The Distributed Deadlock Detection Algorithm

D. Z. BADAL
Hewlett-Packard Laboratories

We propose a distributed deadlock detection algorithm for distributed computer systems. We consider two types of resources, depending on whether the remote resource lock granularity and mode can or cannot be determined without access to the remote resource site. We present the algorithm, its performance analysis, and an informal argument about its correctness. The proposed algorithm has a hierarchical design intended to detect the most frequent deadlocks with maximum efficiency.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.4.1 [**Operating Systems**]: Process Management—*deadlocks*; *synchronization*; D.4.4 [**Operating Systems**]: Communications Management—*network communication*

General Terms: Algorithms

Additional Key Words and Phrases: Communication deadlock, distributed algorithms, distributed deadlock detection, message communication systems, resource deadlock

1. INTRODUCTION

Deadlock is a circular wait condition that can occur in any multiprogramming, multiprocessing, or distributed computer system that uses locking if resources are requested when needed while already allocated resources are still being held. It indicates a state in which each member of a set of transactions is waiting for some other member of the set to give up a lock. An example of a simple deadlock is shown in Figure 1. Transaction T1 holds a lock on resource R1 and requires resource R2; transaction T2 holds a lock on resource R2 and requires R1. Neither transaction can proceed, and neither will release a lock unless forced by some outside agent.

There have been many algorithms published for deadlock detection, prevention, or avoidance in centralized multiprogramming systems. The deadlock problem in those systems has been essentially solved. With the advent of distributed computing systems, however, the problem of deadlock reappeared. Certain peculiarities of distributed systems (lack of global memory and nonnegligible

Author's present address: Distributed Computing Center, Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304.

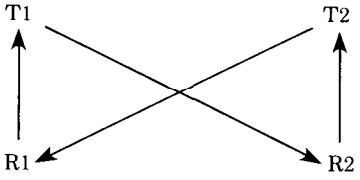


Fig. 1. A simple deadlock cycle.

message delays, in particular) make centralized techniques for deadlock detection expensive. Recently several deadlock detection algorithms for distributed systems have been published [2, 4, 8, 9] (also R. Obermarck, personal communication, 1983). However, most of them have been shown to be incorrect or too complex and expensive to be practical.

In this paper, we propose a new distributed deadlock detection algorithm for distributed computing systems. This algorithm can be seen as an extension of the deadlock detection algorithm proposed in [9]. It requires fewer messages than other published deadlock detection algorithms and differs from existing algorithms in that it uses the concept of a lock history which each transaction carries with it, the notion of intention locks, and a three-staged hierarchical approach to deadlock detection, with each stage, or level, of detection activity being more complex than the preceding one. The third level of the proposed algorithm is essentially the same as the algorithm in [9].

We first present the algorithm, then an informal proof of its correctness, and finally a performance comparison of the proposed algorithm with the algorithm in [9].

2. THE PROPOSED ALGORITHM

2.1 Introduction

The proposed algorithm assumes a distributed model of transaction execution where each transaction has a site of origin (Sorig), which is the site at which it entered the system. Whenever a transaction requires a remote resource (a resource at a site other than the site it is currently at), it *migrates* to the site where that resource is located. Migration consists of creating an *agent* at the new site. The transaction agent then executes and may either create additional agents, start commit or abort actions, or return execution to the site from which it migrated. This transaction model is consistent with recent literature [6, 8] (also R. Obermarck, personal communication, 1983). As the transaction migrates, it carries along its lock history.

Agents can be in any of three states: *active*, *blocked* (waiting), or *inactive*. An *inactive agent* is one that has done work at a site and has created an agent at another site, or one that has returned execution to its creating site and is now awaiting further instructions, such as commit, abort, or become active again. A blocked transaction is one that has requested a resource that is locked by another transaction. An *active agent* is one that is not blocked or inactive. To allow concurrent execution, a transaction may have several active agents.

We assume that all transactions are well-formed and two-phase [4, 8], that is, we assume that any active agent can release a lock only after the transaction has

locked all the resources it needs for its execution and only after it has terminated its execution and the active agent is notified to release the lock during the two-phase commit.

The information contained in a lock table for a resource includes (a) the transaction or agent ID and transaction's site of origin, (b) the type of lock, and (c) if possible, the resource (and type of lock) that the transaction holding a lock intends to lock next. The field containing the current lock is referred to as the *current* field of the lock table, and the field containing the future intentions of the transaction holding the current lock is called the *next* field. The next field always identifies the site(s) to which the transaction migrated.

The proposed algorithm assumes two types of locks: exclusive write (W) and shared read (R). Additionally, the proposed algorithm uses an intention lock (I), which indicates that a transaction wishes to acquire a lock on a resource, either to modify it (IW) or to read it (IR). The intention locks are placed in a resource lock table when an agent is created at a site of a locked resource that it requires, or when a resource at the same site is requested but is already locked by another transaction. The intention locks are also placed in the lock table of the last locked resource(s) if the transaction can determine which resource(s) it intends to lock at a remote site in its next execution step. The intention locks are not the same as the intention modes used by Gray when he discusses hierarchical locks in [4]. (Gray uses the intention mode to tag ancestors of a resource in a hierarchical set of resources as a means of indicating that locking is being done on a "finer" level of granularity, and therefore preventing locking on the ancestors of the resource.)

An example of a lock table is LT(R2B): T1{W(R2B), IW(R3C)}; T2{IW(R2B)}. The lock table for resource R2 at site B shows that T1 holds a write lock on R2, and that T2 has placed an intention write lock on R2. T1 has also indicated that it intends to place a write lock on resource R3 at site C.

The rules for locks in the proposed algorithm are the same as those for conventional locking; that is, any number of transactions or agents may simultaneously hold shared read locks on a particular resource, but only a single transaction or agent may hold an exclusive write lock on a resource. Any number of intention locks (IW or IR) may be placed on a resource, which means that any number of transactions may wait for a resource. Each site must therefore have some method for determining which transaction will be given the resource when it becomes free.

