Distributed Deadlock Detection Algorithm

RON OBERMARCK
IBM San Jose Research Laboratory

We propose an algorithm for detecting deadlocks among transactions running concurrently in a distributed processing network (i.e., a distributed database system). The proposed algorithm is a distributed deadlock detection algorithm. A proof of the correctness of the distributed portion of the algorithm is given, followed by an example of the algorithm in operation. The performance characteristics of the algorithm are also presented.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed databases*; D.4.1 [**Operating Systems**]: Process Management—*deadlocks*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*

General Terms: Algorithms

1. INTRODUCTION

As more distributed processing systems, and especially distributed database systems are implemented, the requirement for distributed deadlock detection becomes more apparent. The majority of the literature speaks in terms of "wait-for" and "resource" graphs, and transactions. This paper employs the same terminology. All words that originate with the author are enclosed in quotation marks at their first mention and appear with initial capital letters throughout.

The definition of a transaction used in this paper is consistent with that defined in [3] and [8]. A *transaction* is a convenient abstraction for the application processing (including the underlying system support) performed to take a database from one consistent state to another consistent state in such a way that the transition appears to be atomic. If a failure occurs during the processing of a transaction, any changes made by (or on behalf of) the transaction are undone, so that the database is returned to a consistent stage. In order to allow concurrent access to the database, while ensuring a consistent view to each transaction, concurrency control mechanisms such as locking are often used. The use of locking for concurrency control introduces the possibility that one transaction may be suspended because it requests a lock held by another transaction. When one transaction is suspended waiting for a second transaction, and the second transaction is waiting (either directly or indirectly) for the first transaction, the result is a circular wait condition called a *deadlock*.

There are many algorithms implemented in centralized database systems for deadlock detection. All of them are based on finding cycles in a transaction wait-for graph (TWFG) in which the nodes of the graph are used to represent

Author's address: IBM San Jose Research Laboratory, 5600 Cottle Road, San Jose, CA 95193.
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0362-5915/82/0600-0187 $00.75

transactions, with the directed edges indicating for which transactions a given transaction is waiting. When cycles are detected in the TWFG, they are broken by choosing a transaction that is involved in the cycle and causing the transaction to *fail* (usually allowing the transaction to restart with its original input). This operation becomes more complex when the TWFG is distributed among multiple sites of a distributed database.

In a distributed database system, although a transaction will usually perform all of its actions at the site in which it originates, it may also perform actions (or actions may be performed on behalf of it) at other than its original site. If this happens, an agent [4] is created at the remote site to represent the transaction at that site. This agent becomes part of the original transaction for concurrency control and recovery purposes.

In Section 2 of this paper we relate the proposed algorithm to prior work. In Section 3 the distributed environment in which the algorithm is expected to operate is presented. Section 4 contains some simplifying assumptions and the formulation of a distributed transaction wait-for graph, which is used throughout the explanation of the algorithm. Section 5 contains the description of the algorithm. In Section 6 a proof of the correctness of the algorithm is presented. In Section 7 the distributed transaction wait-for graph from Section 4 is used in a step-by-step explanation of how the algorithm would be applied. A performance analysis is provided in Section 8, followed by conclusions in Section 9.

2. RELATIONSHIP TO PRIOR WORK

The reader will find that there are similarities between the algorithm proposed in this paper and several published algorithms. Since the proposed algorithm is distributed, this section is generally limited to two distributed algorithms. It must be noted, however, that the proposed algorithm is nearly identical to the centralized algorithm proposed in [3] (the section entitled "How to Find Global Deadlocks"). The significant differences are given in points (3) and (4) below.

The Menasce-Muntz distributed protocol [7] is similar to the new algorithm. Both utilize a transaction wait-for graph (TWFG) and have concurrency control and a deadlock detector at each site in the network. Both protocols transmit wait-for information from site to site. The Menasce-Muntz protocol can be made to work with modification [1], but becomes increasingly complex and a potential performance problem. Key differences are outlined in points (1) and (3) below.

There is also a similarlity to the work of Goldman [2], in that the local TWFG is recreated each time deadlock detection is performed. In addition, the transaction wait-for information transmitted from site to site can be viewed as sets of *ordered blocked process lists.*

The key differences in the proposed algorithm are as follows.

(1) The communications status of transactions existing at a local site (waiting to receive or expected to send) is represented in the transaction wait-for graph at each site by adding a single node to the TWFG, which represents all communications links during local deadlock detection at each site. This node is called "External."

(2) Only potential multisite deadlock information is transmitted from one site to another. (Potential deadlocks are cycles that contain the node External.)

(3) Two optimizations that reduce the deadlock detector-to-deadlock detector message traffic are employed. One optimization ensures that potential deadlock cycle information is transmitted in a single direction along the path of an elementary cycle. The second optimization uses the lexical ordering of the transaction identifiers to reduce the average number of messages by half.

(4) Deadlock detection is assumed to be a service separate from the concurrency control mechanism. Although not covered specifically in this paper, the separation is a key element in allowing the inclusion of more than lock-waits (the normal resource conflict in distributed database examples) into consideration by the deadlock detection algorithm.

3. PROBLEM STATEMENT

Some number of sites exist, connected by a communications network.

Each transaction (and therefore each node of the wait-for graph) has a globally unique identifier. This unique identifier is used by the distributed system as a means to correlate the agents created to do work on behalf of the transaction for both recovery and concurrency control.

A given transaction originates at one site but may migrate to one or more sites to do work. An agent represents the transaction at each site to which it migrates.

In order to perform work, the agent representing a given transaction must contend for resources with agents representing other transactions at the site, and may become suspended because a resource required for the continuation of the transaction's work is held by one or more other transactions. The most commonly mentioned resource class is that termed "Locks." Although there are other resource classes for which contention can occur (*M* of *N* and Pool), this paper does not directly address them.

It follows from this discussion that a transaction must have an agent at the site in which it is waited for. (The waited-for transaction may also have agents at other sites.)

If a transaction has agents at multiple sites, then a communications link exists that connects all the agents of the transaction, either directly or indirectly. That is, if the agents of a transaction were to be considered as nodes of a graph, and the communications links bidirectional edges, one must be able to start at any node, and reach any other node.) The linkage is not necessarily hierarchical.

If an agent representing a given transaction at a given site is waiting for one or more agents representing other transactions (resource-wait), and has a communications link to one or more agents at other sites, the agent that is waiting is expected to transmit on each of the communications links. All other agents for the transaction are waiting to receive a message and therefore cannot be in lock-wait.

Thus a transaction has a single locus of control at any given time. This is similar to the model used by [9].

Each site contains a distributed deadlock detector. It can communicate with the deadlock detectors in each site to which its local transactions have direct communications links. The assumption is made that messages transmitted from deadlock detector to deadlock detector are received.

The problem is to describe a distributed deadlock detection algorithm that will operate correctly within the assumptions stated above.

4. THE EXAMPLE AND SIMPLIFYING ASSUMPTIONS

The proposed deadlock detection algorithm uses a directed graph. The nodes of the graph represent transactions. The directed edges show the wait-for relationship of the nodes (transactions). (An edge directed from node 1 to node 2 indicates that the transaction represented by node 1 waits for the transaction represented by node 2.) Note that there is no representation of the resources being held or waited for. For purposes of deadlock detection, it is an unnecessary complication to carry resource identities in the wait-for graph.

