A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution

ALOK N. CHOUDHARY, STUDENT MEMBER, IEEE, WALTER H. KOHLER, SENIOR MEMBER, IEEE, JOHN A. STANKOVIC, SENIOR MEMBER, IEEE, AND DON TOWSLEY, MEMBER, IEEE

Abstract—This paper presents a modified priority based probe algorithm for deadlock detection and resolution in distributed database systems. The original priority based probe algorithm was presented by Sinha and Natarajan based on work by Chandy, Misra, and Haas. Various examples are used to show that the original priority based algorithm either fails to detect deadlocks or reports deadlocks which do not exist in many situations. A modified algorithm which eliminates these problems is proposed. This algorithm has been tested through simulation and appears to be error free. Finally, the performance of the modified algorithm is briefly discussed.

Index Terms—Concurrency control, deadlock detection, distributed database.

I. INTRODUCTION

THIS paper presents a modified priority based probe algorithm for deadlock detection and resolution in distributed database systems. The original priority based probe algorithm was presented by Sinha and Natarajan in [9] based on work by Chandy, Misra, and Haas [3], [4]. Sinha and Natarajan [9] assigned priorities to transactions and used the priorities to reduce the number of probe messages that are forwarded. Two variations of the algorithm were discussed: a basic algorithm to detect deadlocks when only exclusive lock requests by transactions are allowed and an extended algorithm when shared and multiple lock requests are allowed. Sinha and Natarajan did not implement the algorithm, nor prove its correctness.

In this paper we show that in many situations Sinha and Natarajan's algorithm either fails to detect a deadlock cycle or detects a nonexistent (false) deadlock. We also explain why such situations arise and modify their algorithm in order to make it work correctly. The final result is a refined priority based probe algorithm to detect and resolve distributed deadlocks. We have extensively tested

this algorithm through simulation without finding any errors.

The rest of this paper is organized as follows. The distributed database model is briefly discussed in Section II. For a detailed description of the database model and the original algorithms the reader is referred to [9]. Section III contains examples illustrating those situations in which the original algorithm either fails to detect deadlocks or detects false deadlocks. Modifications to eliminate these problems are suggested with each example. Our modified probe algorithm is then presented in Section IV. The tests performed on our algorithm are described in Section V. The impact of the modifications on the performance of the algorithm is also discussed in Section V. Finally, Section VI summarizes the paper.

II. THE DISTRIBUTED DATABASE SYSTEM MODEL

A distributed database system is modeled as a collection of sites (or nodes) which communicate through messages. The messages arrive at a destination site in the same order in which they were sent from a source site. Messages are neither lost nor duplicated and are transmitted error-free. See Sinha and Natarajan [9] and Chandy, Misra, and Haas [3], [4] for further details.

Transactions execute concurrently in some globally serializable order [1], [2]. Each site has many data items which are accessed by transactions. Each data item is managed by a data manager which has the exclusive right to operate on it. Transactions use the two phase locking protocol (2PL) to access data items [6]. If a transaction wants to lock a data item, it must send a lock request to the data manager managing that data item. A data manager grants a lock request on a data item if it is free (unlocked), otherwise it places the request in a queue, called the Request_Q. A transaction can only access a data item after it has acquired a lock on the data item. A transaction which has obtained a lock on a data item is called the *holder* of the data item, and a transaction waiting in the queue for a lock is called a *requester*. If shared locks are allowed then there may be more than one holder of a data item. However we focus only on the basic algorithm by Sinha and Natarajan which assumes that transactions only request exclusive locks. Although a data manager may manage many data items, it is assumed that a data manager manages only one data item [9] in order to simplify the understanding of the algorithm.

A transaction can be in one of two states: *active* or *wait*. If a transaction has a lock request pending then it is in the wait state, otherwise it is active. A transaction changes its state from active to wait if its lock request cannot be granted. Once a transaction enters the wait state, it cannot request more locks until all of its pending requests are satisfied.

In Sinha and Natarajan's algorithm, transactions are assigned priorities where each transaction has a distinct priority. We use the following notation to denote the priority ordering. For two transactions T_i and T_j, Priority (T_i) > Priority (T_j) iff $i < j$. This will be denoted as $T_i > T_j$.

III. ERRORS AND DEFICIENCIES

In this section we discuss the cases in which the algorithm presented by Sinha and Natarajan [9] either fails to detect a deadlock cycle or detects a false deadlock (a deadlock which does not exist). We also suggest corrections to the algorithm which will eliminate these problems. For the convenience of the reader, the basic algorithm [9, pp. 69-70] is included in the Appendix. However, the reader is refered to [9] for a detailed description of this algorithm.

