Exclusión Mutua en Grupos de Procesos a través de Mensajes

Karina M. Cenci * Jorge R. Ardenghi †

Laboratorio de Investigación en Sistemas Distribuidos (LiSiDi)
miembro del IICyTI
(Instituto de Investigación en Ciencia y Tecnología Informática)
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Resumen

En sistemas distribuidos las aplicaciones realizan trabajos que requieren acceso en forma exclusiva a un recurso o realizan trabajo en forma conjunta para el cual requieren de la utilización de un recurso, para poder mantener estos requerimientos es necesario contar con protocolos que garanticen el acceso a los recursos de procesos que no compiten entre sí. El protocolo puede ser resolver el tradicional problema de exclusión mutua, o puede ser una extensión al problema donde k o un grupo de procesos están al mismo tiempo utilizando el recurso.

En este trabajo se presenta un algoritmo que permite que grupos utilicen el recurso, donde cada uno de los grupos está integrado por un conjunto de procesos y los grupos compiten por acceder al recurso. El algoritmo está formado por dos componentes: grupos y procesos, se basa en algoritmos distribuidos basados en pasajes de mensajes utilizando quorum para acceder a la sección crítica.

Palabras Claves: Sistemas Distribuidos - Exclusión Mutua - Exclusión Mutua de Grupo - Concurrencia

*e-mail: kmc@cs.uns.edu.ar
†e-mail: jra@cs.uns.edu.ar

1. Introducción

El paradigma de la *exclusión mutua* se puede utilizar en una amplia gama de aplicaciones de sistemas centralizados y en los sistemas distribuidos, como por ejemplo para sincronizar tareas, realizar tareas cooperativas.

En un principio, el problema de la exclusión mutua consistía en garantizar que un único proceso utilizara la sección crítica en un instante de tiempo. Surgen extensiones al problema tradicional, una de ellas es el problema de k-exclusión [10], donde k proceso están en la sección crítica a la vez. Otra extensión es la exclusión mutua para grupos, en la literatura, hay gran variedad de trabajos relacionados con el tema [1], [2], [3], [4], [6]; en el cual un conjunto de procesos que están asociados para realizar una tarea o procesos no conflictivos pueden estar al mismo tiempo en la sección crítica. Al igual que en los algoritmos tradicionales existen algoritmos basados en memoria compartida, en pasajes de mensajes: a través de la utilización de mensajes *multicast* o quorums o basados en *token* (en una red de anillo).

Este trabajo presenta un algoritmo de exclusión mutua para grupos basados en quorum, utilizando la forma de armado de quorums del algoritmo de Maekawa [7]. La idea del algoritmo es que garantice la *Exclusión Mutua* y a su vez maximice la *Concurrencia*, esto es, permitir que los procesos no conflictivos compartan un recurso para incrementar la performance del sistema.

2. Fundamentos del Algoritmo

El algoritmo de Maekawa [7] 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 en su algoritmo elige k = $\sqrt{N}$, con esta elección garantiza las propiedades mencionadas.

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 *lockear* 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 mandarían 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 de inanición.

3. Algoritmo de Exclusión Mutua para Grupos

Se considera que se tienen P_1, ..., P_M procesos y G_1, ..., G_N grupos. Para el algoritmo que se presenta se supone que la red es confiable y no necesita reconocimiento.

El modelo está formado por dos componentes que interactúan que son los *procesos* y los *grupos*. El *proceso* es un componente *activo*, ya que es el que selecciona el grupo para acceder al recurso, en la Figura 1, se observan los estados del mismo. Está en RESTO cuando realiza tareas que no involucran la vinculación con algún grupo. Cuando quiere trabajar en un grupo pasa al estado de ENTRADA hasta que puede acceder a la SC[1] por un tiempo determinado finito y luego pasa al estado de SALIDA y vuelve a RESTO y así sucesivamente.

