# PERFORMANCE EVALUATION OF TWO DISTRIBUTED DEADLOCK DETECTION ALGORITHMS

A. Choudhary, W. Kohler, J. Stankovic, and D. Towsley

COINS Technical Report 89-13

November 1988

# Performance Evaluation of Two Distributed Deadlock Detection Algorithms

Alok N. Choudhary [*]
Walter Kohler [†]
John A. Stankovic[‡]
Don Towsley[§]

November 1988

## Abstract

Distributed deadlock detection can require significant overhead, adversely affecting the performance of a distributed database system. SET-based and probe based algorithms have been touted as the more efficient distributed deadlock detection algorithms known. This paper first presents optimizations that enhance the performance of the SET-based approach. A performance analysis is then performed which shows two main results. One, the new SET-based algorithm outperforms probe based algorithms. Two, current analytical models of distributed deadlock detection are inaccurate because they only compute the overhead of deadlock detection when deadlock exists. We show that this overhead cost is only a small portion of the total overall costs, i.e., the cost of running the algorithm when deadlock does not exist dominates the cost of the algorithm when deadlock does exist.

**Index Terms -** Concurrency Control, Distributed Database, Deadlock Detection, Performance Evaluation

[*]Computer Systems Group, Coordinated Science Laboratory, University of Illinois, 1101 W. Springfield Ave., Urbana, IL 61801.

[†]Digital Equipment Corporation, TP Systems Group, Marlboro, MA 01752.

[‡]Department of Computer & Information Science, University of Massachusetts, Amherst, MA 01003.

[§]Department of Computer & Information Science, University of Massachusetts, Amherst, MA 01003.

## 1 Introduction

In a distributed database system, data accesses by concurrent transactions are synchronized in order to preserve database consistency. The synchronization can be achieved using concurrency control algorithms such as two phase locking (2PL), timestamp ordering [6], optimistic concurrency control [17] or a variation of these basic algorithms. In practice, the most commonly used concurrency control algorithm is 2PL. However, if locking is used, a group of transactions may get involved in a deadlock. Consequently, some form of deadlock resolution must accompany 2PL. This paper presents and evaluates two deadlock detection algorithms for distributed databases.

Deadlock detection is very difficult in a distributed database system because no controller has complete and current information about the system and data dependencies. Therefore, information about transactions' dependencies needs to be propagated across sites (or nodes) to look for deadlocks. This can incur significant overhead costs.

Many algorithms have been proposed to detect deadlocks in distributed database systems [8, 9, 20, 14, 23, 15, 4, 26, 3]. Some algorithms detect deadlocks by first constructing and then finding cycles in a transaction wait-for graph (a directed graph whose nodes represent transactions and arcs represent the wait-for relationships). Solutions based on this method are quite expensive because a large amount of information needs to be propagated from site to site [23].

Another method is based on transmitting probes between sites. Probes are special messages used to detect deadlocks. Probes follow the edges of the wait-for graph without constructing a separate representation of the graph [8, 9, 26]. The advantage of this approach is that probe algorithms are more efficient than wait-for-graphs. The disadvantage of the probe approach is that after a deadlock is detected, the constituents of the cycle remain to be discovered. Sinha and Natarajan [26] proposed a probe algorithm that purported to have improved performance over previously known probe algorithms. However, we have previously shown that in many situations their algorithm either fails to detect a deadlock or detects a false deadlock [11]. We also proposed a modified probe algorithm that eliminates these problems. This modified probe algorithm [11] is used in this paper as a basis for the performance evaluation presented. Roesler, et. al. [24] have also proposed a new algorithm to correct the problems with [26].

The third main approach to performing distributed deadlock detection is a combination of the probe and the wait-for-graph approaches. This hybrid approach, referred to as the SET-based approach, is based on propagating more information than in the probe technique, but less information than in the wait-for-graph approach concerning potential deadlock candidates. An algorithm based on this method was presented in [15].

In addition to the extensive literature in these three general approaches to deadlock detection, a number of other papers have addressed the deadlock resolution problem in yet other ways. For example, in [16] a distributed algorithm for deadlock prevention is presented. In [22] an algorithm for detecting deadlocks that assumes a communication model rather than a resource based model (as in the work presented here) is presented. An evaluation study of deadlock resolution in a *centralized* system is described in [2]. Each of these papers while interesting, addresses a different problem than the one addressed in this paper. Consequently, their results are not applicable here.

In this paper we first present extensions to the current SET-based approach to further enhance its performance. A performance evaluation is then performed which shows two main results. First, the new SET-based algorithm outperforms the probe based algorithms. Second, the study indicates that most of the cost of running the algorithm occurs when no deadlock exists, i.e., the cost of running the algorithm when deadlock does not exist, dominates the cost of the algorithm when deadlock does exist.

The rest of this paper is organized as follows. Section 2 presents the distributed database model assumed throughout the paper. The new SET-based distributed deadlock detection algorithm and a summary of the probe algorithm [11] are presented in Section 3. We will refer to the SET-based algorithm as SET algorithm in the rest of this paper. Section 4 outlines the simulation model and discusses the experimental results that compare the performance of the two algorithms. Finally, conclusions are drawn in section 5.

## 2 Distributed Database System Model

A distributed database is a collection of data objects spread across a number of sites which communicate with each other via messages. Each site has data objects, controllers (schedulers), and data managers. Figure 1 shows one such distributed database model [6].

Each node has one or more of the following modules - a Transaction Manager (TM), a Data Manager (DM), a Scheduler (S) and a Transaction Process (T). The scheduler at each site synchronizes the transaction requests and performs deadlock detection.

A transaction may request multiple data objects simultaneously. In this case, all objects must be granted before the transaction can continue. The SET algorithm that we will propose in the next section detects deadlocks when transactions have multiple threads of control.

The following transaction control structure is assumed. A transaction is initiated at one site. It may then initiate one or more remote transactions at other sites. In this case the original transaction is referred to as a *master* transaction and the remote transactions as *slave* transactions (slave transactions are always depicted with a prime notation in the figures and examples). A slave can become a master by creating its own slave transactions. In other words, a transaction may follow a tree structure with all leaf nodes being slaves. In such a transaction system aborting a slave transaction may not necessarily result in aborting the root transaction [19]. Our algorithm suggests a victim to resolve a deadlock cycle, but it does not enforce the condition

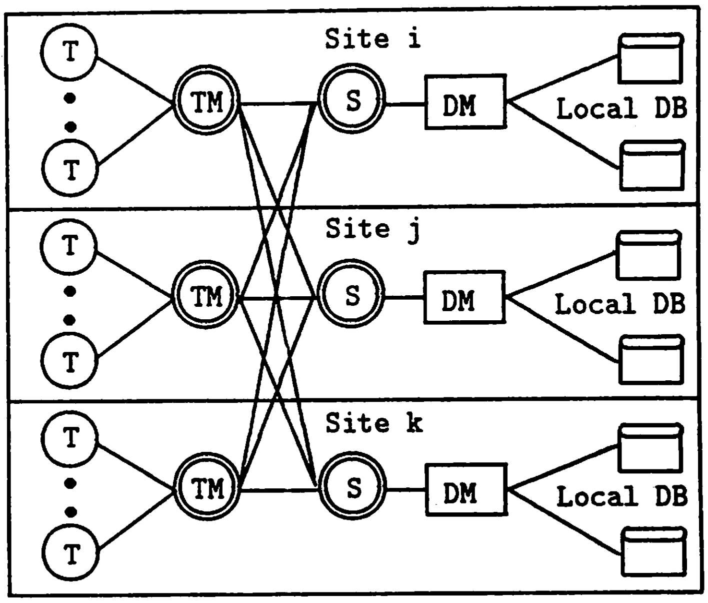

Figure 1: The Distributed Database System Model

that the entire tree structure involving the victim be aborted. For example, if one of the slave transactions is involved in a deadlock then deadlock may be resolved by aborting only that slave, or by aborting the master transaction and all its slaves. This choice depends on the semantics of the transactions and the consistency criteria enforced by the underlying transaction processing system. We also assume a reliable communication system that guarantees the following:

- All messages arrive at their destinations in finite time.
- All messages are transmitted without errors.
- All messages from site $i$ to site $j$ arrive in the same order in which they were sent.

The concurrency mechanism is assumed to be two phase locking. Locks may be requested in shared or exclusive mode. Any number of transactions can lock the same data object in shared mode. At most one transaction can lock a data object in exclusive mode. At that time no other transaction can lock that data object in shared mode.

