Distributed Algorithms for Efficient Deadlock Detection and Resolution Using Resource Allocation Graphs

Yiechan Yang,
Lifeng Hsu
and
Sang H. Son

Computer Science Report No. TR-92-06
February 24, 1992

Distributed Algorithms for Efficient Deadlock Detection and Resolution Using Resource Allocation Graphs

Yiechan Yang, Lifeng Hsu, and Sang H. Son

Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

In this paper we present two efficient distributed deadlock detection and resolution algorithms which combine both path-pushing and edge-chasing techniques to detect and resolve deadlocks in a distributed database system. By locally building up the PRAG (Partial Resource Allocation Graph) information for every transaction on each site through edge-chasing and path-pushing, these algorithms not only have transactions wait-for information but also have resources allocation information. Therefore, as soon as a deadlock cycle is going to form on a site, these algorithms can immediately detect it. The first algorithm starts the deadlock resolution right away whenever a "potential" deadlock cycle is detected. Thus, this algorithm can not completely avoid the detection of false deadlocks. However, it uses an efficient cleanup mechanism to reduce the probability of detecting false deadlocks. The second algorithm does not start the deadlock resolution immediately when a forming of deadlock cycle is detected. Instead, it validates if the potential deadlock cycle is a true deadlock. It starts the deadlock resolution only when a true deadlock is detected. Although this algorithm satisfies the correctness criterion of no false deadlock detection, it delays the true deadlock detection and resolution. We claim that both algorithms are very efficient while comparing them with other distributed deadlock detection algorithms in the literature.

Key Words: distributed database, deadlock detection, transaction, resource allocation graph

This work was supported in part by Office of Naval Research, by NRaD, and by Department of Energy.

1. Introduction

To detect a true deadlock with low communication cost and delays in a distributed database system is very difficult and challenging, because each site of the system simply cannot have complete and up-to-date knowledge of the current state of the system and every inter-site communication involves a finite but unpredictable delay. A simple way to detect a deadlock in a distributed database system is to use a centralized global deadlock detector which periodically takes the union of each site's local TWFG (Transaction Wait-For Graph) to produce a global TWFG and checks it for cycles. However, the single point of failure, the significant communication delay and overhead in assembling all the local TWFGs at the site of global deadlock detector, and the high possibility of detecting a *phantom deadlock* due to the out-of-date global TWFG make global deadlock detection less attractive than a decentralized algorithm [Bern87] [Chou90].

There are many decentralized (distributed) deadlock detection algorithms in the literature. They can be classified into four categories: *path-pushing, edge-chasing, diffusing computations,* and *global state detection* [Knapp87]. In general, the responsibility of detecting a global deadlock in distributed deadlock detection algorithms is shared equally among the sites. The global state of the system is spread over the sites and several sites may participate in the detection of the same deadlock cycle simultaneously. Though distributed algorithms are not vulnerable to a single point of failure and each site will not be swamped with deadlock detection messages, many of them were found to be incorrect by not detecting true deadlocks, or detecting phantom deadlocks, or both. Many more were found very expensive to be implemented, in terms of performance and complexity. And lots of them did not address the problem of deadlock resolution [Chou89] [Knapp87] [Sing89]. It usually ends up that a separate and expensive deadlock resolution phase must be executed for selecting a proper victim to abort when a deadlock is detected, because the site(s) that detect the deadlock may not have enough knowledge of knowing how many transactions are actually involved in the deadlock. This further deteriorates the cost of implementing these algorithms.

In this paper, we present two distributed algorithms which not only detect deadlocks but also resolve the deadlocks they detect immediately without any extra exchanges of messages to select a victim to abort. Both algorithms use edge-chasing and path-pushing techniques to build up resource allocation graph (which includes transaction wait-for information, per se) for every transaction initiated at a site. Instead of starting a *probe computation* while a transaction enters a wait state [Chan83], these algorithms start a sequence of transaction PRAG (Partial Resource Allocation Graph) updates along the edges of virtual TWFGs. Although these algorithms do not build TWFGs for each site of the system, if we consider a transaction PRAG as a partial collection of paths of TWFGs then a transaction PRAG update is just a path-pushing. Hence, our algorithms can be considered as the hybrid approaches that combine both edge-chasing and path-pushing techniques into one unique kind of distributed deadlock detection and resolution algorithms.

The first algorithm in this paper does deadlock resolution immediately whenever the forming of a deadlock cycle is detected. Due to the unpredictability of message delay and the intention to minimize the delay of deadlock resolution, this algorithm does not eliminate the possibility of detecting false (phantom) deadlocks. In order to reduce the probability of detecting false deadlocks, a fast cleanup mechanism is incorporated in the algorithm to cleanup any affected transaction PRAGs after the deadlock victim is aborted. The second algorithm takes one step farther from the first algorithm. It validates the suspicious deadlock cycle whenever the cycle is detected. It starts the deadlock resolution only if the cycle is confirmed to be a true deadlock cycle. It pays the price of one extra round-trip message delay to satisfy the correctness criterion of no false deadlock detection.

By using these algorithms, the integration of deadlock detection and deadlock resolution is easily achieved. The quickness of detecting a deadlock cycle and resolving it gives our algorithms a significant leading edge when compared them with other algorithms in the literature. These algorithms reduce the delays and communication costs of detecting distributed deadlocks to a great extent. They also minimizes the delays and costs of deadlock resolution and any subsequent processing of transactions. Although our

algorithms may consume more memory space on each site due to the construction of transaction PRAGs, and the size of a transaction PRAG update message may be a little bigger than a probe message, we believe that such overhead is negligible while considering the significant performance benefits.

The rest of this paper is organized as follows. In Section 2, we briefly discuss the distributed database model we use in this paper and the underlying assumptions. We also give the definition of a transaction PRAG and its data structure. In Section 3, we present our first distributed deadlock detection and resolution algorithm followed by some examples to demonstrate how the algorithm works. In Section 4, we modify the first algorithm to become the second algorithm which does not detect false deadlocks. Section 5 covers the anatomy of the algorithms. The correctness of the algorithms will be analyzed and discussed in this section. Section 6 gives comparisons between our algorithms and other distributed deadlock detection algorithms in the literature. Finally, Section 7 summarizes the paper.

2. The Distributed Database Model: Definitions and Assumptions

Two types of deadlock have been modeled in the literature. In a resource model, a set of processes is deadlocked if each process in the set requests a resource held by another process in the set. In a communication model, a set of processes is deadlocked if each process in the set is waiting for a message from another process in the set but no process in the set ever sends a message [Chan83] [Sing89]. In this paper, we use the resource model as our distributed database model.

A resource is a passive data item (object) and may be represented by a simple variable or a complex data structure, i,e., it represents an independently accessible piece of information. In this paper, we assume that each resource (data item) is *nonreplicated*, has its own unique identification, and is stored at a site. And a distributed database system is a network, consisting of sites which communicate through messages.

In such a distributed database system, the user accesses data items of the database by executing transactions. A transaction is viewed as a process that performs a sequence of reads and writes on data items. Thus, data items are requested and released by transactions through *inter-site* or *intra-site* message exchanges. The network is assumed to deliver messages error-free. It means that all messages sent from the source sites will arrive at their destination sites in finite time and in the order they are sent. In essence, we assume that there aren't any communication failures and messages are pipelined at their destination sites, neither lost nor duplicated. We also assume that no site failures would occur, which may cause spontaneous transaction aborts.

Each site has a TM (Transaction Manager) to manage those transactions initiated from it and a DM (Data Manager) to manage exclusively the data items stored there. We assume that each TM has the mapping information of the entire system's database, i.e., a TM can always determine where the data item is located based on its identification. We also assume that each DM provides the same locking mechanism which is 2PL (two phase locking protocol) and manages the lock table in the same fashion. The 2PL protocol implies that a transaction must lock a data item before accessing it and cannot release a lock until all requested locks are granted. Locks can be exclusive locks (write locks) or share locks (read locks).

A transaction which has locked the data item is called the *holder* of the data item. When the lock request of a transaction for a data item is rejected by a DM, the DM will put the request transaction in the waiting queue associated with the data item. When a holder unlocks a data item, the DM of that data item chooses a request transaction from the waiting queue and grants the lock to that *requester*. In our algorithms, the scheduling scheme used by a DM is FCFS (First Come First Serve).

Our algorithms are based on the simplest possible distributed database model in which a transaction can have at most one outstanding resource request at a time, it is called one-resource model in [Knapp87]. Thus, the PRAG (Partial Resource Allocation Graph) of a transaction is a directed graph that can be represented as a tree-like structure. The root is the transaction itself. The second layer consists of nodes which denote the resources (data items) having been locked by the root transaction except one node

denotes the outstanding data item that is currently being requested by the root if the situation arises. The third layer, if there is any, consists of nodes which denote the transactions that are waiting for the data items being locked by the root. If any transactions of the third layer themselves also hold some data items while they are waiting for the root transaction to release locks, we will have the fourth layer consisting of nodes of data items again, and so forth. Hence, a transaction PRAG tree consists of alternate layers. The odd layers are transaction nodes and the even layers are resource nodes (see Figure 1). A transaction PRAG tree can be divided into branches based on the data items it holds and requests. There could be many *holding branches*, but at most one *request branch*. Apparently, each PRAG tree has not only transaction wait-for information but also resources allocation information (see Figure 1). The TM of each site manages those transaction PRAG trees which form a *forest*, since every transaction has its own PRAG tree.