El *grupo* es un componente *pasivo*, ya que está esperando que un proceso lo seleccione para participar del mismo, cuando esto ocurre entonces empieza a competir por el recurso, en la Figura 2, se observan los estados del mismo. Inicialmente el componente *grupo* está en el estado INACTIVO esperando que algún proceso lo seleccione y pasa al estado COMPITIENDO en el cual solicita los *locks* necesarios para

[1]SC: Sección Crítica

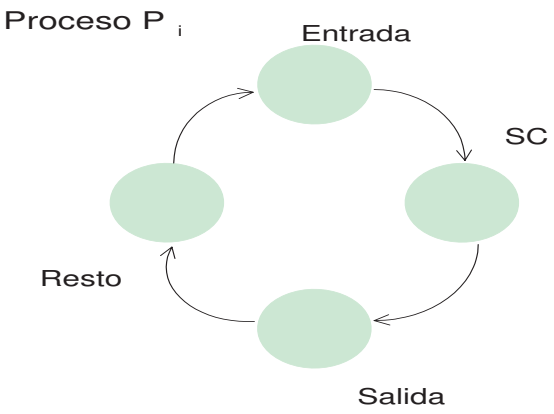

Figura 1: Estados de un Proceso

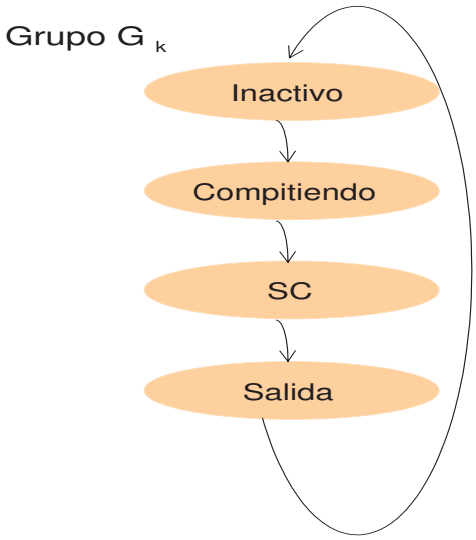

Figura 2: Estados de un Grupo

poder utilizar el recurso y pasar al estado de SC hasta que todos los procesos que lo eligieron terminen su trabajo y luego pasa al estado SALIDA en el cual libera los *locks* y vuelve al estado INACTIVO y así sucesivamente.

Componente Proceso

Cada proceso i, $1 \leq i \leq$ M, realiza los siguientes pasos: selecciona el grupo en el cual va a trabajar, le envía un mensaje de solicitud de requerimiento y espera hasta que su requerimiento es 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; podría vincularse con diferentes grupos a través del tiempo. Se considera que un proceso está vinculado a un grupo un tiempo finito, esto es, para garantizar la propiedad de libre de inanición.

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.

Pasos del Componente Proceso

Componente Grupo

El componente 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 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 componente *grupo* puede recibir un conjunto de mensajes y de acuerdo al requerimiento su estado actual puede modificarse y además generar nuevos mensajes a otros componentes. Teniendo en cuenta el origen del mensaje se lo puede clasificar proveniente de un componente *proceso* o de un componente *grupo*. Los mensajes afectan el estado del *grupo*, 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).

```
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, prio-
    ri)
            Sino
            AgregarListaGrupos(LG, G_l, priori)
```

Pasos del Componente 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.

♯ 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. Sino tiene disponible el *lock* pueden ocurrir dos casos diferentes: (a) La prioridad del mensaje recibido es menor que la prioridad del mensaje cedido el *lock* entonces 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