**Definition 1** *The set of transactions that hold locks on data objects for which transactions $T$ is waiting is called $Block(T)$.*

**Definition 2** *The set of transactions that are waiting for data objects currently locked by transaction $T$ is called $Block\_on(T)$.*

**Definition 3** *A set of transactions $G$ is said to be a deadlocked group iff for all transactions $T_i \in G$, there exists $T_j \in G$ such that $T_j \in Block(T_i)$, and there are no unreceived messages between transactions in $G$. A transaction $T$ is said to be deadlocked iff it is a member of some $G$.*

**Definition 4** *A deadlock cycle is an ordered set of transactions $\{T_0, T_1, \cdots, T_n\}$ such that at a given point in time, $\forall j,\ 0 \leq j < n$ , $T_{j+1} \in Block(T_j)$, $T_j \in Block\_on(T_{j+1})$, $T_0 \in Block(T_n)$ and $T_n \in Block\_on(T_0)$.*

It follows from the above definitions that a deadlock cycle is a subset of a deadlocked group. However, not all transactions within a deadlocked group may reside in a deadlock cycle. There may be some transactions which may wait transitively on a transaction which is a member of a deadlock cycle. Also, in order to break a deadlock cycle, at least one member of the deadlock cycle has to be aborted. In general, if there is more than one cycle, it may be necessary to abort more than one transaction in order to break all the cycles.

It is convenient to distinguish between two types of deadlock cycles: a local deadlock cycle and a distributed deadlock cycle. A deadlock cycle is a local deadlock cycle at a site $m$ if all of its constituent transactions are waiting at that site. A deadlock cycle is a distributed deadlock cycle if it is not a local deadlock cycle.

## 3 The Distributed Deadlock Detection Algorithms

In this section we present a detailed description of the SET algorithm and a brief description of the *probe* algorithm. A more detailed description of the latter algorithm can be found in [11]. Examples are used to illustrate the new SET algorithm.

### 3.1 The SET Algorithm

We present both an informal as well as formal description of the algorithm. Further, we present a proof for the correctness of the algorithm and give some examples of its operation.

We assume that transactions are assigned priorities. The priority of a transaction $T$ is $P(T)$. Assignment of priorities to transactions may be based on any predefined criteria. For example, a transaction requiring only one resource may be given higher priority than one requiring many resources at once. Throughout the remainder of this paper, whenever we are given two transactions $T_i$ and $T_j$, it will be understood $P(T_i) > P(T_j)$ iff $i > j$. The following notation and definitions will be used to describe the algorithm :

- $S = (T_1, T_2, \cdots, T_n)$, a string of transactions. The string of transactions represents a wait-for relationship, $T_i \in Block(T_{i+1})$, $1 \leq i < n$.
- *high*$(S)$ - Highest priority transaction in the string $S$.
- *low*$(S)$ - Lowest priority transaction in the string $S$.
- *last*$(S)$ - Last transaction in the string $S$.
- *first*$(S)$ - First transaction in the string $S$.
- *slave*$(T)$ - Set of slave transactions associated with $T$.
- *substring*$(T', T, S)$ - The substring in string $S$ beginning with $T'$ and ending with $T$.

The reason that we need to define all these terms to describe the algorithm is that the algorithm will create strings of blocked transactions at local sites. These strings are then transmitted to other sites where they are concatenated with other strings and checked for cycles.

**Definition 5** *Let $S_1 = (T_k, \cdots, T_m)$ and $S_2 = (T_m, \cdots, T_n)$ be strings of transactions. Then the concatenation of the strings is given by $S_1 || S_2 = (T_k, \cdots, T_m, \cdots, T_n)$. Furthermore, if $\Sigma_1$ and $\Sigma_2$ are two sets of strings of transactions such that the last transaction of each string in $\Sigma_1$ is the same as the first transaction of each string in $\Sigma_2$ , then $\Sigma_1 || \Sigma_2 = \{S_1 || S_2 | \forall S_1 \in \Sigma_1, S_2 \in \Sigma_2\}$.*

**Definition 6** *A forward dependency group for transaction $T_1$ is a set of transaction strings $F_D(T_1) = \{S_1, S_2, \cdots, S_n\}$ containing strings of the form $S = (T_1, T_2, \cdots, T_m)$ where $T_{i+1} \in Block(T_i)$, $T_i \in Block\_on(T_{i+1})$, $1 \leq i < m$.*

**Definition 7** *A backward dependency group for transaction $T_m$ is a set of transaction strings $B_D(T_m) = \{S_1, S_2, \cdots, S_n\}$ containing strings of the form $S = (T_1, T_2, \cdots, T_m)$ where $T_{i+1} \in Block(T_i)$, $T_i \in Block\_on(T_{i+1})$, $1 \leq i < m$.*

**Definition 8** *A string of transactions $(T_1, T_2, \cdots, T_n)$ is said to be a global candidate $G$ if $T_{j+1} \in Block(T_j)$, $1 \leq j \leq n$, $T_1$ is a slave, $T_n$ is a master, and all of the transactions reside at the same site.*

**Definition 9** *A propagation global candidate (denoted as PGC) is either a global candidate or a concatenation of two or more global candidates.*

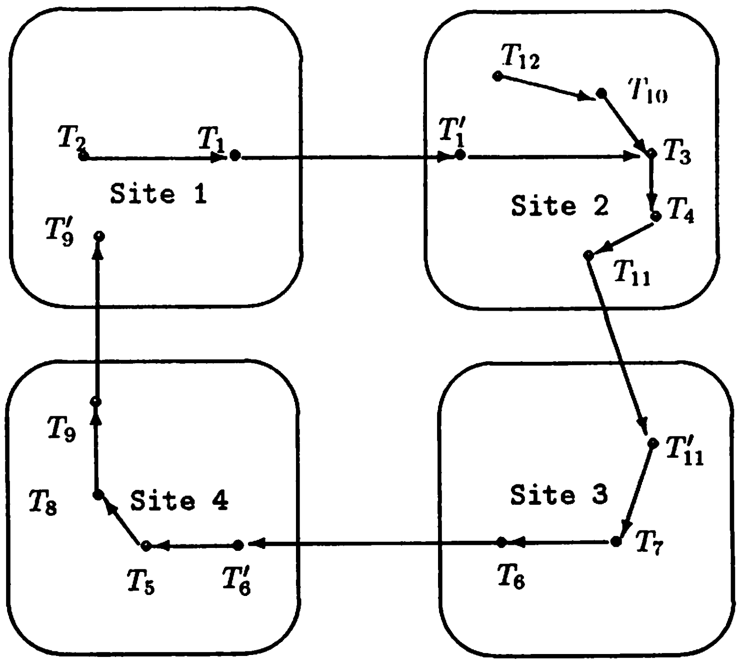

Figure 2: Avoiding Unnecessary Messages.

A global candidate $G$ is potentially part of a deadlock cycle over two or more sites. For example, in Figure 2, at site 2 $(T'_1, T_3, T_4, T_{11})$ and at site 3 $(T'_{11}, T_7, T_6)$ are global candidates, whereas at site 1, $(T_2, T_1)$ is not a global candidate. A substring of a global candidate contributed by one site is not a global candidate. Again, a global candidate has to begin with a slave transaction because that is the only case in which a string can be a part of a global deadlock cycle.

The purpose of the SET algorithm is to identify global candidates at each site and to propagate them among the sites so that they may detect deadlocks involving transactions over two or more sites. Hence, each site is required to maintain a set of global candidates, $GS(i)$ for site $i$. Deletion of global candidates from $GS(i)$ occurs whenever a transaction aborts and whenever a transaction previously waiting for a lock, is awarded the lock and becomes active. At these times, all global candidates within $GS(i)$ that contain these transactions are removed.

The creation of global candidates and their propagation between the sites is described in the remainder of this section. Inclusion of global candidates as a criteria to propagate information across sites is an optimization to the SET algorithm because it avoids sending unnecessary messages across sites. For example, in Figure 2, the string $(T_1, T_2)$ will not be sent to site 2.