The following assumptions are made only for ease of explanation. They do not affect the operation of the algorithm.

(1) Although there are several alternatives and optimizations for the transmission of information between deadlock detectors, the explanation in Section 5, "The Detection Algorithm," assumes that deadlock detection occurs in each site after all deadlock detector-to-deadlock detector communication has been completed for the prior iteration of the algorithm for all sites, and prior to the transmission of deadlock detector-to-deadlock detector results.

    This leads to the following iterative process:

    (a) Each site receives deadlock information from other sites that was produced by the previous deadlock detection iteration.
    (b) Each site performs deadlock detection.
    (c) Each site transmits deadlock results to other sites as required.

    In actual practice, the synchronization between sites would be roughly controlled by an agreed-upon interval between deadlock detection iterations and a timestamp associated with the messages transmitted. The synchronization need not be precise, because deadlocks persist until broken.

(2) A given communications link between two agents of a transaction is considered to be logically synchronous, with one agent receiving (or waiting to receive) and the other agent transmitting (or expected to transmit).

The last simplifying assumption, coupled with the *single locus of control* for each transaction (from the problem statement), produces the following relationship between the agents of a transaction:

> If a transaction has N agents, then only 1 agent may be either active (processing) or in lock-wait.

If $N > 1$, then the one active agent is expected to send a message. $N - 1$ agents of the transaction are waiting to receive a message and may not be either active or in lock-wait.

The wait-for graph in Figure 1 is used as an example throughout the explanation. The graph is shown as it would be seen by a centralized detection algorithm.

The centralized deadlock detection algorithm would detect three elementary deadlock cycles in the graph of Figure 1. (If one assumes that the nodes of a directed graph are numbered in ascending order, an elementary cycle is a unique path formed by following the directed edges from a *lowest* numbered node through other nodes, which ends back at the starting node.) The elementary

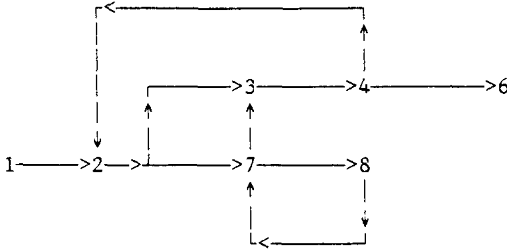

Fig. 1. The wait-for graph as seen in a centralized system

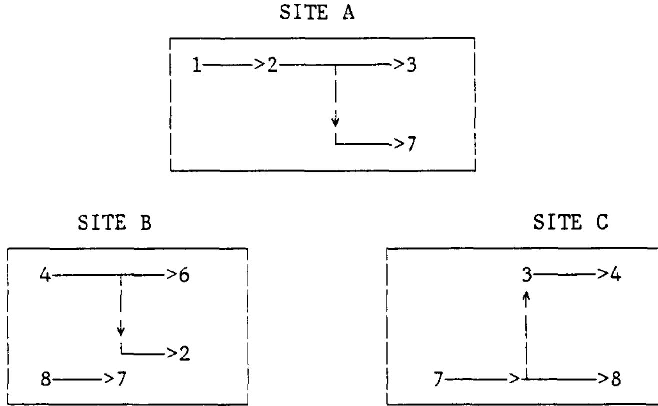

Fig. 2. Wait-for graph split among three sites.

cycles in the above graph are

$$2 \rightarrow 3 \rightarrow 4 \rightarrow 2$$
$$2 \rightarrow 7 \rightarrow 3 \rightarrow 4 \rightarrow 2$$
$$7 \rightarrow 8 \rightarrow 7$$

The centralized deadlock detection algorithm then would cause the deadlock cycles to be broken by choosing a "Victim," to have its request for the resource for which it waits, denied. The Victim would then abort, causing its processing to be undone, and the resources it held to be released. The Victim then could be restarted with its initial input.

In a distributed system, with the transactions processing at many sites, the graph in Figure 1 also becomes distributed. One possible splitting of the graph in Figure 1, which is consistent with the problem description and simplifying assumptions above, is shown in Figure 2.

Here, transaction 2 has done work in site B and then migrated to site A, where it is waiting for a resource shared by transactions 3 and 7. Transaction 3 has done work in site A and migrated to site C. In site C, transaction 3 is waiting for transaction 4. Transaction 4 has migrated to site B, where it is waiting for a resource shared by the agents for transactions 2 and 6.

A similar scenario can be devised to account for each segment of the graph.

By the problem definition, each of the sites A-C must have communications links that connect the various pairs of agents for each transaction. The minimum

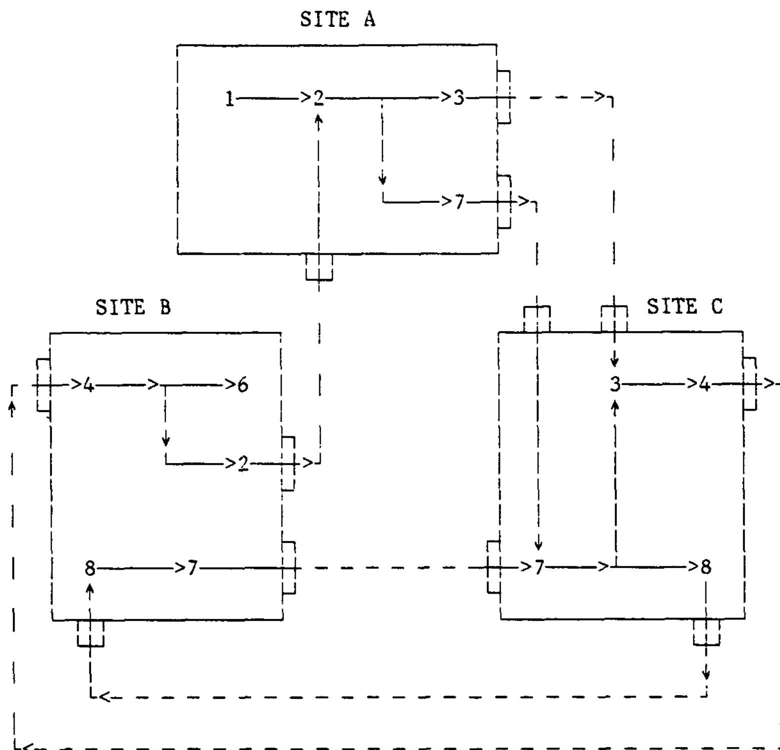

Fig. 3. Same wait-for graph with communications links.

communications links between the agents of each transaction are shown in Figure 3.

The directional arrows point from the agent of each transaction that is waiting to receive, to the agent of the transaction that is expected to send (either an answer or a request). Therefore there is a wait-for relationship established between the various agents of each transaction.

For example, transaction 7 may have originated at site B, and made a request which caused it to migrate to site C. The communications link was established between the two sites, with the agent in site C owing an answer to the request from site B. Transaction 7 in site B is therefore unable to continue until its agent in site C provides the answer.

The agent of transaction 7 at site C then made a request to site A. The communications link was established; the transaction migrated to site A, performed the request, and sent the answer to site C. The agent in site A is shown waiting for the next request from the site C agent.

