# Sobre Algoritmos Distribuidos de Exclusión Mutua para n procesos

Karina M. Cenci[1] Jorge R. Ardenghi[2]

Departamento de Ciencias de la Computación
Universidad Nacional del Sur

## Resumen

Las aplicaciones distribuidas requieren compartir los recursos del sistema. Para hacer uso de los mismos, en algunos casos se necesita tener permiso para poder acceder y utilizarlos. El controlar el acceso a recursos que sólo pueden ser accedidos por un único proceso a la vez requiere de un protocolo de coordinación que garantice esta necesidad. Los algoritmos de exclusión mutua son los mecanismos utilizados para permitir el ingreso a la región que utiliza en forma exclusiva los recursos del sistema. Teniendo en cuenta las condiciones que debe presentar un algoritmo de este tipo se analizan diferentes protocolos y se presenta una alternativa del algoritmo del "Panadero" que satisface exclusión mutua. Los algoritmos están basados en el modelo de memoria compartida asincrónica con la utilización de variables de simple escritura y múltiple lectura.

Palabras Claves : Sistemas Distribuidos, Concurrencia, Exclusión Mutua

[1] e-mail: kmc@cs.uns.edu.ar
[2] e-mail: jra@cs.uns.edu.ar

## Introducción

Con el avance de la tecnología y el abaratamiento de los costos en los equipos surge la necesidad de diseñar aplicaciones que se encuentren dispersas geográficamente en diferentes puestos de trabajo. Los sistemas distribuidos son más complejos de diseñar, testear e implementar. Los modelos son comúnmente utilizados para especificar las propiedades de los mismos. 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 modelos para especificar las propiedades de acuerdo a las características de las mismas. Los más representativos son:

- Funciones Matemáticas
- Máquinas de Estado Finito
- Modelo Gráfico (p.e. Redes de Petri)

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

Para la construcción de este tipo de sistemas se utilizan diferentes tipos de herramientas y/o protocolos para compartir la información, la comunicación, la sincronización, etc.

El trabajo se centra en el control de acceso a los recursos (datos) en un ambiente de memoria compartida distribuida asincrónica.

- El sistema de memoria compartida asincrónica es modelada como una colección de procesos y variables compartidas con interacciones. Cada proceso *i* es una clase de máquina de estado finito de entrada/salida, con un conjunto de $estados_i$ y un subconjunto de estados de $inicio_i$ indicando el estado inicial.

- La exclusión mutua es el punto clave en el diseño de sistemas distribuidos. Es una forma utilizada para resolver conflictos que provocan procesos concurrentes al compartir recursos.

## Exclusión Mutua

El problema de la exclusión mutua, o la definición de las operaciones fundamentales que son posibles para resolver conflictos resultantes de varios procesos concurrentes compartiendo recursos, ha surgido como primer ejemplo de las dificultades asociadas con los sistemas distribuidos.

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 $U_1..U_n$, 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 (sección crítica) sean ejecutadas en forma estrictamente exclusiva.

Un usuario con acceso al recurso es modelado estando en la región 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 región *resto*. Para obtener la admisión a la región crítica, un usuario 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 usuario sigue un ciclo, desplazándose desde la región resto (**R**), a la región de entrada (**T**), luego a la región crítica (**C**) y por último a la región de salida (**E**), y luego vuelve a comenzar el ciclo en la región resto.

Una solución para la exclusión mutua en un ambiente distribuido es mediante la utilización de un coordinador. Cada proceso que solicita acceder a la región crítica, envía su solicitud al coordinador que mantiene una cola de solicitudes y brinda el permiso basado en determinadas reglas, como por ejemplo en estampilla de tiempo. Claramente se observa que el coordinador es un *cuello de botella*. Otra solución para resolver el problema es a través de algoritmos distribuidos utilizando el concepto de memoria compartida asincrónica, que garanticen el acceso de un solo proceso a la región crítica. Los algoritmos presentados en este trabajo tienen esta característica.