Our algorithm uses the lock history (LH) of a transaction, which is a record of all types of locks on any resources that have been requested or are being held by that transaction. Each transaction carries its lock history during its execution. An example of a lock history for transaction T1 is LH(T1): {W(R3C), W(R2B), R(R1A)}. This LH shows that T1 holds a write lock on resource R3 at site C, a write lock on resource R2 at site B, and a read lock on resource R1 at site A. We use lock histories for three reasons: (a) in some cases to avoid global deadlocks, (b) to support the selection of victim transactions for rollback, and (c) to avoid detection of false deadlocks. All of these uses are discussed in greater detail later.

A deadlock can be detected by our algorithm either by constructing a wait-for-graph (WFG) or directly from wait-for-strings (WFSs). A wait-for-graph can be constructed by the deadlock detection algorithm using the lock histories of

transactions that are possibly involved in deadlock cycle. In this paper we omit details of constructing the WFG. Instead, we refer the reader to numerous papers on that subject, for example [2], [3], [4], and [8]. We just mention that there are two types of nodes in the WFG, transactions (or agents) and resources. A directed arc from a resource node to a transaction node indicates that the transaction has a lock on the resource, while a directed arc from a transaction node to a resource indicates that the transaction has placed an intention lock on that resource. A cycle in the WFG indicates the existence of a deadlock.

The WFS is both a list of transaction-waits-for-transaction strings (in which each transaction is waiting for the next transaction in the string), and a lock history for each transaction in the string. For example, the WFS [T1{W(R2A), IW(R3B)}; T4{W(R3B)}] shows that T1 is waiting for T4, and each transaction's lock history is in brackets. A transaction may also bring along other information, such as a metric representing its execution cost, but such information is not included in this paper as it is outside the primary function of the proposed deadlock detector. We assume that each transaction or agent will have a globally unique identifier that indicates its site of origin. The deadlock can be detected directly from WFSs without constructing the WFG by simply detecting whether any transaction recurs more than once in the WFS. This condition is equivalent to having a cycle in the WFG. In the description of our algorithm we use the phrase *check for deadlock* to mean the detection of a cycle or cycles in the WFG or in the WFS.

Each site in the system has a distributed deadlock detector (copy of the same algorithm) that performs deadlock detection for transactions or agents at that site. Several sites can simultaneously be working on detection of any potential deadlock cycle.

The basic premise of the proposed algorithm is to detect deadlock cycles with the least possible delay and number of intersite messages. On the basis of findings by Gray and others [5] that cycles of length 2 occur more frequently than cycles of length 3, and cycles of length 3 occur more frequently than cycles of length 4, and so on, the proposed algorithm uses a staged approach to deadlock detection. We distinguish two types of deadlock cycles: (a) those that can be detected using only the information available at one site, and (b) those that require intersite messages to detect.

In the proposed algorithm, the first type has been divided into two levels of detection activity. Level one of the proposed algorithm checks for possible deadlock cycles every time a remote resource is requested *and* another transaction is waiting for a resource held by the transaction making the remote resource request. Since level one involves data from the lock table of *one resource*, it should be fast and inexpensive. If the requested resource is still not available "after X units of time" [4], then the probability of a deadlock has increased sufficiently to justify a more complex and time-consuming check in level two. Level two requires more time because it attempts to detect the deadlock by using the lock tables of *all* resources at the site. Level three is intended to detect all remaining deadlocks, that is, deadlocks that require intersite communication.

The first level is designed to detect efficiently most cycles of length 2, although certain more complex deadlock cycles could be detected, depending on the topology of the deadlock cycle. This level uses only information available in the

lock table of the last locked resource if the requested resource is at another site, and in the transaction lock histories. Level one of detection activity can efficiently detect direct global deadlocks of cycle length 2. The global deadlock of two transactions T1 and T2 is *direct* when T1 and T2 deadlock at two sites that are also the *last* sites at which T1 and T2 executed, that is, were not blocked. *Indirect* global deadlock is the one that is not direct. Thus, if T1 and T2 execute only at two sites, they can generate only direct global deadlocks. If they execute at more than two sites, they can also generate indirect global deadlocks, and, as shown in the Appendix, the global direct deadlocks of cycle length 2 constitute a majority of all possible cycle length 2 global deadlocks.

In this paper we consider two types of resources. Type I consists of resources whose intention lock can be determined from a remote site, that is, the transaction can determine the remote resource lock granularity and its mode before migrating to the site of the remote resource. Type II consists of resources whose intention lock granularity and mode can be determined only after the transaction has migrated to the remote site. Type I resources are usually those that have just one level of granularity, namely, the whole resource. Type II can have locking on varying levels of granularity such as, for example, pages of a file in a distributed database system.

However, it has been suggested (R. Obermarck, personal communication, 1983) that in a case of a replicated distributed database system the intention locks can be used for propagation of updates to the remaining copies once the transaction has made such changes to the first (or primary) copy. Thus, the resources in the site of the primary copy are of type II and they become resources of type I for remaining copies at different sites.

The algorithm presented here includes an optimization whereby the WFS is sent to the site to which the awaited transaction has migrated only if the first transaction in the WFS has a higher lexical ordering than the transaction that has migrated. This optimization is similar to one used in [9]. When a site deadlock detector receives a WFS, it substitutes the latest lock histories for any transaction for which it has a later version (the longest lock history is the latest). It then constructs a new WFG or WFS and checks for cycles. If a cycle is found, it must be resolved. If any transactions are waiting for other transactions that have migrated to other sites, the current site must repeat the process of constructing and sending WFGs or WFSs to the sites to which the transactions being waited for have migrated, subject to the constraints of the optimization. If these transactions are at this site and active, deadlock detection activity can cease. The deadlock detection activity will continue until a deadlock is found or it is discovered that there is no deadlock.