With the answer to its request to site A, the agent of transaction 7 in site C then proceeded. The agent in site C then required exclusive use of a resource owned jointly by transactions 3 and 8, and is shown waiting for them.

Similar scenarios can be used to explain the remainder of the graph shown in Figure 3.

5. THE DETECTION ALGORITHM

The deadlock detection algorithm at each site builds and analyzes a directed graph, the nodes of which represent transactions, and the directed edges of which represent a given transaction waiting for another given transaction (1 → 2 indicates that transaction 1 waits for transaction 2). Loops (1 → 1) are allowed, but only one directed edge is created from one node to another. (No matter how many times transaction 1 waits for transaction 2, only one edge from node 1 is directed to node 2.)

The deadlock detection algorithm utilizes a distinguished node in the TWFG at each site, called "External." External is used during each site's deadlock detection to represent the portion of the distributed TWFG that is external (therefore unknown) to the site. (External is abbreviated 'EX' when referred to.) The status of each agent's direct communication links at the local site to its cohorts at other sites (at the time deadlock detection is done) determine whether the transaction is waiting for External (waiting to receive) or whether External is waiting for the transaction's agent at this site (expected to send). (This allows the detection of "potential deadlock" cycles of the form 'EX → TRAN1 → TRAN2 → EX' from which "Strings," or global edge lists, are developed.)

The term "String" is used in various steps of the algorithm. A String can be considered to be a list of transaction waits for transaction information that is sent from one site to one or more sites as a result of the sending site's deadlock detection. Each String received by a given site is included in that site's wait-for graph in a specific manner (as described in Step 2 of the algorithm). (For example, the "potential deadlock" cycle 'EX → TRAN1 → TRAN2 → EX' above would produce the String, or global edge list 'EX, TRANID1, TRANID2'.)

The communication protocols used to receive strings from other sites are not critical within the assumptions of the problem statement. Receipt of strings from other sites may be done asynchronously to the deadlock detection itself. If multiple sets of strings are received from a given site between two deadlock detections at the receiving site, the earlier set(s) may be discarded without loss.

In an actual implementation, rough synchronization between the deadlock detectors at various sites could be maintained by a global timestamp, with received strings being retained and used as input for several iterations of a given site's deadlock detection process unless overridden by receipt of newer input.

The detection algorithm at each site performs the following:

(1) Build a wait-for graph using the transaction-to-transaction wait-for relationships (as extracted from lock tables, and any other resource allocation tables or queues from which transaction waits may be obtained).

    Note: This is the same process that a centralized deadlock detection algorithm would perform. Therefore it is not described here. (See [3], "How To Detect Deadlock," for a simple description.)

(2) Obtain and add any Strings of nodes transmitted from other sites to the existing wait-for graph.

    (a) For each transaction identified in a string, create a node of the TWFG, if none exists at this site.
    (b) For each transaction in the string, starting with the first (which is always

External), create an edge to the node representing the next transaction in the string. (The rules covering String creation and transmission are covered in Step 7 below.)

(3) Create wait-for edges from External to each node representing a transaction's agent that is expected to send on a communications link.
(4) Create a wait-for edge from each node representing a transaction's agent, which is waiting to receive from a communications link, to External.
(5) Analyze the resulting graph, listing all elementary cycles. One published algorithm to perform the analysis is described in [6].

The list of elementary cycles is used (as opposed to the TWFG) for the remaining steps of the algorithm.

(6) Select a Victim to break each cycle that does not contain the node External. As each Victim is chosen for a given cycle, remove *all* cycles that include the Victim. Note that the TWFG is no longer used, since all cycles were listed during Step 5.

    Once a Victim is selected, two sets of actions must occur if the chosen Victim is a distributed transaction:

    (a) The site must remember the Transaction ID of the Victim through at least the next deadlock detection cycle. (This enables the site to discard any Strings received between this iteration of the deadlock detection and the next, which contain the Victim)
    (b) If the Victim has an agent at this site, then the fact that the transaction was chosen as a Victim must be transmitted to each site known to contain an agent of the Victim (with which the Victim has a communications link established); otherwise the fact that the transaction was chosen as a Victim must be transmitted to each site from which a "String" containing the Victim's ID was received. (An alternative is to broadcast the Victim to all sites that are in direct communication with this site.)

(7) Examine each remaining cycle that contains the node External (all other cycles have been broken by choosing a Victim). If the transaction identifier of the node for which External waits is greater than the transaction identifier of the node waiting for External, then

    (a) Transform the cycle into a String, which starts with the transaction identifier of the node External, followed by each transaction identifier in the cycle, ending with the transaction identifier of the node that waits for External.
    (b) Send the string to each site for which the transaction terminating the string is waiting to receive.

    For example, a remaining cycle at site (i)

$$\text{'EX'} \rightarrow \text{TRAN}(x) \rightarrow \text{TRAN}(y) \rightarrow \cdots \rightarrow \text{TRAN}(z) \rightarrow \text{'EX'}$$

    would be transformed into the String

$$\text{'EX'}, \text{TRANID}(x), \text{TRANID}(y), \cdots, \text{TRANID}(z)$$

(where the agent for TRAN(z) at site (i) was waiting for a message from an agent at site (j) for the same transaction).

The String would be sent to site (j) only if TRANID(x) > TRANID(z).

6. PROOF OF VALIDITY OF THE ALGORITHM

To show that this deadlock detection algorithm works, we must show that

(1) If there is a global cycle $T(1) \rightarrow T(2) \rightarrow \cdots \rightarrow T(m)$, then it is detected by some site.
(2) If a site detects such a cycle, there actually is a deadlock.

A definition of deadlock is required. Deadlock is defined as that relationship between transactions waiting for other transactions in such a manner that none can proceed. This relationship can be represented by a directed graph in which the nodes of the graph represent the transactions and the directed edges of the graph represent the wait-for relationships between transactions (called a TWFG). Deadlocks are represented in such a graph as cycles.

In order to show that there is actually a deadlock when a site detects a cycle, an assumption, not valid in the real world, must be made:

The portion(s) of the local transaction wait-for graph (TWFG) shipped as String(s) from one site to another does not change until the String has reached and been processed at some *final* site. The final site is defined as

(1) the site at which the String completes a deadlock cycle, or
(2) the most distant site at which a global deadlock can be proved not to exist.

A short discussion of this assumption follows the proofs. It states some reasons why the assumption is not valid, and suggests one method of eliminating *phantom deadlocks* (a cycle detected at some site that is not really a deadlock).

In the following discussion, we use the symbols

$S(n)$	to denote the nth site S,
$T(n)$	to denote the nth transaction T,
$A(n, m)$	to denote the mth agent A at the nth site,
$T[A(n, m)]$	to denote the transaction represented by the mth agent at the nth site,
$TV(c)$	when we refer to the identifier of a transaction,
$TV[A(n, m)]$	when we refer to the identifier of the transaction represented by an agent at a site.

THEOREM 1. *If there is a global elementary deadlock cycle, the cycle will be detected by some site.*

PROOF. Suppose that there is a global elementary deadlock cycle $T(1) \rightarrow T(2) \rightarrow \cdots \rightarrow T(m)$ with contiguous pieces of the cycle represented at N sites.

