A Distributed Algorithm for Generalized Deadlock Detection

Gabriel Bracha
Sam Toueg*

TR 83-558
June 1983
(Revised March 1984)
(Revised July 1984)

Department of Computer Science
Cornell University
Ithaca, New York 14853

*Partial support for this work was provided by the National Science Foundation under grant No. 83-03135.

A Distributed Algorithm for Generalized Deadlock Detection †

Gabriel Bracha

Sam Toueg

Cornell University
Ithaca, New York 14853

Abstract

An efficient distributed algorithm to detect deadlocks in distributed and dynamically changing systems is presented. In our model, processes can request any N available resources from a pool of size M. This is a generalization of the well-known *AND–OR* request model. The algorithm is incrementally derived and proven correct. Its communication, computational, and space complexity compares favorably to those of previously known distributed *AND–OR* deadlock detection algorithms.

1. Introduction

The problem of detecting deadlocks in systems in which processes wait for each other arises in various contexts. In distributed databases, transactions request access to files at different sites. A transaction can proceed only if it gets access to *all* the requested files that are controlled by other sites. In CSP, when executing an alternative command (IF) with input guards, a process waits for messages from a group of processes. A process can exit the statement only if it gets a message from ***any one*** of the processes it is waiting for. In operating systems, processes can request N special resources (disks, tape drives, etc.) out of some pool of size M. A process can proceed only if it gets N of these resources. To maintain consistency in distributed databases with replicated files, Gifford [Giff79] proposed an algorithm where to read (write) a replicated file with k copies, a process must read (write) r (w) out of the k copies such that $r+w>k$. To read or write a file copy, a process must request and obtain a lock on this copy. Therefore, reading (writing) a file generates "r*-out-of-*k" (w*-out-of-*k) locking requests.

In the most general case, a process can make requests described by formulae with *AND*, *OR* and N*-out-of-*M connectives‡ For example, a process p can request services from processes $p_1, \ldots, p_6$ with the following condition to satisfy its request: p_1 *OR* (2*-out-of-*(p_1,p_3,p_5) *AND* 2*-out-of-*(p_2,p_4,p_6)). Process p can proceed only if its requests for service are granted by any combination of processes satisfying the formula.

† Partial support for this work was provided by the National Science Foundation under grant No. 83-03135.

‡ Note that *AND* and *OR* connectives alone can express a N*-out-of-*M request. However, the length of the corresponding *AND–OR* formula is $N \cdot \binom{M}{N}$.

Parsing the formula and the introduction of dummy processes allow us to consider only requests that contains a single connective. Thus, the previous request can be formulated as follows: p issues a (p_1 *OR* q_1) request, q_1 issues a (q_2 *AND* q_3) request, q_2 issues a (2-***out-of***-(p_1,p_3,p_5)) request, and q_3 issues a (2-***out-of***-(p_2,p_4,p_6)) request.

Note that both *AND* and *OR* requests [Chan83a, Herm83] are special cases of the *N*-***out-of***-*M* request. An *OR* request corresponds to $N = 1$, an *AND* request corresponds to $N = M$. Therefore, we can restrict ourselves to systems where processes issue only a single *N*-***out-of***-*M* request.

Using Wait-For-Graphs (WFG) [Holt72] to model three systems of increasing complexity, we incrementally derive a distributed deadlock detection algorithm for dynamic systems with *N*-***out-of***-*M* requests, and prove it correct. The algorithm communication, computational, and space complexity compares favorably to those of previously known distributed *AND-OR* deadlock detection algorithms.

A survey of distributed deadlock detection algorithms is given in [Herm83]. In [Chan82], Chandy gives an excellent description of the classical Distributed Database Deadlock problem as defined in [Mena79], and its relation to the deadlock detection algorithm we present here.

The paper is organized as follows: In Section 2, we give an operational description of our system. In Section 3, we present a corresponding formal model, the WFG. In Section 4, we consider a static system with instantaneous message transmission. A corresponding deadlock detection algorithm is described and proven correct. In Section 5, we consider a static system with transmission delays, i.e., with messages "frozen" in transit in the channels, and we model it with a colored WFG. The deadlock detection algorithm is extended to this system. In Section 6, the algorithm is further extended to a dynamic system whose state changes during the algorithm's execution. The algorithm's performance is analyzed in Section 7, and a discussion of the results concludes the paper.

2. System operational description

A distributed system is a collection of processes that communicate by sending messages to each other. Every message sent will be received within some finite time and messages are received in the order sent.

A process can be either ***active*** or ***blocked***. An ***active*** process is one that is not waiting for any other process. Active processes may issue *N*-***out-of***-*M* requests in the following way. When an active process p requires n_p processes to carry out some action on its behalf, it sends *REQUEST* messages to all the processes that can perform this action (this set of processes is denoted by ***dependent***$_p$). It then becomes ***blocked***, and it waits until the action requested is carried out by at least n_p processes in ***dependent***$_p$. Once a process is blocked, it cannot send any further *REQUEST*s.

Only active processes can carry out a requested action. If a process in ***dependent***$_p$ is active, then, within some finite time, it either becomes ***blocked*** or it will carry out p's requested action. In the latter case, it sends a *REPLY* message to p to notify p that the request was granted. When p receives n_p *REPLY* messages, it becomes active again. It then relinquishes the requests made to the

rest of the processes in $dependent_p$ by sending them *RELINQUISH* messages.

3. The Wait-For-Graph model

The global state of the system is modeled by a Wait-For-Graph (WFG). This is a directed graph where the nodes correspond to processes. A directed edge (v, u) corresponds to a process v that issued a request to a process u, for some service, or some resource that u is holding. In our system, this corresponds to a REQUEST that was sent from v to u, such that v has not received a *REPLY* from u, and u has not received a *RELINQUISH* from v. Each node v is labeled by n_v, the number of *REPLY*s that it needs to receive to become active.

We define OUT_v to be the set of nodes u such that v sent a REQUEST to u, and neither v sent a RELINQUISH to u, nor v received a REPLY from u. In other words, OUT_v is the set of nodes v is still waiting for. Note that $0 \le n_v \le |OUT_v|$, and $n_v = 0$ implies $OUT_v = \phi$. We denote by IN_v the set of nodes w such that v received a REQUEST from w, and neither v sent a REPLY to w, nor v received a RELINQUISH from w. In other words, IN_v is the set of nodes that are requesting a service from v, according to v's point of view. Note that the values of IN_v, OUT_v, and n_v are readily available at a node v in any system.