The following definitions are used in the description of the algorithm:

IL	Intention lock.
W(x)	Exclusive write lock on resource x.
R(x)	Shared read lock on resource x.
IW(x)	Intention lock(write) on resource x.
IR(x)	Intention lock(read) on resource x.
Sorig(T)	Site of origin of transaction T.
LT(R)	Lock table for resource R.

LH(T) Lock history for transaction T.
Next Field in lock table reflecting the resource a transaction intends to acquire next.
Current Field in lock table reflecting the lock currently held by a transaction.

2.2 The Algorithm

1. {Remote resource R requested or anticipated by transaction or agent T}.
   A. If a type I remote resource is requested, place appropriate IL entry in *next* field of the lock table of the current resource (the last resource locked by T, if any) and in LH(T).
   B. {Start level one detection activity at current site}. Construct a WFG or WFS from lock histories of all transactions holding and requesting R, and, if a type I remote resource is requested, check for deadlock.
   C. If no deadlock is detected:
      (1) Have an agent created at the site of the requested resource and ship the WFS (generated at step 1B. or step 4A.) there.
      (2) Stop.

2. {Local resource R requested}.
   A. If resource R is available: {lock it }.
      (1) Place an appropriate lock in lock table of resource R and in LH(T).
      (2) Stop.
   B. If the resource is not available: {Start level two detection activity}.
      (1) Place appropriate IL in lock table of resource R and in LH(T), and delay X time units.
      (2) If the resource is now available:
         (a) Remove IL from lock table and LH(T).
         (b) Go to step 2A.
      (3) If the resource is not available: {Continue level two activity}.
         (a) Construct a WFG or new WFS using the lock histories of the transactions in the WFSs that have been sent from other sites and the lock histories of all blocked or inactive transactions at this site, and check for deadlock.
         (b) If any deadlock is found, resolve the deadlock.
         (c) If no deadlock is found, delay Y units.
         (d) If the requested resource is now available, go to step 2A.
         (e) If the transaction being awaited is at this site and active, stop.
         (f) If the resource is still not available, go to step 3 {Start level three detection activity}.

3. {Wait-for message generation}.
   A. {Start level three detection activity}. Construct a new WFS either by condensing the latest WFG or by combining all WFSs.
   B. Send the WFS to the site to which the transaction being awaited has gone if the awaited transaction in each substring has a smaller identifier than the first transaction in that substring and stop.

4. {Wait-for message received}.
   If wait-for message received DO:
   A. {Start level 3 detection activity}. Construct a WFG or a new WFS from the lock histories of the transactions in the WFSs from other sites and from the lock histories of all blocked or inactive transactions at this site. (Use the latest WFS from each site.) Check for a deadlock. If deadlock is found, resolve it.
   B. If an awaited transaction has migrated to another site that is different from the one that sent the WFS message, go to step 3. {Repeat WFS generation}.
   C. If the awaited transaction is active, stop.

2.3 Explanation of the Algorithm

Step 1. This step is executed any time a transaction (or agent) T requests a remote resource, or when it determines that it will require a remote resource. The lock table of the resource that the transaction is currently using (or has just finished with) is checked to see whether any other transactions are waiting (i.e., have placed intention locks) for that resource. If so, the lock histories of all transactions requesting and holding the resource are combined into a WFG or a WFS and a check for cycles is made. If no cycle is found, T collects the new WFS and causes an agent to be created at the site of the requested resource.

Step 2. This step is executed each time a local resource is requested, either by an agent (transaction) already at that site or by a newly created agent. If the resource is available, appropriate locks are placed and the resource is granted. If the resource is not available, intention locks are placed in the lock table of the requested resource and in the lock history of the requesting transaction. If the resource is not available after the delay, the chance of a deadlock is higher, so the algorithm shifts to another level of detection. It now uses the lock histories from each blocked or inactive transaction at the site, as well as from any WFSs from other sites that have been brought by migrating transactions. If there are no cycles in the new WFG or WFS, and the resource is still not available after a second delay (also tunable by the system users), the possibility of deadlock is again much greater, but the current site has insufficient information to detect it. Therefore the proposed algorithm progresses to the third level of detection (step 3).

Step 3. The wait-for message generated by this step consists of a collection of substrings. Each substring is a list of transactions, each waiting for the next transaction in the substring. The substring also lists the resources locked or intention locked by each transaction in the substring. This step includes the optimization that a WFS is only sent to another site if the transaction that has migrated has a lower lexical ordering than the first transaction in the substring. For example, for the WFG shown in Figure 2, the WFS would be [T2{W(R2B), IW(R3C)}; T3{W(R3C), IW(R4D)}; T4{W(R4D), IW(R1A)}]. T4 has migrated to site A. The WFS would be sent to site A only if T4's identifier is less than T2's identifier.

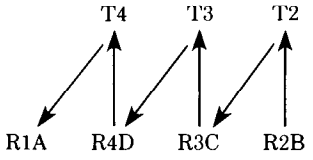


Fig. 2. Example WFG.

Step 4. This step is executed only after a wait-for message has been received. The lock histories of the transactions in the WFSs previously received from other sites, and the lock histories of any blocked or inactive transactions at this site are used to generate a new WFS or WFG. If deadlock is detected, it is resolved; otherwise there is still insufficient information to detect a cycle, and another iteration must be performed. The algorithm repeats by transferring control to step 3. If the transaction being waited for is still active, the algorithm stops.

2.4 Operation of the Algorithm

The operation of the algorithm will be shown by executing it on two examples. In the first example, we assume resources of Type I. T1 is initialized at site A, and it migrates to site B and locks resource R2. It then migrates to site C and locks resource R3. T4 is initialized at site D, and it locks R4 at site D. At this point the lock histories and lock tables are as in Figure 3.