### 3.1.1 Creation of global candidates - The SET algorithm, Part I

The search for new global candidates is initiated whenever a transaction $T$ times out while waiting for a resource. The scheduler begins with a set of transaction strings of the form $\{(T)\}$ where, $T$ is the transaction which has timed out. For example, in Figure 2 if the scheduler at site 2 initiates on behalf of $T_3$ , the initial set will look like $\{(T_3)\}$. Subsequently, the scheduler scans along the wait-for edges in the forward direction looking for local deadlock cycles and builds the forward dependency group $F_D(T_3)$. Strings are discarded from $F_D$ if along the scan path of each string, a non-idle transaction is encountered. Eventually, only those transaction strings within $F_D$ which have a master[1] waiting on a remote request as their last transaction are kept. For example, $F_D(T_3) = \{(T_3, T_4, T_{11})\}$ in Figure 2. Now, in order to find all global candidates, the scheduler scans in the backward direction beginning with the same (initiating) transaction. The scheduler stops when all the slave transactions lying in the path of the backward scan are included in the backward dependency group $B_D(T_3)$. For example, in Figure 2, $B_D(T_3) = \{(T_1', T_3), (T_{12}, T_{10}, T_3)\}$ ; the string $(T_{12}, T_{10}, T_3)$ will be discarded because it cannot be a part of a global candidate. Then the scheduler concatenates strings from forward and backward dependency sets in order to form global candidates $G$. In the above example, $G = (T_1', T_3, T_4, T_{11})$. If a local cycle exists, then the transaction having the lowest priority is chosen as the deadlock victim. If after resolving local deadlocks, the scheduler discovers that one or more global candidates exist, it saves the global candidates for distributed deadlock computation. It propagates a particular global candidate to each site containing a slave of $T_n$ (where $T_n$ is the last transaction a string of $G$ ) if $(P(T_1), P(T_2), \cdots, P(T_n))$ is not a monotonically increasing sequence. This constraint reduces the message overhead because sites will propagate global candidates only if this condition is satisfied. This is another optimization included in the SET algorithm that reduces the number of messages sent across sites. Because of unique priority assignments, not all initiations result in propagation of messages across sites. However, the above criteria is satisfied at least at one site so that a deadlock will be detected eventually. The pseudocode for this part of the algorithm, called Part I, is found in Figure 3.

Some comments regarding the timeout parameter are in order. Its value can be chosen to be any finite value including zero. Although the value of this parameter affects the performance of the algorithm, it has no impact on the correctness of the algorithm. A transaction may time out more than once depending on the amount of time assigned to its timeout period.

### 3.1.2 Upon receiving a Global Candidate - The Set Algorithm - Part II

Part II of the algorithm describes what a scheduler does when it receives global candidates from other sites. First, the algorithm performs Part I if the global candidates at the receiving site have not already been formed. If the highest priority transaction in one or more corresponding

[1] Note that because of our database model it is impossible for a slave to be waiting on a remote request.

(i.e., concatenable) global candidates at the receiving site has a priority greater than that of the highest priority transaction in the received global candidates, then it discards the received global candidates. Otherwise, it checks for distributed deadlocks by forming PGCs by concatenating its own global candidates with the received ones. If combining the received and formed global candidates does not result in the detection of a deadlock then the PGCs are further propagated in a similar manner. For example, in Figure 2, if the global candidate $G = (T'_{11}, T_7, T_6)$ is propagated from site 3 to site 4, then site 4 composes $G = (T'_{11}, T_7, T_6)$ with $(T'_6, T_5, T_8, T_9)$ to get a new PGC $(T'_{11}, T_7, T_6, T_5, T_8, T_9)$. Since this new propagation global candidate is not monotonically increasing in priority, it is also propagated to site 1 where it will be determined that there is no distributed deadlock. The pseudocode for Part II of the algorithm is found in Figure 4.

### 3.1.3 Waiting transaction becomes active

The following is executed whenever a waiting transaction becomes active.

```
if a transaction T enters the active state from a waiting state then
  Delete all global candidates $GC$ such that $T \in GC$.
```

### 3.1.4 Summary of Optimizations

In [15] a SET-based algorithm was presented for distributed deadlock detection. Our SET algorithm adds several optimizations to the original SET algorithm approach. They are as follows:

- Specifically identifying global candidates reduces the number of intersite messages. In [15] strings of transactions are sent across sites even if the strings may not be global candidates. That is unnecessary because unless a string of transactions is a global candidate it can not be a part of a distributed deadlock cycle.
- In our SET algorithm intersite messages are further reduced by the following constraint: A global candidate $(T_1, T_2, ..., T_n)$ is propagated across sites only if $(P(T_1), P(T_2), ..., P(T_n))$ is not a monotonically increasing sequence, i.e., the priorities of transactions in the global candidate is not a monotonically increasing sequence.

### 3.1.5 Example 1

Consider the example shown in Figure 5. together with the pseudo code for the SET algorithm shown in Figure 3 and Figure 4. For notational convenience, subscripts uniquely identify the transactions as well as their priorities. A larger subscript denotes a higher priority. Assume

When a waiting transaction $T$ times out at site $i$, the local scheduler looks for local deadlocks and builds up global candidates to detect distributed deadlocks.

```
Step 1: Build $F_D(T)$ for T.
  if$Block\_on(T) = \emptyset$ then
    Stop.
  Let $C$ be the set of all candidate forward strings with $T$ as the first transaction.
  $C:=\{(T)\}$
  $F_D(T):=\emptyset$
  repeat
    Pick $S$ from $C$
    $C:=C - \{S\}$
    $T':=last(S)$
    foreach $T'' \in Block(T')$ such that $T''$ is waiting
      if$T \in S$ then
        Declare deadlock involving substring $(T'', T', S)$
        $victim:=low(substring(T'', T', S))$
        Remove all strings from $C$ containing the victim
      else
        $S:=S||(T', T'')$
        if$T''$ is a master transaction then
          $F_D(T):=F_D(T) \cup \{S\}$.
        $C:=C \cup \{S\}$
  until$C = \emptyset$
Step 2: Use a similar procedure to construct $B_D(T)$ for T.
Step 3: Form set of global candidates $G(T)$ from $F_D(T)$ and $B_D(T)$. Propagate the set of global candidates $G$ to appropriate sites. Add to $GS(i)$.
  $G(T) := F_D(T)||B_D(T)$
  $GS(i) := G(T) \cup GS(i)$
/*Each string in $G(T)$ is of the form $(T_1, T_2, \cdots, T_n)$ and $n \geq 2$. Note that $T_1$ will always be a slave and $T_n$ will always be a master transaction.*/
foreach global candidate $S \in G(T)$
  if$(P(T_1), P(T_2), \cdots, P(T_n))$ is not a monotonically increasing sequence then
    foreach site containing $T' \in slave(T_n)$, send a copy of $G(T)$ to it.
```

Figure 3: SET Algorithm: Part I - Creation of Global Candidates.

When the scheduler at site $j$ receives a set of global candidates, $G$, from some other site, the scheduler looks for distributed deadlocks. The scheduler also forwards global candidates to other sites if it has its own global candidates which can be concatenated with the received ones.

```
Step 1: Delete all those global candidates from $G$ whose corresponding slaves are not idle at this site.
foreach global candidate $S \in G$
  $T$:=$last(S)$
  ifall slaves at site $j$ associated with $T$are active then
    $G$:=$G - \{S\}$
if$G = \emptyset$ then
  Stop.
Step 2 : Check for distributed deadlocks.
$G'$:=$\emptyset$
foreach global candidate $S \in G$
  foreach global candidate $S' \in GS(j)$
    $T$:=$last(S)$
    if$first(S') \in slave(T)$
      if$high(S') > high(S)$ then
        $G$:=$G - \{S\}$
      else
        $T'$:=$last(S')$
        if$slave(T') \cap G \neq \emptyset$ then
          $S''$:=$S||S'$
          $T''$:=$slave(T)$ closest to $T$ in $S$
          if$high(S'') \in substring(T'', T', S)$ then
              Declare distributed deadlock with cycle $D_l$:=$substring(T'', T', S'')$
              $victim = low(D_l)$
        else
          $G'$:=$G' \cup \{S\}||G$
  foreach string $S = (T_1, T_2, \cdots, T_n) \in G'$
    if$(P(T_1), P(T_2), \cdots, P(T_n))$ is not a monotonically increasing sequence then
      foreach site containing $T' \in slave(T_n)$, send a copy of $G(T)$ to it.

Step 3: It is possible that there exists a global candidate at site j that has not been created when G is received. This step identifies and constructs the global candidates required to complete step 2 above.
foreach global candidate $S \in G$
  $T$:=$last(S)$
  foreach $T' \in slave(T)$
    if($T'$ is idle) and ($T' \notin G$)
      Build global candidates with $T'$ as initiator using Part I
      Repeat step 2 ( PART II) to look for distributed deadlocks.
```

Figure 4: SET Algorithm - Part II.

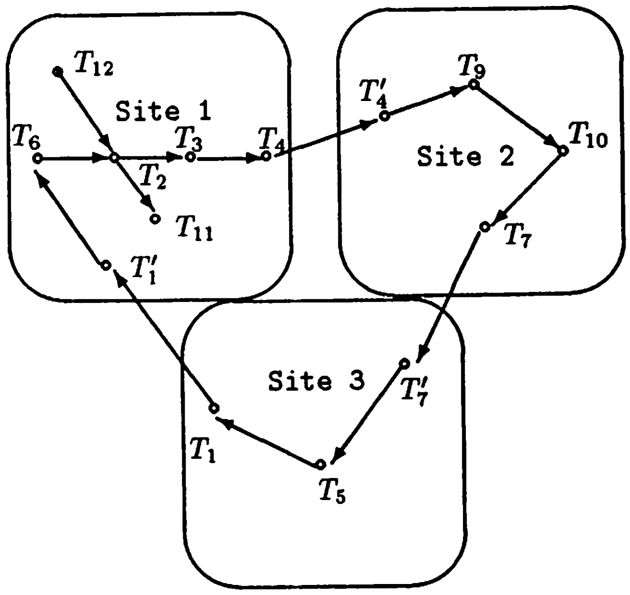

Figure 5: Example Illustrating the Algorithm.

site 1 initiates the deadlock detection when $T_2$ times out and, in parallel, site 2 initiates the algorithm when $T_9$ times out. $G_S(i)$ and $G(i)$ are used to identify the site to which $G_S$ and $G$ belong. $G_S$ denotes a set of global candidates at a site whereas $G$ denotes one global candidate.