A transaction can be in one of two states: *active* or *waiting*. The state of a transaction changes from active to waiting when its lock request for a data item propagates through its TM to the DM of the data item and is queued by the DM, i.e., the lock request is denied. The state of a transaction changes from waiting to active when the DM schedules the transaction from the waiting queue, i.e., the lock is granted. An active transaction can commit or abort. However, a waiting transaction cannot abort itself. If we

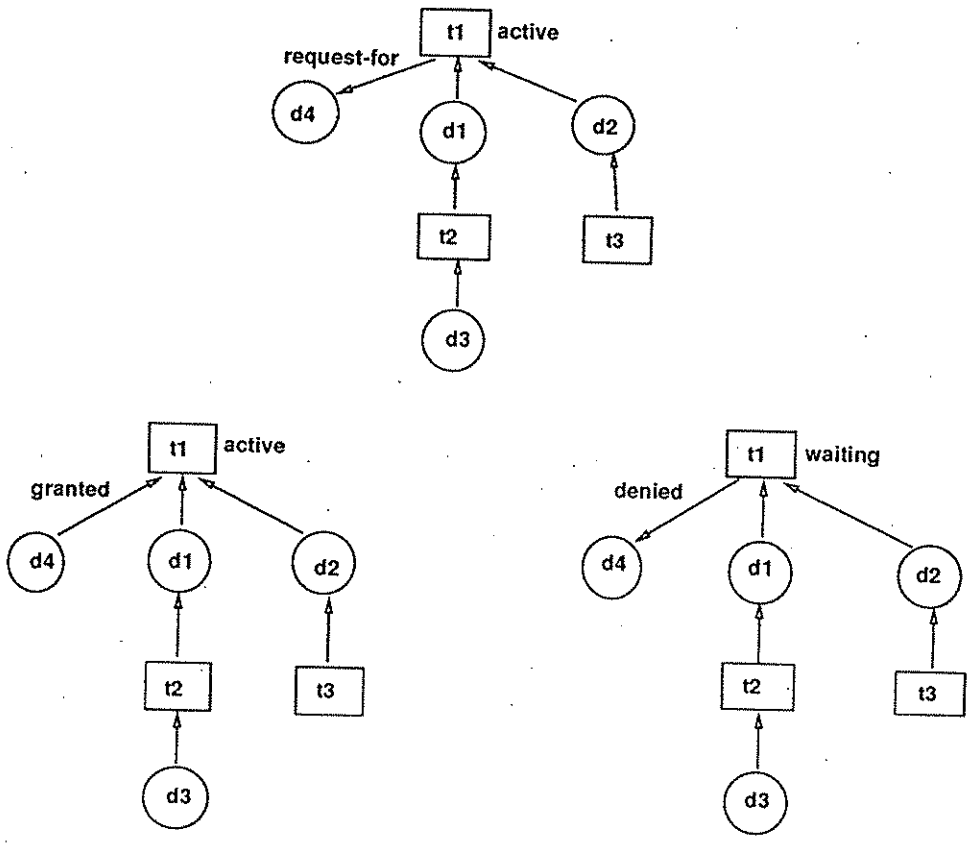

Figure 1: Three types of PRAG trees.

allow a waiting transaction abort itself anytime then there is no way to eliminate the phantom deadlock phenomenon.

Since we use 2PL protocol for concurrency control, the active state of a transaction can be further divided into two substates: *growing* and *shrinking*. The state change of a transaction between active and waiting is actually either from growing to waiting or from waiting to growing. Once a transaction enters a shrinking state it cannot change its state. An active transaction changes its state from growing to shrinking when it is committing, aborting, or being aborted (when a deadlock is detected and the transaction is the selected victim). A waiting transaction can be aborted as well if it is chosen as the victim of deadlock resolution.

We use *events* to model our distributed database system behaviors. Each event occurs with an action of sending or receiving messages. We assume that every site has a logical clock that is monotonically increasing and the various clocks are loosely synchronized such that all of the events of the system can be totally ordered [Lam78]. Each transaction is assigned a unique timestamp when it is initiated from a site. We shall consider the timestamp of a transaction as a part of its identification. We will use the timestamp of a transaction as the priority of the transaction. That means a transaction has higher priority than another transaction if it has smaller timestamp (i.e., it is older). We will use priority to determine the victim of deadlock resolution if a deadlock cycle is detected by a waiting transaction PRAG update. This guarantees that if more than one site detect a deadlock cycle, they will pick the same transaction in the cycle as the victim to abort. When an aborted transaction is restarted, a new timestamp is generated for it. In our algorithms, we do not attempt to reuse the old timestamp for a restart transaction.

Based on the occurrences of events, in this paper, we classify the messages used in the algorithms into the following categories: *lock request messages, lock grant messages, lock deny messages, lock release messages, transaction commit messages, transaction abort messages, transaction not-wait-for messages, transaction PRAG cleanup messages, transaction PRAG update messages, transaction validate messages, transaction not-exist messages*, and *transaction exist messages*. The last three types of messages are only used in the second algorithm for deadlock validations.

3. The First Algorithm and Examples

3.1. The Notations

The following notations will be used throughout this section:

TM(i): The Transaction Manager at site i.

DM(i): The Data Manager at site i.

t(i,j): The jth transaction initiated at site i.

d(i,j): The data item j stored at site i.

t(i,j)<-d(k,l):
Transaction t(i,j) holds a lock on data item d(k,l).

t(i,j)->d(k,l):
Transaction t(i,j) requests a lock on data item d(k,l), same as d(k,l)<-t(i,j).

d(m,n)<-t(i,j)<-d(k,l):
Transaction t(i,j) holds a lock on data item d(k,l) and requests a lock on data item d(m,n); if lock request for data item d(m,n) is granted then the arrow direction will be reversed from "request" to "hold" and t(i,j) remains/becomes active; otherwise, t(i,j) enters/remains waiting.

t(i,j)<-d(k,l)<-t(m,n)<-d(p,q):
Transaction t(i,j) is active, holds a lock on data d(k,l) which is requested by transaction t(m,n) that is holding a lock on data d(p,q); therefore t(m,n) is waiting for t(i,j).

{ X }: A message, sent or received by an event, where X can be any of the following information:

lock request information
Either t(i,j)->d(k,l) received by a TM or REQUEST(PRAG(t(i,j))) received by a DM, where PRAG(t(i,j)) denotes the PRAG tree of transaction t(i,j) with an outstanding request branch.

lock grant information
GRANT(PRAG(t(i,j), d(k,l))), where PRAG(t(i,j), d(k,l)) denotes the new granted d(k,l) holding branch for transaction t(i,j) PRAG tree.

lock deny information
DENY(t(i,j)->d(k,l)), where t(i,j)->d(k,l) denotes transaction t(i,j) is requesting for lock on data item d(k,l).

lock release information
RELEASE(PRAG(t(i,j), d(k,l))), where PRAG(t(i,j), d(k,l)) denotes the existing d(k,l) holding branch of transaction t(i,j) PRAG tree.

transaction commit information
COMMIT(t(i,j)).

transaction abort information
ABORT(t(i,j)).

transaction not-wait-for information
NOTWAIT(t(i,j)->d(k,l)), denotes that transaction t(i,j) is no longer waiting for data item d(k,l) and should be removed from the waiting queue of d(k,l).

transaction PRAG cleanup information
Either CLEANUP(t(i,j), t(k,l)) received by a TM or CLEANUP(d(i,j), t(k,l)) received by a DM. The former one denotes that the waiting-for transaction t(k,l) and its subsequent subtree should be removed from t(i,j) PRAG tree, because t(k,l) is the selected deadlock victim. The later one denotes that the cleanup actions must occur to the holder(s) of d(i,j) such that the deadlock victim t(k,l) and its subsequent subtree will be removed from the PRAG tree(s) of the holder(s), if they are found.

transaction PRAG update information
Either UPDATE(PRAG(t(i,j))) received by a DM, where PRAG(t(i,j)) denotes the new updated PRAG tree of t(i,j) and ready to be propagated to the next transaction for which it is waiting, or UPDATE(PRAG(t(i,j), d(k,l))) received by a TM, where PRAG(t(i,j), d(k,l)) denotes a partial d(k,l) holding branch which may have new resource allocation information for adding to transaction t(i,j) PRAG tree.

3.2. The Algorithm

Now, we present our first algorithm as follows. Note that in the following presentation, any actions that TMs and DMs must perform but are not related to deadlock detection are omitted.

A) Initialization: for each site i,

Creates process TM(i) and initializes an empty transaction PRAG tree table;

Creates process DM(i) and initializes an empty lock table;

Sets every data item waiting queue in DM(i) to empty.

B) For each transaction t(i,j), if

a. t(i,j) is active and requests a lock on data item d(k,l):

Sends a lock request message { t(i,j)->d(k,l) } to TM(i).

b. t(i,j) commits:

Sends a commit message { COMMIT(t(i,j)) } to TM(i).

c. t(i,j) is active and wants to abort:

Sends an abort message { ABORT(t(i,j)) } to TM(i).

d. t(i,j) is waiting:

Does nothing.

C) For each transaction manager TM(i), if

a. TM(i) receives a lock request message { t(i,j)->d(k,l) } from transaction t(i,j) :

Searches the PRAG tree of t(i,j) in the current PRAG tree table to see if d(k,l) is locked by other transaction that is waiting for t(i,j) directly or transitively;

If search succeeds then /* it means that a deadlock is detected while t(i,j) is active */
Sends lock release messages { RELEASE(PRAG(t(i,j), d(x,y))) } to corresponding DM(x), where d(x,y) represents any of the data items currently being locked by t(i,j);

Removes t(i,j) PRAG tree from PRAG tree table;

Kills transaction t(i,j) and triggers the restart counter/timer;
Else
Adds the request branch: t(i,j)->d(k,l) to t(i,j) PRAG tree and sends the lock request message { REQUEST(PRAG(t(i,j))) } to DM(k);
Endif

b. TM(i) receives a commit message { COMMIT(t(i,j)) } or abort message { ABORT(t(i,j)) } :

If t(i,j) is in the transaction PRAG tree table then /* ignores abort message if t(i,j) does not exist */
Sends lock release messages { RELEASE(PRAG(t(i,j), d(x,y))) } to corresponding DM(x), where d(x,y) represents any of the data items currently being locked by t(i,j);

If t(i,j) is waiting then /* means that t(i,j) is selected as the deadlock victim */
Sends not-wait-for message { NOTWAIT(t(i,j)->d(m,n)) } to DM(m) to remove t(i,j) from the waiting queue of d(m,n), where d(m,n) is the data item requested by t(i,j);

Kills transaction t(i,j) and triggers the restart counter/timer;
Endif

Removes t(i,j) PRAG tree from PRAG tree table;
Endif.

c. TM(i) receives a lock grant message { GRANT(PRAG(t(i,j), d(k,l))) } from DM(k) :

Replaces the request branch of t(i,j) PRAG tree with the new holding branch embedded in the received lock grant message;