In a WFG, an ***active node*** corresponds to an active process. State changes in the system, resulting from the issuing and granting of requests, are formally modeled through WFG ***transformations***. If applying a transformation s to a WFG G results in G', we denote this by $G \overset{s}{\vdash} G'$. A ***schedule*** σ for G is a sequence of transformations that can be applied to G. Formally, either σ is the null sequence or $\sigma = s\sigma'$, where $G \overset{s}{\vdash} G'$ and σ' is a schedule for G'. We can now give a formal definition of deadlock. A node v is ***deadlocked in a WFG G*** if there is no schedule σ such that $G \overset{\sigma}{\vdash} G'$, and v is active in G'.

In the next sections, we consider two types of WFG to model systems of increasing complexity.

4. Static systems with instantaneous message transmission

4.1. The model

In this section we consider a simple system where *REQUEST*, *REPLY* and *RELINQUISH* messages are instantaneously delivered. A WFG $G=(V, E)$ models a static "snapshot" of the state of this system that contains no messages in the communication channels. Since message transmission is instantaneous, we have $OUT_v = \{ u \mid (v,u) \in E \}$ and $IN_v = \{ w \mid (w,v) \in E \}$. ~~Note that~~ in this case, we have $v \in IN_u$ if and only if $u \in OUT_v$. We now define active nodes and the WFG transformations that model operational progress in such a system.

A node v is ***active*** in G if $n_v = 0$ (this represents a process v with no outstanding requests, i.e., an active process). Let v be any active node in G; the transformations of G are:

1. Adding k outgoing edges to v and setting n_v to some r $(1 \leq r \leq k)$.
2. Deleting an edge (u, v) and decreasing n_u by 1. If $n_u = 0$ then all the outgoing edges of u are deleted.

Note that (1) models v issuing a r-out-of-k request, and (2) models v sending a *REPLY* to a process u. If this *REPLY* satisfies u, then u relinquishes the rest of its requests.

As we remarked in the previous section, the definitions of an active node and of WFG transformations provide a corresponding formal definition of deadlock.

4.2. The deadlock detection algorithm for this system

We present a distributed algorithm to detect deadlock in a WFG $G=(V, E)$ modeling a static system with instantaneous communication. G represents a static "snapshot" of the system, and therefore it does not change during the execution of the algorithm.

The algorithm starts when some node, which we call the ***initiator***, suspects that it is deadlocked.† In order to distinguish between algorithm invocations started by different initiators, all the messages that are sent in the algorithm are tagged with the initiator identity. All the invocations are executed independently. We consider one such invocation and, to simplify the notation, we omit the initiator identity from the messages.

The algorithm consists of two phases: *Notify* - in which processes are notified that a deadlock detection algorithm has started, and *Grant* - in which active processes simulate the granting of requests. All the processes that are made "active" as a result of this also simulate the granting of requests. Deadlocked nodes are those nodes never made "active" by *Grant*. The *Grant* phase is nested within the *Notify* phase. This nesting ensures that the *Notify* phase terminates only after the *Grant* phase is over.

Each node has the local constants *IN*, *OUT* and n. These constants correspond to the underlying static WFG G as defined earlier. Each node also maintains a few local variables. We denote by var_v the local variable var at node v. The subscript is omitted when there is no ambiguity.

Figure 1 presents the algorithm. It is started when a process (the initiator) invokes the *Notify* procedure. It terminates when this procedure call terminates. At termination, the initiator is not deadlocked if and only if $free_{initiator} = true$. The primitive **send**(w, M) sends the message M to node w, and **await**(w, M) waits to receive a message M from node w. Note that **await** is not blocking, i.e., it does not prevent the reception and processing of *NOTIFY* or *GRANT* messages.

4.3. Correctness of the algorithm

The two phases of the deadlock detection algorithm are very similar. In both phases, messages are propagated in a forest-like pattern, from a core set of nodes to the rest of the graph, according to a well-defined criterion. These phases are only instances of an algorithm that we call *Closure*, and that is a generalization of Chan's "echo" algorithm [Chan80]. Studying *Closure* will enable us

† This can happen after a long wait for a request to be satisfied.

```
Initial state of every node v:
    OUT := {u | (v,u)∈E };
    IN := {u | (u,v)∈E };
    notified, free := false, false ;
    #granted := 0;

Notify:procedure
    notified := true;
    for all w∈OUT send(w,NOTIFY);
    if n = 0 → Grant fi
    for all w∈OUT await(w,DONE);

Upon receipt by v of NOTIFY from u:
    if ¬notified → Notify fi
    send(u,DONE);

Grant:procedure
    free := true;
    for all w∈IN send(w,GRANT);
    for all w∈IN await(w,ACK);

Upon receipt by v of GRANT from u:
    #granted := #granted + 1;
    if ¬free and #granted ≥ n → Grant fi
    send(u, ACK);
```

Figure 1. A deadlock detection algorithm for a colorless WFG $G=(V, E)$.

to treat both phases together.

4.3.1. Closures

Consider a directed graph $G=(V,E)$. Let $IN(v)$ denote the set $\{ u \mid (u,v) \in E \}$. Consider a subset of nodes $S \subseteq V$. The ***closure property*** P is a predicate, $P: V \times 2^V \rightarrow \{true, false\}$, such that, for any v, $P(v,\phi)=false$, and if $S_1 \subseteq S_2$, then $P(v,S_1)=true$ implies $P(v,S_2)=true$. The ***closure of S with respect to P in graph G*** † is denoted as $C(S,P)$, and is recursively defined as follows.

† The reference to the underlying graph G will be omitted whenever it is clear from the context in which graph the closure is computed.

$$C(S,P)^0 = S$$

$$C(S,P)^{i+1} = C(S,P)^i \cup \{ v \in V \mid P(v, IN(v) \cap C(S,P)^i) = true \}$$

$$C(S,P) = \bigcup_{all\ i} C(S,P)^i$$

Informally, the closure of S includes S, and all the nodes that are successively added as follows. If a set T of nodes in the closure are the IN neighbors of a node v and $P(v, T) = true$, then v is added to the closure.

In Figure 2, we describe a distributed algorithm to compute the closure $C(S,P)$ in a graph G. The *Closure* algorithm starts when some node in S calls the *Closure* procedure. We require that all the other nodes in S call the *Closure* procedure, either spontaneously, or following the reception of a *NOTIFY* message (as specified in Figure 2). The *Closure* algorithm terminates when all the nodes in S terminate their *Closure* call.

```
Initial state of every node v:
    OUT := { u |(v,u) ∈ E };
    in_C := false;
    ancestors := φ;

Closure:procedure
    in_C := true;
    for all w ∈ OUT send(w,NOTIFY);
    for all w ∈ OUT await(w,DONE);

Upon receipt by v of NOTIFY from u:
    ancestors := ancestors ∪ { u };
    if ¬ in_C and (v ∈ S or P(v, ancestors)) → Closure fi
    send(u,DONE);
```

Figure 2. *Closure*, an algorithm to compute $C(S,P)$ in a graph $G = (V,E)$.