T1 now attempts to acquire resource R4. By step 1, an IL entry is placed in LH(T1) and in LT(R3) at site C. Since there are no intention locks in LT(R3C), the WFS from site C is collected (at this point none exists), and an agent of T1 is created at site D, with T1 "bringing" LH(T1): {W(R2B), W(R3C), IW(R4D)}. Site D now applies step 2B(1) and places the IL entry in LT(R4D) and LH(T1). After the delay it executes step 2B(3) by combining the lock histories of T1 and T4. No cycles are found, but as T4 is still active at site D, the algorithm stops. The current status of the lock tables and lock histories is as in Figure 4.

T4 now determines that it needs to write into resource R3. It applies step 1 and places an IL entry in LH(T4) and LT(R4D). The lock table for R4 is now LT(R4D): T4{W(R4D), IW(R3C)}; T1{IW(R4D)}, and the lock history for T4 is now LH(T4): {W(R4D), IW(R3C)}. Since T1 is waiting for R4, held by T4, and T4 has made a remote type I resource R3 request, step 1B is executed; that is, T4 and T1 lock histories are combined into a WFG or a WFS. The WFG or WFS contain identical information reflecting the fact that, if T4 should move to a site C, this would result in a deadlock (i.e, in T1{LH}T4{LH}T1{LH} WFS at site C). The algorithm can detect this fact without creating the T4 agent at site C, and it can avoid the deadlock without any intersite messages.

Should the resources in this example be of type II, then the algorithm would allow T4 to create the agent at site C. This agent would bring to site C a WFS T1{LH}T4{LH}. After T4 places a lock at site C and deadlocks with T1, a WFS T4{LH}T1{LH} results. Step 2B(3) of the algorithm detects a deadlock by combining T1{LH}T4{LH} and T4{LH}T1{LH} into a new WFS, T1{LH}T4{LH}T1{LH}. Then the deadlock has to be resolved. Notice that the deadlock was detected with no messages. This was due to a sequential migration of transactions. If T1 and T4 were to move simultaneously to request their remote

Site A	Site B	Site C	Site D
LH(T1): {IW(R2B)}	LH(T1): {W(R2B), IW(R3C)} LT(R2B): T1{W(R2B)}	LH(T1): {W(R2B), W(R3C)} LT(R3C): T1{W(R3C)}	LH(T4): {W(R4D)} LT(R4D): T4{W(R4D)}

Fig. 3. Lock histories and lock tables.

Site A	Site B	Site C	Site D
LH(T1): {IW(R2B)}	LH(T1): {W(R2B), IW(R3C)}	LH(T1): {W(R2B), W(R2C), IW(R4D)}	LH(T4): {W(R4D)} LH(T1): {W(R2B), W(R3C), IW(R4D)}
	LT(R2B): T1{W(R2B)}	LT(R3C): T1{W(R3C), IW(R4D)}	LT(R4D): T4{W(R4D)}; T1{IW(R4D)}

Fig. 4. Lock tables and lock histories.

resources, then level two of the algorithm (step 2) would fail to detect the deadlock and it would be detected with one message by level three.

In the second example, which is intended to illustrate level three of our algorithm, we consider only resources of type II. We assume six transactions and five sites. The topology of the deadlocks is shown in Figure 5.

We assume that Ti is a unique identifier for transaction i, such that Ti < Ti + 1. T1, T2, T3, T4, T5, and T6 execute simultaneously at sites B, C, D, A, C, and E. They then deadlock as indicated in Figure 5; that is, T1 is waiting for T2 and T5 at site C, T2 is waiting for T3 at site A, T4 and T6 are waiting for T1 at site B, and T5 is waiting for T6 at site E. Levels one and two of the proposed algorithm fail to detect the deadlocks. Level three will detect all global deadlocks as follows:

When deadlocks occur, there is the following set of WFSs at different sites:

site C: T1{LH}T2{LH}
T1{LH}T5{LH}

site D: T2{LH}T3{LH}

site A: T3{LH}T4{LH}

site B: T6{LH}T1{LH}
T4{LH}T1{LH}

site E: T5{LH}T6{LH}

Step 3 of the proposed algorithm will generate one WFS message from site B to C. When the message is received at site C, the algorithm at site C will construct the following set of new WFSs:

$$\text{WFS1}\begin{cases}\text{T6\{LH\}T1\{LH\}T2\{LH\}}\\ \text{T4\{LH\}T1\{LH\}T2\{LH\}}\end{cases} \qquad \text{WFS2}\begin{cases}\text{T6\{LH\}T1\{LH\}T5\{LH\}}\\ \text{T4\{LH\}T1\{LH\}T5\{LH\}}\end{cases}$$

Since no deadlock is detected, and because T2 and T5 migrated to sites D and E, step 3 of the algorithm will generate two wait-for messages, WFS1 and WFS2, which are then sent to sites D and E, respectively. Sites D and E, upon receiving these messages, will execute step 4 of the algorithm and generate the following

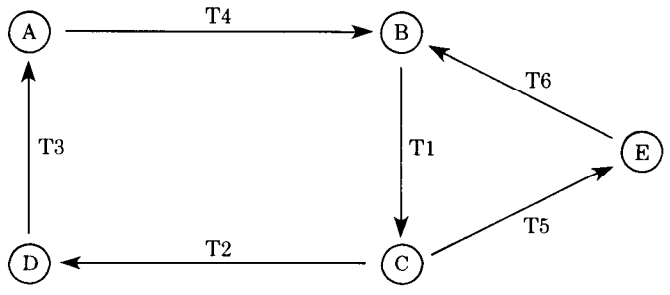


Figure 5

wait-for messages:

site D: T6{LH}T1{LH}T2{LH}T3{LH}
T4{LH}T1{LH}T2{LH}T3{LH} } WFS3

site E: T6{LH}T1{LH}T5{LH}T6{LH}
T4{LH}T1{LH}T5{LH}T6{LH}

At this time, step 4 at site E will detect and resolve a global deadlock T6T1T5T6. However, step 4 at site D will generate wait-for message WFS3 and send it to site A. Site A, upon receiving it, will execute step 4 of the algorithm and generate the following WFSs:

