Self-stabilizing Deadlock Detection Under the OR Requirement Model

Christian F. Orellana[1,2], Cristian Ruz[1], and Yadran Eterovic S.[2]

[1] Escuela de Ingeniería Informática, Universidad Diego Portales
cristian.ruz@udp.cl
[2] Depto. de Ciencia de la Computación, Pontificia Universidad Católica de Chile
{cforella, yadran}@ing.puc.cl

Abstract. This article introduces a self-stabilizing deadlock-detection algorithm for the OR model. The algorithm is complete, because it detects all deadlocks, and it is correct, because it does not detect false deadlocks. Because of the self-stabilization property, the algorithm supports dynamic changes in the wait-for graph on which it works, and transient faults; also, it can be started in an arbitrary state. Previous deadlock-detection algorithms for the OR model are not guaranteed to recover from transient faults, nor can they be started in an arbitrary state. Once the algorithm terminates, each process knows if it is or not deadlocked; moreover, deadlocked processes know whether they cause or only suffer from deadlock.

1 Introduction

One of the main motivations to build distributed systems is the possibility of sharing resources among several processes. A process can acquire and release resources in a sequence that is unknown beforehand. The deadlock problem arises in this setting; being able to detect deadlocks is the first step to take actions and resolve them. A set of processes is said to be deadlocked when each process in the set is blocked, waiting for resources assigned to other processes in the same set. The presence of a deadlock is a stable property of a system; once a set of processes becomes deadlocked, it will remain in that state unless a resolution action is taken.

Knapp classified the deadlock-detection problem in six models, according to the type of requirements a process can make [1]; for most models, deadlock-detection algorithms have been proposed. Under the single-outstanding-request model, a process can request only one resource at a time [2]. Under the AND model, a process can request multiple resources simultaneously; requirements are satisfied when all the requested resources are assigned [2,3]. Under the OR model, a process also can request multiple resources simultaneously, but requirements are satisfied when any of the requested resources is assigned [4,5,6,7]. Under the AND/OR model, a process can request any number of resources in an arbitrary combination of AND and OR requirements [8]. Under the n-out-of-k model, a

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 559–570, 2006.

requirement for n resources is satisfied when k of them are assigned [9]. Under the unrestricted model, no assumption is made about the way in which a process makes its requirements.

Since Dijkstra introduced the concept of self-stabilization in 1974 [10], several self-stabilizing algorithms have been proposed, to solve many problems in distributed systems. Mutual exclusion and leader election are among the classical problems solved with this approach. Schneider wrote an early survey on the subject [11]. In general, a system is said to be self-stabilizing if, regardless of its initial global state, it reaches a legitimate global state in a finite number of steps [10]. The global state of a distributed system is the cartesian product of the local states of every process in the system. The definition of legitimate and illegitimate global states depends on the context of the problem being solved. The ability of regaining a legitimate global state that these systems present, makes them able to support transient faults. A transient fault is one that occurs once, and ceases to occur. Furthermore, self-stabilizing systems can be started in an arbitrary global state, even illegitimate ones, since they will reach a legitimate state nonetheless.

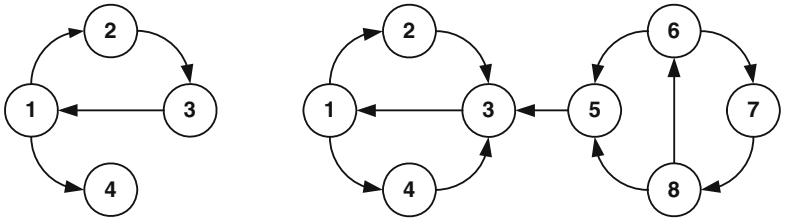
The dynamic nature of resource competition, in which processes are involved in a distributed system, makes the deadlock-detection problem suitable to be treated from a self-stabilizing perspective. In addition, transient-fault tolerance is a desirable property for a distributed deadlock-detection algorithm.