4.3.2. Correctness of the Closure algorithm

In this section, we show that the *Closure* algorithm described in Figure 2 computes $C(S, P)$.

Lemma 1. During an execution of the *Closure* algorithm, a node v calls the *Closure* procedure if and only if $v \in C(S,P)$.

Proof: See Appendix. □

We say that a node *terminates* if its *Closure* call terminates. The *Closure* algorithm terminates if all the nodes in S terminate.

Lemma 2. The *Closure* algorithm terminates. Moreover, it terminates after all the nodes in $C(S, P)$ terminate.

Proof: See Appendix. □

We now consider the communication complexity of *Closure.*

Lemma 3. During the execution of *Closure* on a graph $G=(V, E)$ at most $2|E|$ messages are sent.

Proof: A node sends at most one *NOTIFY* message on any of its outgoing edge, and one *DONE* on any of its incoming edges. □

4.3.3. Deadlock detection as the nesting of two *Closures*

We now show that the deadlock detection algorithm is the nesting of two instances of the *Closure* algorithm. Consider a WFG $G = (V, E)$ and a node denoted *initiator*. Let *ADJ* ("adjacent") be the predicate

$$ADJ(v, D) = \begin{cases} true & |D| \geq 1 \\ false & \text{otherwise} \end{cases}$$

The closure $C(\{initiator\}, ADJ)$ in G is just the set of nodes reachable from *initiator* via outgoing edges in G. To compute $C(\{initiator\}, ADJ)$ the *Closure* algorithm described in Figure 2 can be applied as follows.

The *Closure* procedure and the *in_C* variable are first renamed *Notify* and *notified*, respectively. Now, according to the *Closure* procedure in Figure 2, upon receipt of a *NOTIFY* the following statement

if $\neg notified$ **and** $(v \in \{ initiator \}$ **or** $ADJ(v, ancestors)) \rightarrow Notify$ **fi**

should be executed. However, when the guard is evaluated we must have $|ancestors| \geq 1$. Therefore, $ADJ(v, ancestors) = true$, and the **if** statement can be simplified to

if $\neg notified \rightarrow Notify$ **fi**

The variable *ancestors* can now be removed.

This modified *Closure* program appears in Figure 1 with the statement

if $n = 0 \rightarrow Grant$ **fi**

inserted in the middle of *Notify* procedure. The execution of this program constitutes the *Notify* phase of the deadlock detection algorithm. It is easy to check that Lemma 1 (and its proof) applies to the *Notify* phase even with the inserted **if** statement (provided the execution of this statement does not change any variable of the *Notify* program).

Let *ACTIVE* be the set of active nodes in the set $C(\{initiator\}, ADJ)$ in G, and let *SAT* ("satisfied") be the following predicate

$$SAT(v,D)=\begin{cases} true & |D| \geq n_v \\ false & \text{otherwise} \end{cases}$$

Consider the closure $C(ACTIVE,SAT)$ in the graph $G^T=(V,E^T)$, where $E^T = \{(u,v) \mid (v,u) \in E\}$. We later show that this closure is exactly the set of nodes that are not deadlocked in the WFG G. It can be computed by the *Closure* algorithm as follows.

We first rename the procedure *Closure*, and the variables *in_C* and *OUT* as *Grant*, *free* and *IN*, respectively. We also rename the *NOTIFY* and *DONE* messages as *GRANT* and *ACK*. We then define a variable *#granted* that corresponds to $|ancestors|$. Now, according to the *Closure* algorithm in Figure 2, upon the receipt of a *GRANT* the following statement

$$\textbf{if}\ \neg free\ \textbf{and}\ (v \in ACTIVE\ \textbf{or}\ SAT(v, ancestors)) \rightarrow Grant\ \textbf{fi}$$

should be executed. Note that $SAT(v, ancestors)$ is equivalent to $\#granted \geq n_v$. Moreover, since v received a *GRANT* message through an incoming edge in G^T, then v has at least one outgoing edge in G, so v is not an active node in G. Therefore, $v \in ACTIVE = false$, and the statement can be simplified to

$$\textbf{if}\ \neg free\ \textbf{and}\ \#granted \geq n \rightarrow Grant\ \textbf{fi}\ .$$

This modified *Closure* program is the second part of the algorithm in Figure 1. Its execution constitutes the *Grant* phase of the deadlock detection algorithm, and it computes $C(ACTIVE,SAT)$ in G^T.

4.3.4. Correctness of the deadlock detection algorithm

We first relate the deadlocked nodes in a WFG G to $C(ACTIVE,SAT)$ in G^T

Lemma 4. Let G be a WFG, and v a node in $C(\{initiator\},ADJ)$ in G. Then v is not deadlocked in G, if and only if $v \in C(ACTIVE,SAT)$ in G^T.

Proof: See Appendix. □

We are now able to prove the correctness of our deadlock detection algorithm.

Theorem 1. If an *initiator* starts the deadlock detection algorithm in a WFG G, then the algorithm terminates. Moreover, when the initiator terminates $free_{initiator} = true$ if and only if the initiator is not deadlocked in G.

Proof: The initiator starts the deadlock detection algorithm by calling *Notify*. From our previous discussion of the *Notify* phase, and from Lemma 1, all the nodes in $C(\{initiator\},ADJ)$ in G call the *Notify* procedure, and execute the

$$\textbf{if}\ n=0 \rightarrow Grant\ \textbf{fi}$$

statement. However, only the nodes in ACTIVE call the *Grant* procedure. From our previous discussions, this initiates the *Grant* phase, i.e., the computation of the closure $C(ACTIVE,SAT)$ in G^T.

By Lemma 2, this computation terminates. When a node in *ACTIVE* terminates its ***Grant*** call, it resumes its execution of the *Notify* procedure. By Lemma 2, the initiator terminates its *Notify* call after all the nodes in *ACTIVE* terminate their *Notify* calls. Therefore the initiator terminates after the termination of the *Grant* phase, and by Lemma 2, after the termination of all the *Grant* calls.