If t(i,j) is in waiting state then changes its state back to active.

d. TM(i) receives a PRAG update message { UPDATE(PRAG(t(i,j), d(k,l))) } from DM(k) :

If t(i,j) is active then
  If the PRAG update message is not out of date then /* To know if the message is out of date, we need to check if any data items locked by t(i,j) also appear in the PRAG update message. */
    Updates the d(k,l) holding branch of t(i,j) PRAG tree based on the mismatches found between the received PRAG update message and the d(k,l) holding branch;
  Endif
Else /* t(i,j) is waiting */
  Searches the PRAG update message to see if the data item for which t(i,j) is waiting, say d(m,n), is locked by other transaction;

  If search succeeds then /* deadlock is detected while t(i,j) is idle */
    Determines the deadlock victim by comparing the timestamp of each transaction involved in the cycle and selecting the youngest one, say t(p,q), as the victim;

    /* Note that the selected deadlock victim t(p,q) may be t(i,j) itself. In our algorithm, we assume that TM(i) can send any messages to itself. In the actual implementation, if t(i,j) is not t(p,q) then TM(i) needs to cleanup PRAG tree of t(i,j) and sends an abort message to TM(p). If t(i,j) is t(p,q) then TM(i) must abort t(i,j) the same as the case when it gets an abort message from other site. */

    For other transaction t(x,y), not t(p,q), in the deadlock cycle
      Sends a PRAG cleanup message { CLEANUP(t(x,y),t(p,q)) } to TM(x);
    Endfor

    Sends a transaction abort message { ABORT(t(p,q)) } to TM(p);
  Else
    Updates the d(k,l) holding branch of t(i,j) PRAG tree based on the mismatches found between the received PRAG update message and the d(k,l) holding branch;

    If the PRAG tree of t(i,j) is updated then /* propagate the new info. */
      Sends a PRAG update message { UPDATE(PRAG(t(i,j))) } to DM(m) in where the requested data item d(m,n) is located;
    Endif
  Endif
Endif.

e. TM(i) receives a lock deny message { DENY(t(i,j)->d(k,l)) } from DM(k) :

Sets state of t(i,j) to waiting.

f. TM(i) receives a PRAG cleanup message { CLEANUP(t(i,j), t(k,l)) } :

If t(k,l) is in the PRAG tree of t(i,j) then
Cuts off entire t(k,l) branch;

If t(i, j) is waiting then /* cleanup stops if t(i,j) is active */
Sends a cleanup message { CLEANUP(d(m,n), t(k,l)) } to DM(m), where d(m,n) is the data item being requested by transaction t(i,j);
Endif
Endif.

D) For each data manager DM(i), if

a. DM(i) receives a lock request message { REQUEST(PRAG(t(k,l))) } from TM(k), requesting data item d(i,j) :

/* Note that the requested lock can be either exclusive lock or share lock. */

If lock is available then
Sends a lock grant message { GRANT(PRAG(t(k,l), d(i,j))) } to TM(k) and adds t(k,l) to d(i,j) lock holder list;
Else
Sends a lock deny message { DENY(t(k,l)->d(i,j)) } to TM(k) and puts t(k,l) to the end of the waiting queue of d(i,j);

For each lock holder t(x,y) of d(i,j) /* may be more than one holder if lock is share lock */
Constructs a PRAG update message { UPDATE(PRAG(t(x,y), d(i,j))) } based on the PRAG tree of t(k,l) found in the received lock request message;

Sends this new PRAG update message to TM(x);
Endfor
Endif.

b. DM(i) receives a PRAG update message { UPDATE(PRAG(t(k,l))) } from TM(k) :

Gets the data item d(i,j) requested by t(k,l) from the received message;

For each lock holder t(x,y) of d(i,j) /* may be more than one holder if lock is share lock */
Constructs a new PRAG update message { UPDATE(PRAG(t(x,y), d(i,j))) } based on the PRAG tree of t(k,l) found in the received PRAG update message;

Sends this new PRAG update message to TM(x);
Endfor.

c. DM(i) receives a lock release message { RELEASE(PRAG(t(k,l), d(i,j))) } from TM(k) :

Updates the lock table such that t(k,l) is no longer locking d(i,j);

If d(i,j) is free and its waiting queue is not empty then
Puts the first transaction in the waiting queue into the lock holder list of d(i,j);

If the requested lock of the first transaction in the waiting queue is share lock then
Scans the waiting queue of d(i,j) and removes transactions from it to the holder list until a

transaction which lock request type is exclusive is found;
Endif

For any transaction t(x,y) in the holder list of d(i,j) /* may be just one */
Builds a holding branch PRAG(t(x,y), d(i,j)) by replacing t(k,l) in the received message;

Sends the lock grant message { GRANT(PRAG(t(x,y), d(i,j))) } to TM(x); /* this is necessary because t(x,y) must also inherit up-to-date PRAG information. */
Endfor
Endif.

d. DM(i) receives a not-wait-for message { NOTWAIT(t(k,l)->d(i,j)) } from TM(k) :

Deletes t(k,l) from the waiting queue of d(i,j).

e. DM(i) receives a cleanup message { CLEANUP(d(i,j), t(k,l)) } from TM(m) :

For each lock holder t(x,y) of d(i,j)
Sends a cleanup message { CLEANUP(t(x,y), t(k,l)) } to TM(x);
Endfor

3.3. The Examples

Now, let us show how this algorithm works by simple examples. Note that the event occurrence time in the following examples are idealized for the purpose of demonstration. Also, all of the locks granted or being requested are exclusive locks. In reality, the algorithm itself shall be able to handle any possible event occurrence sequences and either type of locks: exclusive or shared.

Example 1: Assume there are three sites. Each site has one transaction,
one data item, at certain time interval. Thus, we have:
site 1: TM(1), DM(1), t(1,1), d(1,1)
site 2: TM(2), DM(2), t(2,1), d(2,1)
site 3: TM(3), DM(3), t(3,1), d(3,1)
and the priority order is t(1,1) > t(2,1) > t(3,1).

At timestamp V, assume that the system state is as follows:

site	TM: PRAG tree table
1	t(1,1)<-d(3,1); active
2	t(2,1)<-d(1,1); active
3	t(3,1)<-d(2,1); active

site	DM: lock table
1	d(1,1): t(2,1); waiting queue empty
2	d(2,1): t(3,1); waiting queue empty
3	d(3,1): t(1,1); waiting queue empty

At timestamp V+1, assume the event: t(2,1)->d(3,1) occurs (see Figure 2).

Actions triggered:

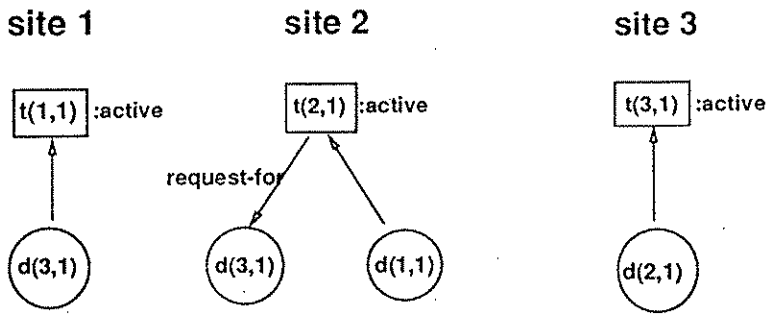

Figure 2: Event t(2,1)->d(3,1) occurs.

(1) t(2,1) sends lock request message { t(2,1)->d(3,1) } to TM(2);

(2) TM(2) checks if d(3,1) is in t(2,1) PRAG tree; then it sends lock request message { d(3,1)<-t(2,1)<-d(1,1) } to DM(3) since it finds no one is holding d(3,1) in its PRAG tree;

(3) DM(3) puts t(2,1) into waiting queue of d(3,1) and sends lock deny message { DENY(t(2,1)->d(3,1)) } to TM(2); it also sends PRAG update message { t(1,1)<-d(3,1)<-t(2,1)<-d(1,1) } to TM(1) since t(1,1) is the holder of d(3,1);

(4) TM(2) changes t(2,1) state from active to waiting;

(5) TM(1) updates t(1,1) PRAG tree to t(1,1)<-d(3,1)<-t(2,1)<-d(1,1).

So, at timestamp W, the system state becomes:

site	TM: PRAG tree table
1	t(1,1)<-d(3,1)<-t(2,1)<-d(1,1); active
2	d(3,1)<-t(2,1)<-d(1,1); waiting
3	t(3,1)<-d(2,1); active

site	DM: lock table
1	d(1,1): t(2,1); waiting queue empty
2	d(2,1): t(3,1); waiting queue empty
3	d(3,1): t(1,1); waiting queue [t(2,1)]

At timestamp W+1, assume event: t(3,1)->d(1,1) occurs (see Figure 3).

Actions triggered:

(6) t(3,1) sends lock request message { t(3,1)->d(1,1) } to TM(3);

(7) TM(3) checks if d(1,1) is in t(3,1) PRAG tree; then it sends lock request message { d(1,1)<-t(3,1)<-d(2,1) } to DM(1) since it finds no one is holding d(1,1) in its PRAG tree;

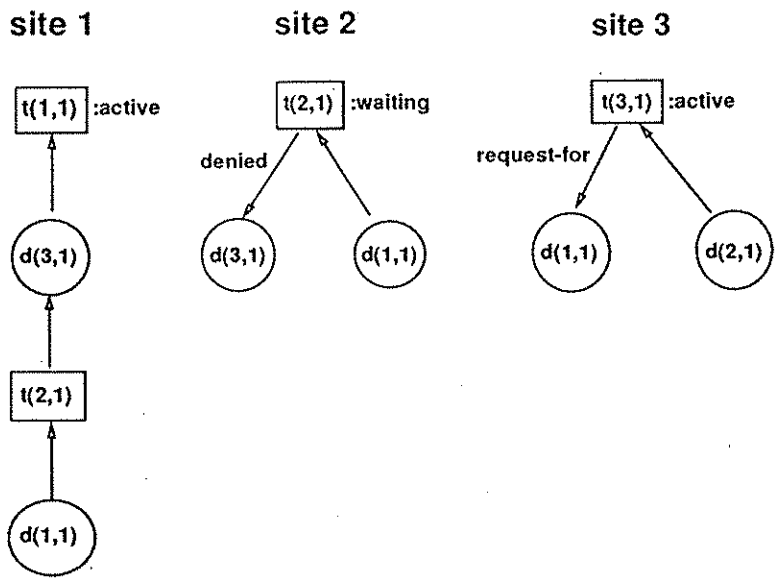

Figure 3: Event t(3,1)->d(1,1) occurs.