There are three basic deficiencies in Sinha and Natarajan's algorithm. First, some deadlocks are not detected because probes are initiated and transmitted only once. Second, there are numerous instances when the algorithm detects false deadlocks or fails to detect some deadlocks because it overlocks the possibility that transactions may wait transitively on a deadlock cycle. Third, there are times when the algorithm detects false deadlocks because transactions and data managers save old probes.

In order to understand the problems with the Sinha and Natarajan algorithm and the new algorithm, we introduce the concept of an *antagonistic conflict* [9].

Definition 1: An antagonistic conflict occurs between two transactions if one transaction (called the holder) locks a data item and the other transaction with a higher priority (called the requester) requests that data item. The requester is said to face an antagonistic conflict.

A deadlock is detected by circulating a message, called the probe, through the deadlock cycle. The occurrence of an antagonistic conflict at a data item triggers the initiation of a probe. A timeout period can also be introduced to delay the initiation of the algorithm. A probe is an ordered pair (*initiator*, *junior*), where *initiator* denotes the requester which faced an antagonistic conflict and triggered the deadlock detection computation. The element junior denotes the transaction with the lowest priority from among the transactions traversed by the probe.

A data manager sends a probe only to the transaction holding its data, while a transaction sends a probe only to the data manager from which it is waiting to receive a lock grant.

The notation

$$T_i \qquad X_k \qquad T_j$$
$$\bullet\!\longrightarrow\!\bullet$$

means that transaction T_i is waiting for a data item X_k, presently held by transaction T_j. The arc between T_i and T_j simply denotes the wait-for relationship. Note that T_i does not communicate with T_j directly. T_i sends its request to the data manager managing data item X_k, denoted as $DM(X_k)$. If $T_i > T_j$ when $DM(X_k)$ initiates a probe (T_i, T_j) and sends it to T_j. Furthermore, $DM(X_k)$ forwards each probe (*initiator*, *junior*) it receives from T_i to T_j if *initiator* > T_j.

Before we proceed with the different examples of how Sinha and Natarajan's algorithm fails, we provide the following concise, high level description of their algorithm. The algorithm consists of two parts. The first part is concerned with the detection of the deadlock through the propagation of probes. Throughout this phase, transactions may be required to store some of these probes. The second part of the algorithm is concerned with resolving a deadlock. This includes first notifying the lowest priority transaction within the cycle that it will be the abort victim and a subsequent phase to remove unnecessary probes stored by other transactions in the cycle. This last phase is initiated by the deadlock victim through the propagation of a special clean message. Once again, we refer the reader to [9] for complete details of the algorithm.
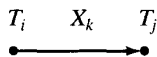
A. Undetected Deadlocks

In this subsection, we describe conditions under which a deadlock cycle will not be detected by the Sinha and Natarajan algorithm. We describe how the algorithm can be modified to avoid this problem.

Consider the example shown in Fig. 1(a). Assume that $DM(X_1)$ initiated a probe (T_1, T_5) that propagated to T_5, then to T_4 and finally to T_3. Transactions T_5, T_4, and T_3 each store the probe (T_1, T_5) in their respective probe_Q's. Now suppose T_3 commits and releases its locks. If T_2 is first in the request_Q it will be granted the lock on X_3.[1] This situation is shown in Fig. 1(b). Now, if T_2 requests a lock for a data item presently held by T_1, say X_4 as shown in Fig. 1(c) using a dashed line, a deadlock cycle will be formed. According to Sinha and Natarajan's algorithm, this deadlock will never be detected. $DM(X_4)$ will not initiate a probe because T_3 does not face an antagonistic conflict. Even if it faced an antagonistic conflict and initiated a probe, the probe would stop at T_1 because T_1 is the highest priority transaction in the cycle. The probe (T_1, T_5), the only probe which can potentially detect this deadlock cycle, was not and will never be propagated to T_2.

The following must be added to part A.1 of Sinha and Natarajan's algorithm [9] (see Appendix).

c) When a transaction completes or aborts it releases its locks. The data manager associated with each released data item assigns the lock for the data item to some transaction waiting for that data item (if one exists). Each data manager then requests all
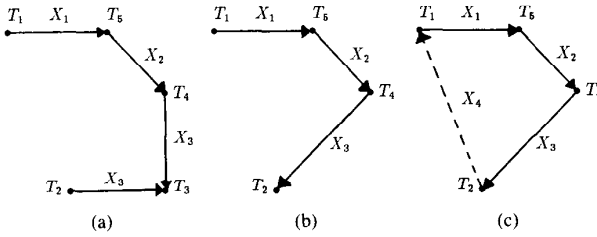
Fig. 1. An undetected deadlock. (a) Initial state. (b) State after release by T_3. (c) Resulting undetected deadlock.