2 The OR Model

This article presents a self-stabilizing deadlock-detection algorithm for the OR requirement model. A process can make an OR request, for example, in a replicated distributed database system, where a read request for a replicated element is satisfied when any copy is read [1]. Also, in a store-and-forward communications network, packets can be forwarded whenever any buffer at the destination node is free [5]. In a similar way, in a message-routing system based on wormhole routing, a router can forward a received message to a neighbor router through one of several channels [12]; a requirement for an output channel is satisfied when any of them becomes available.

A useful way to represent resource requirements is by means of a directed graph, known as *Wait-For Graph* (WFG). In a WFG, each node represents a process in the system. Nodes with outgoing edges represent blocked processes, waiting for resources. On the contrary, nodes without outgoing edges represent active processes. An edge from node i to node j means that process i is waiting for a resource assigned to process j. In general, the deadlock-detection problem can be reduced to that of detecting cyclic structures on this graph. For example, the presence of a directed cycle in the WFG is a necessary and sufficient condition for the existence of deadlock under the AND model [1]. In Fig. 1(a), processes 1, 2, and 3 form a cycle, and are deadlocked.

Under the OR requirement model, the presence of a cycle in the WFG is a necessary — but not sufficient — condition for a deadlock to exist. If the edges



(a) AND model (b) OR model

Fig. 1. Examples of deadlock. (a) Processes 1, 2, and 3 form a cycle, and are deadlocked under the AND model. (b) Processes 1, 2, 3, and 4 form a knot, and are deadlocked under the OR model. Processes 5, 6, 7, and 8, only suffer from deadlock.

represent OR requirements, there is no deadlock in Fig. 1(a), in spite of the cycle, because process 1 is waiting for the resource assigned to process 2 *or* the resource assigned to process 4.

Under the OR requirement model, a process is *blocked* if it has a pending OR requirement. A set of processes is *deadlocked*, if they form a *tie* in the WFG, and all of them are blocked. A tie in a graph is a set of nodes with no directed edges going to nodes outside the set. Another important notion is that of a *knot* in the WFG [13]. A node v is in a knot, if all nodes that are reachable from v by a directed path, can reach node v by a directed path; in that case, the knot is the set of nodes that are reachable from v. That is, a knot is a strongly connected component; moreover, a knot is a tie of blocked processes of which any subset is not a tie. Also, any tie of blocked processes contains at least one knot [5]; there is a path from every node in that tie to at least one knot.

Under the OR requirement model, deadlocked processes can be sorted into two groups. A process *suffers* from deadlock if it is in a tie. A process *causes* a deadlock if it is in a knot. According to these definitions, a process that causes a deadlock also suffers from deadlock. For example, in Fig. 1(b), all processes form a tie of blocked processes. Processes 1, 2, 3, and 4 form a knot, they are deadlocked, and they all cause deadlock. Processes 5, 6, 7, and 8, on the other hand, are not in a knot; they do not cause deadlock, but they are deadlocked nonetheless; they *only suffer* from deadlock. The distinction is important when trying to resolve the deadlock. In order to resolve all deadlocks, a process from each knot must be terminated; it would not help to kill processes that only suffer from deadlock.

Chandy, Misra, and Haas have proposed an algorithm to detect deadlocks under the OR model, based on the technique known as *diffusing computations* [4]. In their proposal, a process starts the algorithm when a request is not granted. Upon termination, a process is guaranteed to know that it is deadlocked only if it was deadlocked when the algorithm started. Nonetheless, in a set of deadlocked processes, at least one of them is able to report it. Cidon, Jaffe, and Sidi [5] proposed an algorithm based on detecting cycles of connected components, which they call *clusters*, and merging them into bigger clusters until a knot is found.