## Consideraciones

Una autómata de I/O puede ser visto como una caja negra desde el punto de vista de un usuario. Lo que el usuario observa es solamente las trazas de la ejecución del autómata (o la ejecución imparcial).
Formalmente, una propiedad de traza P consiste de lo siguiente:

*Sig(P),* a signature que no contiene acciones internas
*Traces(p),* un conjunto de (finitas o infinitas) secuencias de acciones ack(sig(P)[3].

Esto es, una propiedad de traza especifica tanto una interface externa y un conjunto (en otras palabras una propiedad) de secuencias observadas en la interface.

Dos importantes tipos especiales de propiedades de traza son la propiedad de *seguridad* y la propiedad de *vivacidad*.

***Propiedad de Seguridad***: decimos que una propiedad de traza P es una propiedad de traza segura, o una propiedad de seguridad, con tal que P satisfaga las siguientes condiciones:

1.- *traces(P)* no es vacía

[3] En lo sucesivo aparecerá en forma abreviada como equivalencia acts(P)≡ acts(sig(P))

2.- *traces(P)* es prefijo cerrado (prefix-closed), esto es, si ß ∈ *traces(P)* y ß' es un finito prefijo de ß, luego ß ∈ *traces(P)*

3.- *traces(P)* es frontera cerrada (limit-closed), esto es, si ß1, ß2, ... es una secuencia infinita de secuencias finitas en *traces(P)*, y para cada i, ßi es un prefijo ßi+1, luego la única secuencia ß que es límite de ßi bajo la sucesiva extensión ordenada está también en *traces(P)*.

La propiedad de seguridad es a menudo interpretada como diciendo que alguna particular cosa mala/errónea nunca ocurre. Si presumimos que algo malo/erróneo ocurre en una traza, esto ocurre como un resultado de algún evento particular en la traza; por lo tanto, frontera cerrada (limit-closure) es una condición razonable para incluir en la definición. También, si nada malo/erróneo ocurre en la traza, por ende nada malo/erróneo ocurre también en cualquier prefijo de la traza, por esto, prefijo cerrado (prefix-closure) es razonable. Finalmente, nada malo/erróneo puede ocurrir antes que cualquier evento ocurra, esto es, nada malo/erróneo ocurre en una secuencia vacía $\lambda$, por esto, no vacía (nonemptiness) es una condición razonable.

***Propiedad de Vivacidad*** : decimos que una propiedad de traza P es una propiedad de vivacidad de traza, o propiedad de vivacidad, con la condición que cada secuencia finita a través de *acts(P)* tiene alguna extensión en *traces(P)*.

La propiedad de vivacidad es a menudo informalmente entendida como diciendo que alguna particular cosa buena finalmente ocurre.

## Condiciones de un algoritmo correcto

Para una dada 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 sub-secuencia que describe la interacción entre $U_i$ y A está bien formada para i.
- **Exclusión Mutual**: 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 región crítica C.
- **Progreso**: en cualquier punto de una ejecución imparcial:

    1.- (progreso para la región 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 región de salida): si al menos un usuario está en E, en un punto posterior en el tiempo algún usuario entre a R.

Con la condición de progreso se asume que la ejecución del sistema es imparcial, 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 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 propiedad de seguridad, mientras que la condición de progreso es una propiedad de vivacidad.

La propiedad de traza (trace) es otra manera de presentar las condiciones de correctitud. Por ejemplo, se puede definir una propiedad de traza P, donde *sig(P)* tiene todas las acciones *try, crit, exit y rem* como de salida, y *traces(P)* es el conjunto de secuencias ß de estas acciones que satisfacen las siguientes condiciones:

1.- ß está bien formada para cada i.
2.- ß no contiene dos eventos críticos sin la intervención de un evento de salida
3.- En cualquier punto de ß,
(a) Si en algún proceso el último evento es de entrada y no hay proceso con último evento crítico, luego habrá posteriormente un evento crítico
(b) Si en algún proceso el último evento es de salida, luego habrá posteriormente un evento para el resto.