**Part I**

At site 1 : Initially, $C{:=}\{(T_2)\}$ and $F_D(T_2){:=}\emptyset$. $T_3$ and $T_{11} \in \mathit{Block}(T_2)$, but only $T_3$ is waiting. Therefore, from step 1, the first iteration of the repeat-until loop yields $S{:=}(T_2,T_3)$ and $C{:=}\{(T_2,T_3)\}$. The second iteration results in $C{:=}\emptyset$, $S{:=}(T_2,T_3,T_4)$ and therefore, $F_D(T_2) = \{(T_2,T_3,T_4)\}$.
Step 2 uses the same procedure as in step 1 to form $B_D(T_2) = \{(T_1',T_6,T_2)\}$. Note that $(T_{12},T_2) \notin F_D(T_2)$ because $T_{12}$ is not blocking any transaction.
Step 3 combines $F_D(T_2)$ and $B_D(T_2)$ to form a set of global candidates. $G_S(1){:=}B_D(T_2)||F_D(T_2) = \{(T_1',T_6,T_2,T_3,T_4)\}$. The scheduler at site 1 sends $G_S(1)$ to site 2 because site 2 contains *slave*$(T_4)$.
Similarly, the computation at site 2 proceeds as follows: Initially, $C{:=}\{(T_9)\}$ and $F_D(T_9) = \emptyset$. From step 1, $F_D(T_9) = \{(T_9,T_{10},T_7)\}$ and from step 2, $B_D(T_9) = \{(T_4',T_9)\}$. Using step 3, Global candidate $G_S(2) = \{(T_4',T_9,T_{10},T_7)\}$ is sent to site 3.

**Part II**

When the scheduler at site 2 receives $G_S(1)$ it does the following.

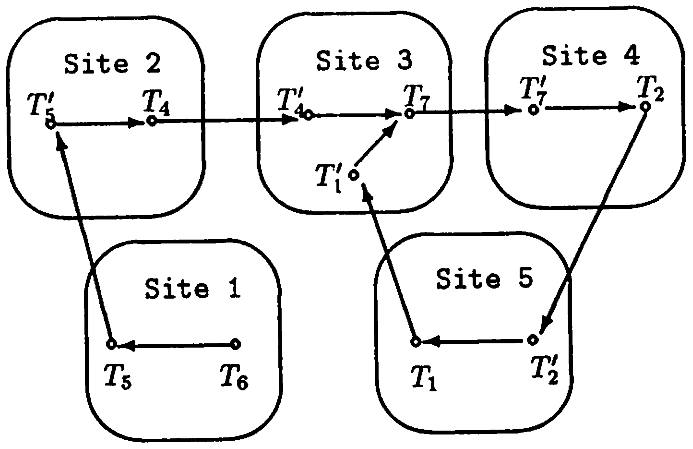

Figure 6: Use of Set Information.

From step 1, for all $G(1) \in G_S(1)$, it checks if $T = last(G(1))$ is waiting. In this example, $last(G(1)) = T_4$, which is waiting. Therefore, it checks if $high(G_S(2)) > high(G(1))$. $High(G_S(2)) = T_{10}$ and $high(G(1)) = T_6$. Since $P(T_{10}) > P(T_6)$, it discards the received global candidate.

At site 3, when scheduler receives $G_S(2)$, it discovers that it does not have its own global candidates, i.e., it has not begun deadlock detection[2]. The scheduler at site 3 executes step 3 to build up global candidate(s) using $last(G(2)) = T_7$ as the initiating transaction. The global candidate formed is $(T_7', T_5, T_1)$.

Now, using step 2 of part II, the scheduler concatenates the received global candidate $G_S(2)$ with $(T_7', T_5, T_1)$ to form the propagation global candidate $G(3) = (T_4', T_9, T_{10}, T_7, T_5, T_1)$. The PGC is sent to site 1 because $T_1$ waits on site 1. When the scheduler at site 1 receives $G(3)$, it executes part II of the algorithm and detects the deadlock $(T_4', T_9, T_{10}, T_7, T_5, T_1, T_6, T_2, T_3, T_4)$. It declares $T_1$ as the deadlock victim because it has the lowest priority. Note that this information is immediately available since the entire string is available to the deadlock detection algorithm.

## 3.2 Example 2

The previous example does not illustrate the possibility of transactions waiting transitively on others that form a cycle. This general condition is illustrated by Figure 6. For example, if $T_4$ initiates deadlock detection at site 2, it will create global candidate $(T_5', T_4)$ and transmit it to site 3. Site 3 will create its own global candidate and concatenate it with the one transmitted

[2]Note that this assumption is made to illustrate Part II - step 3 of the algorithm. If the scheduler at site 3 had a global candidate before it received $G_S(2)$, it would follow Part II - step 2 as done by the scheduler at site 2 when it received $G_S(1)$.

from site 2 resulting in a PGC $(T_5', T_4, T_7)$. This process continues at site 4 and site 5. After site 5 concatenates its own global candidate, the global candidate will contain the following transactions, $(T_5', T_4, T_7, T_2, T_1)$. Site 5 then transmits this latest PGC to site 3. Site 3 then computes $(T_1', T_7)$ and concatenates it with the received PGC to form $(T_5', T_4, T_7, T_2, T_1, T_7)$. At this point site 3 finds the deadlock which is denoted by a substring $(T_7, T_2, T_1, T_7)$ of the final PGC.

One should observe that if the priorities of transactions $T_5$ and $T_7$ are switched, then the algorithm will not detect the deadlock in the way described above but it will of course be detected by an initiation by one of the transactions belonging to the deadlock cycle. The algorithm requires that the highest priority transaction be a member of the deadlock cycle in order to guarantee the uniqueness of the detection of a deadlock.

## 3.3 Formal Properties of the Algorithm

We are interested in the formal properties exhibited by this algorithm as well as its performance. Ideally, one desires a distributed deadlock and resolution algorithm that detects and resolves a deadlock that exists at the time that the transaction chosen for abortion is actually aborted. However it is difficult to develop an algorithm that exhibits this property. For example, a deadlock cycle may be broken by a transaction that aborts for reasons other than to resolve the deadlock (a user at a terminal terminating his session). Another example would be of a transaction chosen as a deadlock victim for a specific cycle which also resolves a second deadlock cycle. This second cycle may have chosen a different transaction as a victim unnecessarily.

In this section we show that in the SET algorithm described above, a deadlock cycle will be uniquely detected and resolved by one of the sites present in the cycle provided that no transaction aborts voluntarily or as a victim chosen by some other overlapping deadlock cycle. Before we formalize this result, we introduce the following terminology.

**Definition 10** *A simple deadlock cycle is a cycle whose constituent transactions only belong to one deadlock cycle.*

**Definition 11** *Two deadlock cycles $C_1$ and $C_2$ are nested if $C_1 \cap C_2 \neq \emptyset$ and $C_1 \neq C_2$.*

**Definition 12** *A transaction abort is said to be voluntary if the transaction is aborted for any reason except to resolve a deadlock.*

**Definition 13** *A transaction performs an incidental abort with respect to a distributed deadlock cycle if it aborts as a result of being chosen as a deadlock victim for some other deadlock cycle.*

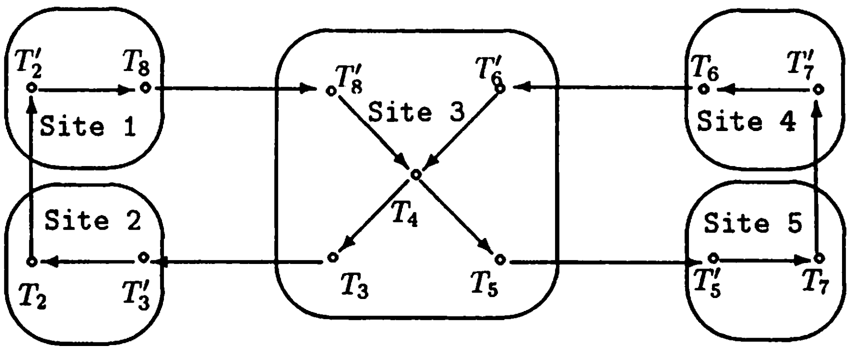

Figure 7: An Incidental Abort.

