El problema de la **exclusión** mutua

1 Introducción histórica al problema

Aunque siempre han existido soluciones de bajo nivel para resolver este problema, p.e. los cerrojos; sin embargo, se han venido estudiando desde 1965 una serie de soluciones-software para resolverlo. Dichas soluciones pretenden ser independientes de las instrucciones de una máquina concreta y permiten asegurar que los procesos cumplen todas las propiedades exigidas a un programa concurrente. Se intenta conseguir, fundamentalmente, soluciones que proporcionen un acceso equitativo de los procesos a la sección crítica. Sólo vamos a considerar soluciones al problema de la exclusión mutua que utilicen las instrucciones básicas de lectura o escritura del valor de una variable.

Los algoritmos que se van a introducir a continucaión representan una muestra cronológica de las propuestas de solución del problema de la exclusión mutua y tienen un indudable valor pedagógico para la comprensión de conceptos básicos de la Programación Concurrente.

En la actualidad vuelve a tener interés la investigación de estos algoritmos, ya que con las nuevas arquitecturas distribuidas es necesario encontrar versiones descentralizadas de los algoritmos clásicos que necesitan de un sistema con memoria común para su programación.

Los algoritmos para resolver el problema de la exclusión mutua se dividen en:

1. *Centralizados:* utilizan variables compartidas entre los procesos que expresan el estado de los procesos (se suele acceder a estas variables en los protocolos de adquisición y restitución, que ejecutan los procesos antes y después de la sección crítica, respectivamente)
2. *Distribuidos:* no utilizan variables compartidas entre los procesos. Suelen utilizar instrucciones de paso de mensajes.

2 Solución al problema con espera ocupada

La espera activa consiste en que los procesos esperan a la entrada de la sección crítica hasta que su continuación sea segura.

Este tipo de implementación utiliza recursos (tiempo del procesador en sistemas monoprocesador y ciclos de memoria en sistemas multiprocesador) sin realizar ningún avance en el

proceso que espera. Se puede considerar una solución aceptable, siempre que el sistema no tenga muchos procesos.

Se podría pensar en una solución al problema de la exclusión mutua que utiliza una variable lógica `mutex` que cambia al valor cierto cuando un proceso está en la sección crítica y a falso cuando no hay ningún proceso en sección crítica:

```
var mutex: boolean:= false;

procedure ProtocoloAdquisicion;
  begin
    while mutex do
        nothing;
    enddo;
    mutex:= true;
  end;

procedure ProtocoloRestitucion;
  begin
    mutex:= false;
  end;
       P1                    P2
     --------              ---------
  ProtocoloAdquisicion;     ProtocoloAdquisicion;
     S.C.                   S.C.
  ProtocoloRestitucion;     ProtocoloRestitucion;
```

La solución anterior no cumple la propiedad de seguridad. Ya que los 2 procesos pueden entrar a ejecutar simultáneamente el protocolo de adquisición, pasar ambos la condición del bucle y entrar en sección crítica.

3 Condiciones que ha de verificar una solución correcta al problema

El problema de la exclusión mutua para 2 procesos fue definitivamente resuelto por Dekker en 1965. Como al principio es un algoritmo difícil de entender, Dijkstra estableció un método para obtenerlo y las condiciones que cualquier otra solución al problema de la exclusión mutua debería cumplir para ser aceptada como tal. Las condiciones de Dijkstra son:

1. *No hacer ninguna suposición acerca de las instrucciones o número de procesos soportados por la máquina.* Sólo podemos utilizar las instrucciones básicas, tales como leer, escribir o comprobar una posición de memoria para resolver el problema. Además, supondremos que son ejecutadas atómicamente. En caso de que 2 instrucciones fueran susceptibles de ser ejecutadas simultáneamente, el resultado es *no-determista*; es decir, es equivalente a la ejecución secuencial de estas en un orden no conocido.

2. *No hacer ninguna suposición acerca de la velocidad de ejecución de los procesos, excepto que no es cero.*

3. *Cuando un proceso está en sección no-crítica* no puede impedir a otro que entre en ella.

4. *La sección crítica será alcanzada por alguno de los procesos que intentan entrar.* Esta condición asegura siempre la alcanzabilidad de la sección crítica por parte de los procesos. Excluye la posibilidad de que en una ejecución del algoritmo se llegara a un interbloqueo.

4 Método de refinamiento sucesivo

Además de las condiciones que ha de cumplir un algoritmo para ser considerado una solución al problema de la exclusión mutua, Dijkstra propone un método para obtener el algoritmo de Dekker en 4 *modificaciones*. Se parte de una *solución* inicial basada en que los procesos alternen su entrada a la sección crítica según el valor de una variable global a la que se asigna el *turno*. Cada una de las *soluciones* encontradas se considera una *etapa* en el proceso de encontrar la solución, que se obtendrá en la quinta etapa.

Primera etapa:
Se utiliza una variable `turno` que contiene el identificador del proceso que puede entrar en sección crítica. Dicha variable puede valer 1 ó 2, inicialmente suponemos que su valor es 1.

```
   P1                      P2
---------               ---------
while true do begin      while true do begin
  Resto ;                  Resto;
```

```
  while turno <> 1 do          while turno <> 2 do
     nothing;                     nothing;
  enddo;                       enddo;
  S.C.                         S.C.
  turno:= 2;                   turno:= 1;
 end                          end
enddo;                       enddo;
```

La solución garantiza el acceso en exclusión mutua de los procesos, ésto es, la solución es segura. No garantiza, sin embargo, la tercera condición, ya que los procesos sólo pueden entrar en la sección crítica alternativamente. Podría pensarse que la solución para evitar la alternancia forzosa en el acceso a la sección crítica consiste, simplemente, en asignar `turno:= i` antes del del bucle de espera activa de cada proceso Pi; pero, es fácil comprobar que la solución dejaría de ser segura en ese caso.