All processes that cause deadlock are detected. In the algorithm proposed by Lee and Lee [6], the *initiator* builds a reduced WFG locally, through receiving the paths from its successors. The initiator uses this graph to decide whether there is deadlock or not. The algorithm proposed by Natarajan [7] is based on the same principle as the one by Chandy, Misra, and Haas, but uses a periodic protocol that allows the choice of exactly one process from a deadlocked set of processes to report the deadlock. Some of these algorithms are dynamic, because they support changes in the WFG; however, they are not guaranteed to recover from transient faults, nor can they be started in an arbitrary state.

In the algorithm proposed in this paper, processes gather enough information about their successors to detect deadlocks. A process that is not deadlocked when the algorithm starts, but becomes deadlocked later, is able to report the deadlock. Thus, in a set of deadlocked processes, every process is able to report it. Additionally, each deadlocked process can decide if it is deadlocked because it is part of a knot, or because it only suffers from deadlock. The algorithm supports dynamic changes in the WFG; furthermore, it supports transient faults and can be started in an arbitrary state.

3 Self-stabilization

In a distributed system, processes are connected to each other according to some underlying network topology, which may be defined by virtual connections on top of a transport protocol. Each process has its own set of local variables, and can communicate with any other process through those connections. The local variables define the state of a process. A process might decide to change its local state depending on its current state and the state of some other processes. In a distributed system, a process can learn the state of other processes through message passing. The ability to change state is called a *privilege*; a process that has a privilege is called a *privileged* process. In a *step*, a privileged process changes its local state.

The cartesian product of the local states of every process defines the global state of the system. Global states can be sorted into two sets: legitimate and illegitimate. A self-stabilizing system converges in a finite number of steps to a legitimate global state, regardless of whether its initial global state is legitimate or not. It is because of this property that self-stabilizing systems can support transient faults. A transient fault is one that changes the local state but not the behavior of a process, and does not continue to occur. Even if a transient fault puts the system in an illegitimate global state, the system will eventually regain a legitimate global state. In addition, it is not necessary to define an initial global state, that is, local variables can be initialized arbitrarily.

In the system defined by the algorithm proposed in this paper, legitimate global states are characterized by the absence of privileges, and by the fact that a process decides that it is deadlocked if and only if it is really deadlocked. In a legitimate global state, every resource request that is not granted and every release of a resource pushes the system into an illegitimate global state, because new privileges appear in the system every time the WFG changes.

Flatebo and Datta [2], and Karaata and Line [3], have proposed self-stabilizing algorithms to solve the deadlock-detection problem under the AND model. Deadlocks are detected by finding cycles in the WFG; each node propagates the information about its predecessor nodes to its successors [3], or the information about its successors to its predecessors [2]. If a predecessor of a node is also a successor of the node, or viceversa, then there is a cycle in the WFG and a deadlock in the system. In both proposals, a global state is legitimate when a process knows that it is deadlocked if and only if it is deadlocked. In both proposals, a change in the WFG puts the system in an illegitimate state; the algorithms support changes produced by the processes that share resources. Moreover, the algorithm proposed by Karaata and Line [3] supports transient faults and arbitrary initialization; on the other hand, the algorithm proposed by Flatebo and Datta [2] does not.

Schneider provided a formalism to prove that a system is self-stabilizing with respect to a predicate over the global states of the system [11]. This state predicate identifies the correct operation of the system, by defining legitimate states. Every state that satisfies the predicate is legitimate, and states that do not satisfy the predicate are illegitimate. According to Schneider, a system is self-stabilizing with respect to a state predicate P, if it satisfies two properties: *closure* and *convergence*. The closure property says that, once the system reaches a state satisfying P, it cannot reach an illegitimate state through execution of the program. The convergence property says that, starting from an arbitrary global state, the system will reach a state satisfying P in a finite number of steps. In this paper, this formalism is used to prove the property of self-stabilization.

4 Self-stabilizing Deadlock Detection

The proposed algorithm is shown in Fig. 2.

Processes make requests for a resource to a distributed component called *resource allocator*. Whenever a resource allocator receives a request, the resource is assigned locally if it is available. In the other case, the request can not be satisfied.