(8) DM(1) puts t(3,1) into waiting queue of d(1,1) and sends lock deny message { DENY(t(3,1)->d(1,1)) } to TM(3); it also sends PRAG update message { t(2,1)<-d(1,1)<-t(3,1)<-d(2,1) } to TM(2) since t(2,1) is the holder of d(1,1);

(9) TM(3) changes t(3,1) state from active to waiting;

(10) Since t(2,1) is waiting, upon receiving the PRAG update message from DM(1), TM(2) checks if d(3,1) requested by t(2,1) can be found in the message; it fails, so it updates t(2,1) PRAG tree to d(3,1)<-t(2,1)<-d(1,1)<-t(3,1)<-d(2,1) and propagates this PRAG tree to DM(3);

(11) DM(3) adds the holder of d(3,1), which is t(1,1), to the received PRAG tree and sends the new PRAG as a PRAG update message to TM(1);

(12) Since t(1,1) is still active, TM(1) simply updates t(1,1) PRAG tree to t(1,1)<-d(3,1)<-t(2,1)<-d(1,1)<-t(3,1)<-d(2,1) and stops the processing of PRAG tree updates.

So, at timestamp X, the system state becomes:

site	TM: PRAG tree table
1	t(1,1)<-d(3,1)<-t(2,1)<-d(1,1)<-t(3,1)<-d(2,1); active
2	d(3,1)<-t(2,1)<-d(1,1)<-t(3,1)<-d(2,1); waiting
3	d(1,1)<-t(3,1)<-d(2,1); waiting

site	DM: lock table
1	d(1,1): t(2,1); waiting queue [t(3,1)]

2 d(2,1): t(3,1); waiting queue empty
3 d(3,1): t(1,1); waiting queue [t(2,1)]

At timestamp X+1, assume event: t(1,1)->d(2,1) occurs (see Figure 4).

Actions triggered:

(13) t(1,1) sends lock request message { t(1,1)->d(2,1) } to TM(1);

(14) TM(1) checks if d(2,1) is in t(1,1) PRAG tree; it finds that d(2,1) is currently locked by t(3,1) which in turn is waiting for t(1,1), shown in t(1,1) PRAG tree; thus, it aborts t(1,1) by sending lock release message to DM(3) and deleting t(1,1) PRAG tree from PRAG tree table;

(15) DM(3), upon receiving the lock release message, removes t(1,1) from the lock holder list of d(3,1) and grants d(3,1) to t(2,1) by sending a lock grant message to TM(2);

(16) TM(2) updates t(2,1) PRAG tree by converting the request branch to a holding branch and changes the state of t(2,1) from waiting to active.

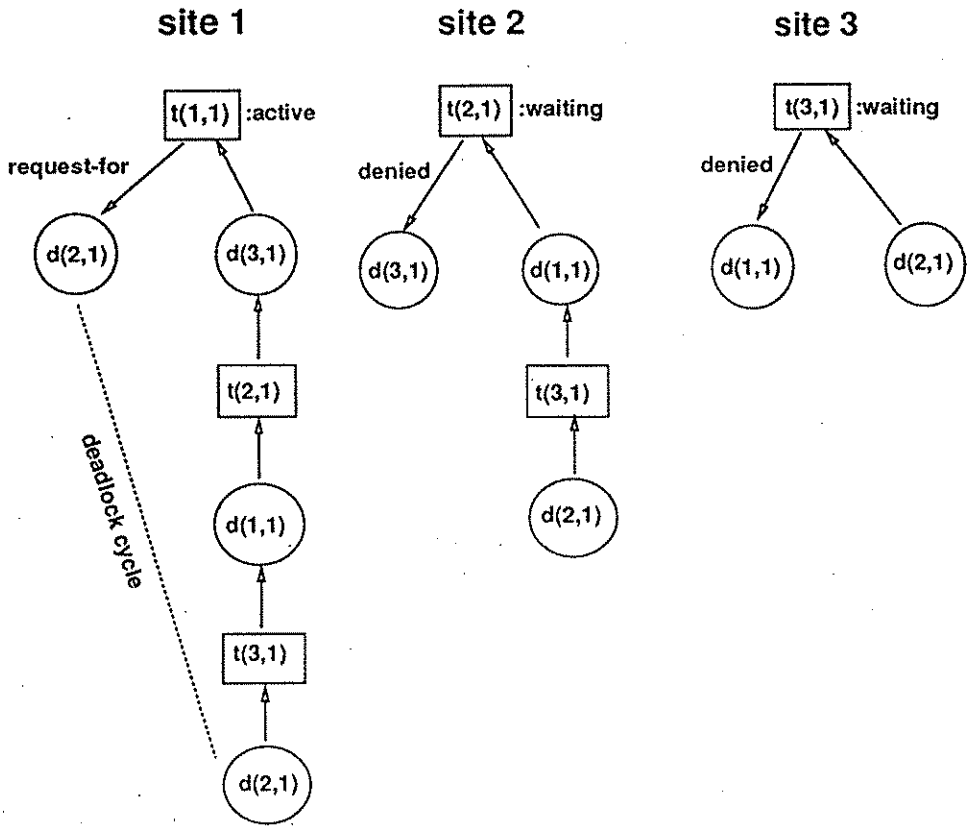

Figure 4: Event t(1,1)->d(2,1) occurs, forming a deadlock cycle.

So, at timestamp Y, the system state becomes:

site	TM: PRAG tree table
1	empty /* t(1,1) is aborted */
2	t(2,1)<-d(1,1)<-t(3,1)<-d(2,1); active <-d(3,1)
3	d(1,1)<-t(3,1)<-d(2,1); waiting

site	DM: lock table
1	d(1,1): t(2,1); waiting queue [t(3,1)]
2	d(2,1): t(3,1); waiting queue empty
3	d(3,1): t(2,1); waiting queue empty

Notice that up to this point, the algorithm has detected a deadlock and resolved it immediately so that the system won't be deadlocked (see Figure 5).

At timestamp Y+1, assume event t(2,1)->d(2,1) occurs.

Actions triggered:

(17) t(2,1) sends lock request message { t(2,1)->d(2,1) } to TM(2);

(18) TM(2) checks if d(2,1) is in t(2,1) PRAG tree; since it finds that d(2,1) is currently locked by t(3,1) which in turn is waiting for t(2,1), shown in t(2,1) PRAG tree, it aborts t(2,1) by sending a lock release message to DM(1) and another lock release message to DM(3); also, it deletes t(2,1)

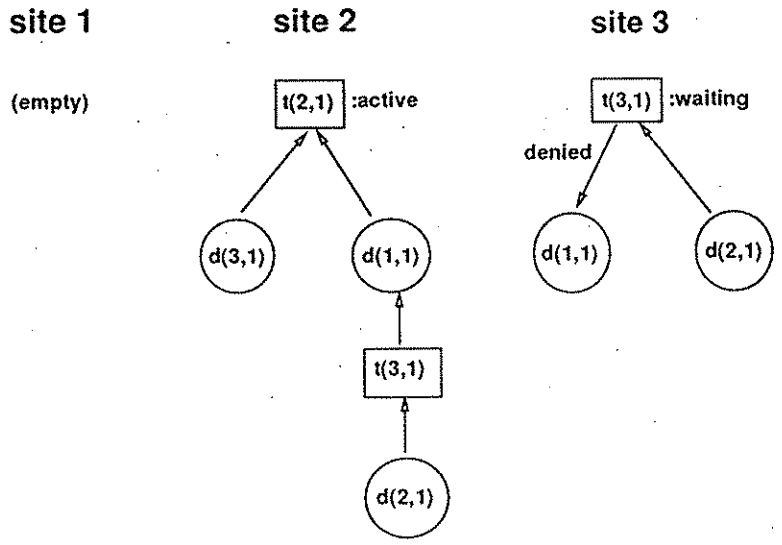

Figure 5: Deadlock is resolved by aborting t(1,1).

PRAG tree from PRAG tree table;

(19) DM(1) removes t(2,1) from the lock holder list of d(1,1) and grants d(1,1) to t(3,1) by sending lock grant message to TM(3);

(20) DM(3) removes t(2,1) from the lock holder list of d(3,1) and d(3,1) becomes free;

(21) TM(3) updates t(3,1) PRAG tree by converting the request branch to a holding branch and changes the state of t(3,1) from waiting to active.

So, at timestamp Z, the system state becomes:

site	TM: PRAG tree table
1	empty /* t(1,1) is aborted */
2	empty /* t(2,1) is aborted */
3	t(3,1)<-d(2,1); active <-d(1,1)

site	DM: lock table
1	d(1,1): t(3,1); waiting queue empty
2	d(2,1): t(3,1); waiting queue empty
3	d(3,1): free; waiting queue empty

In the above example, the events occurred in a perfect timing fashion such that only the active transactions detect deadlocks. In reality, a distributed deadlock detection algorithm shall be able to handle all of the possible event orderings. Now let us use a different event ordering to demonstrate how our algorithm works when the waiting transactions detect deadlocks and multiple sites detect the same deadlock.

Example 2: same configuration as Example 1.

At the same timestamp W as in Example 1, the system state is:

site	TM: PRAG tree table
1	t(1,1)<-d(3,1)<-t(2,1)<-d(1,1); active
2	d(3,1)<-t(2,1)<-d(1,1); waiting
3	t(3,1)<-d(2,1); active

site	DM: lock table
1	d(1,1): t(2,1); waiting queue empty
2	d(2,1): t(3,1); waiting queue empty
3	d(3,1): t(1,1); waiting queue [t(2,1)]

At timestamp W+1, assume event: t(3,1)->d(1,1) occurs and at timestamp W+2, event: t(1,1)->d(2,1) also occurs (see Figure 6).