Segunda etapa:
La alternancia estricta en el acceso a la sección crítica que se producía en la primera etapa era debida a que, para decidir qué proceso entraba en sección crítica, se tenía que almacenar información global del estado del programa en una variable `turno` que sólo la cambiaba un proceso al salir de esta. Para evitarlo, la idea ahora es asociar con cada proceso información una variable *clave* que indique si dicho proceso está en sección crítica o no. Cada uno de los procesos lee la clave del otro pero no puede modificarla.

```
Inicialmente: c1=c2= 1;
      P1                          P2
   ---------                   ---------
   while true do begin         while true do begin
     Resto ;                     Resto;
     while c2=0 do               while c1=0 do
       nothing;                    nothing;
     enddo;                      enddo;
     c1:= 0;                     c2:= 0;
       S.C.                        S.C.
     c1:= 1;                     c2:= 1;
     end                         end
   enddo;                      enddo;
```

En este caso falla la propiedad de seguridad. Ya que si P1 y P2 se ejecutan a la misma velocidad, comprueban que el otro proceso no está en sección crítica y ambos entran en sección crítica. Cuando asignan sus claves a 0 ya es tarde, pues ya han pasado el bucle de espera ocupada. Como esta solución sólo funcionará dependiendo de la velocidad de los procesos, se dice que no cumple la segunda condición de Dijkstra.

Tercera etapa:
El problema de la solución anterior es que un proceso puede comprobar el estado del otro antes de que este lo modifique. Ya que la salida de un proceso de su bucle de espera activa y la modificación de su clave a cero, no se realiza atómicamente.

La solución que ahora se propone consiste en adelantar la sentencia de asignación de la clave del proceso antes del bucle de espera activa. De esta forma, sería imposible que un proceso pase el bucle de espera activa con un valor de su clave distinto de cero.

```
Inicialmente: c1=c2= 1;
       P1                        P2
    ---------                 ---------
    while true do begin        while true do begin
      Resto ;                     Resto;
      c1:= 0;                     c2:= 0;
      while c2=0 do               while c1=0 do
        nothing;                    nothing;
      enddo;                      enddo;
          S.C.                        S.C.
      c1:= 1;                     c2:= 1;
      end                         end
     enddo;                      enddo;
```

Esta solución es segura, pero no se puede considerar correcta, ya que si ambos tienen la misma velocidad se produce interbloqueo, pues ambos procesos pueden cambiar sus claves a cero y detenerse en los bucles de espera ocupada. Por lo tanto no se cumple la cuarta condición.

Cuarta etapa:
El problema de la tercera etapa consiste en que cuando un proceso modifica el valor de su clave no sabe si el otro proceso está haciendo lo mismo concurrentemente con él.

La solución ahora sería que un proceso asigne su clave a cero para así indicar que quiere entrar en la sección crítica. Si inmediatamente después comprueba que el otro proceso tiene también su clave a cero, entonces vuelve a cambiar su clave a uno.

```
Inicialmente: c1=c2= 1;
       P1                        P2
    ---------                 ---------
    while true do begin        while true do begin
      Resto ;                     Resto;
      c1:= 0;                     c2:= 0;
      while c2=0 do               while c1=0 do
       begin                       begin
         c1:= 1;                     c2:= 1;
         while c2=0 do               while c1=0 do
           nothing;                    nothing;
         enddo;                      enddo;
         c1:=0;                      c2:= 0;
       end                         end
       enddo;                      enddo;
         S.C.                        S.C.
       c1:= 1;                     c2:= 1;
    end                           end;
   enddo;                        enddo;
```

Si ambos procesos se ejecutasen a la misma velocidad se podría seguir produciendo interbloqueo, aunque ahora es más improbable que se produzca que en la tercera etapa. Por lo tanto,con esta solución no se cumplen ni la segunda, ni la cuarta condición.

Las variables c1 y c2 pueden interpretarse como unas variables de estado de los procesos, que informan si el proceso está intentando entrar en sección crítica o no. La conclusión a la que se llega después de los intentos anteriores es que el observar el estado de los procesos no es suficiente para dar una solución correcta al problema de la exclusión mutua. Es necesario, por tanto, utilizar un orden previamente establecido para entrar en sección crítica si hay conflicto entre los procesos. Dicho orden lo establece una variable `turno`.

5 Algoritmo de Dekker

El algoritmo de Dekker se basa en la primera y cuarta etapas del método de Dijkstra. La primera etapa era segura pero presentaba el problema de que la variable `turno` era global y no podía ser cambiada por un proceso antes de entrar en su sección, aunque el otro proceso estuviera pasivo, lo que llevaba a la alternacia estricta en el acceso a la sección crítica por parte de los procesos. Como consecuencia de esto no se cumplía la tercera condición. Por otra parte, la cuarta solución con claves separadas no cumple la cuarta condición.

En el algoritmo de Dekker un proceso que intenta entrar en sección crítica asigna su clave a cero. Si encuentra que la clave del otro es también cero, entonces consulta el valor de `turno`; si posee el turno, entonces insiste y comprueba periódicamente la clave del otro proceso. Eventualmente el otro proceso le cede el turno cambiando su clave a 1 y consigue entrar en sección crítica finalmente.

```
     P1                          P2
---------                   ---------
while true do begin         while true do begin
  Resto ;                      Resto;
  c1:= 0;                      c2:= 0;
  while c2=0 do                while c1=0 do
   if turno= 2 then             if turno= 1 then
     begin                        begin
       c1:= 1;                      c2:= 1;
       while turno= 2 do            while turno=1 do
          nothing;                     nothing;
       enddo;                       enddo;
       c1:=0;                        c2:= 0;
     end                          end
    endif                        endif
   enddo;                       enddo;
     S.C.                         S.C.
   turno:= 2;                   turno:= 1;
   c1:= 1;                      c2:= 1;
 end;                         end;
enddo;                       enddo;
```

5.1 Verificación de las propiedades de seguridad

Exclusión mutua:
El proceso `Pi` entra en sección crítica sólo si `cj=1`. La clave de un proceso sólo la puede modificar el propio proceso. El proceso `Pi` comprueba la clave del otro proceso `cj` sólo después de asignar su clave `ci= 0`. Luego, cuando el proceso `Pi` entra, y se mantiene, en sección crítica se cumple que `ci= 0` $\wedge$ `cj=1`. Los valores de estas variables indican que sólo un proceso puede acceder a sección crítica cada vez.

Alcanzabilidad de la sección crítica:
Esquema de la demostración:

1. Si suponemos que un sólo proceso `Pi` intenta entrar en sección crítica, encontrará la clave del otro proceso con valor `cj= 1`; por tanto, el proceso `Pi` puede entrar.
2. Sin embargo, si los procesos `Pi`, `Pj` intentan entrar en sección crítica y el `turno=i`, entonces se tendrá el siguiente escenario:
   (a) `Pj` encuentra la clave `ci= 1`, entonces `Pj` entra en sección crítica;
   (b) `Pj` encuentra la clave `ci= 0` y `turno=i`, entonces se detiene en su bucle interno y pone `cj= 1`;
   `Pi` encuentra la clave `cj= 0`, se detiene en el bucle externo hasta que `cj= 1`;
   por último `Pi` entra en sección crítica.