Figure 7 illustrates the concept of an incidental abort. This Figure contains two nested deadlock cycles $(T_4, T_3, T_2, T_8, T_4)$ and $(T_4, T_5, T_7, T_6, T_4)$. It is possible that the second cycle will be detected first. If it is, then $T_4$ will be chosen as the victim. In the mean time, the algorithm may have been initiated with respect to the first cycle and be past $T_4$ and, by the time it is detected, $T_4$ may be aborted. In this case, $T_4$ performs an incidental abort with respect to the first cycle. This example also illustrates the difficulties involved in proving the correctness of distributed algorithms due to the dynamic behavior of the distributed graphs and timing of the event occurrences.

**Theorem 1** *If there exists a simple distributed deadlock cycle and there are no voluntary aborts and no incidental aborts (with respect to that cycle) then it will be uniquely detected and resolved by one of the sites involved in the cycle.*

**Proof:** Let there exist a deadlock involving global candidates $G_1, G_2, \cdots, G_n$, each formed at a single site. Let the sites involved in the deadlock be $N_1, N_2, \cdots, N_k$.
**Observation:** $k \leq n$ since each global candidate belongs to only one site but one site may contribute more than one global candidate to the deadlock cycle.

We introduce the notion of virtual sites such that each virtual site contributes exactly one global candidate to the deadlock cycle. Therefore, there are $n$ virtual sites $VS_i$, $1 \leq i \leq n$. Note that two or more virtual sites may correspond to the same physical site. Let $VS_i$ correspond to global candidate $G_i$, $1 \leq i \leq n$. We observe that in the set $VS_1, \cdots, VS_n$, no two consecutive virtual sites may correspond to the same physical site because otherwise, the corresponding global candidates would have been collapsed into one global candidate. The above representation helps in distinguishing two or more global candidates from the same site.

**Definition 14** *Let $G$ be a directed graph whose nodes represent the virtual sites that are part of a deadlock and whose directed edges are the wait-for edges of the deadlock then a virtual site $VS_i$ is said to be* closer *than virtual site $VS_j$ to virtual site $VS_k$ iff*

1. *Virtual sites $VS_i, VS_j$ and $VS_k$ are common to the same distributed deadlock and,*

2. *The path length from $VS_j$ to $VS_i$ (i.e., number of edges between $VS_j$ and $VS_i$ in $G$ in the direction of the wait-for edges) is less than the path length from $VS_j$ to $VS_k$ and, the path from $VS_j$ to $VS_i$ is contained in the path from $VS_j$ to $VS_k$.*

Let $C$ denote the deadlock cycle involving virtual sites $VS_1, VS_2, \cdots, VS_{h-1}, VS_h, VS_{h+1}, \cdots, VS_n$ where $VS_h$ denotes the site which contains the highest priority transaction in the cycle. $VS_{h-1}$ and $VS_{h+1}$ are the predecessor and successor sites of $VS_h$ with respect to $C$ and in the direction of wait-for edges. Let initiation of the algorithm by site $VS_i$ detect the deadlock cycle. We consider two cases separately.

**Case 1:** $VS_i = VS_h$
Let $VS_h$ initiate the algorithm at time $t_0$. $VS_h$ will only propagate a global candidate(s) $G_{Sh}$ to $VS_{h+1}$ (from part I of the algorithm). $G_{Sh}$ is of the form $T_i, \cdots, T_j$ such that $T_i \in VS_{h-1}$ and $T_j \in VS_h$ and, $T_j$ waits on $VS_{h+1}$. At time $t_1 (t_0 < t_1)$, $VS_{h+1}$ will receive $G_{Sh}$. From part II of the algorithm, $VS_h$ discards any global candidate(s) it receives which are concatenable with its own (and sent) because $VS_h$ has the highest priority transaction in the cycle C (until a transaction times out again) and has initiated the algorithm itself. From the assumption (Section 2) that messages arrive correctly and in order, $VS_{h-1}$ would eventually receive $G_{Sh}$ and any other global candidates concatenated to it along the way, i.e., global candidates of sites $VS_{h+1}, \cdots, VS_n, VS_1, \cdots, VS_{h-2}$. Note that the assumption is that the deadlock exists and therefore, all the global candidates along the way must exist. It is necessary that they exist at the time global candidates from the previous sites arrive because if they do not, from Part II of the algorithm they will be computed and then will be concatenated to the received global candidates. Therefore, the deadlock cycle will be detected and resolved at site $VS_{h-1}$. More correctly, it is detected and resolved by the physical site corresponding to $VS_{h-1}$.
**Case 2:** $VS_i \neq VS_h$
Let $VS_{i-1}$ and $VS_{i+1}$ be the predecessor and successor sites to $VS_i$ in $C$, respectively. Further, let $\mathcal{V} = \{VS_{i+1}, VS_{i+2}, \cdots, VS_{h-1}, VS_h\}$ be a set of sites which are closer to $VS_h$ than $VS_i$ with respect to $C$. Assume that $VS_i$ initiates the algorithm at time $t_0$, $VS_i$ sends its global candidate(s) $G_{Si}$ to site $VS_{i+1}$, which concatenates its own global candidate to $G_{Si}$ and forwards it and, eventually the global candidates reach $VS_h$, say at time $t_h$. During the time $t_0$ and $t_h$ no other site $VS_k \in \mathcal{V}$ would have initiated the algorithm because otherwise $VS_i$'s global candidate would have been rejected at $VS_h$. The reason is that any other global candidate sent by a site $VS_k \in \mathcal{V}$ would reach $VS_h$ earlier (messages arrive in order). Hence, the arrival of the global candidate due the initiation at site $VS_i$ prompts $VS_h$ to concatenate its own global candidate to the received one and forward it. Note that if $VS_h$ had already initiated the deadlock detection computation, this case will reduce to case 1. Eventually, the concatenated global candidate(s) would arrive at site $VS_{i-1}$, where the deadlock will be detected and resolved. We also observe

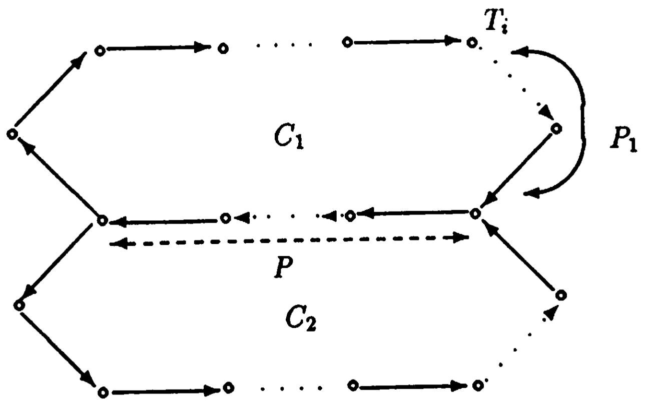

Figure 8: Nested Deadlocks.

that if the initiating site was not a part of the deadlock cycle but was waiting transitively on a site $VS_i$ which is a part of the deadlock cycle, the case reduces to either of the above two cases. This is because as far as the deadlock cycle is concerned, the initiation is done by one of the sites involved in the deadlock cycle no matter what prompts it.
**QED**

The above theorem proves the uniqueness of detection for simple cycles. Let us discuss the case when nested cycles exist. We will discuss the following questions. Can the same initiation detect all the nested cycles? Is it possible to guarantee unique detection and resolution when nested cycles exist? We will discuss the answers to these questions with the help of an example.