Actions triggered:

(1) t(3,1) sends lock request message { t(3,1)->d(1,1) } to TM(3); t(1,1) sends lock request message { t(1,1)->d(2,1) } to TM(1);

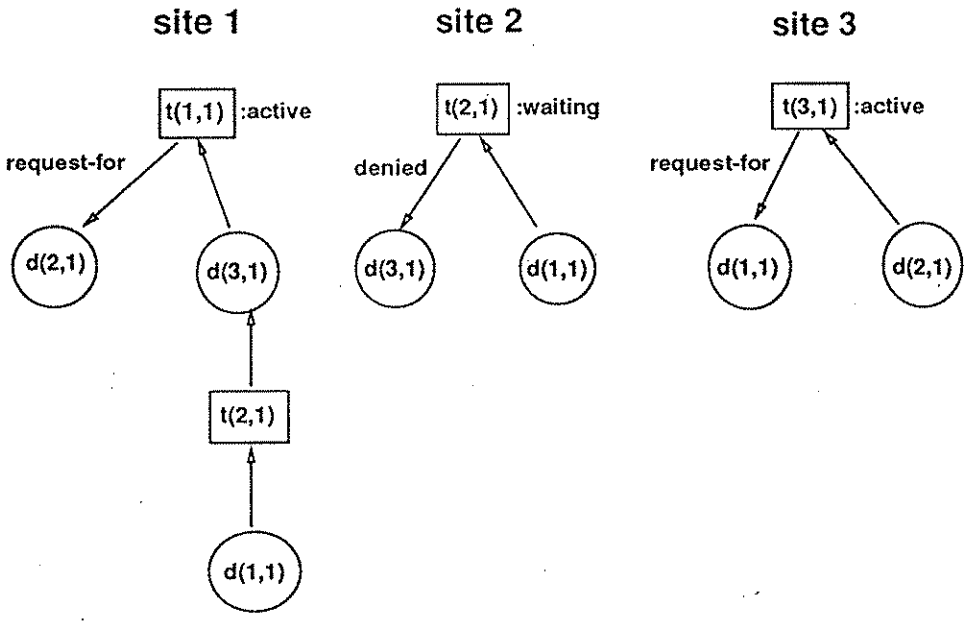

Figure 6: Events t(3,1)->d(1,1) and t(1,1)->d(2,1) almost occur simultaneously.

(2) TM(3) checks if d(1,1) is in t(3,1) PRAG tree; nothing is found, sends lock request message { d(1,1)<-t(3,1)<-d(2,1) } to DM(1); TM(1) checks if d(2,1) is in t(1,1) PRAG tree; nothing is found, sends lock request message { d(2,1)<-t(1,1)<-d(3,1)<-t(2,1)<-d(1,1) } to DM(2);

(3) DM(1) puts t(3,1) into waiting queue of d(1,1), sends lock deny message { DENY(t(3,1)->d(1,1)) } to TM(3), and sends PRAG update message { t(2,1)<-d(1,1)<-t(3,1)<-d(2,1) } to TM(2); DM(2) puts t(1,1) into waiting queue of d(2,1), sends lock deny message { DENY(t(1,1)->d(2,1)) } to TM(1), and sends PRAG update message { t(3,1)<-d(2,1)<-t(1,1)<-d(3,1)<-t(2,1)<-d(1,1) } to TM(3);

(4) TM(3) changes t(3,1) state from active to waiting; TM(1) changes t(1,1) state from active to waiting;

(5) Since t(2,1) is waiting, upon receiving the PRAG update message from DM(1), TM(2) checks if d(3,1) requested by t(2,1) can be found in the message; it fails, so it updates t(2,1) PRAG tree to be: d(3,1)<-t(2,1)<-d(1,1)<-t(3,1)<-d(2,1) and sends this new updated PRAG tree to DM(3); Since t(3,1) is waiting too, TM(3) checks if d(1,1) requested by t(3,1) can be found in the received PRAG update message; it does, therefore, it selects the lowest priority transaction: t(3,1) to abort by sending cleanup messages to TM(1) and TM(2), sending lock release message to DM(2) to release d(2,1) from being locked by t(3,1), and sending not-wait-for message to DM(1) to remove t(3,1) from the waiting queue of d(1,1); it also deletes t(3,1) PRAG tree from the PRAG tree table (see Figure 7);

(6) DM(3) adds the holder of d(3,1), which is t(1,1), to the received PRAG tree and sends the new PRAG update message to TM(1);

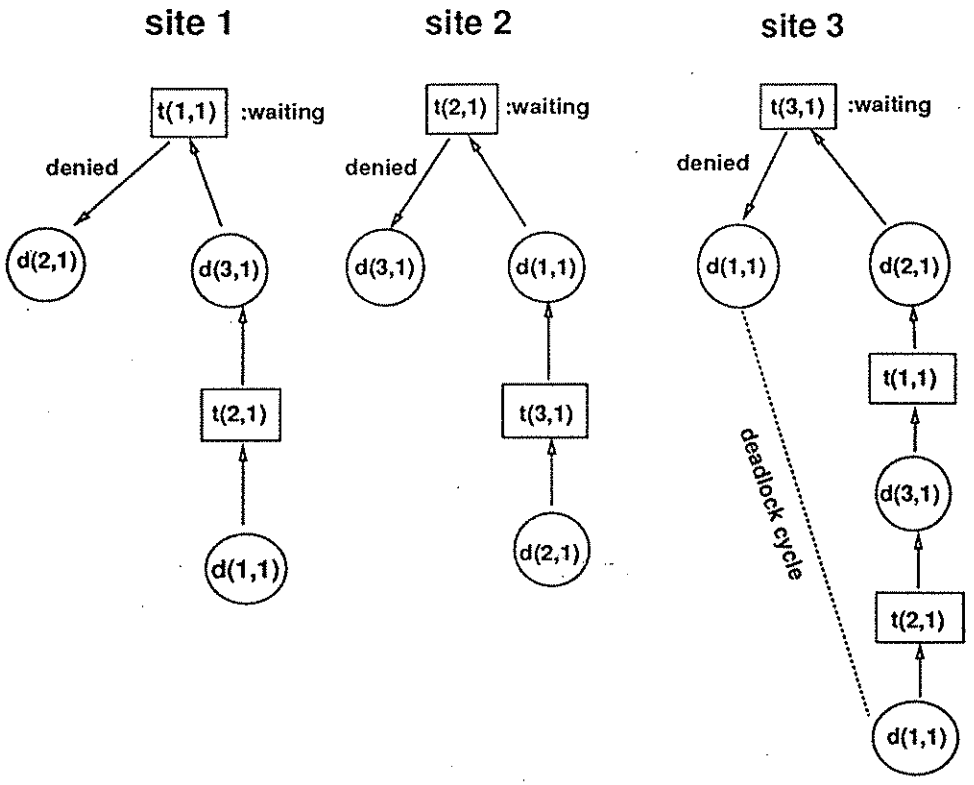

Figure 7: TM(2) propagates PRAG update while TM(3) detects the deadlock.

(7) Assume TM(1) receives PRAG update message for t(1,1) from DM(3) first. Since t(1,1) is waiting, TM(1) checks if d(2,1) requested by t(1,1) can be found in the PRAG update message; the search succeeds, so it also picks t(3,1) as the victim to abort by sending cleanup message { CLEANUP(t(2,1), t(3,1)) } to TM(2), cutting off t(3,1) branch from t(1,1) PRAG tree, and sending a transaction abort message { ABORT(t(3,1)) } to TM(3); Note that TM(1) could receive the cleanup message from TM(3) first. In this case, the cleanup message is discarded since t(3,1) is not in t(1,1) PRAG tree yet. Later, when it receives the PRAG update message from DM(3), the same actions described above would occur.

(8) TM(2) cleans up t(2,1) PRAG tree, based on the cleanup message from TM(3); TM(1) discards the cleanup message from TM(3); DM(2) removes t(3,1) from d(2,1) holder list and sends a lock grant message to TM(1); DM(1) removes t(3,1) from the waiting queue of d(1,1);

(9) TM(2) discards the second cleanup message received from TM(1); TM(3) discards the abort message received from TM(1);

(10) TM(1) wakes up t(1,1), upon receiving the lock grant message from DM(2).

So, at timestamp X', the system state becomes (see Figure 8):

site TM: PRAG tree table

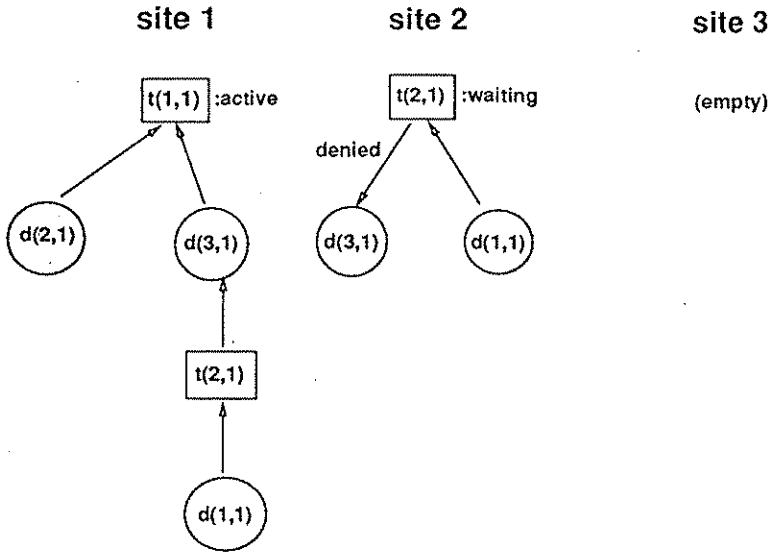

Figure 8: Deadlock is resolved by aborting t(3,1).

1	t(1,1)<-d(3,1)<-t(2,1)<-d(1,1); active <-d(2,1)
2	d(3,1)<-t(2,1)<-d(1,1); waiting
3	empty /* t(3,1) is aborted */

site	DM: lock table
1	d(1,1): t(2,1); waiting queue empty
2	d(2,1): t(1,1); waiting queue empty
3	d(3,1): t(1,1); waiting queue [t(2,1)]

4. The Second Algorithm

4.1. Additional Messages and States

To develop the second algorithm that can completely eliminate the detection of false deadlocks, based on the first algorithm, we need to add two additional transaction states and three types of messages. They are defined as follows:

A transaction changes its state from active to *active-validating* if it is active and requesting a data item which causes its PRAG tree form a potential deadlock cycle. After deadlock validation, an active-validating transaction may become waiting or be aborted as the deadlock victim.