T6{LH}T1{LH}T2{LH}T3{LH}T4{LH} WFS4
T4{LH}T1{LH}T2{LH}T3{LH}T4{LH} WFS5

At this time, the T4T1T2T3T4 deadlock is detected and resolved. Although we do not address deadlock resolution in this paper, it is obvious that T4 should be rolled back and reexecuted after T3 finishes at site A. Similarly, when the T6T1T5T6 deadlock is detected at site E, it should be resolved by rolling back T6 and reexecuting it after T5 finishes at site E. Clearly, if the T4T1T2T3T4 deadlock is resolved as indicated, the algorithm at site A should recognize that removing T4 from WFS5 and WFS4 will result in T3, T2, and T1 terminations. Then WFS4 is reduced to T6{LH}T4{LH} and WFS5 to T4{LH}. By inspecting the T6 and T4 lock histories, it is easy to see that WFS T6{LH}T4{LH} is a false wait-for condition, and this WFS can be discarded. This indicates the usefulness of LHs for detection of false deadlocks. If the described deadlocks occur in a sequential manner, for example, Ti + 1 leaving a site only after Ti gets blocked there by Ti + 1, then the deadlock detection process can catch up with transactions that are still active. In such a scenario the algorithm stops at step 4C. When an active transaction requests a remote resource (step 1), the algorithm will ship the WFS generated at step 4A to a remote resource site (step 1C(1)). This means that all still-executing transactions carry with them information to facilitate deadlock detection, possibly without any additional messages, should they become deadlocked.

3. INFORMAL PROOF OF CORRECTNESS

In general, a deadlock cycle can have many different topologies. For the model of transaction execution used in the proposed algorithm (migration of agents of transactions), these different topologies can be loosely grouped into four

categories. Category A involves local deadlocks in which all the resources and transactions involved in the deadlock are local, that is, located at one site, and thus the transactions involved have not locked any resources at other sites. Category B is the same as category A, with the exception that the transactions are nonlocal, that is, they may have locked resources at other sites. Category C contains all direct global deadlocks of cycle length 2 for resources of type I and all direct global deadlocks of cycle length 2 for resources of type II when transactions migrate sequentially; that is, each transaction involved in a deadlock gets blocked before the blocking transaction leaves the site. Category D is a generalization of category C deadlocks; any number of transactions and resources may be directly or indirectly deadlocked at any number of sites. For each category, it will be argued that the algorithm detects all possible deadlocks in that category, and that the algorithm does not detect "false" deadlocks except in the case in which a transaction that was involved in a deadlock has aborted, but its agents have not yet been notified.

If all the transactions and resources involved in a deadlock are located at the same site and none of the transactions have locked resources at other sites, each transaction's lock history will be an accurate and complete snapshot of the locks placed by that transaction. For this category of deadlock cycles, step 2B(3) (level two) will combine the lock histories of all the blocked or inactive transactions at the site. This information will be a complete and accurate global snapshot of the deadlock cycle, and hence the deadlock will be detected.

Deadlocks in the second category are those in which all the transactions and resources involved are at one site, but the transactions involved may have locked resources at other sites before creating the agent at this site. The argument to show that all deadlocks in this category will be detected by the proposed algorithm is essentially the same as the one used for the first category. Since all the transactions involved in the deadlock are currently at this site, their lock histories are complete and accurate insofar as they pertain to the deadlock cycle. It is possible, in the case of concurrent execution of a transaction's agents, for an agent involved in a deadlock to be unaware of resources locked by other agents of that transaction that are executing concurrently and will probably still be active. The only difference between this case and the preceding is that the WFGs or WFSs constructed by step 2B(3) may contain information about other locks held by the transactions involved, but the necessary information concerning the deadlock cycle will certainly be present.

Deadlocks in the third category will be detected by level one because, if for resources of type I the transaction migrations occur simultaneously, the "next" field of the lock table of the requested resource would show an intention lock on the other resource, and this cycle would be detected by step 2B(3). If the migrations occurred sequentially, the second transaction would, for resources of type I, place an intention lock in the lock table of its last locked resource before it migrates. The level one check of step 1B would cause a WFG or a WFS to be constructed and this would reveal the deadlock cycle. For resources of type II the direct sequential deadlocks are detected by step 2B(3).

The fourth category of deadlock cycles is a generalization of the third. Deadlock cycles in this category may involve any number of transactions and resources at

any number of sites. If level two cannot detect the cycle in step 2B(3) with information at that site, level three causes a WFS containing this site's information to be sent to the site to which the transaction has migrated if the transaction that has migrated has a lower unique identifier than the first transaction in the substring. Steps 3 and 4 cause this process to be continued, with each site adding additional information, until a site contains enough information to detect a deadlock cycle or determine that no deadlock exists, regardless of the number of migrations made by a transaction. To show that this process will continue until the deadlock is detected, we refer to the proof in [9], since the optimization in the proposed and in Obermarck's algorithm is essentially the same.