5.2 Posible inanición de un proceso

En el caso de que `P2`, por ejemplo, fuera un proceso muy rápido y repetitivo, entonces podría estar entrando contínuamente en sección crítica e impidiendo al proceso `P1` el hacerlo. El escenario consistiría en que `P1` sale de su bucle interior, pero nunca puede escribir el valor de su clave c1, ya que P1 siempre la está leyendo y se lo impide.

5.3 Equidad

La equidad del algoritmo del Dekker dependerá de la equidad del hardware. Si existen peticiones de acceso simultáneo a una misma posición de memoria por parte de 2 procesos. Uno pide acceso en lectura y otro en escritura, si el hardware resuelve siempre el conflicto atendiendo primero al proceso que solicita la lectura, entonces el algoritmo de Dekker no sería equitativo.

6 Algoritmo de Dijkstra para la generalización a n procesos

Dijkstra(1965) encontró una solución generalizada al caso de n procesos para el algoritmo de Dekker. Tal como ocurre en este último, el código es simétrico para todos los procesos. Se propone a continuación una forma estructurada de expresar el algoritmo en la que se utilizan variables clave que pueden tomar 3 valores.

Las variables compartidas entre los n procesos del algoritmo son las siguientes:

```
var c: array[0..n-1] of (pasivo, solicitando, en_SC);
turno:0..n-1;
```

Los elementos de `c` son inicializados a *pasivo* y `turno` toma algún valor arbitrario. Cada proceso P_i define una variable local entera j.

```
repeat
  c[i]:= solicitando;

  while turno <> i do
   if c[turno]= pasivo
      then turno:= i
   endif;
  enddo;

  c[i]:= en-SC;
  j:= 0;

  while(j<n) and(j=i or c[j] <> en-SC) do
    j:= j+1;
  enddo;
until j>= n;
<<seccion critica>>
c[i]:= pasivo;
```

6.1 Propiedades de seguridad

Para demostrar la propiedad de exclusión mutua, nótese que P_i sólo puede entrar en su sección crítica cuando $c[j] \neq en - SC$ para todos los valores de $j \neq i$. Como P_i es el único proceso que puede realizar la asignación $c[i] := en - SC$ y dado que sólo comprueba el valor de $c[j]$ después de realizar dicha asignación, se sigue que no puede entrar en sección crítica más de un proceso, en ningún momento.

Para demostrar que ninguna ejecución de los procesos puede llevar al algoritmo a una situación de interbloqueo, comenzaremos con la observación de que cuando P_i intenta entrar

en su sección crítica tenemos que $c[i] \neq pasivo$. Además, $c[i] = en - SC$ no implica que $turno = i$; de hecho, varios procesos pueden encontrar $c[turno] = pasivo$ y dado que *turno* es una variable compartida, contendrá el número del último proceso que le haya asignado un valor, por ejemplo, el valor i. Los procesos P_j que comprueben posteriormente $c[turno]$ realizarán iteraciones y sus variables $c[j]$ mantendrán continuamente el valor *solicitando*. Consideremos ahora a los procesos P_l, ... P_i, ... P_m, tales que su $c[i] = en - SC$, y supongamos que $turno = k$, con $l \leq k \leq m$, entonces P_k entrará en su sección crítica en un tiempo finito. Esto es así, ya que los otros procesos P_i dejarán el segundo bucle interno con $j < n$ y volverán al primer bucle después de haber llevado a cabo la asignación $c[i] :=$ *solicitando*. Posteriormente, P_k podrá volver a realizar iteraciones en el bucle externo o en el segundo bucle (pero no en el primero, ya que $turno = k$) hasta que los otros procesos hayan modificado el valor de $c[i]$; lo que necesariamente harán, si se tiene en cuenta que los procesos avanzan, esto es, que se ejecutan a una velocidad distinta de cero. Cuando los demás procesos hayan modificado los valores de sus claves, P_k se convertirá entonces en el único proceso para el cual $c[i] = en - SC$ y podrá entrar en su sección crítica.

6.2 Inanición de los procesos

El algoritmo anterior garantiza la exclusión mutua y evita el interbloqueo; aunque no evita el peligro de *inanición*. De hecho, si un número determinado de procesos están pidiendo constantemente entrar en sus secciones críticas, no hay forma de impedir que uno de estos procesos sea siempre el último en modificar *turno* cuando compiten por la modificación de dicha variable.

7 Algoritmo de Knuth

La *generalización* del algoritmo de Dekker, conocida como algoritmo de Dijkstra, garantiza la exclusión mutua en el acceso a la sección crítica por parte de los procesos y también la alcanzabilidad de la sección crítica, pero no evita el riesgo de inanición de los procesos. Por lo tanto, no satisface las propiedades de vivacidad exigibles al algoritmo.

7.1 Mejoras respecto del algoritmo de Dijkstra

La primera solución al problema de la exclusión mutua para n-procesos fue propuesta en 1966 por Knuth. Además de las 4 condiciones básicas que ha de cumplir una solución aceptable al problema, se propone una quinta condición:

> 5) Un proceso que intente entrar en su *sección crítica* lo conseguirá en un tiempo finito

La quinta condición se formula a veces de distinta manera, debe existir un límite en el número de veces en que otros procesos pueden adelantarse para entrar en sección crítica a un determinado

proceso. Esta manera de expresar la quinta condición es mejor, ya que nos permite medir el tiempo máximo de espera de un proceso utilizando como unidad de referencia el número de turnos máximo que los otros procesos se le pueden adelantar.

7.2 Esquema general

Se van a utilizar las siguientes variables globales a los procesos:

```
var  c: array[0..n-1] OF (pasivo, solicitando, en_SC);
     turno: 0..n-1;
```

Las variables anteriores se inicializan a `pasivo` y `0`, respectivamente. La variable `j` en el siguiente código es local a los procesos. El proceso-i tiene el siguiente código:

```
repeat
 repeat
  E0: c[i]:= solicitando;
      j:= turno;
  E1: while j <> i do
       if c[j] <> pasivo then j:= turno
                         else j:= (j-1) MOD n
       endif;
      enddo;
  E2: c[i]:= en_SC;
      k:=0;
      while (k<n) and (k=i OR c[k] <> en_SC) do
        k:= k+1;
      enddo;
 until k>=n;
E3: turno:= i;
 <seccion critica>
 turno:= (i-1) MOD n;
 E4: c[i]:= pasivo;
 E5: <<resto instrucciones>>
until false;
```

El segundo bucle interno se puede incluir dentro de una función no atómica:

$$\texttt{testd(i)} = \texttt{TRUE} \leftrightarrow (\forall \texttt{j} \neq \texttt{i} : \texttt{c[j]} \neq \texttt{en\_SC} \wedge \texttt{c[i]} = \texttt{en\_SC})$$

7.3 Verificación de las propiedades del algoritmo

Exclusión mutua
El algoritmo anterior garantiza la exclusión mútua, ya que es fácil comprobar que para 2 procesos distintos `Pi` y `Pj` es imposible que las funciones `testd(i)` y `testd(j)` devuelvan el mismo resultado.