There are situations in which a single initiation will detect all of the nested cycles. Consider the following example. Let there be two deadlock cycles $C_1$ and $C_2$ which are nested together as shown in Figure 8. Let the intersection of the two cycles be a transaction string $P$. From the definition of nested cycles (Definition 11) two cycles are nested if there exists at least one transaction common to the two cycles and the two cycles are not identical. Therefore, the length of $P$ (Number of transactions in the string $P$) can be as small as one transaction (Figure 7). Assume that a transaction $T_i$ which is a part of $C_1$ but does not lie in the intersection of the two cycles, initiates the algorithm. When the deadlock computation reaches $first(P)$, it will produce a transaction string $P_1$. When the deadlock computation reaches $last(P)$, two transaction strings are concatenated to produce a transaction string $P_1||P$. Beyond $last(P)$ the computation splits into two paths: one along $C_1$ and the other along $C_2$. Obviously, cycle $C_1$ will be detected by this initiation because the initiator $T_i$ is a part of $C_1$. We now ask the question, will $C_2$ be detected? The cycle $C_2$ will be detected by this initiation if $high(P_1) < high(C_2)$. In other words, if the highest priority transaction belonging to $C_2$ has priority greater than the priorities of all the transactions that constitute $P_1$ then $C_2$ will also be detected with the same initiation. In summary we can state the following. *There are situations in which a single initiation will detect all the nested cycles. This situation occurs if for each cycle, the* *highest priority transaction of the cycle is also the highest priority transaction of the string that detects it.* In the above example, if $high(P_1) > high(C_2)$ then cycle $C_2$ will not be detected by $T_i'$s initiation because the highest priority transaction of the transaction string containing $C_2$ is not a part of $C_2$. Therefore, we can state the following. *There are situations in which a single initiation will not detect all the nested cycles. This situation occurs if for a cycle, the highest priority transaction of the cycle is not the highest priority transaction of the transaction string that contains the cycle.*

There are situations where deadlocks within nested cycles will be resolved without resolving false deadlocks. Again consider the above example. Let's assume that cycle $C_1$ is detected and resolved before cycle $C_2$. If the victim to resolve $C_1$ belongs to $P$ (i.e., the victim lies in the intersection of $C_1$ and $C_2$) then aborting the victim, in effect, resolves both the cycles. But this information may not be known when $C_2$ is resolved and therefore, an additional transaction may be aborted unnecessarily to resolve $C_2$. The same situation can occur if $C_2$ is detected and resolved before $C_1$. In the other case when the victim of $C_1$ is not in the intersection of the two cycles, resolving $C_1$ does not resolve $C_2$. Hence, in order to resolve $C_2$ another transaction which is a part of $C_2$ must be aborted. Therefore, we can state the following. *There are situations in which resolving one cycle within a nested cycle may resolve more than one cycle. These situations occur when the victim of one cycle lies on the intersection of the nested cycles and, such situations can contribute to resolving false deadlocks. In other situations where the victim of one cycle is not in the intersection of the nested cycles, each cycle in the nested cycles will be detected and resolved uniquely.*

## 3.4 The Probe Algorithm

The original priority based probe algorithm was presented by Sinha and Natarajan in [26] based on work by Chandy, Misra and Haas [8, 9]. Sinha and Natarajan [26] 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 to detect deadlocks when shared and multiple lock requests are allowed. We showed in [11] that these algorithms contained many errors and deficiencies and we described a modified probe algorithm that corrected these errors. In order to better understand the performance results presented later, we give a brief description of that modified probe algorithm.

The probe algorithm consists of two parts. The first part detects deadlock through the propagation of probes. Throughout this phase, transactions are required to store some of these probes in *probe_queues*. A probe is a pair, (initiator, junior), where initiator is the identity of the transaction which is blocked on a data item and the corresponding data manager initiates the probe. The junior is the identity of the current lowest priority transaction through which this probe has passed. The second part of the algorithm resolves 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. The contents of this clean message are used by transactions to delete probes from their probe_queues. Finally, at the end of this second phase, some transactions may be asked to reinitiate probes and/or retransmit probes. Additional details for each of the two phases on the probe algorithm are now given. We refer the reader to [11] for full details.

### 3.4.1 The Deadlock Detection Algorithm Based on Probes - Phase I

Each data item is managed by a data manager. A transaction makes a request to the data manager for a lock on a data item. Transactions as well as data managers participate in the algorithm.

Each data manager maintains a request_queue in which it stores the identities of all the transactions waiting for the data item that it manages. A data manager performs the following tasks.

- It schedules lock requests for its data item.
- It initiates a probe whenever a transaction requests a data item which is already locked by a transaction having a lower priority than the requester.
- If a transaction releases the lock on a data item and there are one or more transactions waiting for a lock on the data item, the data manager gives the data item to one of the waiting transactions and reinitiates probes provided the priority criterion is satisfied.
- It declares a deadlock whenever it receives a probe (initiator, junior) such that initiator holds a data item managed by the data manager, it declares a deadlock.

Each transaction maintains a probe_queue in which it stores all the received probes. If a transaction is waiting then it forwards all the received probes to the data manager where it is waiting. It becomes the junior of a received probe if its priority is less than that of the junior of the received probe. When a transaction enters a wait state because it is blocked, it transmits a copy of all the probes stored in its probe_queues to the data manager where it waits. If a data manager requests a copy of a transaction's probe_queue, the transaction sends its probe_queue to the data manager.

### 3.4.2 Resolution and Post Resolution Computation Phase - II

When a data manager detects a deadlock, it sends an abort signal (victim, initiator) to the victim. In order to clean all the probe_queues of probes containing the victim, the victim sends a clean(victim, initiator) to the data manager where it is waiting. The victim enters the abort phase and withdraws its pending request when the clean message returns to itself, i.e., after all the probe_queues are cleaned.

All transactions that receive the clean message remove probes containing the victim from their probe_queues. When a data manager receives a clean message, it propagates the clean message to its holder; it reinitiates a probe for each request for which the requesters priority is greater than that of the holder; and it requests a copy of the probe_queue from each transaction in its request_queue.

## 4 Experimental Results

In this section we describe a simulation study that evaluates the performance of a distributed database system under both the SET and probe distributed deadlock detection algorithms.

### 4.1 The Environment

There are different approaches to modeling a distributed database system in order to evaluate the performance of different deadlock detection and resolution algorithms. One approach is to model a distributed database system as a network of active resources such as processors and I/O subsystems, and a pool of passive resources representing the data items in the database [1, 7, 25]. In this approach, each primitive operation on the database, e.g., reads and writes, or the overhead of concurrency control, index management etc. is modeled as a consumer of certain amount of processor and I/O processing time. Therefore, contention between transactions for active as well as passive resources is captured in the performance results. However, in such an approach it is very difficult to separate the effects of contention due to active and passive resources. Furthermore, the results are very dependent on the performance characteristics of the underlying system and can not be easily generalized.

We have taken a second approach where we ignore the effects of contention for the physical resources and focus instead on the effects of contention on the data items. Such an approach, for example, has been effectively used in [13] to compare the performance of various concurrency control algorithms. In our simulation model we simulate a distributed database system consisting of $M$ processors where each processor has sufficient processing power available so that there is no conflict between transactions due to active resource requirements.

Figure 9 shows the "life-cycle" of a transaction in the system. A transaction follows this state diagram. The *deadlock-detection* state is not a true transaction state but signifies that deadlock detection is being performed on behalf of a transaction. Briefly, the life-cycle of a transaction can be described as follows. When a transaction is initiated at a processor, it enters the *active* state. While active, the transaction can request resources local to that processor or at some other processor. If the request is granted, the transaction remains in active state (remote active if the resources were elsewhere), otherwise it enters the *wait* state. After the transaction has waited for a sufficiently long time (determined by the time-out period) then deadlock detection is performed. Once the transaction acquires all of the resources it requires, it enters the *commit* state. If a deadlock is discovered and the transaction is chosen as the deadlock victim, it enters the *abort* state. An aborted transaction is restarted later. The time that a transaction remains in a particular state is a random variable. In the case of the active, remote active, new transaction, and commit states, these times were taken to be uniformly distributed random variables between 0 and 2 with an average of 1 unit time. This time signifies the computations performed by transactions between resource requests by transactions, or commit processing, or initial computation in the new transaction state. Zero time between requests represent two successive lock requests without any computation between the requests. The amount of time spent in other states like local wait, remote wait etc. depends on how long a transaction is blocked due to conflict. That is, the time spent in other states depends on the interactions between transactions, load, system state, system throughput, concurrency level, and other system parameters.

The two deadlock detection and resolution algorithms require that transactions be assigned unique priorities. This was accomplished by assigning each transaction a priority equal to its time of initiation (timestamp). Restarted transactions retained their original timestamp. Thus higher priority was given to old transactions.

## 4.2 Input Parameters and Performance Measures

Table 1 summarizes the input parameters. The number of sites in the distributed database, $M$ was fixed at 5 in our simulations. The number of data items, $G$, was fixed at 5000 throughout all of the simulations. A fixed number of transactions, $N$, divided equally between the $M$ sites execute concurrently. Each transaction requests exactly $R$ data items during its execution where $R$ is a uniformly distributed random variable with mean $E[R] \geq 16$ and range $(E[R] - 14, E[R] + 14)$. Each time a transaction requests a data item, it requests it from the local site with probability $p_l$ and from any remote site with probability $(1 - p_l)/(M - 1)$. For example, if $p_l$ is .6 then 60% of the requests will be to the local site and 10% to each remote site if there are a total of 5 sites. Last, data items within a site are chosen with equal probability.