A false deadlock is an anomaly—a nonexistent deadlock cycle is detected by a deadlock detection algorithm—and occurs usually as a result of incorrect or obsolete information. The proposed algorithm uses the latest copy of a transaction's lock history to avoid false deadlock detection. This information cannot be incorrect in the sense of invalid entries, although it may be incomplete. This means that a WFG or WFS constructed from incomplete versions of lock histories may have insufficient information to detect a deadlock at that particular level of detection activity or iteration of level three activity, but that it will not have incorrect information. When a transaction that has agents at two or more sites commits or aborts, however, it is possible that the commit or abort messages to other agents of that transaction may be delayed. Obviously, a transaction that is ready to commit cannot have any of its agents in a blocked state (and therefore in a possible deadlock condition), so its agents can either be only active or inactive. Although inactive agents may be awaited by agents of other transactions, no lock history or lock table can show that an agent of the transaction that is about to commit is waiting for another transaction, so no false deadlocks can exist. Therefore, only the possibility of a transaction that is in the process of aborting and thus causing a false deadlock to be detected must be considered. Suppose an agent of a transaction decides to abort, but before its abort message reaches another agent of that transaction, a deadlock is found involving that transaction. Technically, this could be considered a false deadlock, since one of the transactions involved has aborted, probably breaking the deadlock cycle. If the deadlock cycle is complex, and the proposed algorithm is performing level two or three detection activity, the delays introduced in steps 2B(1) and 2B(3) (c) should allow the abort message to arrive. For what we believe to be the rare occurrences where the abort message does not arrive, it would probably be more efficient to let the deadlock detection algorithm resolve the (false) deadlock rather than have the algorithm perform some explicit action (such as delaying before resolving any detected deadlock cycle) each time it detects a deadlock.

4. PERFORMANCE ANALYSIS

To check the efficiency (in terms of intersite messages) of the algorithm, it was analyzed in several global deadlock scenarios. The algorithm proposed in [9] was also analyzed in these scenarios. Since the majority of deadlocks that occur will be of length 2 or 3, three test cases involving deadlock cycles of those lengths

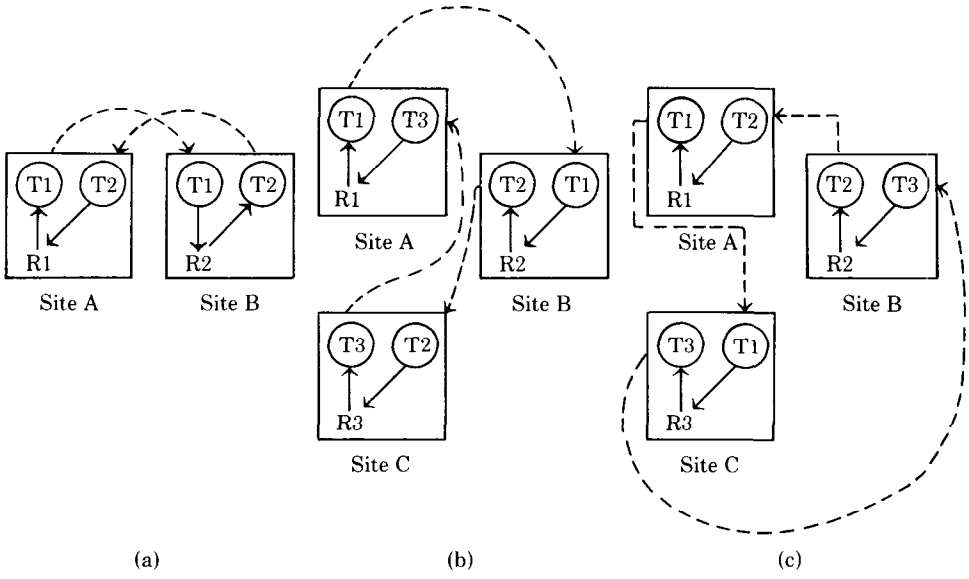


Fig. 6. Deadlock cycles used in performance analysis. (a) Case 1. (b) Case 2. (c) Case 3.

will be used for the comparison. It is assumed that the transactions are lexically ordered T1 < T2 < T3. These cases are shown in Figure 6. T1 originated at site A and holds a lock on R1, and T2 originated at site B and holds a lock on R2. In cases 2 and 3, T3 originated at site C and holds a lock on R3. In case 1, T1 has migrated to site B and requested R2, while T2 has migrated to site A and requested R1. In case 2, T1 has migrated to site B and requested R2, T2 has migrated to site C and requested R3, and T3 has migrated to site A and requested R1. In case 3, T1 has migrated to site C and requested R3, T2 has migrated to site A and requested R1, and T3 has migrated to site B and requested R2.

For case 1, where the deadlock cycle is of length 2 and the resources are of type I, the proposed algorithm requires no additional messages for deadlock detection, while the algorithm in [9] requires one message. If resources are of type II, then the proposed algorithm requires no messages when deadlock is sequential, that is, when one of the transactions reaches a remote site before the other one leaves that site. If the deadlock is not sequential, then the proposed algorithm requires one message, as does the algorithm in [9]. For case 2, with a deadlock cycle of length 3 and resources of type II, the algorithm in [9] requires two messages. The number of messages required by the proposed algorithm is dependent on the timing of the transaction migrations. If the migrations occur at different times (i.e., sequentially), no messages are required. If, however, the migrations happen to occur simultaneously, only one message is generated because of the optimization. If resources are of type II, then the proposed algorithm does not require any messages when the deadlock is sequential. Otherwise, it requires two messages as the algorithm in [9]. We note here that the proposed algorithm will avoid detection of false deadlock, whereas the algorithm in [9] will not. A similar situation occurs in case 3. If we consider

resources of type I and transaction migrations occur simultaneously, two messages will be generated by the proposed algorithm, although one of these messages is redundant; that is, any single message is sufficient to detect deadlock. If transaction migrations occur at different times (i.e., sequentially) then no messages are required. The algorithm in [9] requires three messages, regardless of the timing of the migrations. If we consider resources of type II, then for sequential deadlock the proposed algorithm does not require any messages. For any other deadlocks in scenario 3 it requires three messages, as does the algorithm in [9].

