

# Detección Dinámica de Bloqueos Mutuos bajo el Modelo OR de Requerimientos

Alvaro E. Campos, Christian F. Orellana y María Pía Soto

Departamento de Ciencia de la Computación

P. Universidad Católica de Chile

Casilla 306 - Santiago 22 - CHILE

Fax: +56 2 354-4444

[acampos,cforella,mpsoto]@ing.puc.cl

## Resumen

El problema de la detección de bloqueos mutuos (*deadlocks*) en sistemas distribuidos es uno de los más discutidos en la literatura. Aunque se ha propuesto numerosos algoritmos, se trata de un problema aún abierto. El correcto funcionamiento de los algoritmos depende, en general, del modelo de requerimientos que se considere. Este artículo presenta un algoritmo de detección de bloqueos mutuos para el modelo OR de requerimientos. Este algoritmo es completo, pues detecta todos los bloqueos mutuos, y es correcto, pues no detecta bloqueos mutuos falsos. Además, el algoritmo soporta cambios dinámicos en el grafo de espera sobre el que trabaja. Una vez finalizado el algoritmo, cada proceso sabe si está o no en bloqueo mutuo. Utilizando esta propiedad, se sugiere posibles extensiones al algoritmo, para resolver los bloqueos mutuos.

**Palabras claves:** Sistemas distribuidos, detección de bloqueos mutuos, resolución de bloqueos mutuos, autoestabilización.

## 1 Introducción

UNA de las principales motivaciones para construir sistemas distribuidos es la posibilidad de compartir recursos entre varios procesadores. Un proceso puede adquirir y liberar recursos en un orden que no es conocido de antemano. En un ambiente como éste surge el problema de los bloqueos mutuos, y tener la capacidad de detectarlos es el primer paso para poder tomar acciones y resolverlos.

En términos generales, se dice que un conjunto de procesos se encuentra en bloqueo mutuo cuando cada proceso en el conjunto está bloqueado, esperando por recursos que están asignados a otros procesos en el mismo conjunto [14]. Para que una situación de este tipo ocurra, es necesario que en el sistema se presenten simultáneamente cuatro condiciones relacionadas con la competencia por recursos: los recursos asignados no pueden ser expropiados, debe existir acceso exclusivo a los recursos, los procesos mantienen los recursos asignados mientras esperan por otros y existe espera circular.

Existen tres estrategias para enfrentar el problema de los bloqueos mutuos: prevenirlos, evitarlos y detectarlos [14]. Las dos primeras liberan al sistema de la posibilidad de entrar en un bloqueo

mutuo, pero son poco eficientes. La primera impone restricciones en el modo en que un proceso puede ejecutar, para impedir alguna de las cuatro condiciones mencionadas anteriormente. La segunda es computacionalmente cara, debido a que antes de cada asignación de recursos es necesario comprobar la seguridad del nuevo estado. La tercera, en cambio, consiste en dejar que los bloqueos mutuos ocurran, para luego detectarlos y solucionarlos.

## 2 Modelos teóricos

Knapp ha clasificado el problema de la detección de bloqueos mutuos en seis modelos, de acuerdo al tipo de requerimientos que un proceso puede realizar [10]. Los modelos son los siguientes:

**Sólo un requerimiento pendiente:** En este modelo, un proceso sólo puede requerir un recurso a la vez. Es el modelo de requerimientos más simple. Flatebo y Datta han propuesto un algoritmo para detectar bloqueos mutuos en este modelo [4].

**Modelo AND:** En este modelo, un proceso puede realizar un requerimiento por múltiples recursos simultáneamente. Un proceso ve satisfecha una solicitud de este tipo una vez que se le han asignado *todos* los recursos que había requerido. Algoritmos para detectar bloqueos mutuos bajo este modelo han sido propuestos [4, 9], así como también para resolverlos [6].