The following performance measures are obtained from the simulation (Table 2). First, we are interested in $T$, the rate at which transactions commit. The deadlock detection and resolution algorithms require both communication and processing resources. As our measure of the communication requirements, we recorded, $N_m$, the total number of messages generated by the deadlock detection and resolution algorithm. In order to estimate the processing overhead, we actually measured the CPU time required to execute the algorithms during the simulation. Although the actual times may vary from machine to machine (we used DEC MicroVaxes) they can be used to compare the efficiencies of the two algorithms. We actually report $O$, the processing

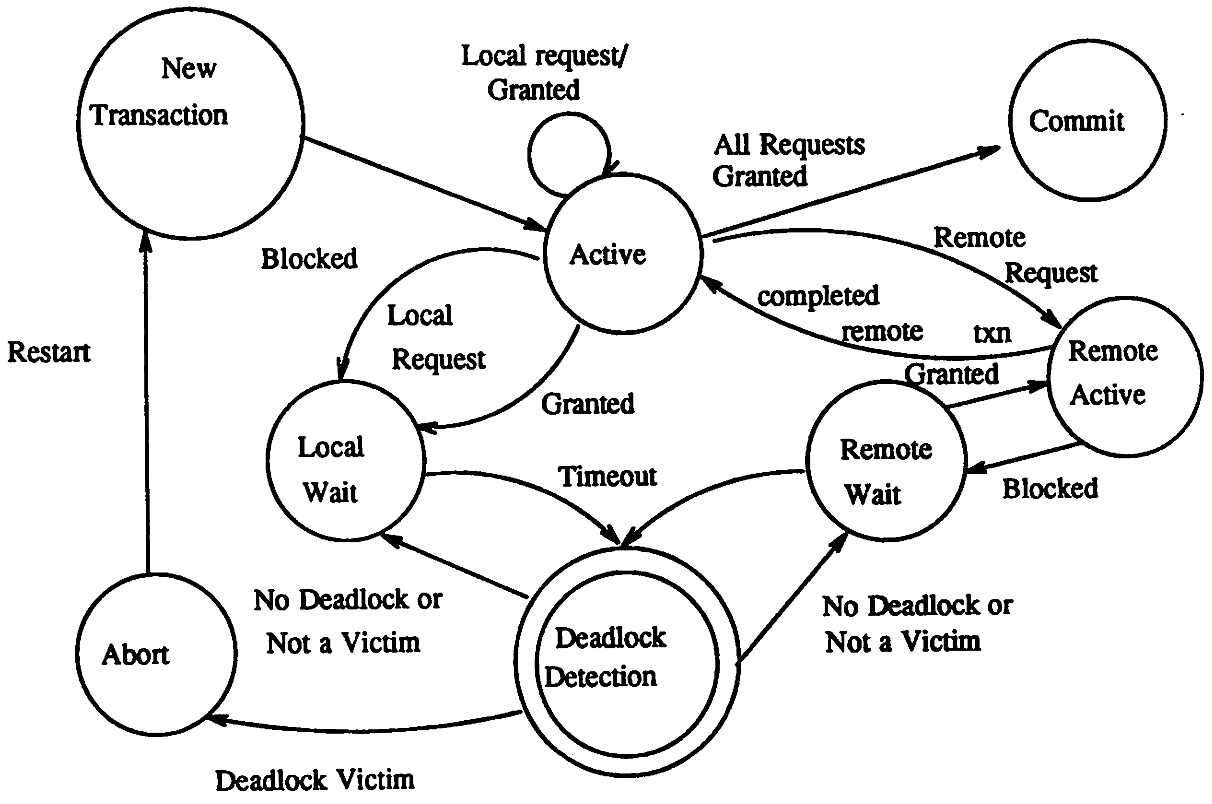

Figure 9: Life-Cycle of a Transaction.

| Parameter | Description | Values |
|---|---|---|
| $M$ | No. of Sites in DDBMS | 5 |
| $G$ | No. of Data Granules | 5000 |
| $N$ | Avg. no. of Txns.D in the System | Range 5 to 200 |
| $R$ | No. of Lock Requests by a Txn. | Uniform $(E[R] - 14, E[R] + 14)$ |
| $p_l$ | Likelihood of a Lock request being local | Range 50% to 95% |
| Access Distribution | Access distribution within a site of data | Uniform |
| Restart Request Pattern | Pattern of Request by a txn after Restart | Same, Different |

Table 1: Input Parameters

requirements per committed transaction.

We measured $P_c$ and $P_d$, the probabilities that requests for data items will have to wait or result in a deadlock. In addition, we measured the average number of transactions in a deadlock cycle, $E[D_l]$. Here $D_l$ is the length of a deadlock cycle.

Last, confidence intervals were estimated for $N_m$, $P_c$, and $P_d$ using the independent replication method. Each experiment was performed at least 10 times, and in each exeriment the stopping criteria was to commit at least 2000 transactions. Therefore, for each experiment 20,000 or more transactions were committed. After performing some initial tests it was discovered that the simulation results exhibited stationary behavior with 2000 transactions committed. Note that for each experiment at least 2000 transactions were committed.

### 4.3 Results

In this section we present performance results obtained from various experiments. We obtained two types of performance measures; deadlock detection algorithm dependent performance measures such as number of messages, computation overhead of deadlock detection etc., and algorithm independent measures such as probability of conflict. Note that probability of conflict is the likelihood of a resource request being blocked and it depends on the system parameters such as concurrency level, transaction size, number of data items and access patterns.

The results reported in this section include the following experiments: overhead of deadlock detection as a function of transaction size, concurrency level, and number of transactions committed with different resource request patterns.

| Parameter | Meaning |
|---|---|
| $N_m$ | No. of Messages used in Deadlock Detection |
| $O$ | Processing Time/Committed txn for deadlock detection |
| $E[D_l]$ | Average Deadlock Length |
| $P_c$ | Probability that a data request will result in a conflict |
| $P_d$ | Probability that a data request results in a deadlock |
| $T$ | No. of Txns Committed/Time |

Table 2: Performance Measures

Figure 10, Figure 12 and Figure 14 show the behavior of different performance measures for the two algorithms as a function of, $N$, the concurrency level. Each transaction generated $R$ requests (between 2 and 30) and each request was likely to go to the local site with probability 1/2 ($p_l = 0.5$), and to any other site with probability 1/8. Figure 11, Figure 13 and Figure 15 show the behavior as a function of average transaction size, $E[R]$. In these experiments, the concurrency level is fixed at $N = 50$ and requests are directed to any site with equal probability.

Figure 10 shows the CPU overhead per committed transaction, $O$, of deadlock detection for both algorithms. We observe little difference when the concurrency level, $N$, is 40 or less. However, the SET algorithm performs better than the probe algorithm when the concurrency level is high. This is because there is enormous overhead required to maintain the probe_queues required by the probe algorithm and, because the probe algorithm is initiated, not only to initiate the deadlock detection, but to also forward probes. Furthermore, when the concurrency level is high, more conflicts and consequently more deadlocks occur. Whenever a deadlock is resolved, probe algorithm requires that the probe_queues of all transactions involved in the deadlock as well as of all the transactions waiting transitively be cleaned and reorganized. This requires significant CPU time.

Figure 11 shows the CPU overhead for a fixed concurrency level $N = 50$ when we vary the average transaction size $E[R]$. There is little difference in the processing requirements of the two algorithms. In both cases the overhead per committed transaction is an increasing function of the transaction size. The disparity between Figure 10 and Figure 11 is due to the fact that the algorithms are more sensitive to the number of transactions in the system than to the transaction size. The reasons are as follows. Suppose there are $N$ transactions and the average transaction size is $E[R]$. Keep the product $NE[R]$ constant. Assume that each transaction currently holds $E[R]/2$ locks (i.e., on an average a transaction holds one half of the required locks). Then a request is likely to collide with $(N-1)E[R]/2$ locks. As $N$ increases and $E[R]$ decreases (while

| | | |
|---|---|---|
| Number of deadlocks ($N_{dl}$) | = | 87 |
| $N_m$ | = | 11250 |
| $E[D_l]$ | = | 4.6 |
| $var(D_l)$ | = | .1444 |

Table 3:

keeping the product $NE[R]$ constant), there are more locks to collide with because a request can only collide with locks held by other transactions. When $E[R]$ increases and $N$ decreases, there are fewer locks to collide with because the requesting transaction itself holds more locks. Furthermore, since a transaction can not request a resource after being blocked, there are fewer transactions to request locks when $N$ is small but more transactions to request locks when $N$ is large. Also, the number of probe queues is an increasing function of $N$ but not of $E[R]$. Therefore, the overhead of maintaining the queues is an increasing function of $N$. Last, the overhead due to the initiation and forwarding of probes is also an increasing function of $N$ but not of $E[R]$.

Figure 12 and Figure 13 provide similar results for $N_m$, the number of messages generated while performing deadlock detection and resolution. These figures also illustrate the 95% confidence intervals for $N_m$. The SET algorithm requires fewer messages than the probe algorithm. However, the message length in the probe algorithm is fixed whereas it varies in the SET algorithm and are generally longer.