Since it is a cycle, $T(m) \rightarrow T(1)$.
Since it is global, $N \geq 2$.

Let S(1), . . . , S (N) be the sites containing contiguous pieces of the deadlock cycle. Note that S(i) may occur several times if S(i) has several discontiguous pieces of the cycle.

From the problem description we derive the following:

(1) A given transaction T(c) is represented at a given site S(i) by an agent A(i, j) if the transaction either originated at site S(i) or if it migrated to site S(i) from some other site.
(2) If a given transaction T(c) has migrated from its site of origin S(i) to perform work at another site S(f), then it is represented by some agent A(i, j) at site S(i), and some agent A(f, k) at site S(f). There is also a wait-for-message relationship between the two agents (either A(i, j) → A(f, k) or A(f, k) → A(i, j)).

Further, if the transaction is waiting for a resource owned by another transaction (T(c) → T(d)), there is a specific relationship between the agents of the waiting transaction T(c) (i.e., if the T(c) → T(d) wait occurs at site S(i), then the agent wait-for-message relationship is A(f, k) → A(i, j)).

This does not preclude more complicated agent wait-for-message structures such as A(f, j) → A(g, k) → A(i, l), which could occur if transaction T(c) originated at site S(f), migrated to site S(g), where agent k was assigned to represent it, and then migrated from site S(g) to site S(i), where it is represented by agent 1.

In the description of the algorithm we use the dummy Transaction ID and node of the local TWFG labeled 'EX' at each site to represent the local agents' wait-for-message status with all other sites during the cycle-detection process.

From the description of the algorithm we obtain the following:

(1) If an elementary cycle is detected at a given site S(y), that is, 'EX' → T(b) → T(c) → · · · → T(d) → 'EX', it is converted into a String.

We use T(n) here rather than A(i, j) because the agent that is actually waiting need not be in the current site. The transaction waits-for transaction relationship may have been provided as input to the deadlock detection algorithm as a String received from another node. Therefore 'EX' → T(b) at a site S(y) means that

> there is some site S(x) ($x \neq y$) with an agent of T(b) that waits-for-message from a site S(w) (w may equal y), where an agent of T(b) waits for a resource owned by another transaction T(c).

T(d) → 'EX' always indicates that an agent A(y, m) of transaction T(d) does exist at site S(y) and waits-for-message from an agent A(z, n) at site S(z) ($z \neq y$) of transaction T(d).

A String is shipped to site S(z) if and only if the lexical ordering of TV(b) > TV(d).

The String contains the 'EX', followed by TV(b), TV(c), . . . , TV(d) in transaction waits-for transaction order.

(2) A site S(z) that receives a String includes the described transaction waits-for transaction relationships in its TWFG. Note that while site S(z) will always have an agent A(z, n) representing transaction T(d), it need not have agents representing any of the other transactions in the String.

Let 'EX' $\rightarrow$ T[A(i, 1)] $\rightarrow$ T[A(i, 2)] $\rightarrow \cdots \rightarrow$ T[A(i, $k(i)$)] $\rightarrow$ 'EX' be the piece of the cycle at S(i) and $k(i)$ be the length of the piece. Note that $k(i)$ might be 1, but this does not affect the following proof.

Because each piece of the cycle at site S(i) ends with the agent of a given transaction waiting to receive a message from its agent at a remote site, TV[A(i, $k(i)$)] = TV[A(i + 1, 1)] for each $i = 1, \ldots, N - 1$ and TV[A(N, $k(N)$)] = TV[A(1, 1)].

Since we are dealing with a cycle, we may rotate the sequence S(1), S(2), ..., S(N − 1), S(N) to be the equivalent cycle S(N), S(1), S(2), ..., S(N − 1). Thus we may label any S(i) as S(N), and so we assume without loss of generality that

(1) TV[A(N, $k(N)$)] > TV[A(i, $k(i)$)] for $i \neq N$ (i.e., A(N, $k(N)$)) represents the transaction with the maximum ID of all the transactions in the cycle that are waiting to receive a message.)
(2) A(N, 1) $\neq$ A(N, $k(N)$). In other words, S(N) has a nontrivial piece of the cycle (i.e., a piece in which there are agents representing at least two transactions with one waiting for the other, rather than a single agent A(N, 1), which is merely waiting to receive a message, which it is then expected to send).

The transactions T(1), ..., T(m) are similarly relabeled so that T(1) = T[A(N, $k(N)$)] (T(1) has the maximum Transaction ID). The two constraints defining S(N) are mutually satisfiable because transaction T(m) waits for transaction T(1) = T[A(N, $k(N)$)] at some site (because there is a deadlock involving T(1)). T(1) = T[A(N, $k(N)$)] is waiting, in turn, for a message from one of its agents A(1, 1).

Consider the first site S(J), which has T[A(J, $k(J)$)] $\neq$ T[A(1, 1)]. The key property of this ordering is that

For each $i = J + 1, \ldots, N$, site S(i) receives from S(i − 1) the String

$$\text{'EX'}, \text{TV[A}(J, 1)], \ldots, \text{TV[A}(i-1, k(i-1))]. \qquad (*)$$

Intuitively, the TV[A(1, 1)] = TV[A(J, 1)] pushes all the edges ahead of it toward the site S(N).

To prove property (*) we need to follow the following observation:

$$\text{TV[A}(i, k(i))] < \text{TV[A}(N, k(N))] \qquad \text{for all} \quad i: J \leq i < N. \qquad (**)$$

This is true because each such T[A(i, $k(i)$)] is either directly or indirectly waited for by T[A(J, 1)] = T[A(N, $k(N)$)] (representing T(1)) and hence represents one of the transactions T(2), ..., T(m). But T(1), ..., T(m) is an elementary cycle so that TV(1) $\neq$ TV(i) for $i = 1, \ldots, N - 1$, and TV[A(N, $k(N)$)] $\neq$ TV[A(i, $k(i)$)].

We prove property (*) by induction on i. It is vacuously true for $i \leq J$ and for $i > N$. Therefore consider site $J + 1$. By the minimality of J, TV[A(1, 1)] = TV[A(J − 1, $k(J - 1)$)] = TV[A(J, 1)]. By the maximality of TV[A(N, $k(N)$)] = TV[A(1, 1)], TV[A(J, 1)] > TV[A(J, $k(J)$)], and so S(J) will send the String 'EX', TV[A(J, 1)], ..., TV[A(J, $k(J)$)] to S(J + 1). This proves the hypothesis for $i = J + 1$.

Assume that the hypothesis is true for some $i: J < i < N$. By hypothesis S(i) will receive the String

$$\text{'EX'}, \text{TV}[\text{A}(J, 1)], \ldots, \text{TV}[\text{A}(i-1, k(i-1))].$$

The wait-for relationships are included in the TWFG at S(i), creating either the elementary cycle

$$\text{'EX'} \rightarrow \text{T}[\text{A}(J, 1)] \rightarrow \cdots \rightarrow \text{T}[\text{A}(i, 1)] \rightarrow \text{T}[\text{A}(i, k(i))] \rightarrow \text{'EX'},$$

if $k(i) > 1$, or the elementary cycle

$$\text{'EX'} \rightarrow \text{T}[\text{A}(J, 1)] \rightarrow \cdots \rightarrow \text{T}[\text{A}(i, 1)] \rightarrow \text{'EX'},$$

if $k(i) = 1$, because $\text{TV}[\text{A}(i-1, k(i-1))] = \text{TV}[\text{A}(i, 1)]$.