**Modelo OR:** En este modelo, un proceso también puede realizar un requerimiento por múltiples recursos simultáneamente, pero este requerimiento se ve satisfecho cuando *alguno* de los recursos solicitados es asignado. Chandy, Misra y Haas [2], y Natarajan [11] han propuesto algoritmos para detectar bloqueos mutuos en este modelo.

**Modelo AND/OR:** Este modelo es una generalización de los dos anteriores. Bajo él, un proceso puede requerir cualquier número de recursos en una combinación arbitraria de requerimientos AND y OR. Por ejemplo, un proceso puede hacer un requerimiento del tipo  $((r_1 \wedge r_2) \vee r_3) \wedge r_4$ , donde  $r_1, r_2, r_3$  y  $r_4$  son recursos en el sistema. Hermann y Chandy han propuesto un algoritmo distribuido para detectar bloqueos mutuos bajo este modelo [7].

**Modelo  $\binom{n}{k}$ :** En este modelo, un proceso ve satisfecho un requerimiento por  $n$  recursos cuando se le otorgan  $k$  de ellos. Corresponde a un modelo generalizado, ya que cuando  $n = k = 1$  el requerimiento corresponde al primer modelo descrito. Si  $n = k \neq 1$ , entonces corresponde a un requerimiento de tipo AND, y si  $k = 1 \neq n$  corresponde a uno de tipo OR. Bracha y Toueg han propuesto un algoritmo para detectar y resolver bloqueos mutuos en este modelo [1].

**Modelo no restringido:** En este caso no se hace ninguna suposición sobre el modo en que un proceso puede realizar requerimientos.

Este artículo presenta un algoritmo de detección de bloqueos mutuos para el modelo OR de requerimientos. Un proceso puede hacer requerimientos de tipo OR, por ejemplo, en un sistema de bases de datos distribuidas replicadas, donde un requerimiento de lectura de un elemento replicado se ve satisfecho al leer *cualquier* copia [10]. De modo similar, en un sistema de enrutamiento de mensajes basado en *wormhole routing*, un *router* que recibe un mensaje puede reenviarlo a un router vecino por alguno de varios canales [13]. Un requerimiento por un canal de salida se ve satisfecho, en este caso, cuando alguno de los canales se encuentra disponible.

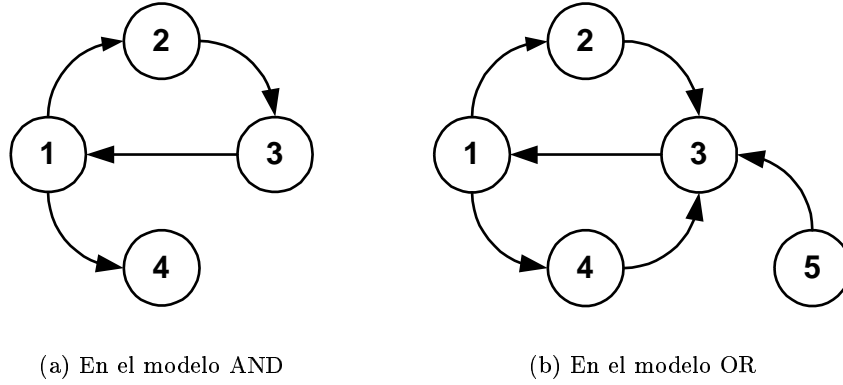


Figura 1: Ejemplos de bloqueo mutuo

Una forma útil de representar los requerimientos de recursos es por medio de un grafo dirigido, conocido como grafo de espera (WFG: *wait-for graph*). En él, cada nodo representa un proceso en el sistema. Nodos con arcos salientes representan procesos bloqueados, que están a la espera por recursos. Por el contrario, un nodo sin arcos salientes representa a un proceso activo. Un arco entre un nodo  $i$  y un nodo  $j$  indica que el proceso  $i$  está esperando por un recurso que actualmente está asignado al proceso  $j$ . El problema de la detección de bloqueos mutuos puede reducirse a detectar estructuras cíclicas en este grafo. Por ejemplo, la presencia de un ciclo en el grafo es condición necesaria y suficiente para la existencia de un bloqueo mutuo en el modelo AND [10]. En la Figura 1(a), los procesos 1, 2 y 3 se encuentran en un ciclo, y están en bloqueo mutuo. Aunque el recurso que tiene asignado el proceso 4 puede ser liberado si éste termina, el proceso 1 necesita también el recurso que tiene asignado el proceso 2 para poder continuar ejecutando.