Un algoritmo regular/bueno de exclusión mutua debería satisfacer:

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

*Libre de inanición* : cada solicitud de acceso a la región crítica debería en un período considerable ser garantizado.

*Imparcialidad* : las solicitudes deberían ser garantizadas basadas en ciertas reglas de imparcialidad. Típicamente, está basado en el tiempo de solicitud determinado por relojes lógicos.

## El algoritmo del “Panadero”

El algoritmo trabaja en forma similar a lo que se realiza en una panadería, donde los cliente obtienen un ticket cuando ingresan y son atendidos según el orden de llegada que corresponde con el número de ticket.

El algoritmo del *Panadero* solamente utiliza registros compartidos de simple escritura / múltiple lectura. En el algoritmo del *Panadero*, la primera parte de la región de entrada, hasta el punto que el proceso i setea *choosing(i)*:=0, es designado como un portal de entrada (*doorway*). Mientras está en el portal, el proceso i selecciona un número mayor que todos los números leídos de los otros procesos. Lee los otros números de procesos uno a la vez, en cualquier orden, luego escribe su propio número. Mientras está leyendo y seleccionando su número, i se asegura que *choosing(i)* =1, como una señal a los otros procesos.

Es posible que dos procesos se encuentren en el portal (*doorway*) al mismo tiempo, el cual puede causar que los dos procesos seleccionen el mismo número. Para seleccionar un solo proceso que entre

en la región crítica se compara la identificación de los nodos y el que tiene un valor menor es el que entra primero a la sección crítica.

En un formato tradicional el algoritmo es el siguiente:

```
Variables Compartidas

Para cada i, 1≤ i ≤ n
    Choosing (i) ∈ {0,1}, inicialmente 0, escrito por i y leído por todo j≠i
    Number (i) ∈ N, inicialmente 0, escrito por i y leído por todo j≠i

Proceso i

** región resto**

try i
choosing(i) := 1
number(i) := 1 + max j≠i number(j)
chossing(i):= 0
para j≠i hacer
    waitfor choosing(j) = 0
    waitfor number(j) = 0 or (number(i),i) < (number(j), j)

crit i

** región crítica **

exit i
number(i) := 0

rem i
```

El algoritmo satisface exclusión mutua y garantiza las propiedades de progreso y está libre de interbloqueo. Además garantiza una condición de imparcialidad de alto nivel denominada FIFO (First In First Out), en realidad esta condición se alcanza después de haber pasado por el portal.

El algoritmo presenta un alto grado de determinismo incluyendo controles que en algunos casos son innecesarios. Se presenta una adaptación de este algoritmo disminuyendo el nivel de determinismo en las condiciones de testeo.

## Adaptación del Algoritmo

(Una aproximación con un mayor grado de libertad).

La idea central en esta variación es lograr un algoritmo que presente un mayor grado de libertad en la ejecución del mismo y tratar de disminuir el tiempo de espera en la región de entrada del protocolo a la región crítica.

Considerando que la asignación del *number(i)* es realizado en una única operación, entonces no es necesario tener que controlar si *choosing(j) = 0* si el *number(j)≠0*. Se presenta en un formato tradicional de código la modificación planteada.

Variables Compartidas

Para cada i, 1≤ i ≤ n
    Choosing (i) ∈ {0,1}, inicialmente 0, escrito por i y leído por todo j≠i
    Number (i) ∈ N, inicialmente 0, escrito por i y leído por todo j≠i

Proceso i

** región resto**

*try i*
choosing(i) := 1
number(i) := 1 + max $_{j≠i}$ number(j)
chossing(i):= 0
∀ j≠i  (waitfor (choosing(j) = 0 and number(j) = 0) or ( (number(j) > 0 and (number(j),i) < (number(j),j)))