Fig. 2. Example of a false deadlock. (a) Initial deadlock. (b) Resolution. (c) False deadlock.

remaining transactions waiting on the new lock to transmit their complete probe_Q's to itself (The identities of these transactions are obtained from the data manager's request_Q.) The data manager forwards each received probe (initiator, junior) to the new holder of the lock for which *initiator* exceeds the priority of the new holder.

Let us return to the example in Fig. 1. At the time that T_3 completes, it releases its locks and $DM(X_3)$ grants T_2 the lock on X_3. $DM(X_3)$ then requests all of the probes in T_4's probe_Q. Transaction T_4 sends probe (T_1, T_5) to $DM(X_3)$ which $DM(X_3)$ in turn transfers to T_2. Therefore, when the deadlock cycle is formed in Fig. 1(c), T_2 will send the probe (T_1, T_5) to $DM(X_4)$ and consequently, $DM(X_4)$ will detect the deadlock cycle.

Similar examples can be constructed such that a deadlock will form which will not be detected following the abortion of a transaction. Consequently, the above actions must also be performed when transactions abort.

B. False Deadlocks

In addition to the above mentioned problems, the algorithm detects false deadlocks. "False deadlock" does not have a universal definition. However, the most widely accepted definition of a false deadlock is "a deadlock which does not exist when reported by an algorithm." We present two examples that illustrate cases where false deadlocks are detected due to external (or transitive) probes and old probes.

1) A False Deadlock Due to External Probe: Consider the case shown in Fig. 2(a). In this example, transactions T_1 and T_2 have locked data items X_1 (not shown in the figure because no other transactions are waiting on them) and X_4, respectively, and T_4 has locked data items X_2 and X_3. In addition, T_1, T_2, and T_4 have requested items X_3, X_2, and X_4, respectively. Although T_1 does not form a part of the deadlock cycle, its probe (T_1, T_4) will be stored in the probe_Q of T_4 and T_2. When the deadlock cycle is detected by the probe initiated by T_2, T_4 is chosen as the transaction to be aborted in order to resolve the deadlock. Before it aborts, T_4 sends a clean $(victim, initiator)$ message to the data manager for which it was waiting $(DM(X_4)$ in this case). This message is transferred between the transactions within the deadlock cycle until it reaches the initiator, T_2, where it is discarded. The purpose of the clean message is to allow transactions to remove probes that contain T_4. The argument for allowing
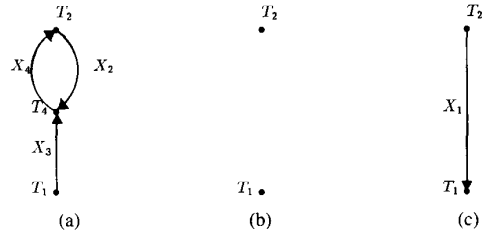
T_2 to discard the clean message is that T_2 should not have any probe in its probe_Q containing the *victim*, T_4, as its *initiator or junior*, because T_2 is the highest priority transaction in the deadlock cycle. This argument, however, is valid only when there are no transactions waiting transitively on a deadlock cycle [such as T_1 waiting on the cycle T_2T_4 in Fig. 2(a)]. As we can observe in this example, T_2's probe_Q contains a probe (T_1, T_4). After the deadlock resolution, both T_1 and T_2 acquire the locks to X_3 and X_2, respectively, and become active again as illustrated in Fig. 2(b). Now assume T_2 requests a data item X_1 held by T_1. It waits for T_1 as shown in Fig. 2(c). According to the algorithm (A.2.b of the original algorithm), T_2 will transmit a copy of the probe (T_1, T_4) to $DM(X_1)$. Upon receipt of this probe, $DM(X_1)$ will declare a deadlock with T_4 as the *victim*. Therefore, we observe that not only is a deadlock detected which does not exist but, in this example, the victim itself does not exist in the system!

The algorithm should be modified in the following way in order to eliminate the false deadlock in the example. Once the transaction is chosen to abort, it should initiate a clean message which should not be discarded until it returns to the transaction to be aborted. Each transaction in the cycle, (initiator included) should use the information contained in the message to clean its probe_Q.