Bajo el modelo OR de requerimientos, la existencia de un ciclo en el WFG es una condición necesaria, pero no suficiente, para la existencia de un bloqueo mutuo. Si los requerimientos son de tipo OR, en la Figura 1(a) no existe bloqueo mutuo a pesar del ciclo, pues el proceso 1 está esperando ya sea por el recurso que tiene asignado el proceso 2 o por el que tiene asignado el proceso 4. El proceso 4 puede terminar, y le otorgará el recurso al proceso 1.

Bajo este modelo, un proceso está *bloqueado* si tiene un requerimiento de tipo OR pendiente [2]. Asociado a cada proceso bloqueado hay un conjunto de procesos, llamado *conjunto dependiente*. Un proceso bloqueado puede continuar ejecutando si alguno de los procesos en el conjunto dependiente le otorga alguno de los recursos por los que esperaba. Un conjunto  $S$  de procesos se encuentra en bloqueo mutuo si se cumplen las siguientes condiciones:

- todos los procesos en  $S$  están bloqueados,
- el conjunto dependiente de cada proceso en  $S$  es un subconjunto de  $S$ , y
- no hay mensajes en tránsito entre procesos en  $S$ .

Detectar un conjunto de procesos en bloqueo mutuo es equivalente a detectar un nudo (*knot*) en el grafo de espera [8]. Por definición, un vértice  $v$  está en un nudo si,

$$\forall \text{ vértice } w, w \text{ es alcanzable desde } v \rightarrow v \text{ es alcanzable desde } w.$$

Para el nodo  $i$ :

```

(1.1)  if    (req not granted)
        then  $Succ := \{(P_1, i), (P_2, i), \dots, (P_k, i)\};$ 
            $Paths := \{(i, P_1, i), (i, P_2, i), \dots, (i, P_k, i)\}$ 
(1.2)  if     $(j, -) \in Succ_n \wedge (j, -) \notin Succ$ 
        then  $Succ := Succ \cup \{(j, n)\}$ 
(1.3)  if     $(j, n) \in Succ \wedge (j, -) \notin Succ_n$ 
        then  $Succ := Succ - \{(j, n)\}$ 

```

Figura 2: El algoritmo propuesto, primera fase

En la Figura 1(b), los procesos 1, 2, 3 y 4 están en un nudo, y están en bloqueo mutuo. Nótese que aunque el proceso 5 no pertenece al nudo, también se encuentra en bloqueo mutuo ya que todos los procesos en su conjunto dependiente lo están.

Chandy, Misra y Haas han propuesto un algoritmo para detectar bloqueos mutuos bajo el modelo OR, basado en la técnica conocida como *diffusing computations* [2]. En él, un proceso inicia el algoritmo cuando algún requerimiento no es satisfecho. Al terminar, el proceso que inició el algoritmo decidirá que se encuentra en bloqueo mutuo sólo si lo estaba en el momento de iniciarlo. En un conjunto de procesos en bloqueo mutuo, al menos uno de ellos es capaz de reportarlo. El algoritmo propuesto por Natarajan [11] se basa en el mismo principio que el de Chandy, Misra y Haas, pero utiliza un protocolo periódico que permite escoger exactamente un proceso de un conjunto de procesos en bloqueo mutuo para reportarlo.

En el algoritmo propuesto en este artículo, un proceso que no se encuentra en bloqueo mutuo al iniciar el algoritmo, pero que cae en ese estado posteriormente, es capaz de detectarlo. En un conjunto de procesos en bloqueo mutuo, todos los procesos son capaces de reportarlo. Adicionalmente, cada proceso puede decidir si se encuentra en bloqueo mutuo por pertenecer a un nudo o porque, sin pertenecer a un nudo, todos los procesos en su conjunto dependiente están bloqueados. Así, los procesos que se encuentran en bloqueo mutuo puedan iniciar una acción de resolución.