*crit i*

** región crítica **

*exit i*
number(i) := 0

*rem i*

Con el modelo tradicional del código presentado no es sencillo poder probar la correctitud y buena formación del misma ya que puede presentar ambigüedades. Este algoritmo se puede traducir a un modelo formal. Para realizar el pasaje al modelo de máquina de estado finito hay que especificar cómo se va a controlar la condición del *Waitfor,* ya que presenta condiciones en la verificación de *choosing(j)* y de *number(j)*. En este caso primero se verificará la variable *choosing* y luego verifica la variable *number*. Además se deberán incorporar estados, variables temporales y un contador de programa. El algoritmo presenta una notación precondición-efecto. En este modelo la región R corresponde al resto; T corresponde a set-choosing, set-number, reset-choosing, check-condición1, check-condición2 y leave-try; C corresponde a crit, y E corresponde a reset y leave-exit. El modelo es el siguiente:

Variables Compartidas

Para cada i, 1≤ i ≤ n
        Choosing (i) ∈ {0,1}, inicialmente 0
        Number (i) ∈ N, inicialmente 0

Acciones de i

| | |
|---|---|
| *Entrada* | *Internas* |
| $Try_i$ | $set\text{-}choosing_i$ |
| $Exit_i$ | $set\text{-}number_i$ |
| *Salida* | $reset\text{-}choosing_i$ |
| $Crit_i$ | $check\text{-}condición1(j)_i, \forall\ j \neq i$ |
| $Exit_i$ | $check\text{-}condición2(j)_i, \forall\ j \neq i$ |

Estados de i

Estado $\in$ {rem, set-choosing, set-number, reset-choosing, check-condición1, check-condición2, leave-try, crit, reset, leave-exit}, inicialmente rem.

S, un conjunto de índices de procesos, inicialmente $\varnothing$

Transiciones de i

*$Try_i$*

Efecto:
Estado:= set-choosing

*$Set\text{-}choosing_i$*

Precondición:
Estado = set-choosing

Efecto:
Choosing(i) := 1
Estado := set-number // Esta acción, setea choosing(i), para especificar al resto de los procesos que //se está el portal de entrada, en la búsqueda del ticket.

*$Set\text{-}number_i$*

Precondición :
Estado = set-number

Efecto:
Number(i) := 1+ max $_{j \neq i}$ number(j)
Estado := reset-choosing // En esta acción se obtiene el número de entrada.

*$Reset\text{-}choosing_i$*

Precondición:
Estado = reset-choosing

Efecto:
Choosing(i) := 0

S := {i}
Estado := check-condición1 // Al setear nuevamente choosing(i), se especifica que se terminó la //búsqueda y que comienza su espera para entrar en la región crítica.

*Check-condición1(j)*$_i$

Precondición:
Estado = check-condición1
$j \notin S$

Efecto:
if choosing(j) = 0 and number(j) = 0 then
S := S $\cup$ {j}
If |S| = n then
Estado := leave-try
else
estado := check-condición2 // S inicialmente está vacío. Cada vez que se analiza el estado //de un proceso j y se comprueba que no está en la región de //entrada del protocolo se lo incorpora en el conjunto. Si la //cantidad de elementos del conjunto es n entonces el proceso //i está en condiciones de ingresar en la región crítica.

*Check-condición2(j)*$_i$

Precondición:
estado = check-condición2
$j \notin S$

Efecto:
if (number(i) < number(j)) or ((number(i)=number(j)) and (i<j)) then
S := S $\cup$ {j}
If |S| = n then
Estado := leave-try
Else
Estado := check-condición1 // Cada vez que se analiza el estado de un proceso j y se //comprueba que está en la región de entrada del protocolo se //verifica si el número de ticket es mayor que el //correspondiente al proceso i entonces se lo incorpora en el //conjunto S; ó el número de ticket es el mismo y el orden de j //es superior a i entonces también se lo incorpora. Si la //cantidad de elementos del conjunto es n entonces el proceso //i está en condiciones de ingresar en la región crítica.

*Crit*$_i$

Precondición:
Estado = leave-try
Efecto:

Estado := crit // Esta acción especifica que el proceso i tiene permiso para acceder a la //sección crítica.

*Exit*$_i$

Efecto:
Estado := reset

*Reset*$_i$

Precondición:
Estado = reset
Efecto:
Number(i) := 0
S := ∅
Estado := leave-exit // El proceso i especifica que libera el uso de la sección crítica.