2) False Deadlocks Due to Old Information: We now consider an example that requires a significant change to the original algorithm. Consider the example in Fig. 3(a). A deadlock exists between T_2 and T_4. Transactions T_1, T_3, and T_5 wait transitively on T_4. Assume T_1's probe (T_1, T_5) has propagated to T_2 and T_4 and is stored in their respective probe_Q's. The cycle contains only two transactions and is detected by T_2's probe. After the resolution, T_2's probe_Q will be cleaned of any probe containing T_4 as a consequence of the modification suggested in Section III-B-1. However, T_2's probe_Q contains the probe (T_1, T_5). Fig. 3(b) shows the wait-for relationship after the resolution. Note that X_3 has been granted to T_3 and X_4 to T_2. Now, assume that T_2 requests data item X_5 held by T_1 as shown in Fig. 3(c). Consequently, $DM(X_5)$ will declare a deadlock because it will receive the probe (T_1, T_5) stored in T_2's probe_Q. Obviously it is a false deadlock and T_5 will be aborted unnecessarily.

In general, each transaction waiting transitively on a deadlock cycle can initiate a probe[2] (like T_1 did in this

---

[2]Antagonistic conflict criteria has to be satisfied. However, in the worst case it can be satisfied for all but one transaction [9].
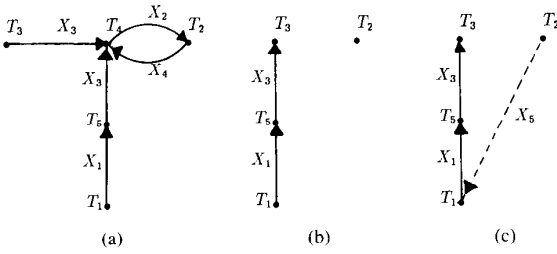
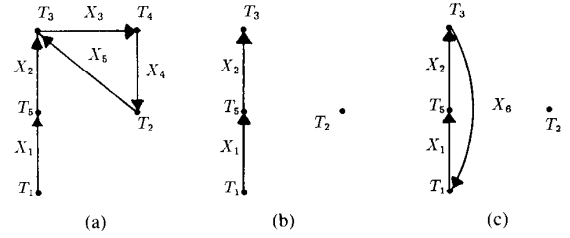Fig. 3. Another false deadlock example. (a) Initial deadlock. (b) After resolution. (c) False deadlock.

Fig. 4. An example illustrating the necessity of reinitiating and retransmitting probes. (a) Initial deadlock. (b) After resolution. (c) Undetected deadlock.

example) which can potentially declare a false deadlock. We can think of two alternative solutions to this problem. The first solution is to ignore this kind of false deadlock under the assumption that it will rarely occur. The second solution is to modify the algorithm in order to eliminate it. If we choose to avoid this type of false deadlock, then the probe_Q's of all the transactions involved in a deadlock cycle should be cleansed of all the probes upon receipt of the clean message. Unfortunately, this cleansing has the following side effect—some future deadlocks may now remain undetected because of the removal of some probes during the cleansing. To avoid this situation, all of the transactions which were involved in the detected deadlock cycle or were waiting for data items held by the transactions involved in the deadlock cycle should retransmit and/or reinitiate the probes. This change will generate a larger number of messages than the original Sinha and Natarajan algorithm thus negating some of the performance benefits claimed in [9].

C. The Necessity of Probe Retransmission and Reinitiation

In this subsection we show why *reinitiation and retransmission* of probes by those transactions which were earlier waiting transitively on a deadlock cycle, is required. Fig. 4 shows a situation similar to one shown in Fig. 3. A deadlock cycle exists between transactions T_3, T_4, and T_2. Transactions T_1 and T_5 wait transitively on the deadlock cycle. Suppose T_1 initiated a probe (T_1, T_5) which is stored in the probe_Q's of T_3, T_4 and T_2. After the deadlock is detected at $DM(X_4)$ by the probe (T_2, T_4), T_4 is aborted as the deadlock victim. The modified resolution step cleans the probe_Q's of T_3 and T_2 of all the probes including the probe (T_1, T_5). Fig. 4(b) shows the situation after the deadlock is resolved. Now, if T_3 starts waiting for T_1, a deadlock will be formed as shown in Fig. 4(c). Since T_3's probe_Q is empty and T_3 is not the highest priority transaction, this deadlock will not be detected unless either T_5 retransmits its probe_Q or $DM(X_1)$ reinitiates a probe on behalf of T_1.[3] Therefore, we observe that after a deadlock resolution, retransmission and reinitiations of probes by those transactions which are waiting transitively on the deadlock cycle is needed.