### 3 Detección de bloqueo mutuo en el modelo OR

En las Figuras 2, 3 y 4 se muestra el algoritmo propuesto. La detección se realiza en tres fases: en la primera, se computan los conjuntos sucesores de cada nodo. En la siguiente, se propagan los caminos del WFG de modo tal que cada nodo consigue una visión parcial de él. Finalmente, se determina localmente si el proceso se encuentra o no en bloqueo mutuo.

El algoritmo se inicia en un nodo cuando un requerimiento realizado por el proceso no es satisfecho. En ese momento, el proceso se bloquea, transfiriendo el control a una hebra que ejecuta el algoritmo de detección.

#### 3.1 Variables

Cada proceso mantiene tres variables locales al ejecutar el algoritmo: *Succ*, *Paths* y *deadlock*. Además, se supone que cada proceso tiene acceso de sólo lectura a las variables locales de los procesos por los cuales espera, representados por los nodos vecinos en el WFG.

Para el nodo  $i$ :

- (2.1) **if**  $(a, \_) \in Succ \wedge (i, a, i) \notin Paths$   
**then**  $Paths := Paths \cup \{(i, a, i)\}$
- (2.2) **if**  $(a, \_) \notin Succ \wedge (i, a, i) \in Paths$   
**then**  $Paths := Paths - \{(i, a, i)\}$
- (2.3) **if**  $(a, b, \_) \in Paths_n \wedge (a, b, \_) \notin Paths$   
**then**  $Paths := Paths \cup \{(a, b, n)\}$
- (2.4) **if**  $(a, b, n) \in Paths \wedge (a, b, \_) \notin Paths_n$   
**then**  $Paths := Paths - \{(a, b, n)\}$

Figura 3: El algoritmo propuesto, segunda fase

La variable  $Succ$  representa al conjunto de sucesores del nodo (proceso) que está ejecutando el algoritmo. Cada elemento de este conjunto es un par ordenado de la forma  $(a, b)$ , donde  $a$  corresponde al identificador del proceso sucesor, y  $b$  corresponde al identificador del vecino desde donde se obtuvo la información.  $Paths$  representa la visión parcial que tiene el nodo del WFG. Cada elemento de este conjunto es un trío ordenado de la forma  $(a, b, c)$ , e indica que existe un camino desde el nodo  $a$  hasta el nodo  $b$ . El identificador del vecino de donde se obtuvo la información corresponde a  $c$ . La variable  $deadlock$  indica si el proceso está o no en bloqueo mutuo.

Inicialmente, la variable  $deadlock$  toma el valor *false*. Las variables  $Succ$  y  $Paths$  deben ser inicializadas como conjuntos vacíos.

### 3.2 Notación

Cada paso del algoritmo tiene la siguiente forma:

(s) **if** <guardia> **then** <movida>

El número (s) es de la forma  $(f.p)$ , en donde  $f$  indica a qué fase pertenece cada paso y  $p$  sirve para enumerar cada uno de ellos. El predicado <guardia> es un predicado booleano sobre las variables a las cuales el proceso tiene acceso: sus propias variables locales y las variables de sus vecinos. Si el predicado es verdadero, entonces es posible ejecutar la acción definida en <movida>. El algoritmo supone la existencia de un coordinador — centralizado o distribuido — que escoge la movida a realizar cuando uno o más predicados son verdaderos. Las movidas son escogidas según la fase a la que pertenecen, teniendo prioridad aquellas que pertenecen a una fase anterior.