Alcanzabilidad de la sección crítica
La alcanzabilidad de la sección crítica se demuestra también. Si ningún proceso entra en la sección crítica entonces el valor de `turno` permanece constante y el primer proceso en el orden cíclico `(turno, turno-1, turno-2, ..., 1, 0, n-1, ... , turno+1)` que intente entrar en la sección crítica podrá hacerlo.

Equidad de los procesos
Para que un proceso `Pi` fuera incapaz de alcanzar la sección crítica, que sabemos que es alcanzable en general, sería necesario que otro proceso `Pj` (o varios) alcanzaran infinitamente a menudo la sección crítica, impidiendo de esta manera que `Pi` entrase en sección crítica. La hipótesis de incorrección anterior significa que cada vez que `Pj` ejecute su protocolo y encuentre `c[i]` $\neq$ `pasivo`, el valor de `turno` que encontró en la segunda línea del algoritmo debe haber sido establecido por un proceso `Pk` que sigue a `Pi` y precede a `Pj` en el turno cíclico, i.e. `(i > k > j)`. Si el proceso `Pk` no existiese entonces `Pi` no está bloqueado o dejará de estarlo muy pronto. Por otra parte, puesto que `Pj` adelanta a `Pi` continuamente (esta es la hipótesis de incorrección) el efecto de permitir que ocurra ésto se debe manifestar también continuamente, ésto es, `Pk` está continuamente entrando en sección crítica antes que `Pj`. Como consecuencia de lo anterior, debe de existir un proceso `Pk`$'$ que sigue a `Pi` y precede a `Pk`, i.e. `(i>` k' `> k)`, que asigne el turno que encuentra `Pk`, y así sucesivamente. Pero el razonamiento anterior nos lleva a un absurdo, ya que el número de procesos es finito (`n`) y alguna vez tiene que entrar el proceso `Pi` en la sección crítica. Por lo tanto la ausencia de inanición de los procesos con la solución anterior queda garantizada.

El retraso máximo que se produce en la entrada a la sección crítica por cualquier proceso del algoritmo anterior es r(n) $= 2^{n-1} - 1$ turnos, donde **n** es el número de procesos definido en el algoritmo y *turno* se decrementa cada vez que un proceso entra en su sección crítica. Nótese que los valores sucesivos de turno no son siempre consecutivos.

Escenario que muestra cómo, para `n=4` procesos, un proceso solicitando entrar en sección crítica podría llegar a tener que esperar hasta un máximo de 7 turnos ($= 2^{n-1} - 1$).

posición	c[1]	c[2]	c[3]	c[4]	turno	en SC
inicialmente: P_4 en `E1` P_1, P_2, P_3 en `E2`	solicitando	solicitando	solicitando	solicitando	0	
P_1 : `E2` $\rightarrow$ `E5`	pasivo	solicitando	solicitando	solicitando	4	si
P_2 : `E2` $\rightarrow$ `E5`	pasivo	pasivo	solicitando	solicitando	1	si
P_1 : `E5` $\rightarrow$ `E2`	solicitando	pasivo	solicitando	solicitando	1	
P_1 : `E2` $\rightarrow$ `E5`	pasivo	pasivo	solicitando	solicitando	4	si
P_3 : `E2` $\rightarrow$ `E5`	pasivo	pasivo	pasivo	solicitando	2	si
P_1 : `E5` $\rightarrow$ `E2`	solicitando	pasivo	pasivo	solicitando	2	
P_2 : `E5` $\rightarrow$ `E2`	solicitando	solicitando	pasivo	solicitando	2	
P_1 : `E2` $\rightarrow$ `E5`	pasivo	solicitando	pasivo	solicitando	4	si
P_2 : `E2` $\rightarrow$ `E5`	pasivo	pasivo	pasivo	solicitando	1	si
P_1 : `E5` $\rightarrow$ `E2`	solicitando	pasivo	pasivo	solicitando	1	
P_1 : `E2` $\rightarrow$ `E5`	pasivo	pasivo	pasivo	solicitando	4	si

8 Una solución simple al problema de la exclusión mutua: el algoritmo de Peterson

La solución al problema de la exclusión mutua para n-procesos, conocida como *algoritmo de Dijkstra*, es una *generalización mítica* del algoritmo de Dekker. ¡No hay más que comparar lo diferentes que son los protocolos de ambos algoritmos! Peterson en 1981 propuso una forma simple de resolver el problema de la exclusión mutua, que además permitía una fácil generalización al caso de n-procesos, manteniendo, al mismo tiempo, la estructura del protocolo.

8.1 Solución para 2 procesos

Las variables compartidas por los procesos son:

```
var
   c: array [0..1] of boolean;
   turno: 0..1;
```

El array `c` se inicializa a `FALSE` e indica si el proceso `Pi` intenta o no entrar en sección crítica. La variable `turno` sirve para resolver conflictos cuando ambos procesos intentan entrar simultáneamente.

Supuestos los valores `i= 0` y `j= 1` correspondientes a los identificadores de los procesos `Pi` y `Pj`. El código para el proceso `Pi` es el siguiente:

```
...
c[i]:= true;
turno:= i;
while(c[j] and turno=i) do
  nothing;
enddo;
 <seccion critica>
c[i]:= false;
...
```

Exclusión mutua
Para demostrar que el algoritmo anterior cumple la propiedad de exclusión mutua se sigue un razonamiento por reducción al absurdo. Suponer que `P0` y `P1` se encuentran ambos en sus secciones críticas, entonces los valores de sus claves han de ser necesariamente, `c[0]= c[1]= TRUE`; sin embargo, las condiciones de espera de los dos procesos no han podido ser simultáneamente ciertas, ya que la variable compartida `turno` ha debido tomar el valor final 0 ó 1 antes de que ambos procesos realicen iteraciones. Por tanto, sólo uno de los procesos ha podido entrar en sección crítica, digamos que es `Pi` ya que éste encontró `turno= j`. `Pj` no ha podido haber entrado en sección crítica junto con `Pi`, ya que para esto hubiera necesitado encontrar `turno=i`; sin embargo, la única asignación que el proceso `Pj` pudo haber hecho a la variable `turno` le era desfavorable.

