita. Local coteries and a distributed resource allocation algorithm. *Transactions of Information Processing Society of Japan*, Agosto 1996.

[80] H. Kakugawa and M. Yamashita. A universal self-stabilizing mutual exclusion algorithm. *Transactions of Information Processing Society of Japan*, 0(0), 2000.

[81] P. Keane and M. Moir. A general resource allocation synchronization problem. In *21st International Conference on Distributed Computing Systems*, pages 557–564, 2001.

[82] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):673–685, Julio 2001.

[83] Y. J. Kim and J. H. Anderson. A time complexity bound for adaptive mutual exclusion (extended abstract). In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 1–15, Octubre 2001.

[84] K. J. Kristoffersen, F. Laroussinie, K. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In *Proceedings 7th. International Joiny Conference Theory and Practice of Software Development (TAPSOFT'97)*, pages 565–579, Abril 1997.

[85] A. Kumar. Performance analysis of a hierarchical quorum consensus algorithm. In *Proceedings of 10th IEEE International Conference on Distributed Computing Systems*, pages 378–385, Mayo 1990.

[86] A. Kumar. Hierarchical quorum consensus: A new method for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1104, Septiembre 1991.

[87] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, Agosto 1974.

[88] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, Julio 1978.

[89] L. Lamport. The mutual exclusion problem. part I: A theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, Abril 1986.

[90] L. Lamport. The mutual exclusion problem. part II: Statement and solutions. *Journal of the ACM*, 33(2):327–348, Abril 1986.

[91] L. Lamport. A fast mutual exclusion algorithm. *ACM on Transactions on Computer Systems*, 5(1), Febrero 1987.

[92] G. Le Lann. Distributed systems-towards a formal approach. In *Information Processing 77 (Toronto, Agosto 1977), vol. 7 of Proceedings of IFIP*, pages 155–160. North-Holland, Amsterdam, Bruce Gilchrist, 1977.

[93] S. Lodha and A. Kshemkalyani. A fair distributed mutual exclusion algorithm. *IEEE Transaction on Parallel and Distributed Systems*, 11(6):537–549, Junio 2000.

[94] W. S. Luk and T. T. Wong. Two new quorum based algorithms for distributed mutual exclusion. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems (ICDCS '97)*, pages 100–107, 1997.

[95] N. A. Lynch. *Distributed Algorithms*. Morgan-Kaufmann, 1996.

[96] N. A. Lynch. I/O automaton models and proofs for shared-key communication systems. Technical report, Technical Report MIT/LCS/TR-789, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, Agosto 1999.

[97] N. A. Lynch and M. Tuttle. Hierarchical correctness proofs for distibuted algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (Versión abreviada)*, Agosto 1987. Technical Report MIT/LCS/TR-387.

[98] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quaterly*, 2(3):219–246, Septiembre 1989.

[99] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, Mayo 1985.

[100] K. Makki, P. Banta, K. Been, N. Pissinou, and E. Park. A token based distributed k mutual exclusion algorithm. In *IEEE Proceedings of the Symposium on Parallel and Distributed Processing*, pages 408–411, Diciembre 1992.

[101] Y. Manabe, R. Baldoni, M. Raynal, and S. Aoyagi. k-arbiter: A safe and general scheme for h-out of-k mutual exclusion. *Theoretical Computer Science*, 193(1–2):97–112, Febrero 1998.

[102] Y. Manabe and N. Tajima. (h-k)-arbiter for h-out of-k mutual exclusion problem. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 216–223, 1999.

[103] Y. Manabe and N. Tajima. (h,k)-arbiters for h-out of-k mutual exclusion problem. *Theoretical Computer Science*, 310(1–3):379–392, Enero 2004.

[104] C. Mittermaier, T. Prokosch, and I. Rents. Algorithms for distributed systems maekawa's voting algorithm for mutual exclusion, Febrero 1999. http://www.risc.uni-linz.ac.at/ software/daj/Maekawa/automaton.ps.

[105] M. Mizuno, M. Neilsen, and R. Rao. A token based distributed mutual exclusion algorithm based on quorum agreements. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS '91)*, pages 361–368, 1991.

[106] M. Mizuno, M. Nesterenko, and H. Kakugawa. Lock-based self-stabilizing distributed mutual exclusion algorithms. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems (ICDCS '96)*, pages 708–716, 1996.

[107] M. Moir and J. Anderson. Fast, long-lived renaming (extended abstract). In *Proceedings of the 8th International Workshop on Distributed Algorithms, WDAG '94*, pages 141–155. Terschelling, The Netherlands, September 29 - October 1, 1994.

[108] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programing*, 25(1):1–39, Octubre 1995. También en Proceedings 8th International Workshop on Distributed Algorithms, pp. 141–155, Septiembre 1994.

[109] M. Moir and J. A. Garay. Fast, long-lived renaming improveed and simplified. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, Octubre 1996.

[110] S. Mullender, editor. *Distributed Systems.* Addison-Wesley, 2da. edition, 1993.