En las Figuras 2, 3 y 4, la variable  $i$  representa al identificador del proceso que está ejecutando el algoritmo, y  $n$  representa al identificador de alguno de sus vecinos. El número de vecinos es  $k$ , y sus identificadores son representados por variables de la forma  $P_j$ , con  $1 \leq j \leq k$ . Las variables locales del vecino  $n$  son designadas por  $Succ_n$ ,  $Paths_n$  y  $deadlock_n$ .

Las tuplas de la forma  $(a, \_)$  representan a cualquier par ordenado cuyo primer elemento es  $a$ . Del mismo modo, las de la forma  $(a, b, \_)$  representan a cualquier trío ordenado cuyos primeros elementos son  $a$  y  $b$ .

Para el nodo  $i$ :

```

(3.1)  if     $Paths \neq \emptyset \wedge (\forall j, (i, j, -) \in Paths \rightarrow (j, i, -) \in Paths) \wedge deadlock = false$ 
        then  $deadlock := true$ 
(3.2)  if     $(deadlock_1 \wedge deadlock_2 \wedge \dots \wedge deadlock_k = true) \wedge deadlock = false$ 
        then  $deadlock := true$ 
(3.3)  if     $(req\ granted)$ 
        then  $deadlock := false;$ 
            $Succ := \emptyset;$ 
            $Paths := \emptyset$ 

```

Figura 4: El algoritmo propuesto, tercera fase

### 3.3 El algoritmo

La fase 1 comienza con el paso (1.1), que es ejecutado por el proceso  $i$  cuando un requerimiento no es satisfecho. El proceso entonces calcula su conjunto inicial de sucesores, con los identificadores de sus vecinos. Dado que el proceso se encuentra bloqueado en ese momento, no es posible que se agreguen más vecinos. Al mismo tiempo, el conjunto de caminos se inicializa con los caminos directos desde el nodo  $i$  hacia sus vecinos.

El conjunto de sucesores de un nodo va propagándose en el paso (1.2) hacia los nodos antecesores. El objetivo de esta propagación es determinar todos los sucesores de cada nodo, tanto directos como indirectos. El nodo completa la información de sus sucesores indirectos guardando el identificador del nodo desde donde obtuvo la información. De este modo, en el paso (1.3) es posible eliminar todas las propagaciones de tuplas que, debido a algún cambio en el WFG, sean inválidas. Al terminar esta fase, cada nodo conoce exactamente a todos sus sucesores.

La fase 2 comienza con el paso (2.1), donde se completa el conjunto de caminos desde el nodo  $i$  hacia todos sus sucesores. Los caminos agregados se propagan hacia los antecesores, en un modo similar que en la primera fase, en el paso (2.3). Los pasos (2.2) y (2.4) sirven para propagar los cambios en el WFG. Una vez terminada esta fase, los nodos tienen una visión parcial del WFG que les permite decidir localmente si están o no en bloqueo mutuo.

La fase final comienza con el paso (3.1), que corresponde a determinar la existencia de un nudo en el WFG. Si existe, el proceso sabe que se encuentra en bloqueo mutuo. El paso (3.2) sirve para que los nodos que no se encuentran en un nudo, pero que esperan por procesos que sí lo están, sepan que están también en bloqueo mutuo. El paso (3.3) finaliza el algoritmo, en caso de que el recurso por el cual el nodo esperaba le sea otorgado.

## 4 Propiedades del algoritmo

A continuación se presenta y demuestra algunas propiedades del algoritmo propuesto.

**Lema 1** Si en el WFG  $i$  es un predecesor de  $j$ , entonces  $(j, -) \in Succ_i$  al terminar el algoritmo.

**Demostración** Por inducción sobre  $d(i, j)$ , la distancia entre el nodo  $i$  y el nodo  $j$ .

Caso base:  $d(i, j) = 1$ . En este caso,  $j$  es un vecino del nodo  $i$ , y  $(j, i) \in Succ_i$  por el paso (1.1) del algoritmo.

Hipótesis de inducción:  $\forall i$ , si  $d(i, j) = n$  entonces  $(j, \_) \in Succ_i$ .