Ausencia de interbloqueo
Suponer que `P0` está continuamente bloqueado esperando entrar en su sección crítica. Suponer también que el proceso `P1` está pasivo y no intenta ejecutarse, o bien que el proceso `P1` está esperando entrar en sección crítica continuamente. En el primer caso `c[1]= FALSE` y `P0` puede entrar en su sección crítica. El segundo caso es imposible, ya que la variable compartida `turno` ha de ser 0 ó 1, y hará a alguna de las condiciones de espera cierta, permitiendo entrar en sección crítica al proceso correspondiente. Por lo tanto, el suponer que ambos procesos están bloqueados indefinidamente lleva a contradicción.

Equidad de los procesos
Suponer ahora que `P0` está bloqueado esperando entrar en sección crítica y `P1` está entrando una y otra vez en ella, y por lo tanto monopolizando su acceso a la sección crítica. Sin embargo, ésto también lleva a contradicción, ya que si `P1` intenta entrar de nuevo en sección crítica hará la asignación `turno:= 1` que validará la condición para que `P0` entre en sección crítica.

8.2 Solución para n procesos

El principio de generalización del algoritmo de Peterson es simple: se utiliza repetitivamente (n-1) veces la solución de 2 procesos para eliminar al menos 1 proceso cada vez hasta que sólo quede uno, el cual entra en la sección crítica.

8.2.1 Esquema general

Las variables compartidas necesarias son:

```
var c: array[0..n-1] of -1..n-2; /*los valores del array c representan
                                   las etapas por las que pasa un proceso
                                   antes de entrar en s.c.*/
   turno: array[0..n-2] of 0..n-1; /*representa el no. de proceso que tiene el
                                    turno en cada etapa, para resolver posibles
                                    conflictos*/
```

Los valores iniciales de los arrays `c` y `turno` son -1 y 0, respectivamente. `c[i]= -1` representa que `Pi` está pasivo y no ha intentado entrar en el protocolo. `c[i]= j` significa que `Pi` está en la etapa j-ésima del protocolo.

El código siguiente pertenece al proceso `Pi`. Se utilizan las variables locales `i` y `j`. Los números de procesos se toman en el rango `i= 0..n-1`.

```
while true do
  begin
   Resto de las instrucciones;
   (1) for j=0 TO n-2 do
   (2)   begin
   (3)     c[i]:= j;
   (4)     turno[j]:= i;
```

```
(5)        while((∃k ≠ i : c[k] ≥ j) ∧ turno[j] = i)do

   (6)       nothing;
   (7)    end;
   (8)   enddo;
   (9) c[i]:= n-1; /*meta-instruccion*/
   (10) <seccion critica>
   (11) c[i]:= -1
  end
enddo;
```

La implementación del algoritmo anterior representa una cola con `n-1` posiciones para entrar en sección crítica, excepto que los procesos bloqueados en una etapa pueden ser adelantados por los demás cuando salen de la sección crítica y vuelven a entrar al protocolo.

8.2.2 Verificación formal de las propiedades del algoritmo

Se dice que el proceso `Pi` *precede al* proceso `Pk` sii el primer proceso está en una etapa más adelantada que el segundo, i.e. `c[i] > c[k]`.

La verificación de que el algoritmo satisface las propiedades que serían deseables (tanto seguridad como vivacidad), se realiza demostrando los cuatro lemas siguientes.

Lema 1
Un proceso que precede a todos los demás puede avanzar al menos una etapa.

Sea `Pi`, cuando dicho proceso evalúa su condición de espera, encuentra que `j= c[i] > c[k]`, *para todo k distinto de i*, ya que precede a todos los demás. Por lo tanto, la condición para terminar la espera se hace cierta y puede avanzar a la etapa `j+1`.

Debido a que cuando un proceso comprueba su condición no realiza una instrucción atómica, puede ocurrir que durante dicha comprobación uno o varios procesos alcancen la misma etapa que `Pi`, pero tan pronto como el primero de dichos procesos modifique la variable `turno[j]` ocasionará que el proceso `Pi` pueda continuar. También podría ocurrir que el proceso `Pi` fuera adelantado en la etapa siguiente, si más de un proceso llega a la etapa `j` del algoritmo.

Lema 2
Cuando un proceso pasa de la etapa `j` *a la etapa* `j+1`*, se han de verificar alguna de las condiciones siguientes.*

(a) precede a todos los demás,
(b) o bien, no está solo en la etapa `j`

Suponer que `Pi` está a punto de avanzar a la etapa siguiente, entonces: o bien se da la condición del Lema 1, y por tanto también se verifica (a), o bien no se da dicha condición. En este último caso, para que el proceso `Pi` pudiera avanzar ha de cumplirse que `turno`[`j`] <> `i`, luego el proceso `Pi` está acompañado por al menos otro proceso en la etapa `j` y además no ha sido el último proceso en llegar a dicha etapa. Independientemente del número de procesos que modificaron `turno[j]` después de `Pi`, existirá un proceso `Pr` que fue el último en modificar

`turno[j]`, dicho proceso no satisface la condición para salir de la espera, ya que `turno[j]= r`, y se queda bloqueado. El proceso `Pr` hace que la condición (b) del lema sea cierta.

También puede suceder que el proceso `Pi` vaya a avanzar de etapa porque ha encontrado que la condición (a) es cierta y mientras tanto otro proceso alcance su misma etapa, haciendo falsa dicha condición, pero en ese caso se hace cierta la condición (b).

Lema 3
Si existen al menos dos procesos en la etapa `j`*, entonces existe al menos un proceso en cada una de las etapas anteriores.*

La demostración se hace por inducción sobre la variable j.

Para (`j=0`) no tiene sentido dicha demostración. Para demostrar el caso (`j=1`) se utiliza el Lema 2. Si suponemos que existe un proceso en la etapa `j= 1`, entonces otro proceso (o más de uno) se unirá a él cuando el que se une deje atrás un proceso en la etapa `j=0`. Este proceso que se ha quedado en la etapa 0, mientras que esté solo no puede avanzar.