As pointed out in [9] it is apparent that in the overwhelming majority of cases the global deadlocks are of cycles of length 2 involving two sites. For resources of type I no messages are required to detect direct global deadlocks of cycle length 2 by the proposed algorithm. In order to provide the evaluation of both algorithms for global deadlocks with cycle length $n > 2$ and for resources of type I, we assume that n nonlocal (or global) transactions are involved in the global deadlock such that at each of n sites only one transaction is blocked by another transaction and each transaction needs to execute only at two sites. Then the number of messages needed by the proposed algorithm for the worst case scenario (when all the transactions involved migrate simultaneously) and for resources of type I can be shown to be $N - 1$, where $N = \sum_{k=1}^{n} (n - k)$. Under the same circumstances it can be shown that the algorithm in [9] requires N messages regardless of sequencing of transaction migrations; that is, the number of messages depends only on the number of transactions involved in the deadlock. Thus for a cycle of length 3, the number of messages required for the worst case would be two for the proposed algorithm and three for the algorithm in [9].

For resources of type II the proposed algorithm generates the same number of messages (N) as the algorithm in [9] only when all transactions involved migrate simultaneously and the lexical ordering of the transactions is such that $n - 1$ messages are sent on the first iteration. In all other cases the proposed algorithm generates fewer messages than the algorithm in [9].

Our analysis has been limited to a number of intersite messages needed to detect global deadlocks because we consider messages to be the most critical performance parameter for distributed systems. However, we must point out that the proposed algorithm minimizes the number of messages at the cost of larger messages and at the cost of doing some computation before each remote resource request. Thus the proposed algorithm has a constant overhead that may not be present in other algorithms.

This constant overhead (see step 1B of the proposed algorithm) consists of constructing WFSs from lock histories of all transactions waiting for a resource whenever a transaction holding that resource makes a request for a remote resource. If the request is for the resource of type I, the deadlock detection is executed as well. The construction of WFSs is a simple operation of copying a queue of waiting transactions and their lock histories into a remote resource request message buffer. The deadlock check is an equally simple operation consisting of one scan of the WFS.

The constant overhead exhibited by the proposed algorithm is due to dissipating transaction state information within a distributed system. The potential

payoff comes in the form of (a) avoiding most frequent global deadlocks on resources of type I, (b) avoiding sending a remote message and the creation of a remote agent, and (c) reduction of deadlock detection messages for resources of type II.

We estimate the cost of one message processing at both sites to be 3500 instructions. Our estimate is based on published data in [1] and on our distributed UNIX[1] prototype measurements. Our data indicate that the remote agent creation or activation can vary widely depending on whether such process is in user or system space. In the former case the cost can be as large as 35,000 instructions and in the latter as small as 2000 instructions or less. We estimate context switch due to wait periods in the proposed algorithm to be about 150 instructions and therefore negligible. We estimate the minimal cost of deadlock detection to be two messages and the minimal cost of deadlock resolution to be one message. Thus the minimal cost of allowing the deadlock to occur is one remote resource request message that deadlocks the transaction and three messages to detect and resolve the deadlock, plus the setup of a remote agent. Depending on a remote agent implementation, the minimal cost of deadlock is 16,000 or 49,000 instructions.

Let us assume resources of type I and that deadlocks occur with a probability of 0.04, that is, 1 in 250 remote resource requests will cause global deadlock [5, 9]. In order to justify the proposed algorithm overhead, the algorithm would have to check 250 times the waiting transaction queue in fewer than 16,000 or 49,000 instructions. Thus one WFS construction and its scan for a deadlock should consume fewer than 64 or 196 instructions, depending on remote agent implementation.

For resources of type II the proposed algorithm saves some deadlock detection and resolution messages. If we assume that the cycle of length-2 global deadlocks on resources of type II also occur with a probability 0.04 and that half of these are sequential deadlocks, then the algorithm saves three messages for each sequential deadlock occurrence. Thus for resources of type II the algorithm should copy the waiting transactions queue into a remote resource request message buffer in fewer than 21 instructions. As an example, we point out that on MC 68000, 21 instructions move 84 bytes of data.

In any case we consider adding 21 or even 100 instructions into the remote resource request message execution path a reasonable overhead with respect to a message setup overhead of 3500 instructions. Moreover we believe that in some applications, like office automation, where the deadlocks can occur more frequently than in traditional applications, the proposed algorithm should have a distinct performance advantage.

5. CONCLUSIONS

The proposed algorithm has been shown to be able to detect deadlock with a smaller number of intersite messages than existing algorithms for deadlock detection in distributed computing systems. We have shown that for the deadlock scenarios analyzed in this paper the proposed algorithm requires from 0 to

[1] UNIX is a trademark of AT&T Bell Laboratories.

$N - 1$ (where $N = \sum_{k=1}^{n} (n - k)$) messages for resources of type I and from 0 to N messages for resources of type II, where n is the number of transactions and sites involved in the deadlock cycle. It requires no messages when the transaction migrations leading to the deadlock occur sequentially. This is because when a transaction migrates, it "brings along" all pertinent wait-for information from its current site, and thus each site has more information than the sites would have using the algorithm described in [9].

The most important point is that the proposed algorithm can detect the most frequent deadlocks with a minimum of intersite messages. When resource locks can be determined without access to a site where the resource is located, then the proposed algorithm requires no intersite messages for direct global deadlocks of cycle length 2. Moreover, one-half of such deadlocks can be avoided before they occur. Similarly, for resources of type I, no messages are required for deadlocks of cycle length >2 when (a) a sequential migration of transactions in order of their lexically ordered unique identifiers has occurred, regardless of the number of transactions or sites involved, or (b) the deadlock is direct and involves only two sites where at one site only two transactions conflict and an arbitrary number of transactions conflict at the other site. For all other types of deadlocks, the proposed algorithm requires one less message than the algorithm in [9].