D. Summary of Deficiencies

We have presented examples illustrating those cases where either the Probe Algorithm [9] fails to detect deadlocks, or detects false deadlock. Let us summarize the reasons for such deficiencies in the algorithm. First, the major cause for detecting false deadlock or failing to detect some deadlocks is that the algorithm overlooks the possibility that transactions may wait transitively on a deadlock cycle. Second, the algorithm fails to include those cases in which, after a transition releases locks, new transactions which now acquire those locks may get involved in a deadlock. This is because probes are either not initiated or transmitted more than once. Third, there are times when the algorithm detects false deadlocks because transactions and data managers save old probes in probe_Q's in order to reduce message overhead. This unfortunately introduces the possibility of retaining old probes, which may later result in false deadlocks. All three of these problems were illustrated by the examples in Section III-B through Section III-B-2. The next section contains the statement of the modified priority based probe algorithm in which these problems do not arise.

IV. MODIFIED PROBE ALGORITHM

In this section we present a modified probe algorithm which incorporates all the modifications we have suggested in the previous section. We keep the basic structure of the original algorithm presented by Sinha and Natarajan [9] and incorporate the changes at the appropriate places. The changes are highlighted using boldface text. The algorithm makes no assumptions about the scheduling policy of a data manager. When two or more transactions simultaneously wait for a data item, the data manager may assign the lock for that data item to any transaction. In the case that the owner of a lock, *holder* releases the lock and it is assigned to some other transaction, we shall refer to the second transaction as *new holder*.

A. The Revised Basic Deadlock Detection Algorithm

The basic deadlock detection algorithm now has the following steps.

1) A data manager initiates, **propagates, or reinitiates** a probe in the following situations.

a) When a data item is locked by a transaction, if a lock request arrives from another transaction, and *re-*

---

[3]If T_1 was directly waiting on the deadlock cycle then reinitiation would be necessary; in this example retransmission of probes by T_5 is sufficient.

quester > holder, the data manager initiates a probe and sends it to the *holder*.

b) When the current *holder* releases a data item, the data manager schedules a "waiting lock request." If there are more lock requests still in the request_Q, then for each lock request for which *requester > new holder*, the data manager initiates a probe and sends it to the *new holder*.

When a data manager initiates a probe it sets

initiator := requester;
junior := holder;

c) When a transaction completes or aborts it releases its locks. The data manager associated with each released data item assigns the lock for the data item to some transaction (heretofore referred to as *new holder*) waiting for that data item (if one exists). Each data manager then requests all remaining transactions waiting on the new lock to transmit their complete probe_Q's to itself. (The identities of these transactions is obtained from the data manager's request_Q.) The data manager forwards each received probe (*initiator, junior*) to *new holder* the lock for which *initiator > new holder*.

2) Each transaction maintains a queue, called a probe_Q, where it stores all probes received by it. The probe_Q of a transaction contains information about the transactions which wait for it directly, or transitively. Since a transaction follows 2PL, the information contained in the probe_Q of a transaction remains valid until it aborts or commits.

After a transaction enters the second phase of the 2PL, it does not discard the probe_Q. However, during the second phase, any probe received is ignored.

Otherwise, a transaction sends a probe **or a copy of its probe_Q to the data manager, where it is waiting in the** following **three** cases.

a) When a transaction *T* receives probe (*initiator, junior*), it performs the following.

if (*junior > T*)
then *junior := T*;
save the probe in the probe_Q;
if *T* is in wait state
then transmit a copy of the saved probe to the data manager where it is waiting;

b) When a transaction issues a lock request to a data manager and waits for the lock to be granted (i.e., it goes from active to wait state), it transmits a copy of each probe stored in its probe_Q to that data manager.

c) If a transaction is waiting and receives a request for its probe_Q from the data manager where it is waiting, it sends a copy of its probe_Q to the data manager. (This may occur as a consequence of part A.1.c.)

3) When a data manager receives probe (*initiator, junior*) from one of its requesters, it performs the following.

if holder > *initiator*
then discard the probe

else if *holder < initiator*
    then propagate the probe to the *holder*
    else declare deadlock and initiate deadlock resolution;

When a deadlock is detected, the detecting data manager has the identities of two members of the cycle, *initiator* and *junior*, i.e., the highest and lowest priority transactions, respectively. The *junior* is chosen as the deadlock victim.

B. The Deadlock Resolution and Post Resolution Computation

This consists of the following three steps.

1) To abort the *victim*, the data manager that detects the deadlock sends an abort signal to the *victim*. The identity of the initiator is also sent along with the abort signal: abort (*victim, initiator*). Since *victim* is aborted, it is necessary to discard those probes (from the probe_Q of various transactions) that have *victim* as their *junior or initiator*. Hence, on receiving an abort_signal, the *victim* does the following.

a) It initiates a message, clean (*victim, initiator*), sends it to the data manager where it is waiting.

b) The *victim* enters abort phase only when its clean message returns to itself. Once it enters the abort phase, the *victim* releases all the locks it held, withdraws its pending request, and aborts. During this phase, it discards any probe or clean message that it receives.