Suponer ahora que el Lema 2 se satisface en la etapa `j-1`. Si hay 2 ó más procesos en la etapa `j`, entonces se ha de satisfacer que en el instante en que el último de dichos procesos alcanzó la etapa `j` existían al menos dos procesos en la etapa `j-1` (por el Lema 2). Por hipótesis de inducción todas las etapas anteriores han de estar ocupadas y por el Lema 2 ninguna de dichas etapas ha podido quedar vacía desde entonces.

Lema 4
El número máximo de procesos que puede haber en la etapa `j` *es* `n-j`*, con* `0 <= j <= n-2`.

Aplicando el Lema 3, si las etapas `0..j-1` tienen al menos 1 proceso, entonces la etapa `j` tiene como máximo `n-j` procesos. Si alguna de las etapas anteriores estuviese vacía (caso de que un proceso se adelante a los demás y consiga avanzar siempre), entonces por el Lema 3 habrá como máximo un proceso en la etapa j. Por lo tanto, en la etapa `n-2` hay como máximo 2 procesos.

Exclusión mutua
Supongamos que hay 1 proceso en la etapa `n-2` y 1 proceso en la etapa `n-1` (sección crítica), entonces aplicando el Lema 2, el proceso de la etapa `n-2` no puede entrar en sección crítica ya que no precede a todos los demás y está solo en dicha etapa. Cuando el proceso que está en la etapa `n-1` abandone la sección crítica, se cumplirá la condición de que el proceso en la etapa `n-2` precede a los demás y entonces podrá avanzar.

Supongamos que hay 2 procesos en la etapa `n-2`, sólo uno de ellos podrá avanzar, ya que para el otro cuando éste avance no se cumplirán las condiciones del Lema 2.

Ausencia de interbloqueo
La hipótesis de incorrección sería que los procesos se quedaran bloqueados al llegar a una etapa y no consiguiesen avanzar. Si suponemos que un proceso precede a todos los demás, entonces por el Lema 1 no puede quedarse bloqueado en ninguna etapa. Si ahora suponemos que un proceso llega a una etapa donde hay otros procesos, entonces los procesos que se encontraban en dicha etapa pueden avanzar.

Equidad
Supongamos que todos los procesos intentan entrar en sección crítica y que en la etapa 0 se encuentra el proceso `Pi` que ha sido el último en asignar `turno[0]` y por tanto se bloquea en dicha etapa. En el caso más desfavorable los restantes `n-1` procesos entran en sección crítica y vuelven al protocolo de entrada. Sea `Pl` el último de dichos procesos en asignar `turno[0]:= l`, luego dicho proceso hará cierta la condición de `Pi` para abandonar la espera. Como consecuencia `Pi` se desbloqueará y pasará a la etapa 1. La misma situación se aplicará a los `n-1` procesos restantes.

El número de turnos que un proceso cualquiera tendría como máximo que esperar con el algoritmo anterior, sería: $r(n) = n - 1 + r(n-1) = \frac{n(n-1)}{2}$ turnos

9 El problema de la exclusión mutua en un entorno distribuido

En un sistema distribuido, la especificación de un algoritmo de exclusión mutua no se puede realizar utilizando variables en memoria compartida, sino sólo operaciones de intercambio de mensajes y un protocolo que proporcione a los procesos un acceso equitativo a la sección crítica.

Normalmente, los algoritmos distribuidos suelen formularse haciendo las siguientes hipótesis acerca de la red de comunicaciones que sirve para transmistir los mensajes entre los procesos concurrentes.

- La red de comunicaciones está completamente conectada, ésto es, desde cualquier nodo se puede encontrar un camino hacia el resto de los nodos de la red.
- Se supone que la transmisión de los mensajes se realiza sin errores.
- El retraso en la entrega de un mensaje es variable, ya que depende del número de mensajes que se estén transmitiendo en cada momento y de la planificación del acceso de los procesos a la red; pero, dicho retraso será siempre un tiempo acotado, por lo tanto, no se pierden mensajes durante la transmisión de estos.
- Podrían ocurrir desencuenciamientos en la transmisión de los mensajes, es decir, los mensajes pueden ser recibidos en un orden diferente del que fueron enviados.

9.1 Algoritmo original de Ricart-Agrawala (1981)

Este algoritmo se propone para reducir el número de mensajes necesario, respecto de otras propuestas anteriores, para programar un protocolo distribuido para resolver el problema de la exclusión mutua. El algoritmo asume las suposiciones anteriores respecto de la red de comunicaciones. Admite, en concreto, un tiempo variable en la transmisión de los mensajes y que estos se puedan adelantar unos a otros mientras circulan por la red. El algoritmo necesita 2×(n-1) mensajes para implementar correctamente el protocolo de exclusión distribuido; esto

es, un proceso necesita (n-1) mensajes para indicar al resto su intención de entrar en sección crítica y otros (n-1) mensajes para obtener la respuesta favorable de estos para poder acceder.

Cuando un proceso P_i quiere entrar en su sección crítica genera un número de secuencia *ns* y difunde un mensaje de petición, de tipo *req*, a todos los demás procesos que contiene dicho número de secuencia. Si un proceso P_j recibe tal mensaje puede, o bien responder favorablemente enviándole un mensaje de tipo *rep*, o posponer su respuesta provocando la espera de P_i. Un proceso sólo puede entrar a la sección crítica cuando recibe un mensaje de cada uno de los otros procesos. Al salir de la sección crítica, el proceso enviará los mensajes de respuesta favorable a todos los procesos que lo estuvieran esperando. La decisión de responder favorablemente de forma inmediata o de posponer dicha contestación se basa en un mecanismo de prioridad entre los procesos que viene definido por la siguiente regla: si un proceso no tiene intención de entrar en su sección crítica, envía la respuesta favorable inmediatamente; si no, si el proceso también intenta entrar, compara el número de secuencia de su petición actual con el de la última petición recibida. Si la propia petición del proceso es más reciente (se "hizo antes"), entonces decide posponer la respuesta favorable a la petición recibida.

```
                Version Concurrente del algoritmo de Agrawala
var
    ns: 0..+INF; (* numero de secuencia generado por P(i) *)
    mns: 0..INF:= 0; (* mayor numero de secuencia generado o recibido *)
    numrepesperadas:0..n-1;
    intentaSC: boolean;
    prioridad: boolean;
    repretrasadas: array[1..N] of boolean;

    Proceso Main(i)
(1) wait(s);
(2)     intentaSC:= TRUE;
(3)     ns:= mns +1;
(4)     numrepesperadas:=n-1;
(1) signal(s);
(6) for j:= 1 to n do
(7)     if j <> i then
(8)       send(j, pet, ns, i);
(9) wait(sinc);
 <<seccin crtica>>
(10) wait(s);
(11)     intentaSC:= FALSE;
(12) signal(s);
(13) for j:=1 to n do
(14) if repretrasadas[j] then begin
(15)   repretrasadas[j]:= FALSE;
(16)   send(j, rep);
     end;
```

```
Proceso Pet(i)
(1) receive(pet, k, j);
(2) wait(s);
(3)   mns:= max(mns, k);
(4)   prioridad:= (intentaSC) and (k>ns or (k=ns and i<j));
(5)   if prioridad then
(6)     repretrasadas[j]:= TRUE
(7)     else send(j, rep);
(8)  signal(s);

Proceso Rep(i)
(1) receive(rep);
(2)   numrepesperadas:= numrepesperadas -1;
(3)   if numrepesperadas= 0 then (* el proceso puede entrar en S.C.*)
(4)      signal(sinc);
```