A transaction changes its state from waiting to *waiting-validating* if it is waiting and its TM finds a potential deadlock cycle while trying to update its PRAG tree. A waiting-validating transaction would not propagate (push) any updates to its PRAG tree to other transactions. After deadlock validation, a

waiting-validating transaction may change back to waiting or trigger a deadlock resolution.

transaction validate message
: In the form of { VALIDATE(t(i,j), t(k,l)) }, where t(i,j) denotes the transaction which PRAG tree is found to form a deadlock cycle and t(k,l) is one of the waiting transaction found in the cycle.

transaction not-exist message
: In the form of { NOTEXIST(t(i,j), t(k,l)) }, where t(i,j) denotes the transaction which PRAG tree is found to form a deadlock cycle and t(k,l) is one of the waiting transaction found in the cycle, but has been validated that it does not exist in TM(k) any more.

transaction exist message
: In the form of { EXIST(t(i,j), t(k,l)) }, where t(i,j) denotes the transaction which PRAG tree is found to form a deadlock cycle and t(k,l) is one of the waiting transaction found in the cycle which has been validated its existence in TM(k).

4.2. The Algorithm

Now, we present our second algorithm as follows. Note that in the following presentation, only the modified portions of the first algorithm and the additional portions are shown.

C) For each transaction manager TM(i), if

a. (modified) TM(i) receives a lock request message { t(i,j)->d(k,l) } from transaction t(i,j) :

Searches the PRAG tree of t(i,j) in the current PRAG tree table to see if d(k,l) is locked by other transaction that is waiting for t(i,j) directly or transitively;

If search succeeds then /* it means that a potential deadlock is detected while t(i,j) is active */
Sets t(i,j) state to active-validating;

Collects the deadlock cycle members, other than t(i,j), into a list and sets total-count equal to the list size; also, sets message-count equal to total-count;

For every transaction t(x,y) in the deadlock cycle member list
Sends transaction validate message { VALIDATE(t(i,j), t(x,y)) } to TM(x);
Endfor
Else
Adds the request branch: t(i,j)->d(k,l) to t(i,j) PRAG tree and sends the lock request message { REQUEST(PRAG(t(i,j))) } to DM(k);
Endif

d. (modified) TM(i) receives a PRAG update message { UPDATE(PRAG(t(i,j), d(k,l))) } from DM(k) :

If t(i,j) is active or active-validating then
If the PRAG update message is not out of date then /* To know if the message is out of date, we need to check if any data items locked by t(i,j) also appear in the PRAG update message. */
Updates the d(k,l) holding branch of t(i,j) PRAG tree based on the mismatches found between the received PRAG update message and the d(k,l) holding branch;
Endif
Else /* t(i,j) is waiting or waiting-validating */
Updates the d(k,l) holding branch of t(i,j) PRAG tree based on the mismatches found between

the received PRAG update message and the d(k,l) holding branch;

If t(i,j) is waiting and t(i,j) PRAG tree is changed then
Searches t(i,j) PRAG tree to see if the data item for which t(i,j) is waiting, say d(m,n), is locked by other transaction;

If search succeeds then /* deadlock is detected while t(i,j) is waiting */
Sets t(i,j) state to waiting-validating;

Collects the deadlock cycle members, other than t(i,j), into a list and sets total-count equal to the list size; also, sets message-count equal to total-count;

For every transaction t(x,y) in the deadlock cycle member list
Sends transaction validate message { VALIDATE(t(i,j), t(x,y)) } to TM(x);
Endfor
Else
Sends a PRAG update message { UPDATE(PRAG(t(i,j))) } to DM(m) in where the requested data item d(m,n) is located;
Endif
Endif
Endif.

g. (new) TM(i) receives a transaction validate message { VALIDATE(t(k,l), t(i,j)) } from TM(k) :

If t(i,j) is in the transaction PRAG tree table then
Sends a transaction exist message { EXIST(t(k,l), t(i,j)) } to TM(k);
Else
Sends a transaction not-exist message { NOTEXIST(t(k,l), t(i,j)) } to TM(k);
Endif.

h. (new) TM(i) receives a transaction exist message { EXIST(t(i,j), t(k,l)) } or a transaction not-exist message { NOTEXIST(t(i,j), t(k,l)) } :

If the message is exist message then
Decrements t(i,j) total-count of deadlock members by 1;
Else
Remove t(k,l) branch from t(i,j) PRAG tree;
Endif

Decrements t(i,j) message-count of deadlock validation by 1;

If total-count becomes 0 then /* this is a true deadlock */
If t(i,j) is active-validating then /* abort t(i,j) */
Sends lock release messages { RELEASE(PRAG(t(i,j), d(x,y))) } to corresponding DM(x), where d(x,y) represents any of the data items currently being locked by t(i,j);

Removes t(i,j) PRAG tree from PRAG tree table;

Kills transaction t(i,j) and triggers the restart counter/timer;
Else /* t(i,j) is waiting-validating */
Determines the deadlock victim by comparing the timestamp of each transaction involved in

```
        the cycle and selecting the youngest one, say t(p,q), as the victim;

        For other transaction t(x,y), not t(p,q) or t(i,j), in the deadlock cycle
           Sends a PRAG cleanup message { CLEANUP(t(x,y),t(p,q)) } to TM(x);
        Endfor

        Sends a transaction abort message { ABORT(t(p,q)) } to TM(p);

        If t(i,j) is not t(p,q) then
           Removes t(p,q) branch from t(i,j) PRAG tree;

           Searches t(i,j) PRAG tree again to see if the data item for which t(i,j) is waiting, say
           d(m,n), is locked by other transaction;

           If search succeeds then /* another deadlock is detected */
              Collects the deadlock cycle members, other than t(i,j), into a list and sets total-count
              equal to the list size; also, sets message-count equal to total-count;

              For every transaction t(x,y) in the deadlock cycle member list
                 Sends transaction validate message { VALIDATE(t(i,j), t(x,y)) } to TM(x);
              Endfor
           Else
              Sets the state of t(i,j) back to waiting and sends a PRAG update message {
              UPDATE(PRAG(t(i,j))) } to DM(m), where d(m,n) is the data item requested by t(i,j);
           Endif
        Endif
     Endif
  Else if message-count becomes 0 then /* a false deadlock */
     Searches t(i,j) PRAG tree again to see if the data item for which t(i,j) is waiting, say d(m,n), is
     locked by other transaction;

     If search succeeds then /* another deadlock is detected */
        Collects the deadlock cycle members, other than t(i,j), into a list and sets total-count equal to
        the list size; also, sets message-count equal to total-count;

        For every transaction t(x,y) in the deadlock cycle member list
           Sends transaction validate message { VALIDATE(t(i,j), t(x,y)) } to TM(x);
        Endfor
     Else
        If transaction t(i,j) is active-validating then
           Set t(i,j) state back to active;

           Adds the request branch: t(i,j)->d(k,l) to t(i,j) PRAG tree and sends the lock request mes-
           sage { REQUEST(PRAG(t(i,j))) } to DM(k);
        Else
           Sets the state of t(i,j) back to waiting and sends a PRAG update message {
           UPDATE(PRAG(t(i,j))) } to DM(m), where d(m,n) is the data item requested by t(i,j);
        Endif
     Endif
  Endif.
```

5. Anatomy of The Algorithms

A deadlock detection algorithm is said to be correct if it satisfies 1) no undetected deadlocks and 2) no false deadlocks. The first criterion must be enforced in any deadlock detection algorithm. However, the second criterion sometimes can be relaxed if the detection of false deadlocks would be rare. In facts, the tradeoffs between a complicate algorithm (e.g., our second algorithm) that detects no false deadlocks and a less complicate algorithm (e.g., our first algorithm) that fails to meet this criterion but can minimize the delays and costs of deadlock detection and resolution, usually lead us to favor the simpler approach. In this section, we present observations and analyses of our two algorithms to discuss their properties, correctness, and differences.

In our algorithms, deadlock can be detected in two occasions (events). The first deadlock detection occurs when an active transaction is making a lock request and its TM detects that the lock request will form a deadlock cycle in its PRAG tree. The other deadlock detection occurs when the TM of a waiting transaction receives a PRAG update message and detects the forming of a deadlock cycle in that transaction PRAG tree.

In the first case, only the site where the active transaction resides in would detect the deadlock and the active transaction is selected to abort. We do not use priority to choose the victim for deadlock resolution in this case, because aborting the top transaction of the wait-for chain is the cheapest and simplest way to resolve the deadlock. No cleanups are necessary since all the PRAGs on other sites are still valid after the deadlock is resolved. However, if the cost of aborting the current active transaction outweighs the cost of aborting anyone of other involved waiting transactions for resolving a deadlock, we can modify our algorithms by adding a mechanism that can select the best victim to abort. This mechanism can be more sophisticated than simple priority.

In the second case, once a deadlock cycle is detected all transactions involved in the deadlock are in the waiting state because an active transaction never appears in the PRAGs of other transactions. Therefore, it is possible that several sites detect the same deadlock at different time before the deadlock is resolved. We use priority to solve this problem by selecting the involved transaction with the lowest priority (i.e., the highest timestamp) as the victim to abort. Hence, the site which detects the deadlock first can start the deadlock resolution right away. It does not need to worry about whether other sites may detect the same deadlock and they may select a different victim to abort because there is only one lowest priority transaction in the deadlock cycle. With regard to the redundant transaction abort messages, our algorithm simply discards them. In fact, the operation of a transaction abort in our algorithm is *idempotent* since the restart of an aborted transaction does not use the same transaction identification (timestamp).