The algorithm starts at a process, when a request is not granted. The requesting process blocks, and control is transferred to a thread that runs the detection algorithm. These threads maintain exact, up-to-date information about their neighbors in the WFG. The set of neighbors of a node v changes when one of them releases a resource, which is then reallocated to some waiting node. If it is reallocated to v, v is no longer blocked; otherwise, it has a different set of neighbors. The resource allocator can inform the detection-algorithm thread of these changes through atomic updates of the local variables *Succ* and *Pred*. No other event can change the set of neighbors, since the process is blocked.

4.1 Variables

Each process mantains eight local variables when executing the algorithm: *Succ*, *Pred*, *Succ**, *Pred**, *Deadlocked**, *Knot*, *Tie*, and *deadlocked*, which it can read

For node i:

(0.1) **if** $Succ = \emptyset \wedge (Succ^* \neq \emptyset \vee Deadlocked^* \neq \emptyset \vee Knot \neq \emptyset \vee Tie \neq \emptyset)$
then $Succ^* := \emptyset; Deadlocked^* := \emptyset; Knot := \emptyset; Tie := \emptyset$
(1.1) **if** $Succ^* \neq (\cup_{j \in Succ} Succ_j^*) \cup Succ$
then $Succ^* := (\cup_{j \in Succ} Succ_j^*) \cup Succ$
(1.2) **if** $Pred^* \neq (\cup_{j \in Pred} Pred_j^*) \cup Pred$
then $Pred^* := (\cup_{j \in Pred} Pred_j^*) \cup Pred$
(2.1) **if** $Succ^* \neq \emptyset \wedge Knot \neq \{i\} \wedge Succ^* \subseteq Pred^*$
then $Knot := \{i\}$
(2.2) **if** $Knot \neq \emptyset \wedge Succ^* \not\subseteq Pred^*$
then $Knot := \emptyset$
(2.3) **if** $Succ^* \neq \emptyset \wedge Tie \neq \{i\} \wedge Succ^* \subseteq (Deadlocked^* \cup Pred^*)$
then $Tie := \{i\}$
(2.4) **if** $Tie \neq \emptyset \wedge Succ^* \not\subseteq (Deadlocked^* \cup Pred^*)$
then $Tie := \emptyset$
(2.5) **if** $Succ^* \neq \emptyset \ \wedge Deadlocked^* \neq ((\cup_{j \in Succ} Deadlocked_j^*) - \{i\}) \cup Knot \cup Tie$
then $Deadlocked^* := ((\cup_{j \in Succ} Deadlocked_j^*) - \{i\}) \cup Knot \cup Tie$
(3.1) **if** $Succ^* \neq \emptyset \wedge (Succ^* \subseteq Deadlocked^*) \neq deadlocked$
then $deadlocked := (Succ^* \subseteq Deadlocked^*)$

Fig. 2. The deadlock-detection algorithm

and write. Also, it is assumed that each process has read-only access to the local variables $Succ^*$, $Pred^*$, and $Deadlocked^*$ of its neighbors. Since the algorithm is self-stabilizing, there is no need to set specific initial values for the variables.

Variable *Succ* represents the set of successors of the node i that is executing the algorithm, while variable *Pred* represents the set of its predecessors. Variable $Succ^*$ represents the set of nodes that are reachable from the node i that is executing the algorithm, while variable $Pred^*$ represents the set of nodes that reach node i. Variable $Deadlocked^*$ represent the set of reachable nodes that are probably deadlocked. Variables *Knot* and *Tie* are sets, and by execution of the algorithm can get two values: empty, or the identifier of the node that is executing the algorithm. Boolean variable *deadlocked* indicate whether the process that is executing the algorithm is deadlocked or not.

Transient faults can change the value of any variable but *Succ* and *Pred*, which are kept up to date by the resource allocator, and represent the view that a node has of the local connections on the WFG.