By following the String-shipping algorithm, S(i) constructs the String 'EX', TV[A(J, 1)], ..., TV[A(i, $k(i)$)] because we are dealing with an elementary cycle. But

$$\text{TV}[\text{A}(J, 1)] = \text{TV}[\text{A}(N, k(N))] \geq \text{TV}[\text{A}(i, k(i))].$$

By property (**)

$$\text{TV}[\text{A}(i, k(i))] \neq \text{TV}[\text{A}(N, k(N))],$$

so

$$\text{TV}[\text{A}(i, k(i))] < \text{TV}[\text{A}(N, k(N))],$$

and the String-shipping algorithm will cause S(i) to send the String 'EX', TV[A(J, 1)], ..., TV[A(i, $k(i)$)] to S(i + 1). By induction this proves the property (*) above.

Applying (*) to site N shows that S(N) will receive the String 'EX', TV[A(1, 1)], ..., TV[A(N − 1, $k(N-1)$)]. This, concatenated with the piece $\text{T}[\text{A}(N, 1)] \rightarrow \cdots \rightarrow \text{T}[\text{A}(N, k(N))] = \text{T}[\text{A}(1, 1)]$ at site S(N), will cause a cycle to appear in the TWFG at site S(N). Hence S(N) will detect the deadlock. This proves the theorem. □

THEOREM 2. *If an elementary global cycle is detected at a site, it is a deadlock.*

Given the definition of deadlock, and the assumption above, if a site constructs a cycle in its TWFG, the cycle represents a deadlock.

6.1. False Deadlocks

The assumption that portions of the distributed TWFG transmitted as Strings will remain frozen until after global deadlock has either been determined or the piece has reached some "final" destination is not valid in a real-world distributed environment. In fact, it will often be the case that the local TWFG will not be a true picture of the wait-for relationships in the single site.

Depending on the amount of concurrent processing in a centralized system, the deadlock detector may obtain the information that transaction 1 is waiting for transaction 5 (a true state at the time). While the deadlock detector is obtaining transaction-wait information for transactions 2–4, transaction 5 could release the resource for which transaction 1 was waiting, and proceed to request a resource

held by (the now no longer waiting) transaction 1. When the deadlock detector requests wait information for transaction 5, transaction 5 is shown as waiting for transaction 1. A false deadlock will now be detected (1→5→1) when the TWFG is analyzed.

Centralized systems deal with this potential problem by

(1) either suspending all resource allocation and release activity from the time that the deadlock detector begins collecting transaction waits-for transaction information, until deadlock detection has completed, or
(2) validating the detected cycles.

In the distributed deadlock detection environment, if transactions 1 and 5 are distributed transactions, the deadlock detector in site A may detect the potential global deadlock EX→5→1→EX during one deadlock detection cycle, and send the String EX, 5, 1 to site B. Any time between the first deadlock detection cycle and the next, transaction 1 at site A may receive a message from its agent at site B, process, and release the resource for which transaction 5 was waiting, allowing transaction 5 to respond to its agent in site B.

By the time the deadlock detector is run at site B, the local portion of site B's TWFG may show 1→5. When the string EX, 5, 1 is added to the TWFG, a false deadlock would appear.

There are two main approaches to the problem of false deadlocks. The first approach is to treat false deadlocks as if they were valid. This approach is acceptable as long as the number of false deadlocks is low.

An alternate approach is based on the fact that real deadlocks will persist until broken. When a deadlock cycle is detected, validation of the transaction wait-for relationships that make up the cycle can be performed. In the single-site deadlock, one method might be to redo the deadlock detection to determine if the same deadlock cycle is formed.

When a global deadlock cycle is detected by a given site, it can be validated by sending it to each site, in turn, that contains a local edge of the cycle (follow the cycle backward, for example). If the cycle constitutes a true deadlock, it will return to the detecting site for validation of the last edge. This works for the same reason that the original detection works.

If an error occurs (a site failure, or communications failure), the cycle will be broken by the site or sites detecting the failure (we assume that the correction of the error will abort the transactions involved). Therefore we can ensure that we do not break deadlocks that never existed. There is still the possibility that the elementary deadlock cycle being broken at site $S(x)$ no longer exists, because it could have been broken asynchronously by some other site (e.g., a transaction in the cycle at site $S(y)$ is canceled by its originator).

7. EXAMPLE OF DEADLOCK DETECTION PROCESSING

Using Figure 3 as an example, with a starting assumption that deadlock detection has not been performed before, the application of the distributed deadlock detection algorithm would perform as follows.

The first detection iteration (shown in Figure 4) would result in the activity and status at each site. Since this is the first iteration of the deadlock detection

```
SITE  STRING REC'D      WAIT FOR GRAPH          CYCLES     STRING SENT
          (FROM)                                (BROKEN)        (TO)

                        EX<-------┐
                        |         ↑             EX,2,3,EX
                        ↓         |             EX,2,7,EX
 A        --            1->2->┬->3->┤                            ---
                              |     ↑
                              ↓     |
                              └->7->┘

                              ┌->6
                              |
                        ┌->4->┴->2->┐                          EX,4,2
                        ↑           |                            (A)
 B        --            |           |
                   EX->┴->8->7->┐   |           EX,4,2,EX      EX,8,7
                   ↑            |   |           EX,8,7,EX       (C)
                   |            ↓   ↓
                   └<-----------┴<--┘

                        ┌------>┬->3->4->┐
                        ↑       ↑        |
                        |       |        |      EX,3,4,EX
 C        --       EX->┴->7->┴->8->┐     |      EX,7,3,4,EX    EX,7,3,4
                   ↑               |     |      EX,7,8,EX       (B)
                   |               ↓     ↓
                   └<--------------┴<----┘
```

Fig. 4. Iteration 1 of detection algorithm.

algorithm at each site, there are no input Strings to be added to the local wait-for graph at any site.

At site A, External (depicted as EX) is shown as waiting for transaction 2. This is due to the state of the communications link (expected to write) associated with the agent representing transaction 2 at this site. Transactions 3 and 7 are shown as waiting for External because the agents representing these transactions at site A are both waiting to receive a message.

Two cycles would be detected at site A, and they are shown in the CYCLES column. Both cycles contain the special TWFG node External, and are listed in transaction waits for transaction order. Cycles listed in this column will always be shown with the start and end transaction equal.

Since the lexical ordering of the transaction waited for by External is less than that of the transaction which waits for External in each cycle, no string will be sent from site A.

Both sites B and C have similar potential cycles, but owing to the lexical ordering of the transactions within the cycles, they will send Strings. Site B will send the String EX, 4, 2 to site A, because the transaction waiting for External is waiting to receive a message from site A. The String EX, 8, 7 likewise will be sent to site C.

Although site C has three potential cycles, only the String EX, 7, 3, 4 will be sent to site B.