It is also possible for those sites which are involved in a deadlock or those sites which have transactions that own share locks on the same data items locked by the members of a deadlock, to contain out-of-date, invalid transaction PRAGs due to the abort of the deadlock victim and the unpredictable delays of message passings. Thus, in our algorithms, when a TM detects a deadlock and starts the deadlock resolution, transaction PRAG clean-up messages are sent to other involved sites as well. This implies that there are possible redundant clean-up messages queued up on a deadlocked site. Fortunately, the operation of clean-up in our algorithms is idempotent too.

In our algorithms, the transaction PRAG update messages and cleanup messages stop their propagations when the target transactions of the messages are active or waiting but the messages do not affect their PRAG trees (i.e., their PRAG trees are already up-to-date).

Because our algorithms use edge-chasing mechanism to build up PRAG trees and we don't allow any transaction in waiting state to abort itself, the PRAG trees would truly record the wait-for relationship among existing transactions. Though, some PRAG trees may not reflect the current state of the system due to message delays, none of them would contain "invalid" wait-for information among existing transactions. The only possible invalid wait-for information recorded in a transaction PRAG tree comes from a deadlock resolution. When the system is resolving deadlocks, the out-of-date wait-for data in a

transaction PRAG tree due to the abort of a deadlock victim (nonexisting) can only result in the detection of a phantom deadlock. In other words, the first criterion of algorithm correctness: no undetected deadlocks, is met in both algorithms. The formal proof of the correctness of probe based edge-chasing algorithm can be found in [Chan82]. In terms of detecting a deadlock, our algorithms works the same as Chandy's algorithm if we consider PRAG update messages the same as probe messages. The differences between theirs and ours are that their algorithm is only for deadlock detection, hence it avoids dealing with the problems of deadlock resolution that would affect the subsequent deadlock detections, and that their algorithm tends to send more probe messages (more cost) and is slower to detect a deadlock (more delays) than ours.

Can our algorithms satisfy the second criterion of correctness: no false deadlocks? If our algorithms had separated deadlock detection from deadlock resolution and during the deadlock resolution those deadlocked sites were halt (i.e., no active transactions were allowed to proceed until the deadlock was resolved), then the second criterion of correctness were ease to met. Because our algorithms use 2PL to synchronize the accesses of data items in the database, unless spontaneous transaction aborts occur due to site failure, it is impossible to have phantom deadlocks [Bern87]. However, the algorithms would be mediocre. As a matter of fact, our algorithms aggressively integrate deadlock resolution into deadlock detection. They allow the deadlocked sites to continue processing of active transactions while the deadlock resolution is underway. Also, they allow data locks to be exclusive (write) locks or share (read) locks. The first algorithm minimizes the cost and the delay of detecting and resolving a deadlock, but fails to guarantee that every deadlock it has detected is a true deadlock. The second algorithm achieves the goal of detecting no false deadlocks by validating each deadlock cycle it detects before starting a deadlock resolution.

Although the first algorithm has the problem of possibly detecting phantom deadlocks due to unpredictable cleanup message delays and the acquisitions of share locks by some deadlocked transactions, we believe that the probability of detecting a phantom deadlock which could result in an unnecessary transaction abort, would be very low. In our algorithm, when a deadlock cycle is detected, the cleanup messages are multicasted to all the involved deadlocked sites before the deadlock victim is actually aborted. The process of cleanup should be very efficient because cleanup messages are processed in parallel, not in a slow, serial fashion. Besides, when multiple sites detect the same deadlock cycle, an identical victim is selected to abort. Thus, even though the deadlocks they detect are phantom deadlocks, no additional transactions are aborted for this kind of phantom deadlocks. Furthermore, the out-of-date PRAG information can only leak out of the deadlock cycle through the share locks, not through the transactions involved in the deadlock. Because when a deadlock cycle is broken, the transaction wait-for states in the cycle remain unchanged except the transaction which was waiting for the aborted victim. This transaction becomes the end transaction of a wait-for chain. If this transaction obtains the released lock and becomes active then it will throw away any out-of-date PRAG update messages it receives to stop the propagation. If it remains waiting because the released lock is assigned to someone else then any out-of-date PRAG update messages will eventually stop there as well. Because when the invalid update message arrives, the same deadlock will be detected again and another round of redundant, harmless deadlock resolution will begin. Therefore, the invalid PRAG information will remain inside the deadlock cycle until it is cleaned up, if there are not any share locks involved in the deadlock. All of these certainly reduce the probability of detecting a phantom deadlock which would cause unnecessary transaction abort.

Based on the first algorithm, we know that a detected deadlock cycle could be a phantom deadlock and since a true deadlock would be persistent, the second algorithm validates each detected deadlock cycle to eliminate the possibility of detecting phantom deadlocks. The validation procedure is very efficient as well. Like the cleanup messages, the validate messages are multicasted to every site that is involved in the deadlock cycle. Therefore, we minimize the extra delay of detecting a true deadlock to not greater than the worst-case round-trip message delay in the system. However, due to the validation delay, the algorithm becomes more complicate. Two intermediate states are needed during the process of

validation. While a TM detects the forming of a deadlock cycle in one of its transaction (active or waiting) PRAG trees and starts the deadlock validation, it needs to keep that transaction PRAG tree up-to-date during the validation, if there are any related transaction PRAG update messages arrive. Otherwise, some useful wait-for information may be lost such that the phenomenon of undetecting true deadlocks may arise. Also, after the validation, if the deadlock is proved to be false or the transaction is not the victim of a true deadlock, we need to search the PRAG tree again to make sure that there is no other cycle exists before we release the leash of updates propagation. This becomes necessary for the second algorithm because the PRAG updates during the validation delay could form another deadlock cycle. Even though the deadlock validation itself can cleanup out-of-date information from the existing transaction PRAG tree if the deadlock is proved to be false, we don't bother to eliminate the cleanup process from the algorithm because we think that the cleanup process can prevent a lot of expensive but unnecessary deadlock validations from execution.

Our algorithms present a simple solution to both problems: deadlock detection and deadlock resolution. When a deadlock resolution starts, the deadlocked sites need not put other active transactions on halt or prevent new transactions from being initiated until the deadlock is resolved. They do not throw away those accumulated, useful PRAG trees either, as long as they remain valid after a victim is aborted. By utilizing the same PRAG information, the algorithms can detect subsequent deadlocks very fast and minimize the cost of communications to a great extent. It makes the separation of deadlock detection and deadlock resolution totally unnecessary.

Our algorithms use an event-driven, message passing, client-server model to accomplish the distributed decision making. Every process is triggered by an event. Once a process receives the triggering message, it does not need any more information through communications to make progress. Once a process completes its task, it either terminates itself or starts other event(s) by sending message(s) to the appropriate process(es). There are two types of message passing: intra-site and inter-site message passing. We shall only consider the cost of inter-site communications. Most of the messages used in our algorithm can be either intra-site or inter-site. The size of each message varies. Since for typical applications, over 90% of WFG cycles are of length two [Bern87], most of the PRAG trees we use in our algorithms is very small. This implies that most of the messages generated by our algorithms will not exceed the size limit of a message packet imposed by the underlying communication links. Thus, although we generate PRAG update messages instead of probe messages for detecting deadlocks, the message size overhead shall be ignorable. Whereas, the savings on reducing the number of inter-site messages and on the quickness of detecting a deadlock and resolving it, far outweigh the extra cost of using larger messages.

6. Comparisons with Other Distributed Algorithms

In this section, we compare our algorithm to other distributed deadlock detection (with/without deadlock resolution) algorithms. The comparisons are based on the four classes of distributed deadlock detection algorithms, defined in [Knapp87].

6.1. Comparison to Edge-chasing Algorithms

In [Chan83], an edge-chasing distributed deadlock detection algorithm was presented in a similar resource model. We would use our model to interpret their algorithm. We don't intend to compare the differences between their model and our model.

In order to determine if a waiting transaction is deadlocked, its TM initiates a *probe computation.* In a probe computation, controllers (TMs + DMs) send messages called probes to one another. A probe is a triple (i, j, k) denoting that it belongs to a probe computation initiated for transaction i, and is being sent from TM j to DM k or from DM j to transaction k. Probes are concerned exclusively with deadlock detection and are distinct from resource requests and replies. Probe computations may be initiated for several transactions by their TMs when these transactions become waiting, and the same transaction may have several probe computations initiated for it in sequence.

Their algorithm works as follows. When a transaction becomes waiting, its TM initiates a probe computation for it and sends a probe message to the DM, where the waiting-for resource resides. The DM updates and passes the probe message to the TM, where the holder (transaction) of that resource resides. The TM records the dependency of the two transactions. If the holder transaction is active then the probe message is discarded; otherwise, if the dependency is a new one, then the TM updates and propagates the probe message to the DM, which has the resource that is requested by the holder but owned by other transaction, and so forth. Notice that the initiator (transaction) in the probe computation is never changed. When a waiting transaction becomes active, its TM discards all the recorded dependency data of that transaction. It follows that if the TM of the initiator receives a probe (i, j, i) for any j, then a deadlock is detected.

When we compare their algorithm to our algorithms, the differences are as follows. They use fixed length probe messages and we use variable length PRAG update messages. They need an auxiliary dependency set for each transaction to record which transactions are waiting-for it. We don't need those since the PRAG trees have that kind information already. The number of PRAG update messages generated by our algorithms is less than the number of probe messages generated by theirs. Because our algorithms detect a deadlock as soon as the deadlock cycle is formed, the PRAG update computation need not go back to the initiator site. In their algorithm, a deadlock is detected whenever the first probe computation completes its cycle. For example, in the case of an active transaction making a request that forms a deadlock cycle, our algorithms will detect the deadlock immediately without any message exchanges. But, their algorithm needs to start a complete round of probe computation in order to detect the same deadlock. Their algorithm is only for deadlock detection because it lacks of enough information to know which transactions are actually involved in the deadlock. Therefore, it cannot select a proper victim to start the deadlock resolution. On the contrary, our algorithms can resolve deadlock immediately after the deadlock cycle is found. There is no need to have a separate phase for deadlock resolution. This further speeds up detection and resolution of future deadlocks.