4.2 Notation

Each step of the algorithm is written as a guarded command. The guard is a predicate over the variables that the process can read: its own local variables and the ones from its neighbors. If the predicate is true, then there is a privilege in the system, and it is possible to execute the associated action. Actions are executed atomically until there are no more true guards at the node, with the non-local variables being read once, before evaluating the guards. When there

are more than one true guard at a node, the action executed is always the one with minor number.

In Fig. 2 variable i represents the identifier of the process that is executing the algorithm. The local variables of neighbor j are represented as $Succ_j^*$, $Pred_j^*$, and $Deadlocked_j^*$.

4.3 The Algorithm

The algorithm begins at a node i when the process blocks waiting for resources and, therefore, it acquires a set of successors. Step (1.1) locally computes the set $Succ^*$, using the information available in variable $Succ$ and the information in variable $Succ^*$ of every successor. Because of this step, any change in the set $Succ$ is reflected in the local variable $Succ^*$, and propagated to predecessors nodes. In a symilar manner, step (1.2) computes the set $Pred^*$ using the information available in variable $Pred$ and the information in variable $Pred^*$ of every successor.

Step (2.1) sets the local variable $Knot$ to a set with i as its only element, when all successors of i are also its predecessors. Step (2.2) sets the local variable $Knot$ to empty when i has at least one successor node that is not a predecessor at the same time.

Step (2.3) sets the local variable Tie to a set with i as its only element, when all successors of i are deadlocked, or can reach i back. If that is not the case, step (2.4) sets the local variable Tie to empty.

Step (2.5) includes in local variable $Deadlocked^*$ the information in variables $Knot$ and Tie, and the information in variables $Deadlocked^*$ of every successor, and propagates this information to predecessor nodes.

Step (3.1) allows a node to decide whether it is deadlocked or not, setting boolean variable *deadlocked* accordingly.

When a blocked process becomes active, step (0.1) reset all variables that depend on $Succ$ back to empty.

5 Properties of the Algorithm

This section proves the main theorem of this paper, which states that the proposed algorithm is complete, correct, and self stabilizing.

Lemma 1. *Once there are no privileges in the system, the following three statements are equivalent:*

1. *There is a path from node i to node j in the WFG*
2. $j \in Succ_i^*$
3. $i \in Pred_j^*$

Proof. (1⇒2) Assume there are no privileges in the system, and there is a path $x_0, x_1, \ldots, x_{n-1}, x_n$ in the WFG, with $x_0 = i$ and $x_n = j$. The local resource allocator ensures that $j \in Succ_{x_{n-1}}$. If $j \notin Succ_{x_{n-1}}^*$ then node x_{n-1} would be

privileged; the guard from step (1.1) would be true. Since there are no privileges in the system, $j \in Succ^*_{x_{n-1}}$. Following the same reasoning, $j \in Succ^*_{x_{n-2}}$, or else x_{n-2} would be privileged. The same is true for all nodes in the path, including i.

(1⇒3) The proof is similar to the one given for (1⇒2). The local resource allocator ensures that $i \in Pred_{x_1}$, so $i \in Pred^*_{x_1}$ or else x_1 would be privileged. The same is true for all nodes in the path, including j.

(2⇒1) Let $j \in Succ^*_i$. Then, $j \in Succ_i$ or $j \in Succ^*_k$ for some $k \in Succ_i$, or else i would be privileged. If $j \in Succ_i$ then there is a path of length 1 from i to j in the WFG. Otherwise, if $j \in Succ^*_k$ then $j \in Succ_k$ or $j \in Succ^*_{k'}$, for some $k' \in Succ_k$. If $j \in Succ_k$ then there is a path of length 1 from k to j, and a path of length 2 from i to j. When there is a node m such that $j \in Succ_m$, it is possible to find a path from i to j in the WFG. Note that there is always a node m such that $j \in Succ_m$, or else j would never be included in a variable $Succ^*$.