[111] M. L. Neilsen and M. Mizuno. Coterie join algorithm. *IEEE Transaction on Parallel and Distributed Systems*, 3(5):759–765, Septiembre 1992.

[112] M. Nesterenko and M. Mizuno. A quorum-based self-stabilizing distributed mutual exclusion algorithm. *Journal of Parallel and Distributed Computing 62*, pages 284–305, 2002.

[113] W. K. NG and C. V. Ravishankar. Coteries templates: A new quorum construction method. In *15th. International Conference on Distributed Computing Systems (ICDCS'95)*, pages 92–100, 1995.

[114] M. L. Nielsen. Measures of importance and symmetry in distributed systems. In *5th IEEE Symposium on Parallel and Distributed Computing*. Dallas, TX, Diciembre 1993.

[115] M. L. Nielsen and M. Mizuno. Nondominates k-coteries for multiple mutual exclusion. *Information Processing Letters*, 1994.

[116] T. Parka and H. Y. Yeom. Application controlled checkpointing coordination for fault-tolerant distributed computing systems. *Parallel Computing*, 26(4):467–482, 2000.

[117] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, Junio 1981.

[118] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. *ACM Annual proceedings of Theory of Computing*, pages 91–97, 1977.

[119] S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *Programming Concepts and Methods, IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 424–443, Junio 1998.

[120] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30(4):189–193, Febrero 1989.

[121] K. Raymond. A tree-based mutual algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[122] M. Raynal. Algorithms for mutual exclusion. Technical report, MIT Press, Cambridge, MA, 1986.

[123] M. Raynal. A distributed solution for the k-out of-m resources allocation problem. In *Proceedings 1st Conference on Computing and Information, Lecture Notes in Computer Sciences*, volume 497, pages 599–609, Mayo 1991.

[124] I. Rhee, S. Y. Cheung, P. W. Hutto, and V. S. Sunderam. Group communication support for distributed collaboration systems. *Cluster Computing*, 2(1):3–16, 1999. Proceedings of ICDCS (Mayo 1998).

[125] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Comunications of the ACM*, Enero 1981.

[126] B. A. Sanders. The information structure of distributed mutual exclusion algorithms. *ACM Transaction on Computer Systems*, 5(3):284–299, Agosto 1987.

[127] P.C. Saxena and J. Raib. A survey of permission-based distributed mutual exclusion algorithms. *Computer Standards & Interfaces*, 25(2):159–181, Mayo 2003.

[128] C. L. Seitz. Resources in parallel and concurrent systems. *ACM Press*, 1990.

[129] D. Shou and S. D. Wang. An efficient quorum generating approach for distributed mutual exclusion. *Journal of Information Science and Engineering*, 9:201–227, Junio 1993.

[130] A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison-Wesley, 5ta. edition, 1998.

[131] P. K. Srimani and R. L. N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 41(1):51–57, Enero 1992.

[132] W. Steiner and M. Paulitsch. The transition from asynchronous to synchronous system operation: An approach for distributed fault-tolerant systems. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 329–336, 2002.

[133] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transaction on Computer Systems*, 3(4):344–349, Noviembre 1985.

[134] P. Thambu and J. Wong. An efficient token-based mutual exclusion algorithm in a distributed system. *J. Systems Software*, 1995.

[135] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy of database. *ACM Transaction Database Systems*, 4(2):180–209, Junio 1979.

[136] K. Vidyasankar. A simple group mutual l-exclusion algorithm. *Information Processing Letters*, 85(2):79–85, Enero 2003.

[137] J. E. Walter. A k-mutual exclusion algorithm for ad hoc mobile networks. Technical report, Technical report 00-022, Texas A&M University, 2000.

[138] J. E. Walter, J. L. Welch, and N. H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Wireless Networks*, 9(6):585–600, Noviembre 2001. Baltzer Wireless Networks Journal, special issue on DialM papers, 2001.

[139] Y. E. Walter and M. Mohanty G. Cao. A k-mutual exclusion algorithm for wireless ad hoc networks. In *Proceedings of the first annual Workshop on Principles of Mobile Computing (POMC 2001)*, pages 29–39, Agosto 2001.

[140] J. Wu. *Distributed System Design*. CRC Press LLC, 1999.

[141] K. P. Wu and Y. J. Joung. Asynchronous group mutual exclusion in ring networks. In *13th International Parallel Processing Symposium / 10th Symposium on Parallel and Distributed Processing (IPPS / SPDP '99). Proceedings. IEEE Computer Society*, pages 539–543, Abril 1999.

[142] M. Y. Wu and W. Shu. An efficient distributed token-based mutual exculsion algorithm with central coordinator. *Journal of Parallel and Distributed Computing*, 62:1602–1613, 2002.

[143] Y. Yan, X. Zhang, and H. Yang. A fast token-chasing mutual exclusion algorithm in arbitrary network topologies. *Journal of Parallel and Distributed Computing*, 35:156–172, 1996.

[144] J. H. Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, Agosto 1995.