Paso inductivo: Si  $d(i, j) = n + 1$  existe un camino entre el nodo  $i$  y el nodo  $j$  en el WFG. Sea  $k$  el nodo que sigue a  $i$  en este camino. Entonces,  $d(k, j) = n$  y por la hipótesis de inducción,  $(j, \_) \in Succ_k$ . Como  $k$  es un vecino de  $i$ , por el paso (1.2) del algoritmo, necesariamente  $(j, k) \in Succ_i$ .  $\square$

**Lema 2** *Si al terminar el algoritmo  $(j, \_) \in Succ_i$  entonces existe un camino entre  $i$  y  $j$  en el WFG.*

**Demostración** Si existe en  $Succ_i$  una tupla de la forma  $(j, i)$ , entonces fue agregada en el paso (1.1) y  $j$  es un vecino de  $i$  en el grafo.

Si existe en  $Succ_i$  una tupla de la forma  $(j, n)$  con  $n \neq i$  significa, por el paso (1.2), que hay una tupla de la forma  $(j, \_)$  en  $Succ_n$ . Como el nodo  $n$  es un vecino de  $i$ , existe un camino entre  $i$  y  $n$ . Inductivamente para  $n$ , se puede encontrar un camino entre  $n$  y  $j$ , por lo que se deduce que existe un camino entre  $i$  y  $j$ . Si este camino no existiera, entonces  $j$  sería eliminado de  $Succ_i$  por el paso (1.3) del algoritmo.  $\square$

**Lema 3** *Al terminar el algoritmo, cada nodo conoce todos los caminos del WFG que se inician en él.*

**Demostración** De los Lemas 1 y 2 se sabe que, al terminar el algoritmo, cada nodo conoce a todos sus sucesores, tanto directos como indirectos. Por el paso (2.1) del algoritmo, se agrega al conjunto *Paths* todos los caminos que se inician en el nodo  $i$ , y que terminan en algún nodo en el conjunto  $Succ_i$ . Además, por el paso (1.1), los caminos a los sucesores inmediatos del nodo  $i$  son también agregados. Por el paso (2.2) se eliminan todos los caminos que terminan en un nodo que ya no es alcanzable desde  $i$ .  $\square$

**Lema 4** *Al terminar el algoritmo, cada nodo conoce todos los caminos del WFG que se inician en todos sus sucesores.*

**Demostración** Por el Lema 3, cada nodo conoce exactamente todos los caminos del WFG que se inician en él. Por el paso (2.3), estos caminos se propagan hacia los antecesores hasta que todos los antecesores conocen todos los caminos que se inician en alguno de sus sucesores en el grafo. Si el WFG cambia, por los pasos (2.2) y (2.4) se eliminan los caminos necesarios y se propagan los cambios hacia los antecesores.  $\square$

**Lema 5** *Al terminar el algoritmo, cada nodo sabe si está o no en bloqueo mutuo.*

**Demostración** De los Lemas 3 y 4 se puede concluir que el nodo  $i$  conoce todos los caminos que involucran a sus sucesores y a él mismo. Si el predicado del paso (3.1) es verdadero el nodo  $i$  sabe que se encuentra en un nudo. Por la definición de bloqueo mutuo en el modelo OR, la existencia de un nudo es una condición suficiente para la existencia de un bloqueo mutuo. Por esto, el nodo que pertenece al nudo sabe que está en bloqueo mutuo.

Por el paso (3.2), los nodos tales que todos sus sucesores se encuentran en bloqueo mutuo, también actualizan el valor de su variable local *deadlock* a *true*.

Así, si el algoritmo termina, todos los procesos involucrados en un bloqueo mutuo tienen sus variables *deadlock* con valor *true*.  $\square$

Se ha demostrado que el algoritmo detecta todos los bloqueos mutuos. A continuación se demuestra que el algoritmo no detecta bloqueos mutuos falsos.