By Lemma 1, $v \in C(ACTIVE, SAT)$ in G^T if and only if v calls the *Grant* procedure, i.e., if and only if v sets $free_v$ to ***true***, before the initiator terminates. From Lemma 4, we conclude that when the initiator terminates, a node v in $C(\{initiator\}, ADJ)$ is not deadlocked in G if and only if $free_v = true$. □

5. Static systems with messages in transit

5.1. The model

In this section we consider a system where message passing is not instantaneous. Any attempt to capture the state of the system must take into account the messages in the communication channels. A colored WFG, $G = (V, E)$, models a static "snapshot" of such a system. The messages in the channels are modeled by introducing colors on the edges. An edge (u, v) is:

grey: If u has sent *REQUEST* to v, v has not yet received it, and u has not sent *RELINQUISH* to v.

black: If v has received *REQUEST* from u, v has not sent *REPLY* to u, and u has not sent *RELINQUISH* to v.

white: If v has sent *REPLY* to u, u has not yet received it, and u has not sent *RELINQUISH* to v.

translucent: If u has sent *RELINQUISH* to v, and v has not yet received it.

Remember that every node v knows IN_v and OUT_v. We further assume that every node v knows the colors of all the edges it has to (from) nodes in OUT_v (IN_v). The explicit mechanism that provides the color information will be described later.

In order to map the colored WFG to the colorless WFG, we have to interpret the colored edges, either as existing edges, or as nonexistent edges. We choose to consider the grey, white, and translucent edges as nonexistent. This interpretation yields a very simple algorithm, but it does not conform with our operational notions about grey edges.

For example, consider an isolated cycle of grey edges. Already at that point, one can realize that this is a deadlock situation. But since we choose to consider grey edges as nonexistent, we cannot yet detect that deadlock. However, the only effect of this decision is that in some cases the deadlock will not be detected at the earliest possible time. Within some finite time all the grey edges will turn black, and the deadlock will be detected by the next invocation of the algorithm.

Following our interpretation of colored edges we assume that all the requests corresponding to grey and white edges are granted. Of course if there is a deadlock under this assumption then there

is a deadlock in the colored graph. Let $\#greywhite_v$ denote the number of grey and white edges in OUT_v. In order to reflect our interpretation of colored edges, we have to modify n_v , the number of pending requests, to $n_v - \#greywhite_v$. Thus a node v is *active* if $n_v - \#greywhite_v \leq 0$.

Let v be an active node in a WFG G; the transformations of G are:

1. adding k grey edges to a node v with no outgoing edges, and assigning $n_v := r$ $(1 \leq r \leq k)$.
2. Changing a grey edge into a black one.
3. Changing a black (u,v) edge into white.
4. Removing a white edge (u,v) and decreasing n_u by 1. If $n_u = 0$, then all outgoing edges of u are made translucent.
5. Removing a translucent edge.

These transformations model the issuing and receipt of the *REQUEST* (1. and 2.), *REPLY* (3. and 4.); and *RELINQUISH* (4. and 5.) messages. Note that the new graph transformations together with the new definition of active nodes redefine deadlock.

5.2. The algorithm

An algorithm to detect deadlock in a colored WFG is presented in Figure 3. The algorithm is an adaptation of the algorithm in Figure 1 to a colored WFG that reflects our interpretation of the colored edges. The only changes are as follows: messages are sent only along black edges, $n_v - \#greywhite_v$ is used instead of n_v, and the guards for initiating *Grant* reflect our new definition of an active node. As mentioned before, each node v knows the colors of the edges in IN_v and OUT_v sets.

Theorem 2. If an *initiator* starts the deadlock detection algorithm in a colored WFG G, then the algorithm terminates. Moreover, when the initiator terminates $free_{initiator} = true$ if and only if the initiator is not deadlocked in G.

Proof: The proof is obtained directly from Theorem 1 and our interpretation of the colored edges. □

5.3. Color information

The deadlock detection algorithm assumes that each node v knows the colors of its edges to (from) OUT_v (IN_v). A node has to determine the colors of its edges when it receives the first message of the deadlock-detection algorithm (*NOTIFY* or *GRANT*) in order to evaluate $\#greywhite$. In this section we present the explicit mechanism that achieves that.

A node v can determine the color of its edges by exchanging *COLOR* messages with all the nodes in IN_v and OUT_v. A *COLOR* message sent from v to u tells u whether $u \in IN_v$ and whether $u \in OUT_v$. By exchanging *COLOR* messages over an edge (u, v) two nodes u, v can determine the edge's color as follows. An edge (u,v) is black if $u \in IN_v$ and $v \in OUT_u$. It is translucent if $u \in IN_v$ and $v \notin OUT_u$. If $v \in OUT_u$ and $u \notin IN_v$ it can be either white or grey. Since the colored WFG

Initial state of every node v:

$OUT_{black} := \{u \mid (v,u) \in E \text{ and is black}\}$;
$IN_{black} := \{u \mid (u,v) \in E \text{ and is black}\}$;
$notified, free := false, false$;
$\#granted := 0$;

Notify:**procedure**
$notified := true$;
for all $w \in OUT_{black}$ **send**$(w, NOTIFY)$;
if $n - \#greywhite \leq 0 \rightarrow Grant$ **fi**
for all $w \in OUT_{black}$ **await**$(w, DONE)$;

Upon receipt by v of *NOTIFY* from u:
if $\neg notified \rightarrow Notify$ **fi**
send$(u, DONE)$;

Grant:**procedure**
$free := true$;
for all $w \in IN_{black}$ **send**$(w, GRANT)$;
for all $w \in IN_{black}$ **await**(w, ACK);

Upon receipt by v of *GRANT* from u:
$\#granted := \#granted + 1$;
if $\neg free$ **and** $\#granted \geq n - \#greywhite \rightarrow Grant$ **fi**
send(u, ACK);

Figure 3. A deadlock detection algorithm for a colored WFG $G=(V, E)$.

algorithm uses just *#greywhite*, the nodes are now provided with all the necessary color information.

5.4. Optimization

A node v need not delay the propagation of *NOTIFY* and *GRANT* messages in order to first find the colors of its edges. It first assumes that all its edges are black, and it sends those messages to all the nodes in OUT_v and IN_v. If some node u receives a message from a node that is not in IN_u or OUT_u, it marks the message as a reject, and sends it back to v. There are two possible cases:

1. If a node v receives a rejected message back, it can deduce the color of the edge.

(i) If the edge turns out to be translucent (as a response to a rejected *GRANT*) the node reevaluates the guard for terminating its *Grant* phase.

(ii) If the edge turns out to be grey or white, (as a response to a rejected *NOTIFY*) the node reevaluates *#greywhite* and the guards for starting its *Grant* phase.

2. Otherwise, v receives a *DONE* or *ACK* message back, and it realizes that the edge is black. In either case, v determines the colors of its edges before its *Notify* call terminates. Therefore, the *Grant* phase always terminates before the *Notify* phase, and the optimized algorithm is proven correct as before.