(3⇒1) The proof is similar to the one given for (2⇒1). □

Lemma 2. *Once there are no privileges in the system, if node i causes deadlock then $deadlocked_i = true$.*

Proof. If node i causes deadlock, then it is in a knot. All nodes that are reachable from i by a directed path in the WFG are in variable $Succ^*_i$ (by Lemma 1). All nodes that reach i by a directed path in the WFG are in variable $Pred^*_i$ (by Lemma 1). Since i is in a knot, all reachable nodes from i can reach i back. Then $Succ^*_i \subseteq Pred^*_i$ and, after one execution of step (2.1), $Knot_i = \{i\}$. This is also true for all nodes in the knot, that is, $Knot_j = \{j\}$ for all j in $Succ^*_i$.

Because of step (2.5), $j \in Deadlocked^*_j$ for all nodes j such $Knot_j = \{j\}$. The information that each node keeps in variable $Deadlocked^*$ is propagated backwards in the graph in step (2.5), just like the information in variable $Succ^*$ in step (1.1).

Once there are no privileges in the system, all nodes in $Succ^*_i$ are also in $Deadlocked^*_i$. Therefore, $Succ^*_i \subseteq Deadlocked^*_i$ and, after one execution of step (3.1), variable $deadlocked_i = true$. □

Theorem 1 (Completeness). *Once there are no privileges in the system, if node i suffers from deadlock then $deadlocked_i = true$.*

Proof. If node i suffers from deadlock, then it is in a tie of blocked processes in the WFG. Let d_{ik} be the length of the longest simple path from i to a reachable knot k that does not include edges in k. There is at least one reachable knot. Let d_i be the maximum d_{ik} over all k. If $d_i = 0$ then i belongs to a knot and $deadlocked_i = true$ by Lemma 2. If $d_i = n > 0$ then for all successors v of i, $d_v < n$ or there is a path from v to i. For if $d_v \geq n$ and there is no path from v to i, there would be a longer path from i to a knot through v, and d_i would be strictly larger than n. Inductively, if $d_v < n$ then $deadlocked_v = true$ and $v \in Deadlocked^*_v$. Because of step (2.5), the information in variable $Deadlocked^*$ is propagated backwards in the WFG so, in time, $v \in Deadlocked^*_i$. If $d_v \geq n$, then $v \in Pred^*_i$ by Lemma 1. Hence, at some time, every successor v of i belonged

to $Deadlocked_i^*$ or $v \in Pred_i^*$. Thus, the guard of step (2.3) had to be true and after the execution of the step, $Tie_i = \{i\}$. This is true for all nodes in the tie.

Since all nodes j in $Succ_i^*$ are in the tie, $Tie_j = \{j\}$. Because of step (2.5), $j \in Deadlocked_j^*$ and, in time, $j \in Deadlocked_i^*$. Therefore, $Succ_i^* \subseteq Deadlocked_i^*$ and, after one execution of step (3.1), variable $deadlocked_i = true$. □

Theorem 2 (Correctness). *Once there are no privileges in the system, if $deadlocked_i = true$ then node i suffers from deadlock.*

Proof. Let $deadlocked_i = true$ and suppose that i does not suffer from deadlock. Since i does not suffer from deadlock, then it reaches a node j that is not blocked. Thus $j \in Succ_i^*$ (by Lemma 1). Since j is not blocked, $Succ_j = \emptyset$. Because of step (0.1), $Deadlocked_j^* = \emptyset$, $Knot_j = \emptyset$ and $Tie_j = \emptyset$. Because of step (2.5), no other node appart from j can include j in its own variable $Deadlocked^*$, and j can not execute step (2.5) because $Succ_j^* = \emptyset$. Thus, j can not be included in variable $Deadlocked^*$ at any node, in particular i. Then, $j \notin Deadlocked_i^*$ and $j \in Succ_i^*$, that is $Succ_i^* \not\subseteq Deadlocked_i^*$. Because of step (3.1), variable $deadlocked$ can not be true, or i would be privileged. Since there are no privileges, $deadlocked_i$ must be false, leading to a contradiction. □

Lemma 3. *A privileged node looses its privilege in a finite number of steps.*

Proof. A privileged node has at least one true guard. After the execution of one step, the guard associated to that step becomes false.