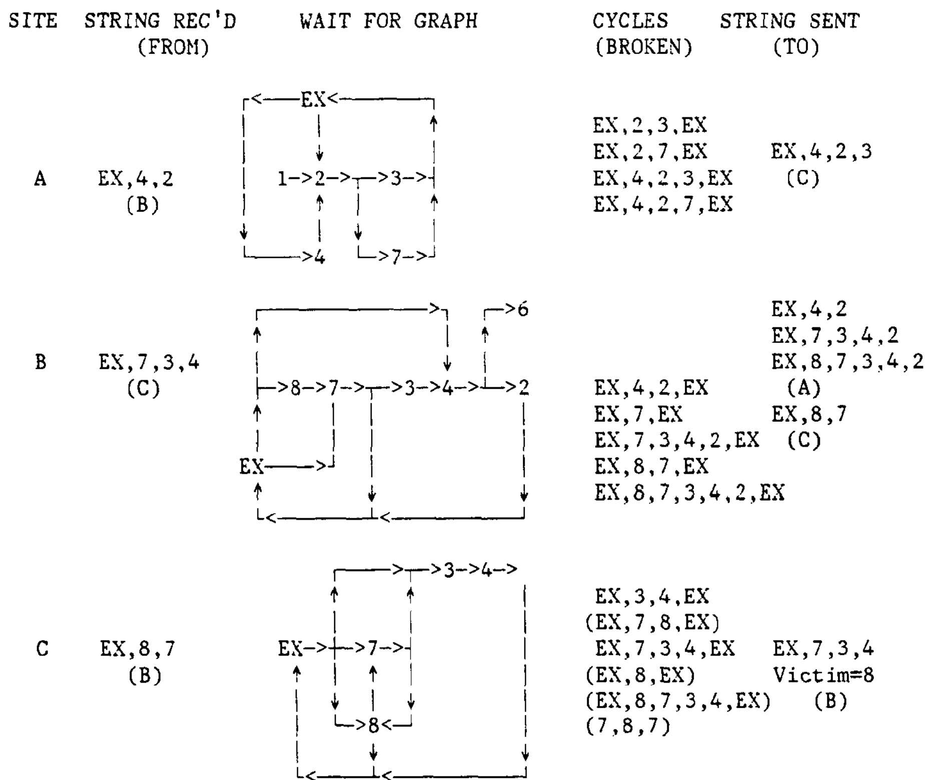

Fig. 5. Iteration 2 of detection algorithm.

Figure 5 shows the second iteration of the distributed deadlock detection algorithm. Note that all Strings sent from the first iteration have been received by their destination sites, and will be incorporated into the TWFG of that site.

When the String EX, 4, 2 (received from site B) is included in the TWFG at site A, two additional potential cycles are created. The first additional potential cycle, EX, 4, 2, 3, EX, is converted to a String to be sent to site C.

At site B, the String EX, 7, 3, 4 received from site C causes the creation of the additional Strings (EX, 7, 3, 4, 2 and EX, 8, 7, 3, 4, 2) that will be sent to site A.

The receipt of the String EX, 8, 7 from site B completes the global deadlock cycle 8, 7, 8. Transaction 8 was arbitrarily chosen as the deadlock Victim, breaking the listed cycles in parentheses, since they each contain transaction 8. The String EX, 7, 3, 4 is still sent to site B, as well as the notification that transaction 8 has been chosen as a deadlock Victim. The fact that transaction 8 is a deadlock Victim is also retained at site C through the next deadlock detection iteration.

The third iteration of the deadlock detection is shown in Figure 6.

Receipt of the three Strings from site B (extended by information received during the previous iteration) causes detection of the 12 elementary cycles listed at site A. The "real" deadlock cycles (2, 3, 4, 2 and 2, 7, 3, 4, 2) are broken first. Transaction 3 was arbitrarily chosen as the deadlock Victim, thereby removing those cycles in parentheses.

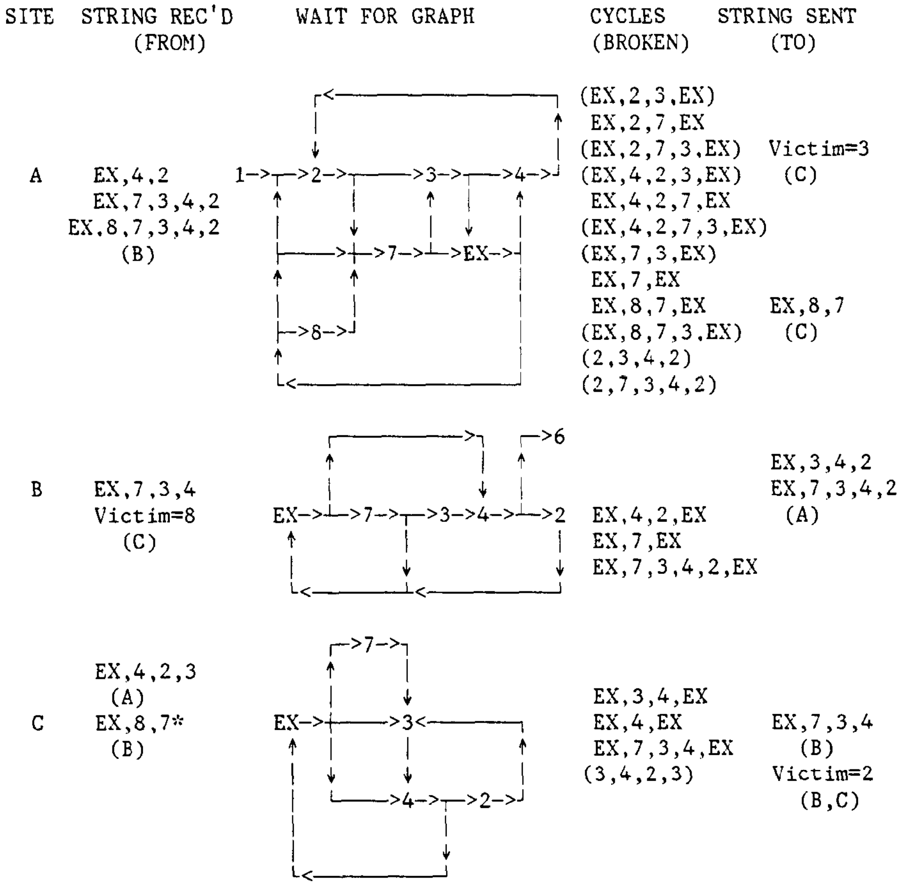

Fig. 6. Iteration 3 of detection algorithm.

It is interesting to note that although both of the remaining elementary deadlock cycles were detected at site A, one of them (2, 3, 4, 2) has been detected at site C as well because the String EX, 7, 3, 4, 2 contains the wait-for information, which, when combined with the local TWFG at site A, completes both elementary cycles. The extended String sent from site A to site C during the prior iteration, contains the information to complete only the elementary cycle 2, 3, 4, 2 at site C. At site C, transaction 2 was arbitrarily chosen to break this cycle.

The String EX, 8, 7 received at site C is not included in the TWFG of that site because it contains the remembered Victim, transaction 8, chosen by site C during the previous iteration.

The final iteration of the deadlock detection algorithm is shown in Figure 7.