Algoritmo original de Ricart-Agrawala(1981)

```
var
    osn : 0 .. + ;
    hsn : 0.. +  ;
    numrepexpected : 0 .. n-1;
    csrequested : boolean;
repdeferred : array [1..n] of boolean;
    priority: boolean;

csrequested <- true;
osn <- hsn + 1;
numrepexpected <- n-1;
for j <> i, j  1..n
        send(req, osn, i) to j;
wait(numrepexpected = 0);
<critical section>;
csrequested <- false;
for j = 1 to n do
    if repdeferred[j] then begin
        repdeferred[j] <- false;
        send(rep) to j;
        end;
    endif;

on receipt:

    of (req, k, j) do
        begin
```

```
    hsn <- max(hsn, k);
    priority <- csrequested
[(k > osn) (k = osn   i < j)];
    if priority then
            repdeferred[j] <- true
            else send(rep) to j
        endif;
    end;

of (rep)    do
numrepexpected  numrepexpected-1
```

9.2 Demostración de la corrección del algoritmo

Dado cualquier par de procesos, P_i, P_j, del algoritmo, siempre se cumplirá que uno ha de dejar la sección crítica antes de que el otro pueda entrar a ella. Veamos que ésto es así razonando por *reducción al absurdo*. Si los 2 procesos anteriores consiguen entrar a la vez en sus regiones críticas, entonces cada uno de los procesos ha debido de transmitir un mensaje *pet* al otro y ha debido igualmente de recibir de este un mensaje *rep* de contestación favorable. En ese caso, habría que discutir varios casos para llegar a la conclusión de la imposibilidad de que ambos procesos lleguen a estar simultáneamente en sección crítica.

1. P_i ha enviado su contestación favorable a una petición de P_j antes de asignar su número de secuencia. Esta última es una operación previa al envío de su propio mensaje de petición. Pero en ese caso el número de secuencia que genere P_i será necesariamente mayor que el que contiene la petición de P_j. Por tanto, cuando P_j reciba la petición de entrada en sección crítica de P_i, decidirá posponer la contestación favorable hasta que él mismo salga de ella. Llegamos por tanto a una contradicción con la hipótesis de que pudiera existir un escenario en el que ambos procesos accedan simultáneamente a la sección crítica.

2. Este caso es el proceso P_j quien envía su contestación favorable a P_i y este pospondrá la petición de P_j cuando la reciba, de una forma simétrica al caso anterior.

3. Por último, supongamos que cada uno de los 2 procesos envíase una respuesta favorable al otro. Ambos procesos tendrán su variable `intentaSC` con el valor `true` cuando reciban la petición del otro y, por tanto, los procesos podrá determinar su prioridad para entrar en la sección crítica con respecto al otro. Pero, sólo uno de los 2 procesos se verá retrasado en al acceso a la sección crítica, ya que los números de secuencia de todos los procesos del algoritmo están globalmente ordenados. En caso de "empate" decide el que tenga asignado el menor número de nodo.

La propiedad de alcanzabilidad de la sección crítica se demuestra también por reducción al absurdo. Suponemos que ningún proceso puede entrar en sección crítica porque está siempre esperando una contestación favorable de otro proceso que nunca llega. Esta situación, sin embargo, no se puede mantener por mucho tiempo, ya que la decisión de retrasar indefinidamente una contestación favorable a la petición de un proceso es incompatible con la ordenación total

de las peticiones de los procesos que induce la condición de *prioridad* programada en la línea (4) del proceso `Pet(i)`. De las que esperan ser autorizadas, siempre existirá una petición con el menor número de secuencia que recibirá todas las contestaciones favorables del resto de los procesos y, por tanto, su proceso podrá entrar a la sección crítica.

Siguiendo un razonamiento similar al de la demostración anterior, se puede demostrar también la ausencia de inanición de los procesos en cualquier escenario de ejecución del algoritmo. Como todas las peticiones se encuentran globalmente ordenadas según su antigüedad, toda petición terminará por recibir la respuesta favorable del resto de los procesos trascurrido un tiempo finito.

Para determinar la propiedad de equidad en la ejecución de los procesos del algoritmo, hay que obtener el mayor número de *turnos* que un proceso puede llegar a tener que esperar, según la hipótesis de ejecución del algoritmo que le sea más desfavorable, antes de entrar en la sección crítica. Si consideramos que los mensajes de tipo *rep* pueden adelantarse a los mensajes de petición (*pet*), el máximo retraso que puede experimentar un proceso vendría dado por: $d_1(n) = \frac{n(n+1)}{2} - 1$, para n procesos; ya que si los mensajes de tipo *rep* se adelantan a los de tipo *pet*, entonces se puede dar que el proceso P_i adelante un número máximo de turnos al proceso P_j que es igual a la diferencia entre sus números de secuencia ($= n_{s_j} - n_{s_i}$). En el caso de que los mensajes sean recibidos siempre en el orden en el que se transmiten, entonces un proceso no puede adelantarse a los demás y entrar más de 2 veces seguidas en sección crítica; es decir, una vez entraría debido a que su número de secuencia es menor que el del resto y la otra debido a que ahora lo es su número de nodo. En este caso, por tanto, el máximo retraso vendrá dado por la fórmula: $d_2(n) = 2 \times (n - 1)$.