6. A dynamic system

In this section, we consider the "real-life" situation where the WFG is dynamically changing during the execution of the deadlock detection algorithm. This implies that even if the initiator is not deadlocked when the algorithm starts, it can deadlock during the execution of the algorithm. Likewise, the algorithm may decide that the initiator is not deadlocked, but by the time the initiator realizes that, it becomes deadlocked. Therefore, a deadlock detection algorithm in a dynamic system can ensure only the following:

1. If the initiator is deadlocked at the time it invokes the algorithm, a deadlock will be detected.
2. If the algorithm detects a deadlock, then the initiator is deadlocked at the time the algorithm terminates.

The following scheme is used to overcome the changes of the WFG that occur during the execution of the algorithm: Special *FREEZE* messages are propagated throughout the system. When a process receives a *FREEZE* message, it takes a ***snapshot*** of its local state and stores it. The snapshot contains the sets *IN* and *OUT* and n. The distributed deadlock detection algorithm of Figure 3 (that was developed for static systems) is then executed on these fixed snapshots. During this execution, the processes' local states may change.

Because of the time it takes the *FREEZE* to propagate, the collection of snapshots does not describe the state of the system at any particular point in time. Not every collection of snapshots describes a meaningful WFG, and the outcome of the deadlock detection algorithm will not necessarily satisfy the correctness requirements stated above. In the following sections we discuss what constitutes a consistent and valid collection of snapshots, and how to obtain it.

6.1. Consistent states

The notion of consistent states is due to [Chan83b, Chan83c], and the following discussion is a condensed version of it. A distributed system is a collection of processes that send messages to each other according to some underlying algorithm. An ***event occurs at process p*** when p sends or receives a message. The ***local state*** of process p can be represented as the history of all the events that occurred in p. A ***state of the system*** is a collection of the local states of all the processes. We

define *BEFORE*, a partial relation on events, as the closure of following base relation: e_1 *BEFORE* e_2, if

1. both e_1 and e_2 occurred at p, and e_1 occurred before e_2, or
2. e_1 is the sending of a message and e_2 is the receipt of this message.

We represent the progress of the system by a diagram as in Figure 4:

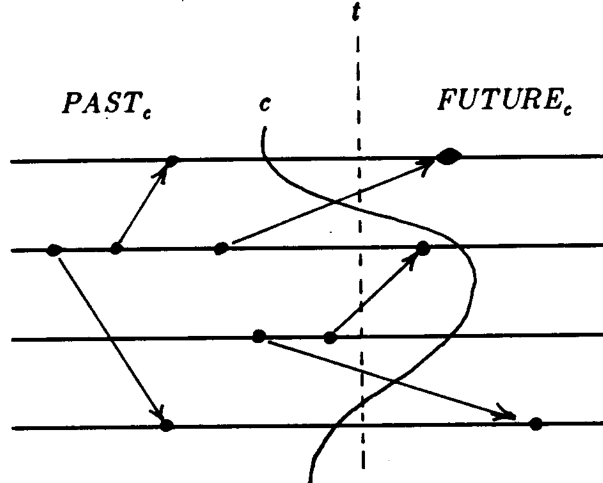

Figure 4. A consistent cut in a distributed system.

The horizontal lines represent the time axis of the processes, the points represent events, and the arrows represent the *BEFORE* relation.

A ***cut*** c is a partition of the graph into two sets of events, $PAST_c$ and $FUTURE_c$. A cut c is ***consistent***, if $FUTURE_c$ is closed under *BEFORE*. A cut defines a state of the system, i.e., the collection of the local states that are represented by the events in $PAST_c$. A consistent cut defines a ***consistent state***. From here on, we refer to cuts and states interchangeably.

A special type of consistent state is S_t, the ***state at time t***, which is the collection of the local state of all the processes at time t. The S_t states are more of a theoretical construct since they require some outside observer to instantaneously capture the local states of all the processes. In contrast, the consistent states can be obtained from within the system by message passing. Consistent states are meaningful because they describe a possible view of the system under different propagation times of the messages, i.e., they are "potential" S_t states.

We can extend the *BEFORE* relation to consistent states. Let S_1 and S_2 be consistent states. We say S_1 *BEFORE* S_2 if $PAST_{S_1} \subseteq PAST_{S_2}$. If the system is in consistent state S and e is an event such that $PAST_S \cup \{e\}$ defines a consistent state S', then we denote this as $S \overset{e}{\vdash} S'$. A schedule σ for a consistent state S is a sequence of events that can successively occur after the system is in state S. We can now extend the $\vdash$ relation by denoting $S \overset{\sigma}{\vdash} S'$ if the application of σ to S results in S'. One can show that:

Lemma 5 [Chan83c]. if S *BEFORE* S', then there is a schedule σ such that $S \overset{\sigma}{\vdash} S'$.

In the context of deadlock detection, the states of the system are WFGs, and schedules are schedules of WFG transformations. We say that a node is ***deadlocked at time t*** if it is deadlocked in G_t (the WFG at time t). The following lemma allows us to apply the algorithm to consistent WFGs instead of G_ts.

Lemma 6. If G *BEFORE* G' and v is deadlocked in G, then v is deadlocked in G'.

Proof: By Lemma 5, $G \overset{\sigma}{\vdash} G'$ for some schedule σ. Suppose, for contradiction, v is not deadlocked in G'. From our definition of deadlock, there must a schedule σ' such that $G' \overset{\sigma'}{\vdash} G''$ and v is active in G''. Hence $G \overset{\sigma \cdot \sigma'}{\vdash} G''$, and v is not deadlocked in G, a contradiction. □

6.2. Obtaining a consistent WFG

The following scheme obtains a collection of snapshots that represents a consistent WFG: The initiator sends *FREEZE* messages to all the nodes in its *IN* and *OUT*. When a process p receives the first *FREEZE* message, it takes a snapshot of its local state and it sends *FREEZE* messages to all the processes in its IN_p and OUT_p. Furthermore, whenever p sends a message to a process that joined IN_p or OUT_p after p has taken its snapshot, the message will be preceded by a *FREEZE*.

Lemma 7. The scheme described above provides a consistent collection of snapshots.

Proof: Clearly the collection of snapshots defines a cut c. It suffices to show that c is consistent. Suppose c is not consistent, then there are processes p and q such that some message m from p is recorded in q's snapshot; but in p's snapshot, m is not recorded as sent in $PAST_c$. Process p received a *FREEZE* message before sending m. Therefore, it sent a *FREEZE* to q before it sent m. Because messages are received in the order they are sent, q received the *FREEZE* before m. Therefore, m cannot appear in q's snapshot, a contradiction, and c is consistent. □

When the deadlock detection algorithm is initiated, *FREEZE* messages are sent before the *NOTIFY* messages. Messages are received in the order sent. Therefore, whenever a deadlock-detection message (*NOTIFY*, *GRANT*, *DONE* or *ACK*) arrives at a node, the node has already taken its snapshot.