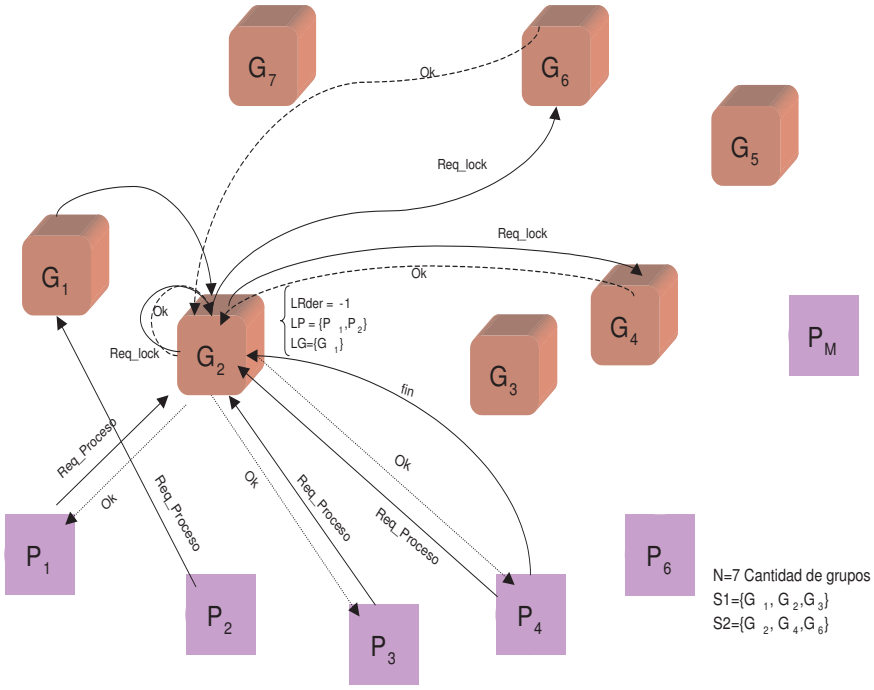

Figura 3: Ejemplo

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*

Ejemplo

En la Figura 3, 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)), 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, 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 3, también se observa que el proceso P_4 ha finalizado su trabajo y 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.

4. 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.

Supongamos que esto es así, 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, esto es una contradicción. Por lo tanto, el algoritmo garantiza la exclusión mutua.

La espera para ingresar a la sección crítica está limitada ya que cada solicitud de *lock* tiene una prioridad y esta es única, 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.

5. Complejidad del Algoritmo

La complejidad de los algoritmos se pueden medir de acuerdo a diferentes consideraciones, como puede ser el número de operaciones de memoria compartida requeridas para ingresar a la sección crítica, o el intervalo de tiempo entre entradas a la sección crítica, o 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 $k = \sqrt{N}$. La cantidad 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 requiere *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 deban 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 requerirían *3l + 5(k-1)* mensajes.

6. Conclusiones

En este trabajo se presenta un algoritmo para exclusión mutua de grupos de procesos, basado en pasajes de mensajes utilizando quorums para acceder a la sección crítica. El algoritmo permite que un conjunto de procesos cooperativos compartan el acceso, esto es, maximizando la concurrencia. La complejidad para *l* procesos que requieran trabajar en el mismo grupo, en el caso que no se requiera devolver el *lock*, es de $3l$+3(k-1) mensajes. Se considera que cada proceso permanece en la sección crítica un tiempo limitado, de esta manera se evita la inanición. El algoritmo funciona en un ambiente que no tenga fallas de comunicaciones entre los componentes proceso y grupo. Puede soportar alguna falla en el caso de grupos sino forman parte del quorum.

Referencias

[1] Yuh-Jzer Joung *Asynchronous Group Mutual Exclusion (extended abstract).* In Proc. 17 th. ACM PODC, Junio 1998.

[2] Kue-Pin Wu, Yuh-Jzer Joung *Asynchronous Group Mutual Exclusion in Ring Networks.* IEE Proc. Computers and Digital Techniques.

[3] Yuh-Jzer Joung *The Congenial Talking Philosophers Problem in Computer Networks (extended abstract).* In Proc. 13th International Symposium on Distributed Computing, 1999.

[4] Yuh-Jzer Joung *Quorum-Based Algorithms for Group Mutual Exclusion* IEEE Transactions on Parallel and Distributed Systems, Mayo 2003.

[5] Karina Cenci, Jorge Ardenghi *Exclusión Mutua para Coordinación de Sistemas Distribuidos.* CACIC 2001

[6] Karina Cenci, Jorge Ardenghi *Algoritmo para Coordinar Exclusión Mutua y Concurrencia de Grupos de Procesos* CACIC 2002

[7] Mamoru Maekawa *A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems.* ACM Transactions on Computer Systems, Mayo 1985

[8] Wai-Shing Luk, Tien-Tsin Wong *Two New Quorum Based Algorithms for Distributed Mutual Exclusion*

[9] Mitchell L. Nielsen, Masaaki Mizuno *Nondominates K-Coteries for Multiple Mutual Exclusion*

[10] K. Vidyasankar *A simple group mutual* l-*exclusion algorithm* Information Processing Letters, 2003.