2) When a data manager receives clean (*victim, initiator*) message, it does the following.

a) It propagates the clean message to its *holder*.

b) It reinitiates probes for each *requester* for which *requester > holder*.

c) It requests each transaction in the request_Q to retransmit its probe_Q. This corresponds to part c) of A.1.

3) When a transaction *T* receives clean (*junior, initiator*) message, it acts as follows.

purge every probe from its probe_Q;
if *T* is in wait state
    then if *T = junior*
        then enter the abort phase and release all locks
        else propagate the clean message to the data manager where
T is waiting
else discard the clean message.

V. COMMENTS ON THE PERFORMANCE AND CORRECTNESS OF THE ALGORITHM

The performance of the modified algorithm will be degraded compared to the original algorithm by Sinha and Natarajan [9] because additional overhead is incurred in order for the algorithm to work correctly. We briefly discuss the impact of the modifications on the message overhead and the delay involved for detecting deadlocks.

1) Message Overhead: The message overhead is the

number of messages that must be sent in order to perform deadlock detection. It is no longer true that if a deadlock was detected and resolved and, if later another deadlock forms involving all or a subset of the members of the first cycle, it will be detected using fewer messages as claimed in [9]. The reason is that after a deadlock is resolved, all members of the cycle discard all the probes from their probe_Q's. Therefore, to detect the second deadlock involving the same transactions or a subset of these transactions, the number of messages are not reduced because probes need to be retransmitted. In fact, the number of messages to detect the second deadlock can be greater than the number required to detect the first deadlock. Transactions waiting transitively also have to send probes again, even though they were not a part of the cycle earlier and may not be a part of a cycle in the future.

2) Delay: The delay denotes the time required to detect a deadlock once the deadlock is formed. This also increases over that in the original algorithm because probes need to be retransmitted. Since the probe initiated on behalf of the highest priority transaction detects a deadlock, the delay depends on when the highest priority transaction enters the wait state. However, if the highest priority transaction has to retransmit or reinitiate a probe, the delay can be large even if the highest priority transaction entered the wait state much earlier than the other members of the cycle. Fig. 4(c) is a good example of this situation. For the second deadlock in Fig. 4(c) it does not matter when T_1 entered the wait state. It has to retransmit its probe because the earlier probe was cleaned by all the transactions when the first deadlock was resolved.

In addition to the delay introduced due to reinitiation and retransmission, another form of delay is incurred in aborting a transaction. Part B1.a of the modified algorithm suggests that a deadlock *victim* releases its resources only after its clean message returns to itself. Therefore, transactions waiting on resources held by the *victim* can not acquire the resources even though the deadlock has been detected and resolved. This is not exactly a delay for detecting a deadlock but a delay in the subsequent processing by transactions. This delay will always be proportional to the deadlock cycle length.

We do not have a formal proof of correctness for the modified probe algorithm. Lacking a formal proof, the algorithm has been extensively tested through simulation. In fact, we discovered the deficiencies of the Sinha and Natarajan algorithm for exclusive locks while implementing that algorithm in a distributed database simulator. As we discovered the deficiencies described in Section III, we modified the algorithm until it became the algorithm found in Section IV. After this point in time, the algorithm neither failed to discover a deadlock nor detected a false deadlock. The algorithm has been tested by simulation for a distributed database system on five nodes each containing 1000 data items. The number of users on the system ranged from 2 to 200. Each user repeatedly executed transactions until approximately 20 000 transactions were committed. Each transaction requested on the average 16 exclusive locks. The number of users

on the system was used to control the frequency of deadlock occurrence. When the number of users was large (~ 200), the probability that a transaction conflicts with another transaction on *one data item request* was 0.3. In this case, the probability of a transaction becoming part of a deadlock cycle *on a single data item request* lock was as high as 0.02. In these cases, the probability that a transaction would become part of a deadlock during its lifetime was approximately 0.3. These simulations produced many deadlock cycles with an average length of ten. In addition many situations arose where transactions waited on deadlock cycle transitively. In each of these cases the modified algorithm performed correctly. The reader is referred to [5] for further details on the simulation study.

VI. Conclusions

In this paper we presented a modified probe algorithm based on priorities to detect and resolve deadlocks in distributed database systems. We illustrated through various examples how the algorithm presented by Sinha and Natarajan [9] failed to work correctly in many situations. We observed that the major cause for detecting false deadlock or failing to detect some deadlocks is that the algorithm overlooks the possibility of transactions waiting transitively on a deadlock cycle. Also, the algorithm fails to include those cases in which, after a transaction releases locks, new transactions which now acquire those locks may get involved in a deadlock. We also suggested modifications to the algorithm to eliminate the errors. Although we have not formally proven that the modified algorithm works in all possible cases, extensive simulation evidence leads us to believe that it is correct.