**Lema 6** *El algoritmo no detecta bloqueos mutuos falsos.*

**Demostración** Si un proceso tiene su variable *deadlock* con el valor *true*, significa que se ha ejecutado al menos una vez el paso (3.1) o el paso (3.2). Para que el paso (3.2) sea ejecutado, es necesario que otro proceso haya ejecutado el paso (3.1) antes. Si un proceso ejecuta el paso (3.1) es porque existe un nudo en el WFG. Si no existe un nudo en el grafo, entonces el guardia del paso (3.1) nunca es verdadero por los Lemas 3 y 4, y la movida nunca se ejecuta. Si el paso (3.1) no se ejecuta por ningún proceso, tampoco puede ejecutarse el paso (3.2), pues no hay procesos con su variable *deadlock* en *true*.

Luego, no es posible que un proceso tenga su variable *deadlock* en *true* si no existe un nudo en el sistema, por lo que el algoritmo no detecta bloqueos mutuos falsos.  $\square$

**Lema 7** *El algoritmo termina.*

**Demostración** Se considera dos casos:

- El nodo que ejecuta el algoritmo no se encuentra en bloqueo mutuo. Esto significa que el nodo eventualmente recibirá alguno de los recursos por los que espera. Cuando este evento ocurra, se ejecuta el paso (3.3) y el algoritmo termina.
- El nodo que ejecuta el algoritmo se encuentra en bloqueo mutuo. En este caso, el nodo puede o no formar parte de un nudo en el grafo. Como todos los procesos que se encuentran en un nudo están bloqueados, no pueden agregarse más arcos salientes de cada nodo en el nudo. Por lo tanto, tras un número finito de ejecuciones del paso (1.2), los guardias de los pasos (1.2) y (1.3) no serán verdaderos para ningún nodo en el nudo. Lo mismo ocurrirá con los guardias de los pasos de las fases 2 y 3. Finalmente, no habrá más movidas que hacer y el algoritmo terminará.

Si el nodo no pertenece a un nudo pero está en bloqueo mutuo es porque todos sus sucesores están bloqueados. En este caso, tampoco pueden agregarse nuevos sucesores ni caminos, por lo que tras un número finito de ejecuciones de los pasos del algoritmo, los guardias dejarán de ser verdaderos y el algoritmo terminará.  $\square$

De todos los lemas anteriores se concluye el siguiente teorema.

**Teorema 1** *El algoritmo es completo y correcto.*

## 5 Resolución de bloqueos mutuos

Para resolver un bloqueo mutuo es necesario eliminar uno de los procesos que participan en él. Una vez detectado el bloqueo, es necesario escoger una víctima para ser eliminada. Sin embargo,



la eliminación de un proceso cualquiera no necesariamente resuelve el bloqueo. Considere la Figura 1(b). Si se elimina al proceso 5, los demás procesos siguen en bloqueo mutuo pues aún hay un nudo en el grafo. Es necesario escoger uno de los procesos que forman parte del nudo.

Una vez terminado el algoritmo, cada proceso sabe si está o no en bloqueo mutuo. Adicionalmente, el algoritmo puede ser modificado ligeramente, de manera tal que el proceso sepa, además, si forma parte de un nudo. Esto puede lograrse agregando una variable booleana *knot*, la que debe ser inicializada en *false* en el paso (1.1). En el paso (3.1), si el predicado es verdadero, entonces la variable debe ser actualizada en *true*, pues sólo en este paso el proceso puede determinar que forma parte de un nudo. Finalmente, en el paso (3.3), la variable debe tomar el valor *false*.

Los procesos que al terminar el algoritmo saben que forman parte de un nudo, pueden iniciar un algoritmo para escoger una víctima tal que, al ser eliminada, se resuelva el bloqueo mutuo. Por ejemplo, un algoritmo de elección de líderes como el descrito por Ghosh y Gupta es suficiente [5].

## 6 Comentarios finales

En este artículo se ha presentado un algoritmo de detección de bloqueos mutuos para el modelo OR de requerimientos. El algoritmo posee las siguientes características:

- Es dinámico, ya que soporta cambios en el WFG tales como la inclusión o eliminación de nodos durante la ejecución del algoritmo.
- Es correcto, pues se ha demostrado que no detecta bloqueos mutuos falsos.
- Es completo, porque también se ha demostrado que detecta todos los bloqueos mutuos.