If the execution of a step could make true guards associated to later steps then, in the worst case, each step will be executed once and, eventually, the privilege will be lost.

In the proposed algorithm all steps can only make true guards associated to later steps. The only exception is step (2.5) which could also make true the guard of step (2.3) or the guard of step (2.4). Note that the guards of steps (2.3) and (2.4) can not be both true at the same time.

If after one execution of step (2.5) the guard of step (2.3) becomes true, then $Succ^* \subseteq Deadlocked^* \cup Pred^*$. After the execution of step (2.3), $Tie = \{i\}$ and the guard of step (2.5) could become true again. If step (2.5) is executed again, the value of variable Tie does not change, and variable $Deadlocked^*$ now includes i. The guard of step (2.3) can not become true again, because variable Tie has not changed. The guard of step (2.4) can not become true because $Succ \subseteq Deadlocked^* \cup Pred^*$ still holds.

On the other hand, if after one execution of step (2.5) the guard of step (2.4) becomes true, then $Succ^* \not\subseteq Deadlocked^* \cup Pred^*$. After the execution of step (2.4), $Tie = \emptyset$ and the guard of step (2.5) could become true again. If step (2.5) is executed again, the value of variable Tie does not change, and variable $Deadlocked^*$ now does not includes i. The guard of step (2.4) can not become true again, because variable Tie has not changed. The guard of step (2.3) can not become true because $Succ \not\subseteq Deadlocked^* \cup Pred^*$ still holds.

Since the execution of one step can only make true a finite number of guards, then a privileged node looses its privilege after a finite number of steps. □

When a privileged node changes its state and produces privileges in neighbour nodes, it is called a *propagation of privilege* in this paper.

A node can propagate privileges only when it changes its local variables $Deadlocked^*$, $Succ^*$, and $Pred^*$. These variables can change initially when the WFG is modified.

Changes in variables $Deadlocked^*$ and $Succ^*$ can only propagate privileges to predecessor nodes; changes in variable $Pred^*$ can only propagate privileges to successor nodes. These privileges can not be propagated indefinitely, because once a node has received the privilege and updated its local variables, it will not receive the privilege because of the same change again.

A transient fault can generate privileges at a node. These privileges will be used locally (by Lemma 3) and will not be propagated. Thus, the effects of transient faults are always corrected locally by the algorithm.

This observation along with Lemma 3 conclude the following theorem.

Theorem 3 (Extintion of privileges). *Privileges produced in the system are eventually lost.*

An algorithm is said to be self-stabilizing with respect to a state predicate P if it satisfies the properties of closure and convergence, as defined by Schneider [11]. In the system defined by the proposed algorithm, legitimate states are defined by the following predicate:

> *P: There are no privileges in the system and, for every node i, $deadlocked_i = true$ if and only if i forms part of a tie in the WFG.*

The following two lemmas show that P satisfies both the closure and convergence properties.

Lemma 4 (Closure). *Once P is established, it is not falsified by execution of the algorithm.*

Proof. When P becomes true, there are no privileges in the system; therefore, no actions are executed and the state remains the same. Hence P is not falsified. □

Lemma 5 (Convergence). *Starting from an arbitrary initial state, once transient faults cease to occur, the system reaches a global state satisfying P within a finite number of steps.*

Proof. By Theorem 3, privileges eventually disappear from the system. Therefore, the first part of P is satisfied. Once there are no privileges in the system, by Theorems 1 and 2, the second part of P is satisfied. Hence P holds after a finite number of steps, once transient faults cease to occur. □

Lemmas 4 and 5 prove the following theorem.