We would like to point out that the modifications we suggested for the basic algorithm are also required for the extended algorithm proposed by Sinha and Natarajan to detect deadlocks when shared and multiple lock requests are allowed. Since the Sinha and Natarajan extended algorithm includes the basic algorithm in its entirety, and those extensions do not resolve the deficiencies, all the modifications we suggested apply. A detailed discussion can be found in [5].

Finally, we would like to point out that it is important to implement and test a distributed algorithm in order to gain a high degree of confidence in whether it performs correctly *if no formal proof of correctness is developed.* This was apparently not done for the original Sinha and Natarajan algorithm.

APPENDIX
The Priority Based Probe Algorithm

The following is a description of the basic deadlock detection algorithm reported by Sinha and Natarajan [9, pp. 69-70].

A. The Basic Deadlock Detection Algorithm

The basic deadlock detection algorithm has three steps.

1) A data manager initiates a probe in the following two situations.

a) When a data item is locked by a transaction, if a lock request arrives from another transaction, and *requester > holder*, the data manager initiates a probe and sends it to the *holder*.

b) When a *holder* releases the data item, the data manager schedules a waiting lock request. If there are more lock requests still in the request_Q, then for each lock request for which *requester > new holder*, the data manager initiates a probe and sends it to the new *holder*.

When a data manager initiates a probe it sets

initiator := requester;
junior := holder;

2) Each transaction maintains a queue, called a probe_Q, where it stores all the probes received by it. The probe_Q of a transaction contains information about the transactions which wait for it directly, or transitively. Since a transaction follows 2PL, the information contained in the probe_Q of a transaction remains valid until the transaction aborts or commits.

After a transaction enters the second phase of the 2PL, it can never get involved in a deadlock. Hence, when it enters the second phase, it discards the probe_Q. During the second phase, any probe received is ignored.

A transaction sends a probe to the data manager, where it is waiting in the following two cases.

a) When a transaction *T* receives probe (*initiator, junior*), it performs the following.

if (*junior > T*)
then *junior := T*;
save the probe in the probe_Q;
if *T* is in wait state
then transmit a copy of the saved probe to the data manager where it is waiting;

b) When a transaction issues a lock request to a data manager and waits for the lock to be granted (i.e., it goes from active to wait state), it transmits a copy of each probe stored in its probe_Q to that data manager.

3) When a data manager receives probe (*initiator, junior*) from one of its requesters, it performs the following.

if *holder > initiator*
then discard the probe
else if *holder < initiator*
   then propagate the probe to the *holder*
   else declare deadlock and initiate deadlock resolution;

When a deadlock is detected, the detecting data manager has the identities of the two members of the cycle, *initiator* and *junior*, i.e., the highest and lowest priority transactions, respectively. The *junior* is chosen as the deadlock *victim*.

B. The Deadlock Resolution and Post Resolution Computation

This consists of the following three steps.

1) To abort the *victim*, the data manager that detects the deadlock sends an abort signal to the *victim*. The iden-

tity of the initiator is also sent along with the abort signal: abort (*victim, initiator*). Since *victim* is aborted, it is necessary to discard those probes (from the probe_Q of various transactions) that have *victim* as their *junior or initiator*. Hence, on receiving an abort_signal, the *victim* does the following.

a) It initiates a message, clean (*victim, initiator*), sends it to the data manager where it is waiting, and enters the abort phase. Since the *initiator* is the highest priority transaction of the deadlock cycle, its probe_Q will never contain any probe generated by the other members of the cycle. Consequently, probe_Q's of transactions, from *initiator* to *victim* in the direction of the probe traversal, will not contain a probe having *victim* as *junior* or *initiator*. And hence, the clean message carries the identity of the *initiator* beyond which it need not traverse.

b) In abort phase, the *victim* releases all the locks it held, withdraws its pending request, and aborts. During this phase, it discards any probe or clean message that it receives.

2) When a data manager receives clean (*victim, initiator*) message, it propagates the message to the *holder* of its data.

3) When a transaction *T* receives clean (*victim, initiator*) message, it acts as follows.

purge from the probe_Q every probe that has *victim* as its *junior* or *initiator* of that probe;
if *T* is in wait state
   then if *T = initiator*
      then discard the clean message
      else propagate the clean message to the data manager where it is waiting
   else discard the clean message;

A transaction discards the clean message in the following two situations: 1) the transaction is in active state or, 2) the transaction is the same as the *initiator* of the clean message received.