9.3 Algoritmo de Suzuki-Kasami (1982) o de Ricart-Agrawala (1983)

En este algoritmo el permiso para que un proceso pueda entrar en sección crítica vendrá dado por la posesión de un *token*. Cualquier proceso que tenga el *token* puede entrar sin tener que esperar una respuesta favorable del resto de los procesos, a diferencia de lo que ocurría en el caso del algoritmo de Ricart-Agrawala (1981). El *token* es asignado inicialmente, de forma arbitraria, a uno de los procesos. Un proceso que intenta entrar en la sección no sabe cual de los otros procesos posee el *token* en ese momento y lo tendrá que solicitar mediante la difusión de un mensaje, en el que incluirá un número de secuencia, al resto de los procesos. El *token* es un array cuyo *k-ésimo* elemento guarda el valor del número de secuencia vigente en el proceso P_k cuando lo consiguió la última vez. Al salir de la sección, el proceso que actualmente posee el token comienza a buscar, según el orden $j+1$, $j+2$, ... n, 1, 2, ..., en un array local donde se han apuntado las peticiones de los otros procesos, hasta que encuentra uno que ha solicitado el *token* después de la última vez que lo obtuvo. Si encuentra dicho proceso, le transfiere el *token*; si no, lo mantiene y vuelve a acceder a la sección.

```
                Version concurrente del algoritmo:
token_presente:boolean:=FALSE;
enSC:boolean:= FALSE;
token: array[1..N] of 0..+INF;
peticion: array[1..N] of 0..+INF;
```

```
              Proceso Pi
(0) wait(s);
(1) if NOT token_presente then begin
(2)   ns := ns+1;
(3)   broadcast(pet, ns, i);
(4)   receive(acceso, token);
(5)   token_presente := true;
(6) end;
(7) enSC := true;
(8) signal(s);
<<seccion critica>>
(9)  token[i] := ns;
(10)    enSC := false;
(11)    wait(s);
(12)      for j:= i+1 to n, 1 to i-1 do
(13)        if peticion[j] > token[j] and token_presente then begin
(14)            token_presente := false;
(15)            send(j, acceso, token);
(16)          end;
(17)    signal(s);

            Proceso Pet(i)
(1) receive(pet, k, j);
(2) peticion[j]:= max(peticion[j], k);
(3) wait(s);
(4)  if token_presente and NOT enSC then
<<< repetir (12)- (16) >>>
    signal(s);

    Algoritmo original de Suzuki-Kasami
var
    osn : 1 .. +INFTY;
    tokenpresent : boolean;
    inside : boolean;
    token : array[1..n] of 0..+INFTY;
    requests : array[1..n] of 0.. +INFTY;

if  !tokenpresent then
    begin
        osn <- osn +1;
        broadcast(req, osn, i);
        wait(tkn, token);
    end;
endif;
inside <- true;
tokenpresent <- true;
<critical section>;
```

```
token[i] <- osn;
inside <- false;
for j = i+1 to n step 1, = 1 to i-1 step 1 do
    if (request[j] > token[j]) &&
   tokenpresent then begin
        tokenpresent <- false;
        send(tkn, token) to j;
    end;
endif;

on receipt
    of (req, k, j) do
        begin
            request[j] <- max(request[j], k);

    if tokenpresent && !inside
    then
        for j = i+1 to n step 1, = 1 to
     i-1 step 1 do
        if (request[j] > token[j]) &&
     tokenpresent then begin
                tokenpresent <- false;
                    send(tkn, token) to j;
            end;
endif
```

9.4 Demostración de la corrección del algoritmo

Demostrar que este algoritmo satisface la propiedad de exclusión mutua es equivalente a probar que en cualquier momento de su ejecución el número máximo de variables *token-presente* con valor *true* es uno. Puesto que inicialmente sólo se asigna dicho valor a una sola variable, podremos demostrarlo si probamos que la condición de posesión única del token se conserva cada vez que este es transmitido.

Si consideramos el protocolo de adquisición de P_i, veremos que la variable $\texttt{token\_presente}_i$ sólo cambia (de *false* a *true*) cuando este recibe el token. De manera análoga, si ahora consideramos el protocolo de restitución de un proceso que envía el *token*, veremos que dicho proceso ha podido enviarlo porque lo poseía y, por tanto, $\texttt{token\_presente}_i$ cambiará después de *true* a *false*. Ésto establece la condición de exclusividad en la posesión del token, así como la existencia de una sola variable con valor *true*. Nótese que todas las variables $\texttt{token\_presente}_i$ tienen el valor *false* mientras el el *token* está siendo transmitido por la red, y sólo en ese caso.

La alcanzabilidad de la sección crítica se puede demostrar haciendo la suposición de incorrección de que todos los procesos llegan a un estado en el que quieren entrar en la sección, pero ninguno posee el *token*. En dicho escenario todos los procesos estarían bloqueados indefinidamente esperando el *token*. Pero, si ésto es así, el *token* se encontraría en tránsito por la red

llegando, al cabo de cierto tiempo, a ser recibido por uno de los procesos que se desbloquearía. Este hecho es contradictorio con la hipótesis de incorrección.

La equidad se sigue del hecho de que todos los mensajes han de ser entregados en un plazo de tiempo finito (una de las suposiciones del modelo de sistema distribuido que inicialmente realizamos). El protocolo de restitución de cualquier proceso P_i obliga a transmitir el *token* al primer proceso que se lo haya pedido, siguiendo el orden $j+1, j+2, \ldots, n, n-1, \ldots$. Si los retrasos de los mensajes de todos los mensajes están acotados, es decir no se puede perder ninguno, entonces tarde o temprano el resto de los procesos llegarán a saber de la petición de un proceso que intenta entrar en la sección y eventualmente se pondrán de acuerdo en dejarlo entrar, cuando llegue su turno.

Un proceso que no posea el *token* necesitará intercambiar *n* mensajes, *n-1* para difundir su solicitud y 1 para obtener el *token*, con el resto de procesos. Los mensajes pueden ser de 2 tipos: los del primero representan a las solicitudes de los procesos y se componen de tres campos (tipo, número de secuencia, identidad del proceso); los del segundo tipo, que contienen al *token*, tienen n+1 elementos: el tipo del mensaje y *n* valores de secuencia. A pesar de la longitud de los mensajes que transmiten el *token*, éste algoritmo es mejor que el de Ricart-Agrawala (1981), ya que el número de mensajes que se intercambian es mucho menor.

Si la red no fuera fiable y se perdieran mensajes, el algoritmo seguiría *funcionando*, aunque provocando la inanición de algunos procesos o degenerando en una situación de interbloqueo si se pierde el mensaje con el *token*.