When resource locks, that is, their granularity and their mode, can be determined only at a site where the resource is located, as is the case in distributed database management systems, then only one-half of all global direct deadlocks of cycle length 2 can be detected by the proposed algorithm without any messages. The Appendix shows that the global direct deadlocks of cycle length 2 constitute a majority of all possible cycle-length-2 global deadlocks.

However, we must point out that the proposed algorithm has a constant overhead due to more information in lock tables and to a frequent checking for cycle-length-2 deadlocks. If deadlocks do occur very rarely, then the constant overhead in our algorithm makes it less desirable despite its lower number of messages. Thus the proposed algorithm would most likely be useful for distributed applications with most resources of type I or for applications where deadlocks occur more frequently than in conventional applications. However, we have also shown that the proposed algorithm performance is comparable with other algorithms even for conventional applications.

The proposed algorithm could be modified by combining levels one and two, if the number of resources and transactions in the system are small, and therefore the cost of creating WFGs or WFSs at level two would be comparable with the cost of the level-one WFG construction. The delays that have been built into the algorithm can be adjusted empirically to determine the optimum delays for a particular implementation.

APPENDIX

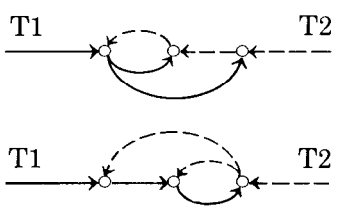
I is a number of indirect global deadlocks and D is a number of direct global deadlocks.

S is a number of sites at which transactions execute and conflict for global deadlocks of length 2.

$S = 2$:

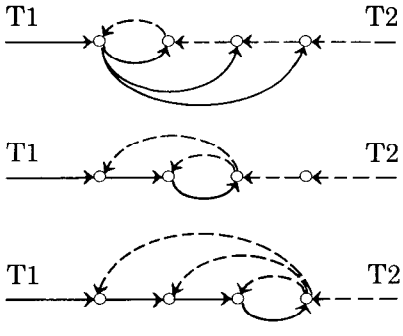
$$\frac{I}{D} = \frac{0}{2} = 0.$$

$S = 3$:



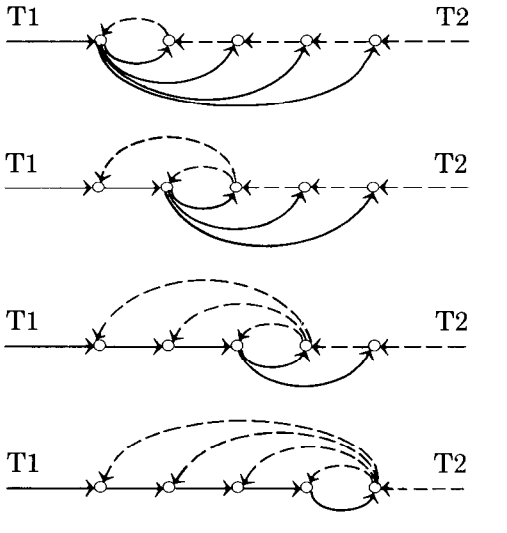
$$\frac{I}{D} = \frac{2}{6} = \frac{1}{3}.$$

$S = 4$:



$$\frac{I}{D} = \frac{6}{12} = \frac{1}{2}.$$

$S = 5$:



$$\frac{I}{D} = \frac{12}{20} = \frac{3}{5}.$$

We can conclude that for deadlocks of cycle length 2 the ratio

$$\frac{I}{D} = \frac{(S-2)(S-1)}{S(S-1)} = \frac{S-2}{S}.$$

Thus the direct deadlocks are a majority of all global deadlocks of cycle length 2.

ACKNOWLEDGMENTS

The author would like to express his appreciation to the referees and to Ron Obermarck of IBM Research, San Jose, for their comments, which considerably contributed to the improvement of the final version of the paper. The author also wishes to acknowledge the support of Joel Tesler and other members of the distributed UNIX project team at Hewlett-Packard Labs in obtaining needed measurements.

REFERENCES

1. BIRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.
2. GLIGOR, V., AND SHATTUCK, S. On deadlock detection in distributed systems. *IEEE Trans. Softw. Eng. SE-6* (Sept. 1980), 435–440.
3. GOLDMAN, B. Deadlock detection in computer networks. Tech. Rep. TR-MIT/LCS/TR-185, Massachusetts Institute of Technology, Cambridge, Mass., Sept. 1977.
4. GRAY, J. Notes on data base operating systems. Research Rep. RJ2188 (30001), IBM Research Division, San Jose, Calif., Feb. 1978.
5. GRAY, J., HOMAN, P., KORTH, H., AND OBERMARCK, R. A straw man analysis of the probability of waiting and deadlock in a distributed database system. Paper presented at *5th Berkeley Workshop on Distributed Data Management and Computer Networks* (San Francisco, February). Lawrence Berkeley Laboratory, University of California, Berkeley, and U.S. Department of Energy, 1981.
6. GRAY, J. The transaction concept: Virtues and limitations. Tech. Rep. TR81.3, Tandem, June 1981.
7. LINDSAY, B., SELINGER, P., GALTIERI, C., GRAY, V., LORIE, R., PUTZOLU, F., TRAIGER, I. L., AND WADE, B. Notes on distributed databases. Tech. Rep. RJ2571, IBM Research, San Jose, Calif., July 1979.
8. MENASCE, D., AND MUNTZ, R. Locking and deadlock detection in distributed data bases. *IEEE Trans. Softw. Eng. SE-5*, 3 (May 1979), 195–202.
9. OBERMARCK, R. Distributed deadlock detection algorithm. *ACM Trans. Database Syst. 7*, 2 (June 1982), 187–209.
10. TSAI, W. C., AND BELFORD, G. Detecting deadlock in a distributed system. In *Proceedings INFOCOM* (Las Vegas, Nev., Apr. 1). IEEE, New York, 1982.

Received March 1983; revised August 1983, May 1984, August 1985; accepted June 1986