Adicionalmente, se ha discutido ligeras modificaciones al algoritmo que permiten resolver bloqueos mutuos una vez identificados los nodos que participan en él.

Desde que Dijkstra introdujo el concepto de autoestabilización en el año 1974 [3], se han propuesto algoritmos autoestabilizantes para resolver muchos problemas en sistemas distribuidos. Entre los problemas clásicos, se encuentran los de exclusión mutua y elección de líderes. Schneider ha escrito un completo *survey* sobre el tema [12].

En términos generales, un sistema se dice autoestabilizante si, independientemente de si su estado global inicial es legal o ilegal, es capaz de alcanzar un estado final global legal en un número finito de pasos [3]. Se entiende por estado global al producto cartesiano de los estados locales de todos los procesadores en el sistema. La definición de estado global legal o ilegal depende en gran medida del contexto del problema que se trate.

La habilidad de recuperar un estado global legítimo que estos sistemas presentan, los hacen capaces de soportar fallas transientes. En general, se entiende por falla transiente a aquella falla que ocurre una vez y no vuelve a repetirse.

Algunos algoritmos autoestabilizantes para resolver el problema de la detección de bloqueos mutuos han sido propuestos [4, 9]. Sin embargo, las definiciones clásicas de algunos de los conceptos utilizados en algoritmos autoestabilizantes no son claramente aplicables en este contexto. La principal dificultad está en la definición de estados globales legales e ilegales. Además, tampoco existe una definición satisfactoria de fallas transientes.

Debido a lo anterior, el paso siguiente es definir adecuadamente los conceptos previamente planteados, con el objetivo de presentar un algoritmo que soporte fallas transientes, y que sea autoestabilizante.

## Referencias

- [1] Gabriel Bracha and Sam Toueg. A distributed algorithm for generalized deadlock detection. In *Symposium on Principles of Distributed Computing*, pages 285–301, Vancouver, British Columbia, Canada, August 1984.
- [2] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.
- [3] Edsger Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
- [4] Mitchell Flatebo and Ajoy Kumar Datta. Self-stabilizing deadlock detection algorithms. In *Proceedings of the 1992 ACM Annual Conference on Communications*, pages 117–122, Kansas City, Missouri, April 1992.
- [5] Sukumar Ghosh and Arobinda Gupta. An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, September 1996.
- [6] José R. González de Mendiivil, Federico Fariña, José R. Garitagoitia, C. F. Alastruey, and J. M. Bernabeu-Auban. A distributed deadlock resolution algorithm for the AND model. *IEEE Transactions on Parallel and Distributed Systems*, 10(5):433–447, May 1999.
- [7] T. Hermann and K. Chandy. A distributed procedure to detect AND/OR deadlock. Technical Report TR LCS-8301, Department of Computer Science, University of Texas, Austin, Texas, 1983.
- [8] Richard C. Holt. Some deadlock properties on computer systems. *ACM Computing Surveys*, 4(3):179–196, September 1972.
- [9] Mehmet H. Karaata and Jeffery C. Line. Self-stabilizing algorithms for deadlock detection and identification in distributed systems. In *Proceedings of the ISCA Thirteenth International Conference on Parallel and Distributed Computing*, pages 320–325, Las Vegas, Nevada, August 2000.
- [10] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, December 1987.
- [11] N. Natarajan. A distributed scheme for detecting communication deadlocks. *IEEE Transactions on Software Engineering*, SE-12(4):531–537, April 1986.
- [12] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.
- [13] Loren Schwiebert. Deadlock-free oblivious wormhole routing with cyclic dependencies. *IEEE Transactions on Computers*, 50(9):865–876, September 2001.
- [14] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied Operating System Concepts*. John Wiley & Sons, New York, NY, first edition, 2000.