6.3. The complete algorithm for a dynamic system

Whenever a process p suspects that it is deadlocked, it initiates a new invocation of the algorithm. If it is the k-th such invocation, all the ensuing messages are tagged with (p,k). Process p first takes a snapshot of its local state and tags it with (p,k). It then sends *FREEZE* messages to its neighbors, as described in the previous section. Immediately after, it starts the deadlock detection algorithm for colored WFGs (i.e., for static systems with messages in transit) on its (p,k) snapshot.

Similarly, when a process receives its first (p,k) *FREEZE* message, it takes a (p,k) snapshot of its local state, it propagates *FREEZE* messages, and then executes the distributed deadlock detection algorithm according to its (p,k) snapshot.

Theorem 3. Let t_1 be the time the algorithm is invoked, t_2 the time it terminates, and G_{t_1} and G_{t_2} the respective WFGs.

1. If the initiator is deadlocked at t_1 then $free_{initiator} = false$ at t_2.

2. If $free_{initiator} = false$ at t_2 then the initiator is deadlocked at t_2.

Proof: Let G be the WFG described by the collection of snapshots that were obtained by the invocation of the algorithm. By Lemma 7, G is consistent. It is also clear that G_{t_1} *BEFORE* G.

1. If the initiator is deadlocked in G_{t_1}, then, by Lemma 6, the initiator is deadlocked in G. Since the algorithm is applied to G, by Theorem 2, $free_{initiator} = false$ when the algorithm terminates, i.e., at t_2.

2. Suppose $free_{initiator} = false$ at t_2. From Theorem 2, the initiator is deadlocked in G. There are two cases:

(i) If G *BEFORE* G_{t_2} then, by Lemma 6, the initiator is deadlocked at t_2.

(ii) G *BEFORE* G_{t_2} does not hold, i.e., processes keep propagating *FREEZE* messages even after t_2. Let t_3 be the time the last snapshot in G was taken. Clearly, G *BEFORE* G_{t_3} and $t_2 \leq t_3$. By Lemma 6, the initiator is deadlocked at t_3. Note that already at t_2, the algorithm correctly determined that the initiator is deadlocked at t_3. This is possible only if the initiator was already deadlocked at t_2. □

6.4. Optimization

There is no need for an explicit wave of *FREEZE* messages to herald the deadlock-detection messages. The *FREEZE* messages can be incorporated into the deadlock-detection messages and into the WFG messages (*REQUEST, REPLY* and *RELINQUISH*). Each deadlock-detection message is tagged with the invocation identifier. Each WFG message, sent from p to q, will also carry all the invocation identifiers that p has seen and has not yet sent to q. When q receives a message with a new invocation identifier, it interprets it as a *FREEZE,* and takes a snapshot of itself. Only then q changes its state according to the message content.

There is no need for processes to store snapshots for all the previous invocations of the algorithm by a single initiator since all but one of them are obsolete. When a node receives a message from the k-th invocation by process p, all earlier invocations by p have terminated and their snapshots can be discarded. Thus, processes have to store only n snapshots, where n is the number of processes, and only the latest invocation numbers are carried with the WFG-messages.

One can further reduce the number of snapshots stored as follows:

1. First we add another phase to the algorithm in which the initiator notifies all the nodes that the *Grant* phase has terminated. When any node receives such a message it can find out whether it is deadlocked by checking its variable *free*.

2. We maintain a priority order on all the invocations of the deadlock detection algorithm (Logical Time [Lamp78] is one such ordering). Each process stores only the snapshot of the highest priority invocation and abort activity on all the rest.

No specific invocation is bound to terminate, since it might be aborted by a later one. However, the number of consecutive aborts is limited by the number of processes. Therefore, within finite time from the first aborted invocation, some invocation will terminate. By the first rule above, all the processes involved will then be able to check whether they are deadlocked.

7. Performance

Given a WFG with n nodes, e edges, and diameter d, we first consider the complexity of an invocation of the static algorithm by a single initiator. At most four messages are sent over each edge, i.e., at most a total of $4e$ messages are sent. The size of each message is a small constant number of bits. A node v of degree k needs $o(k)$ bits of local storage, and spends $o(k)$ time in local computation.

We model the running time of the algorithm by assuming that all the message transmissions are synchronized and cost one unit of time, a *hop*. The colored WFG algorithm and its optimized version take $4d$ hops to terminate.

In the best previously published algorithm for *AND−OR* requests [Herm83], a single algorithm invocation sends up to n messages over each edge, i.e., a total of up to $n \cdot e$ messages, and the size of each message is up to $o(n \log n)$ bits. The algorithm requires $o(n^2 \log n)$ bits of local storage and a considerable amount of local computation. However, it takes only $2d$ hops to detect a deadlock.

The best previous solution that supports *N−out−of−M* requests is a non-distributed algorithm [Ober80] and it requires costs that are exponential in the number of nodes.

In a dynamic system, multiple invocations of the algorithm by different initiators incur some additional costs. The ongoing propagation of *FREEZE* messages may require up to n^2 messages. However, we already showed that *FREEZE*s can be piggybacked on the WFG messages, and thus do not require an independent transmission. We also have to store the snapshots. A snapshot takes, without its identifyer, $o(n)$ bits of local storage; thus we need $o(n^2)$ bits for n snapshots. If we maintain a priority ordering of invocations as explained in the previous section, then $o(n)$ bits of storage per process are sufficient.

8. Discussion and conclusions

We have presented an efficient algorithm for deadlock detection in distributed dynamically changing systems. The algorithm supports both *AND* and *OR* requests. Also, it directly supports $N\text{-}out\text{-}of\text{-}M$ requests without an exponential increase in the complexity.

The algorithm was incrementally developed and proven correct for three systems of increasing complexity. We first derived an algorithm for a static system with instantaneous message transmission. This algorithm was then extended to work on a static system with messages "frozen" in transit. Finally, to deal with a dynamically changing system, we proposed an algorithm that takes consistent local-state snapshots of the processes, and concurrently executes the static system deadlock detection algorithm on these snapshots.

This approach is quite general and seems widely applicable. It simplifies the complexity of directly compensating for system changes that occur during the execution of an algorithm. We are currently applying it to other distributed algorithms for dynamic systems.

In [Chan83c] an algorithm for obtaining a consistent global state is presented, with the assumption that the processes intercommunication topology is fixed, and every process knows all its incoming channels. In in our system, a node may be unaware of some of its incoming channels (e.g., a process u holding a certain file lock may not know about the existence of another process v that just sent a lock request to u).