With this final iteration, all deadlock cycles have been broken, and the final state in which there is no further deadlock detection message traffic occurs. Any following iterations (given the assumption that the transactions remaining will not incur further conflicts) do not show potential deadlock cycles.

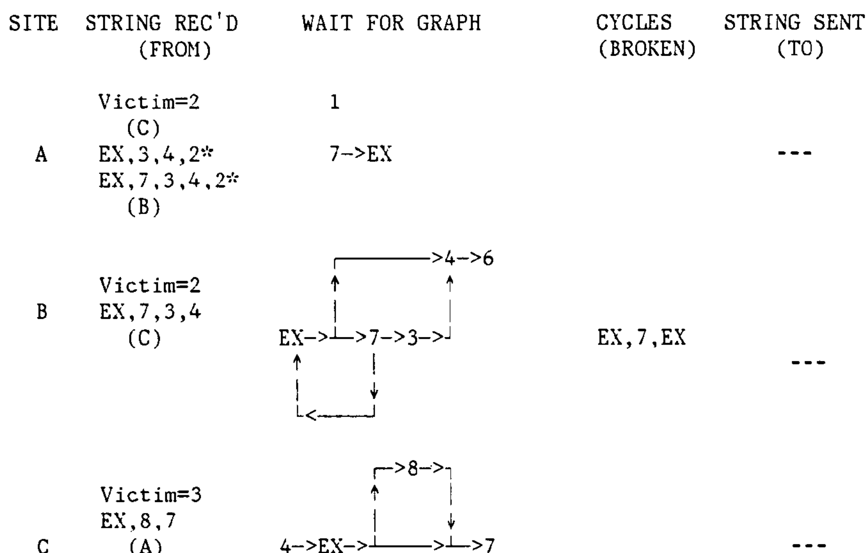

Fig. 7. Iteration 4 of detection algorithm.

8. PERFORMANCE CONSIDERATIONS

The method described above detects global deadlock cycles by transmitting transaction wait information (Strings) only when the potential for a global deadlock is discovered at a given site.

In a global deadlock cycle which involves S sites, the cycle is detected after a maximum of $S(S - 1)/2$ messages have been transmitted. We include Strings that are sent more than once by successive iterations of the deadlock detection algorithm. However, we assume that all Strings to be sent from one site to another following a given iteration of the deadlock detection algorithm are sent as a single message. By transmitting messages along each potential global deadlock cycle in the direction from an agent waiting to receive a message to the agent of the transaction that is expected to send it, the deadlock message follows the path of the potential global cycle. (From the problem statement, while a given agent of a transaction may be expected to send a message to many other agents, it can be waiting for a message from only one other agent at a time.)

The message traffic is halved (on the average) because potential cycle information is transmitted only when the first transaction's identifier in the potential deadlock cycle is lexically *greater* than the last transaction's identifier.

People who submit transactions usually expect a timely response. Therefore either one or more of the submitters would have canceled their request, or timeout mechanisms within the local systems would have aborted one or more of the transactions involved in the "worst-case" situation before the number of sites could get too large.

Although the worst-case performance characteristics of any implementation must be considered, the normal-case performance is a key criterion. In the case described above, if the number of sites becomes large, the probability is high that

the cycle will be broken by some other means before it is complete. Therefore the expectation of the number of global edges (Strings) that will be formed per site is of interest.

8.1 Equations for Calculation of Expected Normal Case

In calculating the expected, or normal case, we use a rough probabilistic analysis [5]. If we choose one average site to represent the network, and determine the probability of having a global edge, or String, be produced on a given deadlock detection iteration at that site, we then can determine the message load to be expected for deadlock detection.

Given

L = the number of concurrently processing local transactions per site (no requirement to access data at other sites to complete successfully),

D = the number of concurrently processing distributed transactions per site (either originated at this site and require access to data at another site, or vice versa),

A + 1 = the total number of concurrently processing agents in a given site,

PW = probability that an agent experiences a lock-wait any time while processing a given transaction (we do not worry about when the lock-wait occurs, but assume that all waits will be seen by the deadlock detector),

SW = fraction of D's waiting to receive a message (which, because of our original problem statement, excludes them from the possibility of being in lock-wait),

E = expected number of global edges (Strings) per site,

we can form the following picture of all the agents at a given site. The agents can be divided into three classes, as shown in Figure 8:

(1) those representing local transactions L (presumably the majority) that are either active or in lock-wait;
(2) those representing distributed transactions that are in a "waiting to receive" state (neither active nor in lock-wait) D1;
(3) those representing distributed transactions that are either actively processing, or are in lock-wait.

The lock-waits in which we are most interested are those that form potential global cycles. (This requires that we have the situation of $D2 \rightarrow \cdots \rightarrow D1$.) The probability that an agent representing a distributed transaction from the class of either active or in lock-wait distributed transactions (D2) is in lock-wait for a resource held by a distributed transaction that is waiting to receive (D1) can be expressed by

$$D2 \rightarrow D1 = D2 * PW * D1/A$$

since it must not only wait for a resource, but for a resource owned by one of the distributed agents that are waiting to receive, out of all agents.

The probability that an agent representing a distributed transaction in the class of either active or in lock-wait distributed transactions (D2) is in lock-wait

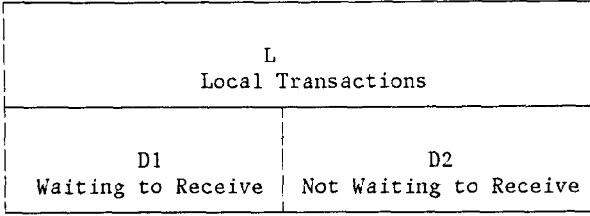

Fig. 8. Transaction States.

for a resource held by a local transaction can be expressed by

$$D2 \rightarrow L = D2 * PW * L/A$$

since it must not only wait for a resource, but for a resource that is owned by a member of the set of local agents, out of the set of all agents.

Because the global detection algorithm considers only wait-for relationships of the form $D2 \rightarrow \cdots \rightarrow D1$ (where there is a string starting with a D2, and ending with a D1, with the possibility of one or more local agents in the wait-for string), the probability for a *specific* local agent waiting for any D1 is required:

$$(\text{a specific L}) \rightarrow D1 = PW * D1/A.$$

The *specific* local agent waiting for a D1 in which we are interested is the one which is being waited for (either directly or indirectly) by a D2, thereby creating a global edge.

For longer strings, the probability that a *specific* local agent is waiting for another local agent is

$$(\text{a specific L}) \rightarrow L = PW * L/A.$$

The probability that an agent representing a distributed transaction from the set of those not waiting to receive (D2) is in lock-wait for a local agent, which is, in turn, in lock-wait for a resource held by a distributed transaction that is waiting to receive, can be expressed by

$$D2 \rightarrow L \rightarrow D1 = (D2 * PW * L/A) * (PW * D1/A).$$

The probability that an agent representing a distributed transaction out of those not waiting to receive (D2) is in lock-wait for a local agent, which is, in turn, in lock-wait for a second local agent, which is, in turn, in lock-wait for a resource held by a distributed transaction that is waiting to receive, can be expressed by

$$D2 \rightarrow L \rightarrow L \rightarrow D1 = (D2 * PW * L/A) * (PW * L/A) * (PW * D1/A)$$
$$= D2 * (PW * L/A)^2 * (PW * D1/A).$$

It then follows that the expected number of global edges that will normally occur in a given site (and therefore contribute to the communications overhead of the distributed deadlock detection algorithm) can be determined by

$$E = D2 * PW * D1/A * (1 + PW * L/A + (PW * L/A)^2 + \cdots).$$

<table>
<tr><td colspan="2">L
Local Transactions
Total 18, Active 16.2, Lock-Wait 1.8</td></tr>
<tr><td>D1
Waiting to Receive
Total 4</td><td>D2
Not Waiting to Receive
Total 2, Act 1.8, LKW .2</td></tr>
</table>

Fig. 9. Probable number of transaction per state.