Theorem 4 (Self-stabilization). *The proposed algorithm is a self-stabilizing deadlock-detection algorithm under the OR requirement model.*

6 Deadlock Resolution

In order to resolve a deadlock, at least one of the deadlocked processes must be terminated. Therefore, once a deadlock is detected, it becomes necessary to choose a victim to terminate. Terminating just any process does not necessarily resolve the deadlock. In Fig. 1(b), if process 5 were terminated, there would still be deadlock, because the knot in the WFG remains. To resolve a deadlock, it is not enough to kill a process that only suffers from deadlock; it is necessary to terminate one process from each knot.

Once there are no privileges in the system, each process knows whether it is deadlocked or not. And, in addition, deadlocked processes also know whether they are part of a knot or not. Variables *Knot* and *Tie* compute precisely that information. Processes that are part of a knot can start an algorithm to choose a victim such that, when terminated, the knot disappears.

Processes that are part of a knot have the same set of successors, formed by all processes in the knot. Thus, if all the nodes in the knot apply a rule —like $victim = \min(Succ^*)$— it is possible to choose exactly one victim to terminate from each knot.

No special actions need to be taken once a deadlock has been resolved. The immediate predecessors of the terminated processes would see a change in their variable *Succ*, and the detection algorithm would recompute for the new WFG.

7 Concluding Remarks

This article presents a self-stabilizing deadlock-detection algorithm for the OR requirement model. The algorithm is self-stabilizing, that is, it supports changes to the WFG, transient faults, and arbitrary initialization; previous algorithms for the OR model are not guaranteed to recover from transient faults or arbitrary initialization. The algorithm is complete and correct since it detects all deadlocks and it detects no false deadlocks, respectively. Hence, every process knows whether it is deadlocked or not and, moreover, deadlocked processes know whether they cause or only suffer from deadlock. In addition, the algorithm provides enough local information to implement actions and resolve the deadlocks detected.

References

1. Knapp, E.: Deadlock detection in distributed databases. ACM Computing Surveys **19**(4) (1987) 303–328
2. Flatebo, M., Datta, A.K.: Self-stabilizing deadlock detection algorithms. In: Proceedings of the 1992 ACM Annual Conference on Communications, Kansas City, Missouri (1992) 117–122
3. Karaata, M.H., Line, J.C.: Self-stabilizing algorithms for deadlock detection and identification in distributed systems. In: Proceedings of the ISCA Thirteenth International Conference on Parallel and Distributed Computing, Las Vegas, Nevada (2000) 320–325

4. Chandy, K.M., Misra, J., Haas, L.M.: Distributed deadlock detection. ACM Transactions on Computer Systems **1**(2) (1983) 144–156
5. Cidon, I., Jaffe, J.M., Sidi, M.: Local distributed deadlock detection by knot detection. In: Proceedings of the ACM SIGCOMM Conference on Communications Architecture & Protocols, ACM Press (1986) 377–384
6. Lee, S., Lee, Y.: A distributed algorithm for deadlock detection under OR-request model. In: Proceedings of the 18^{th} IEEE Symposium on Reliable Distributed Systems, IEEE Press (1999) 298–299 Work in Progress, Fast Abstracts.
7. Natarajan, N.: A distributed scheme for detecting communication deadlocks. IEEE Transactions on Software Engineering **SE-12**(4) (1986) 531–537
8. Herman, T., Chandy, K.: A distributed procedure to detect AND/OR deadlock. Technical Report TR LCS-8301, Department of Computer Science, University of Texas, Austin, Texas (1983)
9. Bracha, G., Toueg, S.: A distributed algorithm for generalized deadlock detection. In: Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada (1984) 285–301
10. Dijkstra, E.: Self-stabilizing systems in spite of distributed control. Communications of the ACM **17**(11) (1974) 643–644
11. Schneider, M.: Self-stabilization. ACM Computing Surveys **25**(1) (1993) 45–67
12. Schwiebert, L.: Deadlock-free oblivious wormhole routing with cyclic dependencies. IEEE Transactions on Computers **50**(9) (2001) 865–876
13. Holt, R.C.: Some deadlock properties of computer systems. ACM Computing Surveys **4**(3) (1972) 179–196