This difference affects the two solutions: In [Chan83c] each process is responsible for reporting the state of all its incoming channels. This is possible because processes know all their incoming channels. In our solution, each process has to locally find out the states of its outgoing edges, since these determine its actions in the algorithm.

As a final remark, our model assumes that an active process simultaneously satisfies all the requests to it. One may consider the case where requests can be only granted one by one, serially. It easy to show that the two models are equivalent; there is a deadlock in the case of simultaneous granting of requests if and only if there is a deadlock in the serial case. Thus our algorithm is able to handle both models.

Acknowledgement

The authors are grateful to Ken Perry for suggesting many improvements in the presentation of the algorithms.

Appendix

Lemma 1. During an execution of the *Closure* algorithm, a node v calls the *Closure* procedure if and only if $v \in C(S,P)$.

Proof: We number the *Closure* calls that occur during the algorithm execution according to their order of invocation in time†, and we denote by $s(v)$ the number of the *Closure* call initiated by v.

We first claim that if v calls the *Closure* procedure, then $v \in C(S,P)$. The proof is by induction on $s(v)$. If $s(v)=1$ then v is the first node to call *Closure* and therefore $v \in S \subseteq C(S,P)$.

Suppose the claim holds for all v such that $0 \leq s(v) \leq r$. Consider v with $s(v)=r+1$. If $v \in S$, then we are done. If $v \notin S$, then when v calls *Closure* we have $P(v, \mathit{ancestors}_v)=\mathit{true}$, where $\mathit{ancestor}_v$ is the set of nodes that sent a *NOTIFY* message to v. If $w \in \mathit{ancestor}_v$, then $w \in IN(v)$, and w called *Closure* before v, i.e., $s(w) \leq r$. By induction hypothesis, $w \in C(S,P)$. Therefore, $\mathit{ancestors}_v \subseteq IN(v) \cap C(S,P)^k$, for some k. Since $P(v, \mathit{ancestors}_v) = \mathit{true}$, then $v \in C(S,P)^{k+1}$.

We now claim that if $v \in C(S,P)$ then v calls the *Closure* procedure. Let $l(v)$ denote the smallest i such that $v \in C(S,P)^i$. The proof is by induction on $l(v)$. If $l(v)=0$, then $v \in S$, and therefore v calls *Closure*.

Suppose our claim holds for all v such that $0 \leq l(v) \leq r$, and consider v such that $l(v)=r+1$. Since $v \in C(S,P)^{r+1}$, there must be $v_1, \ldots, v_k$ in $IN(v) \cap C(S,P)^r$ such that $P(v, \{v_1, \ldots, v_k\})=\mathit{true}$. By induction hypothesis, all the v_j's $(1 \leq j \leq k)$ call *Closure*, and therefore send a *NOTIFY* to v. When the last such *NOTIFY* is received by v, we have $\{v_1, \ldots, v_k\} \subseteq \mathit{ancestors}_v$, $P(v, \{v_1, \ldots, v_k\})=\mathit{true}$. At this point, either $\mathit{in_C}_v = \mathit{false}$ and v calls the *Closure* procedure, or $\mathit{in_C}_v = \mathit{true}$ and v already called *Closure*. □

Lemma 2. The *Closure* algorithm terminates. Moreover, it terminates after all the nodes in $C(S, P)$ terminate.

Proof: Consider an execution of the *Closure* algorithm. A node v is the ***engager*** of a node u if v sends a *NOTIFY* to u that causes u to call the *Closure* procedure. Note that v must call *Closure* before it sends *NOTIFY* messages, and therefore before u calls *Closure*. Moreover, a node w cannot call *Closure* more than once (since the first call sets $\mathit{in_C}_w$ to *true*). Therefore, there can be no cycles in the "v is the engager of u" relation, and a node cannot have more than one engager.

Consider the subgraph $F = (V, \{(v,u) \mid v \text{ is the engager of } u\})$, induced by the execution of the algorithm. From our previous remarks, F must be a forest, and the roots of the trees in F are nodes in S. We define the height $h(v)$ of a node v in F, as the length of the longest path from v to a leaf in F. Formally, if v is a leaf in F, then $h(v)=0$, otherwise $h(v) = \max_w\{h(w) \mid (v,w) \in F\} + 1$. We say that u is a ***descendent*** of v, if there is a path from v to u in

† Concurrent calls can be arbitrarily ordered.

F.

Let v be a node that calls the *Closure* procedure. We claim that v terminates, and that all v's descendents terminate before v. The proof is by induction on $h(v)$. If $h(v)=0$, then v is a leaf in F, and it has no descendents. There are two possible cases:

1. $OUT_v = \phi$, and v terminates.

2. During the *Closure* call, v sends *NOTIFY* messages to all the nodes in OUT_v, but, since v is a leaf in F, it is not the engager of any node in OUT_v. It is then clear that v receives a *DONE* back from all of the nodes in OUT_v, and v terminates.

Suppose the claim holds for all v such that $0 \leq h(v) \leq r$. Consider the *Closure* call of a node v such that $h(v) = r+1$, and let $u \in OUT_v$. During this call, v sends a *NOTIFY* to u. There are two possible cases:

1. v is the engager of u, therefore u calls *Closure*, and $h(u) \leq r$. By induction hypothesis, u terminates, and all u's descendents terminate before u. Note that after u terminates, it sends a *DONE* to its engager v.

2. v is not the engager of u, and u replies to v's *NOTIFY* with a *DONE*.

Therefore v receives a *DONE* from every u in OUT_v, and then v terminates. Consequently, v's descendents must terminate before v, and the claim is proved.

All the nodes in S call the *Closure* procedure. From our claim:

1. all the nodes in S terminate, and therefore the algorithm terminates, and
2. since all the nodes in F are descendents of nodes in S, they terminate before the algorithm does.

From Lemma 1, $v \in C(S, P)$ if and only if v calls *Closure*. So, either v has an engager and therefore $v \in F$, or $v \in S$. In both cases, v terminates before the algorithm. □

Lemma 4. Let G be a WFG, and v a node in $C(\{initiator\}, ADJ)$ in G. Then v is not deadlocked in G, if and only if $v \in C(ACTIVE, SAT)$ in G^T.

Proof: Let G and v be as in Lemma 4. We claim that if v is not deadlocked in G then $v \in C(ACTIVE, SAT)$ in G^T. The proof is by induction on $t(v)$, the length of a shortest schedule σ such that $G \overset{\sigma}{\vdash} G'$, and v is active in G'. If $t(v)=0$ then v is active, and, since $v \in C(\{initiator\}, ADJ)$ in G, then $v \in ACTIVE \subseteq C(ACTIVE, SAT)$ in G^T.