8.2 Calculation of an Expected Case

A realistic view of distributed systems, based on measurements of lock-waits in both production and experimental database systems shows that the following is true:

$$PW \ll 1.$$

From the transaction volume and response-time requirements placed upon current production database systems, coupled with the communications delays that would occur in a distributed system, it also follows that $D/(A + 1) \ll 1$. In other words, transactions that access data at only their local or original site are expected to be the majority.

Also, since only one agent of a distributed transaction can be active, with the remaining agents waiting to receive, $D - SW \leq 0.5 * D$.

Since the major delay and overhead of distributed deadlock detection is involved with the transmission of global edges (or Strings) from site to site, let us suppose that

(1) there are 20 transactions originating at a given site;
(2) 10 percent of all transactions are distributed;
(3) each distributed transaction has three agents (involves three sites);
(4) 10 percent of potentially active agents experience lock-wait;
(5) $D - SW = 0.3 * D$.

Figure 9 shows the application of the probabilities to the three classes of agents expected at a given site.

There would then be

$A + 1 = 24$ agents per site,
$L = 18$ local agents (16.2 active and 1.8 in lock-wait),
$D = 6$ agents representing distributed transactions,
$D2 = 2$ (1.8 active and 0.2 in lock-wait),
$D1 = 4$ $(D - D2)$,
$A = 23$,

$$D2 \rightarrow D1: \quad 2 * 0.1 * 4/23 = 0.035$$

$$D2 \rightarrow L \rightarrow D1: \quad 2 * 0.1 * 18/23 * 0.1 * 4/23 = 0.003$$

$$D2 \rightarrow L \rightarrow L \rightarrow D1: \quad 2 * (0.1 * 18/23)^2 * (0.1 * 4/23) = 0.0002$$

$$\Rightarrow \text{Expected Number of Global Edges} < 0.04.$$

Therefore the expected number of Strings shipped per site would be < 0.02, because the detection algorithm would ship only half (on the average) of the global edges. It is interesting to note that the probabilities favor the short, D2→D1 cycle. In fact, we can expect that almost all global cycles will be length 2, therefore involving only two sites.

8.3 Comparison with a Centralized Algorithm

A centralized global detection algorithm (e.g., that described in [3]) requires global edge information from each distributed site. For each global deadlock detection iteration, two messages per distributed site are required:

(1) one message to request the global edges from each distributed site, and
(2) the reply from each of the distributed sites, with either the site's global edges, or a message stating, "none this time."

In a three-site network, this would give a total of six messages per iteration. As long as the number of sites involved in a given global deadlock cycle is expected to be low (and the average number of global edges per site is less than 2), the distributed detection algorithm will send fewer messages than the centralized algorithm. By the calculations above,

(a) almost all cycles are of length 2, and
(b) the expected number of global edges (0.04) $\ll 2$.

Therefore the distributed detection algorithm will require fewer messages than the centralized algorithm. (A rough guess indicates an improvement by a factor of 3.)

By changing the percentage of transactions that are distributed and the probability that an agent will experience lock-wait to 50 percent each, the expected number of global edges produced in a given site increases to approximately 2.77. Since the number of Strings to be transmitted from one site to another is expected to be one-half of the number of global edges (1.39 in this case), this is still within acceptable limits.

While the expectation for global edges is low, the performance of the proposed distributed deadlock detection algorithm is expected to be good.

9. CONCLUSIONS

The alogrithm has several advantages over other distributed deadlock detection algorithms. They are as follows:

(1) Only potential multisite deadlock cycle information and Victims need be transmitted.
(2) The deadlock information is transmitted only in a single direction along the path of the potential deadlock cycle. This has the advantage that no new communications links are required for deadlock detection.
(3) A given site processes its own (local) transaction wait-for graph plus the specific nodes and edges that it receives from other sites.
(4) It is not dependent on some "distinguished" site that must be in communication with all other sites, and is therefore less prone to failure.

As stated in the proof, detection of false deadlock cycles is possible with this algorithm. On the basis of the expectation of global edges, as shown in the performance section, one could apply either of the major approaches to the potential problem mentioned in the proof section.

ACKNOWLEDGMENTS

I am deeply indebted to the assistance of three fellow project members who made significant contributions to this paper: to Jim Gray, who not only gave me the equations used for the "normal case" performance calculations, but who spent several hours explaining them until I understood them, and who also did the majority of the final version of the proof of correctness of the distributed algorithm; to Bruce Lindsay, who discovered the optimization that allows the algorithm to reduce the average number of String messages by half; and to Patricia Griffiths/Selinger, who supplied the initial version of the proof of the correctness of the algorithm. In addition, all three reviewed several drafts of this paper, making constructive comments.

REFERENCES

1. GLIGOR, V.D., AND SHATTUCK, S.H. On deadlock detection in distributed systems. Computer Science Tech. Rep. 837, University of Maryland, College Park, Md., Dec. 1979.
2. GOLDMAN B. Deadlock detection in computer networks. Tech. Rep. M.I.T.-LCS TR-185, Massachusetts Institute of Technology, Cambridge, Mass., Sept. 1977.
3. GRAY, J.N. Notes on data base operating systems. In *Operating Systems An Advanced Course*, R. Bayer, R.M. Grahm, and G. Segmuller, (Eds.), Lecture Notes in Computer Science, vol. 60, Springer-Verlag, Berlin and New York, 1978.
4. GRAY, J.N. A discussion of distributed systems. Res. Rep. RJ2699(34594), IBM Research Division, Sept. 1979.
5. GRAY, J.N., HOMAN, P., OBERMARCK, R., AND KORTH, H. A straw man analysis of probability of waiting and deadlock. Res. Rep. RJ3066(38112), IBM Research Division, Feb. 1981 (presented at the 5th Berkeley Workshop on Distributed Data Management and Computer Networks, Feb. 1981).
6. JOHNSON, D.B. Finding all the elementary cycles of a directed graph. *SIAM Comput. 4*, 1 (March 1975), 77–84.
7. MENASCE, D., AND MUNTZ, R. Locking and deadlock detection in distributed data bases. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 195–202.
8. OBERMARCK, R. Distributed data base. IBM Palo Alto Systems Center Tech. Bull. G320-6019, IBM, Palo Alto, Calif., Sept. 1978.
9. ROSENKRANTZ, D.J., STEARNS, R.E., AND LEWIS, P.M. II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst. 3*, 2 (June 1978), 178–198.

Received June 1980; revised April 1981; accepted May 1981