After "cleaning up" its probe_Q as described above, each member transaction of the deadlock cycle retains the remaining probes in its probe_Q. In the future, if the remaining members (or any subset of them) get involved in a deadlock cycle, it will be detected with fewer number of messages, since probes have already traversed some edges of the cycle.

REFERENCES

[1] P. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, no. 2, pp. 185–221, June 1981.
[2] ——, "A sophisticates's introduction to distributed database concurrency control," in *Proc. 8th Int. Conf. Very Large Data Bases*, Sept. 1982.
[3] K. M. Chandy and J. Misra, "A distributed algorithm for detecting resource deadlocks in distributed systems," in *Proc. ACM SIGACT-*

SIGOPS Symp. Principles of Distributed Computing, Ottawa, Ont., Canada, Aug. 1982.

[4] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," ACM Trans. Comput. Syst., vol. 1, pp. 144–156, May 1983.

[5] A. N. Choudhary, "Two distributed deadlock detection algorithms and their performance," Master's thesis, Dep. ECE, Univ. Massachusetts, Amherst, Feb. 1986.

[6] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notion of consistency and predicate locks in a database system," Commun. ACM, vol. 19, pp. 624–633, Nov. 1976.

[7] V. Gligor and S. H. Shattuck, "On deadlock detection in distributed systems," IEEE Trans. Software Eng., vol. SE-6, no. 5, Sept. 1980.

[8] R. Obermarck, "Distributed deadlock detection algorithm," ACM Trans. Database Syst., vol. 7, pp. 187–208, June 1982.

[9] M. K. Sinha and N. Natarajan, "A priority based distributed deadlock detection algorithm," IEEE Trans. Software Eng., vol. SE-11, no. 1, pp. 67–80, Jan. 1985.

Alok N. Choudhary (S'88) received the B.E. (Hons.) degree in electrical engineering from Birla Institute of Technology and Science, Pilani, India, in 1982. He worked as a Systems Analyst and Designer from 1982 to 1984. He received the M.S. degree in electrical and computer engineering from the University of Massachusetts, Amherst, in 1986.

He is currently with the Coordinated Science Laboratory, University of Illinois, Urbana-Champaign, where he is working toward the Ph.D. degree. He was a Visiting Scientist at IBM, T. J. Watson Research Center, Yorktown Heights, NY, during the summers of 1987 and 1988. His research interests include computer architecture, parallel processing, parallel architectures for integrated vision systems, high speed transaction processing, and distributed computing.

Mr. Choudhary is a student member of the Association for Computing Machinery.

Walter H. Kohler (S'66–M'73–SM'86) received the B.S.E., M.S., M.A., and Ph.D. degrees in electrical engineering from Princeton University, Princeton, NJ.

From 1968 to 1969 he was a member of the Technical Staff of Bell Telephone Laboratories, Holmdel, NJ. Since 1972 he has been with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst. He has also been a consultant to research and development groups within the Digital Equipment Corporation since 1979. His research interests include the design, measurement, and modeling of distributed computer systems.

Dr. Kohler is a member of the IEEE Computer Society and the Association for Computing Machinery.

John A. Stankovic (S'77–M'79–SM'86) received the B.S. degree in electrical engineering and the M.S. and Ph.D. degrees in computer science from Brown University, Providence, RI, in 1970, 1976, and 1979, respectively.

He is currently an Associate Professor in the Department of Computer and Information Science, University of Massachusetts at Amherst. His current research interests include investigating various approaches to distributed scheduling on general-purpose but highly integrated distributed systems, developing means for controlling and scheduling tasks on distributed hard real-time systems, and developing database partitioning protocols.

Prof. Stankovic received an Outstanding Scholar Award from the School of Engineering, University of Massachusetts. He has recently published a tutorial text entitled Reliable Distributed System Software (IEEE Computer Society), and was the General Chairman for the 1986 Real-Time Systems Symposium.

Don Towsley (M'78) received the B.A. degree in physics and the Ph.D. degree in computer sciences from the University of Texas at Austin in 1971 and 1975, respectively.

From 1976 to 1985 he was a member of the faculty of the Department of Electrical and Computer Engineering at the University of Massachusetts, Amherst, where he achieved the rank of Associate Professor. He is currently an Associate Professor of Computer and Information Science at the University of Massachusetts. During 1982–1983, he was a Visiting Scientist at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY. His research interests are in computer networks, distributed computer systems, and performance evaluation.

Dr. Towsley is currently an Associate Editor of Networks and IEEE TRANSACTIONS ON COMMUNICATIONS. He is also a member of the Association for Computing Machinery and the Operations Research Society of America.