Suppose that our claim holds for all nodes u such that $0 \leq t(u) \leq r$. Let v be a node as in Lemma 4, such that $t(v) = r+1 \geq 1$, and let σ be a schedule of length $t(v)$ as described above. Note that v is not active in G, therefore $n_v = d$ for some $d > 0$ (in G). On the other hand, v is active in G', and therefore $n_v = 0$ (in G').

The only graph transformation that can decrease the label n_v (by exactly one) is the transformation of type 2, when an edge (v,u) leading to an active node is deleted. Since σ decreases n_v from d to 0, σ must include d such transformations s_j, $1 \leq j \leq d$. Let (v, u_j) be the edge deleted

by s_j, such that u_j is active (in the graph that s_j was applied to). Note that u_j is not deadlocked in G, and $t(u_j) \leq r$. We claim that all (v, u_j)'s are edges in G. A consequence of this claim is that $u_j \in C(\{initiator\}, ADJ)$ in G, for all j's, and by induction hypothesis (and our previous remark), $u_j \in C(ACTIVE, SAT)$ in G^T, $1 \leq j \leq d$.

To prove the claim we note that v is not active in any graph resulting from the application of a prefix of the schedule σ to G (this would contradict the definition of σ). Therefore, σ does not include any transformation (of type 1) that adds outgoing edges to v, and the claim is proved. We also conclude that the u_j's are all distinct nodes.

We showed that $u_j \in IN(v) \cap C(ACTIVE, SAT)$ in G^T, for all j, $1 \leq j \leq d$. Therefore, $v \in C(ACTIVE, SAT)$ in G^T.

Now suppose $v \in C(ACTIVE, SAT)$ in G^T, and let $l(v)$ be the minimum i such that $v \in C(ACTIVE, SAT)^i$ in G^T. We claim that, for all $k \geq 0$, there is a schedule σ_k such that

1. $G \overset{\sigma_k}{\vdash} G_k$,
2. σ_k is a sequence of edge deletions (i.e., transformations of type 2), and,
3. for all v, if $l(v) \leq k$ then v is active in G_k, and if $l(v) > k$ then (v, u) is in G if and only if it is in G_k.

The proof is by induction on k.

Let $k = 0$. For all v such that $l(v) = 0$, then $v \in ACTIVE$, so v is active in G, and our claim holds for the null schedule $\sigma_0 = < >$. We now assume the claim holds for $k = r$, and show it also holds for $k = r + 1$.

Consider a node v such that $l(v) = r + 1$. By definition of $l(v)$, we have $| IN(v) \cap C(ACTIVE, SAT)^r | \geq n_v$, i.e., there is a set $\{ u_1, \ldots, u_{n_v} \}$ of nodes u_j such that (v, u_j) is an edge in G and $l(u_j) \leq r$. By induction hypothesis, there is a schedule σ_r consisting of edge deletions such that $G \overset{\sigma_r}{\vdash} G_r$, all the nodes u with $l(u) \leq r$ are active in G_r, and the edges of v in G_r are the same ones that v had in G.

Starting from the WFG G_r, we apply a sequence t of transformations of type 2 that deletes all the edges (w, u) where $l(w) = r+1$ and u is active in G. This results in a graph G_{r+1}, where $G \overset{\sigma_{r+1}}{\vdash} G_{r+1}$, and $\sigma_{r+1} = \sigma_r \cdot t$.

From our previous remarks, every node v with $l(v) = r + 1$ have at least n_v (v, u_j) edges that are deleted by the transformations in t. Therefore, t decreases n_v to 0, and v must be active in G_{r+1}. It is easy to check that σ_{r+1} satisfies the other requirements of the induction hypothesis. □

References

Beer81 C. Beeri and R. Obermarck, A resource class independent deadlock detection algorithm, *Research Report RJ3077,* IBM Research Laboratory, San Jose, California, March 1981.

Brac84 G. Bracha and S. Toueg, A distributed algorithm for generalized deadlock detection, Tech. Rep. 83-558, Computer Science Department, Cornell University, Ithaca, New York, June 1983.

Chan80 E. Chang, Echo algorithms: depth parallel operations on general graphs, *IEEE Transactions on Software Engineering,* vol. SE-8, no. 4, pp. 391-401, July 1980.

Chan82 K. M. Chandy and J. Misra, A distributed algorithm for detecting resource deadlocks in distributed systems, *Proc. of the 1st ACM Symposium on Principles of Distributed Computing,* Ottawa, Canada, pp.157-164, Aug. 1982.

Chan83a K. M. Chandy, L. M. Haas and J. Misra, Distributed deadlock detection, *ACM Transactions on Computer Systems,* vol. 1, no. 2, pp. 144-156, May 1983.

Chan83b K. M. Chandy, Presentation given at the Department of Computer Science, Cornell University, Ithaca, New York.

Chan83c K. M. Chandy and L. Lamport, Detecting stability in distributed systems, to be published.

Gary82 G. S. Ho and C. V. Ramamoorthy, Protocols for deadlock detection in distributed database systems, *IEEE Transactions on Software Engineering,* vol. 8, no. 6, pp. 554-557, Nov. 1982.

Glig80 V. G. Gligor and S. H. Shattuck, On deadlock detection in distributed systems, *IEEE Transactions on Software Engineering,* vol. 6, no. 5, pp. 435-440, Sept. 1980.

Haas83 L. M. Haas and C. Mohan, A distributed deadlock detection algorithm for a resource-based system, *Research Report RJ3765,* IBM Research Laboratory, San Jose, California, Jan. 1983.

Herm83 T. Herman and K. M. Chandy, A distributed procedure to detect AND/OR deadlock, Tech. Rep. LCS-8301, Department of Computer Sciences, University of Texas, Austin, Texas, Feb. 1983.

Holt72 R. C. Holt, Some deadlock properties of computer systems, *ACM Computing Surveys,* vol. 4, no. 3, pp. 179-196, Sept. 1972.

Kim84 J. K. Kim, Deadlock detection algorithms in distributed database systems, Tech. Rep. R-84-1162, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, Feb. 1984.

Lamp78 L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM,* vol. 21, no. 7, pp. 558-565, July 1978.

Mena79 D. A. Menasce and R. R. Muntz, Locking and deadlock detection in distributed data bases, *IEEE Transaction on Engineering,* vol. 5, no. 3, pp. 195-202, May 1979.

Ober80 R. Obermarck, Deadlock detection for all resource classes, *Research Report RJ2955,* IBM Research Laboratory, San Jose, California, Oct. 1980.

Ober82 R. Obermarck, Distributed deadlock detection algorithm, *ACM Transactions on Database Systems,* vol. 7, no. 2, pp. 187-208, June 1982.