*Rem*$_i$

Precondición:
Estado = leave-exit
Efecto:
Estado := rem

El algoritmo está bien formado. Para la exclusión mutua, la idea está en que accede solamente un proceso al área de sección crítica, el que tiene el menor valor en *number* o si lo comparte entonces tiene el menor número de identificación.

Para que el proceso i sea el ganador y pueda acceder a la región crítica se debe verificar lo siguiente:

*number*(i) < *number*(j) i≠j ó
*number*(i) = *number*(j) y i<j i≠j ó
*number*(j) = 0 i≠j, en los casos que el proceso j no se encuentre en la región de entrada del protocolo

Supongamos que se alcanza en un estado, que dos procesos, i y j, están ambos en la sección crítica.
Si esto ocurre, entonces
number(i) <= number(j) e i<j,
y al mismo tiempo debería ocurrir que
number(j) <= number(i) y j<i.

Partiendo de la hipótesis inicial se alcanza una contradicción, por lo tanto un solo proceso puede estar en la sección crítica.

## Conclusiones

El crecimiento de las aplicaciones distribuidas requiere que se estudien y analicen los protocolos existentes para modificarlos o elaborar nuevos protocolos de acuerdo a las necesidades. Este tipo de sistemas comparten los recursos y necesitan tener acceso a los mismos. Dependiendo de los recursos y de las operaciones a utilizar pueden tener libre acceso y sin ninguna restricción hacer libre uso del mismo. Pero hay un grupo de recursos que sólo pueden ser accedidos por un usuario a la vez, y para lograr esto es necesario contar con un protocolo que lo garantice. El trabajo se basó en la problemática de la exclusión mutua. Se presentan las condiciones que debe tener un buen algoritmo, libre de interbloqueo, libre de inanición e imparcialidad. Teniendo en cuenta las características que debe presentar se analizan varios protocolos, principalmente el algoritmo del "panadero"(Lamport), que se caracteriza por la utilización de variables compartidas de simple escritura y múltiple lectura. Partiendo de esta solución para resolver el problema de exclusión mutua se presenta una alternativa de este protocolo utilizando un modelo formal de máquina de estado finito. La ventaja del modelo es que presenta menor grado de determinismo en la ejecución de la región de entrada a la sección crítica, evitando en algunos casos verificar la variable *choosing* para los procesos que compiten en el acceso a la región crítica. El algoritmo se basa en el modelo de memoria compartida asincrónica para su implementación brindando un alto grado de abstracción.

## Bibliografía


[1] Jie Wu. *Distributed System Design*, 1999.

[2] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, Junio 1981.

[3] Nancy A Lynch. *Distributed Algorithms,* 1997.

[4] Sape Mullender. *Distributed Systems*, 2da. Ed. 1993.

[5] Abraham Silberschatz y Peter Galvin. *Operating System Concepts,* 5ta. Ed. 1998.

[6] Leslie Lamport. Provinding the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, Marzo 1977.

[7] Michael Raynal. *Algorithms for Mutual Exclusion*. MIT Press, Cambridge, 1986.

[8] M. Ben Ari. *Principles of Concurrent Programming*. Prentice Hall, Englewood Cliffs, 1982.

[9] Clyde P. Krustal, Larry Rudolph y Marc Snir. Efficient synchronization on multiprocessors with shared memory. In *Proceedings of the Fifth Annual Symposium on Principles of Distributed Computing*, Agosto 1986.

[10] Bowen Alpern y Fred B. Schneider. Defining liveness. *Information Processing Letters*, Octubre 1985.

[11] Cenci, Karina y Ardenghi, Jorge. Exclusión Mutua en la Implementación Memoria Compartida Asincrónica. VI Congreso Internacional de Ingeniería Informática, ICIE 2000, 26 al 28 de Abril 2000 – Facultad de Ingeniería – UBA.

[12] L. Lamport, A New Solution of Dijkstra's Concurrent Programming Problem. In *Communications of the ACM,* Agosto 1974.