In [Sinha85], another edge-chasing algorithm was presented. It was a priority based distributed deadlock detection algorithm. It tried to reduce the number of probe messages generated for probe computations by introducing priorities (timestamps) for transactions and only allowing those transactions that have *antagonistic conflicts* to initiate probe computations. An antagonistic conflict occurs when a transaction waits for a data item locked by a lower priority (younger) transaction. The algorithm provides deadlock resolution as well. However, the algorithm was proved incorrect by [Chou89]. It fails to satisfy both criteria of correctness: no undetected deadlocks and no false deadlocks.

In [Chou89], a modification to Sinha's priority based probe algorithm is proposed to correct the problems they identified. But, the modified algorithm has to separate deadlock detection from deadlock resolution again and the message overhead is greatly increased because after a deadlock is resolved, all members of the cycle discard all the probes from their probe queues and thus, probes need to be retransmitted. In addition to the delay introduced due to reinitiation and retransmission of probes, the modified algorithm also "suggests" that a deadlock victim releases its resources only after its cleanup message returns to itself. Therefore, transactions waiting on resources held by the victim cannot acquire the resources even though the deadlock has been detected and resolved. This further increases the delay of deadlock resolution. In addition, their cleanup method has to chase the edges of deadlocked TWFG again due to the lack of TWFG information. This is a serial method. It is slow and affected by the length of deadlock cycle. And the suggestion of waiting for the cleanup message to return to itself then the deadlock victim can release its resources, is incorrect either. Because when a deadlock victim receives its own cleanup message, it still can not guarantee that the cleanup process is done. That means their algorithm will suffer phantom deadlock problems as well. we believe that our algorithm not only would outperform their algorithm but also is easier to implement.

6.2. Comparison to Path-pushing Algorithms

The basic problem for a path pushing algorithm is that in general each site does not know where to send its paths. If each site needs to send its paths to all other sites then the communications cost could easily overshadow the benefit of detecting short deadlock cycles more quickly [Bern87]. In [Ober82], two optimizations that reduce the path pushing message traffic are employed. One optimization ensures that potential deadlock cycle information is transmitted in a single direction along the path of an elementary cycle. The second optimization uses the lexical ordering of the transaction identifiers to reduce the average number of messages by half.

A more serious problem is that because each site maintains a local TWFG, whenever a site decides to push paths to other sites it needs to decompose the local TWFG into paths and selectively sends portions of the list of paths to other sites. Not to mention the overheads of decomposition and optimization, by the time the pushed paths arrive the destination sites the snapshots of local TWFGs may be out-of-date already. Therefore, the problems of failing to detect true deadlocks or detecting false deadlocks or both, can be found in many path-pushing algorithms. In [Ober82], an unrealistic assumption was made: portions of the distributed TWFG transmitted as Strings will remain frozen until after global deadlock has either been determined or the piece has reached some "final" destination.

Our algorithms do not maintain local TWFG on each site. Thus, the path decomposition and pushing optimization problems do not exist in our algorithms. It uses edge-chasing mechanism to update transaction PRAGs. If we treat each transaction PRAG update message as a decomposed path then the algorithms always know where and when to push the path. They also know where and when to stop the push. Therefore, we believe that the message overhead and delay of detecting a deadlock in our algorithm are lower than Obermarck's algorithm.

There are two main approaches to the problem of false deadlocks. The first approach is to treat false deadlocks as if they were real ones. This approach is acceptable as long as the number of false deadlocks is low. Our first algorithm is a good example for this approach. It makes the number of possible false deadlocks be very low. An alternate approach is based on the fact that real deadlocks will persist until broken. When a deadlock cycle is detected, validation of the transaction wait-for relationships that make up the cycle can be performed. The proposed approach for validation in [Ober82] is: when a global deadlock cycle is detected by a given site, it can be validated by sending it to each site, in turn, that contains a local edge of the cycle. If the cycle constitutes a true deadlock, it will return to the detecting site for validation of the last edge. Unlike this backward, serial edge-chasing method, our second algorithm uses a parallel approach to do validation. Our method minimizes the delay of detecting a true deadlock to roughly a round-trip message delay, no matter what is the length of the validating deadlock cycle.

6.3. Comparison to Diffusing Computation Algorithms

The algorithm in [Chan83] for the communication deadlock model is an application of the diffusing computation technique. The diffusing computation grows by sending *query* messages and shrinks by receiving *replies*. The communication model is also referred to as the OR model in [Knapp87]. The OR model allows replicated data items on different sites of the system. Therefore, discovery of a cycle is insufficient for deadlock detection. Since the basic database models are different, we should not compare these algorithms. For more detailed information about diffusing computation, see [Chan83], [Knapp87].

6.4. Comparison to Global State Detection Algorithms

The global state detection algorithms are based on the results developed by [Chan85]. A key notion of global state detection is that a consistent global state can be determined without temporarily suspending ("freezing") the underlying database computation. If a deadlock is detected in a "consistent" global state of the system (a carefully taken snapshot, not an arbitrary snapshot, of the system) then the deadlock must also exist in the real time. [Chan85] shows how to obtain a consistent global state of a distributed

system by propagating markers along the channels of the system. Since such a snapshot of the system by definition is a static object, there are no problems in conjunction with message delays. Thus, deadlock detection becomes much easier. In the context of deadlock detection, the consistent global state can be a global TWFG.

If we consider that building up transaction PRAG trees is to maintain a consistent global state of the system, then our algorithms can be classified as global state detection algorithms. In our algorithms, each site maintains a table of transaction PRAG trees which is always consistent but may not be current during deadlock detection time. The only time the transaction PRAG trees may not be consistent is during deadlock resolution time. However, we have proved that the inconsistency can be fixed by imposing deadlock validation in our second algorithm.

The global state detection algorithm in [BRA84] is applied to the very generalized distributed database model, the (n,k) model. The (n,k) model allows the specification of requests to obtain any k available resources out of a pool of size n. A transaction becomes blocked upon sending out n request messages. It becomes executing again when it receives k grant messages. Since our algorithms use one-resource model (i.e., a transaction can have at most one outstanding resource request at a time), it will be inappropriate to compare our algorithms to their algorithm here.

7. Conclusions

In this paper, we have presented two distributed algorithms that solve both deadlock detection and deadlock resolution problems, using a simple distributed resource model. The algorithms lay their theoretical ground on edge-chasing, path-pushing, and PRAGs (Partial Resource Allocation Graphs), in term of deadlock detection. The first algorithm is very attractive while comparing it to other distributed deadlock detection algorithms in the literature. Although, it may consume more local memory than other algorithms since each transaction on a site owns a PRAG tree and the PRAG update messages may be longer than probe messages generated by other algorithms, the overall benefits are countless. It not only requires less inter-site message communications but also detects deadlocks and resolves them very fast. The only drawback in our first algorithm is that it can not completely eliminate the detection of false deadlocks which may cause unnecessary transaction aborts.

The second algorithm satisfies both criteria of correctness. Unlike the first algorithm, this algorithm does not start the deadlock resolution immediately when the forming of a deadlock cycle is detected. Instead, it validates every potential deadlock and only resolves those deadlocks that have been proved to be true deadlocks. While comparing it to the first algorithm, the second algorithm merely pays the price of an extra round-trip message delay to achieve the goal of no false deadlock detections. Therefore, while comparing both algorithms to other algorithms in the literature, we believe that our algorithms have a lot of advantages in terms of communication costs and the delays of deadlock detection and resolution.

Currently we plan to conduct a performance study on our algorithms and other related distributed algorithms under the categories such as the algorithm complexity, the average message size, the number of inter-site messages required to detect the same deadlock, the memory size required for each site to do deadlock detection, the time delay to detect a deadlock, and the time delay to resolve a deadlock, etc. With such experiments, we will be able to provide quantitative results to show the power of our algorithms.

There are a number of issues remain to be addressed. Future researches should focus on developing a formal proof of the algorithm correctness, generalizing the algorithms to accommodate the AND model, the OR model, and the AND-OR model [Knapp87], if possible. Other possible research area is extending the algorithms to handle site failures and communication failures.

References

[Bern81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Vol. 13, June, 1981.

[Bern87] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and recovery in Database Systems*, Addison Wesley, 1987.

[Bra84] G. Bracha and S. Toueg, "A Distributed Algorithm for Generalized Deadlock Detection", *Proc. ACM Symp. on Principles of Distributed Computing*, August, 1984.

[Chan82] K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", *Proc. ACM Symp. on Principles of Distributed Computing*, August, 1982.

[Chan83] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computer Systems*, Vol. 1, No. 2, May, 1983.

[Chan85] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 1, February, 1985.

[Chou89] A. N. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley, "A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution", *IEEE Transactions on Software Engineering*, Vol. 15, No. 1, January, 1989.

[Chou90] A. N. Choudhary, "Cost of Distributed Deadlock Detection : A Performance Study", *Sixth IEEE International Conference on Data Engineering*, Febrary, 1990.

[Ho82] G. S. Ho and C. V. Ramamoorthy, "Protocols for Deadlock Detection in Distributed Database Systems", *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 6, November, 1982.

[Knapp87] E. Knapp, "Deadlock Detection in Distributed Databases", *ACM Computing Surveys*, Vol. 19, No. 4, December, 1987.

[Lam78] L. Lamport, "Time, Clocks and Ordering of Events in Distributed Systems", *Communications of ACM*, Vol. 21, No. 7, July, 1978.

[Mena79] D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Databases", *IEEE Transactions on Software Engineering*, Vol. SE-5, May, 1979.

[Mitch84] D. A. Menasce and R. R. Muntz, "A Distributed Algorithm for Deadlock Detection and Resolution", *Proc. ACM Conf. Principles of Distributed Computing*, August, 1984.

[Ober82] R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, Vol. 7, No. 2, June, 1982.

[Sing89] M. Singhal, "Deadlock Detection in Distributed Systems", *IEEE Computer*, November, 1989.

[Sinha84] M. K. Sinha and N. Natarajan, "A Distributed Deadlock Detection Algorithm Based on Timestamps", *Proc. of the 4th International Conference on Distributed Computing Systems*, IEEE, New York, 1984, pp. 546-556.

[Sinha85] M. K. Sinha and N. Natarajan, "A Priority Based Distributed Deadlock Detection Algorithm", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, January, 1985.

[Wuu85] G. Wuu and A. Bernstein, "False Deadlock Detection in Distributed Systems", *IEEE Transactions on Software Engineering*, August, 1985.