As discussed earlier, most analytical results on the number of messages generated by a deadlock detection and resolution algorithm are worst case results based on the existence of a deadlock. We have observed that most messages that are generated do not result in the detection of a deadlock. We present the following example to illustrate this. Consider the "probe" algorithm. When there are 50 concurrent transactions, each of average size 16, we obtained the statistics shown in Table 3 after committing 2000 transactions.

Consider a deadlock containing $D_l$ transactions. The worst case number of messages occurs if they reside on different sites and is $D_l \times (D_l - 1)$, where $D_l$ is the number of transactions involved in the deadlock cycle [26, 11]. Applying this to our example yields,

$$\begin{aligned} Number\ of\ intersite\ messages &\leq N_{dl} \times E[D_l \times (D_l - 1)] \\ &= N_{dl}(var(D_l) + E^2[D_l]), \\ &= 1453. \end{aligned}$$

This value is an order of magnitude smaller than the number of messages actually transmitted during the simulation which is 11250 messages as shown in Table 3.

The reason is as follows: messages are sent out even if there is no deadlock. This occurs much more frequently than the transfer of messages when there is deadlock. In addition to the message overhead, this observation is true for processing overhead of deadlock detection. Thus analyses of the performance of deadlock detection and resolution algorithms should focus on the message overhead when there are no deadlock as well as when there are deadlocks. Moreover, one of the goals in designing an algorithm should be to minimize the overall number of messages even at the expense of increasing the overhead when a deadlock exists.

Figure 14 illustrates the behavior of $P_c$ and $P_d$ as a function of the concurrency level in the system. As mentioned earlier, these performance measures are independent of the deadlock detection algorithms because they depend on the system parameters. However, the results reported here were obtained when the probe algorithm was used for deadlock detection. We confirmed these results by using the SET algorithm as the deadlock detection algorithm. We observed that both $P_c$ and $P_d$ are largely independent of the underlying distributed deadlock detection algorithm because they represent transaction interactions and mostly depend on the system parameters. However, deadlock detection will have a small impact on their values because the longer a transaction remains in conflict or deadlocked, the higher the corresponding probabilities. We observe that initially both $P_c$ and $P_d$ increase rapidly as concurrency level increases, but level off for higher values. The reason is that even though the conflict rate is high, not many transaction are in active state and, therefore, the number of requests made is small. Hence, the rate of increase in $P_c$ and $P_d$ is lower. Figure 15 shows $P_c$ and $P_d$ as a function of average transaction size. Similar characteristics are obtained as in Figure 14 because as the average transaction size increases the conflict rate increases.

The last six figures ( Figure 16 to Figure 21 ) show the number of messages as a function of the number of transactions committed for various transaction request patterns and restart request patterns. Figure 16 shows the number of intersite messages as a function of transactions committed when $p_l = .5, .6$ The same restart request pattern used by a restarted transaction means that if a transaction is aborted because it was chosen as a deadlock victim, when restarted, it will request exactly the same resources that it did before aborting. We observe from Figure 16, the SET algorithm performs better than the "probe" algorithm.

Figure 17 shows the number of messages when a restarted transaction follows a different request pattern from what it did prior to aborting. This might correspond to the behavior of an air line reservation system. There may be several alternative flights between the source and the destination. If making a reservation(s) on a flight or air line fails due to a deadlock, the same transaction can restart and try other alternatives thereby requesting a different set of data items in the database. However, the performance improvement in this case is not significant because the conflict rate was not very high and therefore, the number of restarted transactions were small. It should be noted that a performance improvement in the latter case is expected because if a restarted transaction follows the same request pattern, it is likely to conflict again with the existing transactions in the system because the transactions remain in the system for some amount of time with which it had originally conflicted and was chosen as a deadlock victim in the first place.

Figure 18 and Figure 19 show the message overhead when $p_l = .7, .8$. Obviously the number of messages reduce for both algorithms as compared to the previous two figures because the number of remote request and therefore, the number of remote waits are less. However, the SET algorithm performs much better than the "probe" algorithm. For the case of $p_l = .7$, the number of messages sent by the probe algorithm is 3.5 times more than the SET algorithm and for the case of $p_l = .8$, the number of messages sent by the probe algorithm is 5 times larger than the SET algorithm. This can be explained as follows. The SET algorithm sends messages only when global candidates exist. Global candidates are less likely to exist as the number of remote requests decrease. However, in the "probe" algorithm, messages are sent even if no global candidates exist. As a result, the probe algorithm generates many more unnecessary messages than the SET algorithm. Last, we observe that global candidate strings will be small so that the length of messages will be comparable under each algorithm. Figure 20 and Figure 21 show the corresponding output when $p_l = .9, .95$. The SET algorithm clearly performs much better in terms of number of messages.

During the course of all the experiments, approximately two million transactions were committed. For each reported deadlock by any algorithm, we checked whether or not a deadlock really existed so that if an algorithm reported a false deadlock it was always noted. We observed that all the reported deadlocks really existed and therefore, we believe that this provides some empirical evidence as to the correctness of the algorithm. Additional discussion on these experiments as well as results from additional experiments are reported in [10].

## 5 Conclusions

In this paper we present optimizations to current SET-based deadlock detection algorithms. These optimizations are based on the notion of assigning priorities to transactions. In general, by using priorities, fewer messages need to be forwarded. We also formally prove that the SET-based algorithm uniquely detects a simple deadlock cycle. We also discuss those situations where a single initiation can detect all the nested cycles and conditions under which each cycle in nested cycle will be detected and resolved uniquely. We compare the performance of the SET algorithm with a probe algorithm. The performance results indicate that when the concurrency level is low, the two algorithms perform similarly; but when the concurrency level is high, then the SET-based algorithm outperforms the probe based algorithm. Further, we showed that analytical performance evaluation of distributed deadlock detection algorithms are quite optimistic because they only model the cost of the algorithm when deadlock exists. In fact, this cost is a small portion of the overall cost, i.e., the total overhead of a distributed deadlock detection algorithm is dominated by the case when deadlock detection is initiated, but no deadlock exists.

## References

[1] R. Agrawal, "Models for Studying Concurrency Control Performance: Alternatives and Implications," *ACM SIGMOD*, pp. 108-121, 1985.

[2] R. Agrawal, M. J. Carey, and L. W. McVoy, "The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," *Trans. on Soft. Engg.*, SE-13, No. 12, Dec. 87.

[3] D. G. Badal, "Distributed Deadlock Detection," *ACM Trans. on Computer Systems*, Nov. 86.

[4] G. Barcha and S. Toueg, "A Distributed Algorithm for Generalized Deadlock Detection," *ACM 3rd Proceedings on Distributed Computing*, pp. 285-301, Aug. 1984.

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

[6] P. Bernstein and N. Goodman, "A Sophisticate's Introduction to Distributed database Concurrency Control," *Proc. 8th International Conference on Very Large Databases*, Sept. 1982.

[7] R. Agrawal and M.J. Carey, "The Performance of Concurrency and Recovery Algorithms for Transactions-Oriented Database Systems," *Database Engineering*, Vol. 8, pp. 58-66, June 1985.

[8] K.M. Chandy and J.Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems," *Proc. of ACM SIGACT-SIGOPS Symp. on Principles of Dist. Compu.*, Ottawa, Canada, Aug. 1982.

[9] K.M. Chandy, J. Misra, L.M. Haas, "Distributed Deadlock Detection," *ACM Trans. on Computer Systems*, Vol. 1, pp. 144-156, May 1983.

[10] A.N. Choudhary, Two Distributed Deadlock Detection Algorithms and Their Performance. Master's Thesis, ECE Department, University of Massachusetts, Amherst, Feb. 1986.

[11] A.N. Choudhary, W.H. Kohler, J.A. Stankovic, D. Towsley, "A Modified Priority Probe Algorithm for Distributed Deadlock Detection and Resolution," *IEEE Trans. on Soft. Engg.*, Jan. 1989.

[12] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, pp 624-633, ACM, Nov. 1976

[13] P. Franaszek and J.T. Robinson, "Limitation of Concurrency in Transaction Processing," *ACM Trans. on Database Systems*, Vol. 10, pp. 1-28, March, 1985.

[14] V. Gligor and S.H. Shattuck, "On Deadlock Detection in Distributed Database Systems," *IEEE Trans. on Soft. Engg.*, Vol. SE-6, Sept. 1980.

[15] L.M. Haas and C. Mohan, "A Distributed Deadlock Detection Algorithm for Resource Based Systems," IBM Research Report RJ 3765, Jan. 1983.

[16] S. Katz and O. Shmueli, "Cooperative Distributed Algorithm for Dynamic Cycle Prevention," *IEEE Trans. on Soft. Engg.*, No. 5, Vol SE-13, May 87.

[17] H. T. Kung and J. T. Robinson, "Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems*, Vol. 6, pp. 213-226, June 1981.

[18] L. Lamport,"Time, Clocks and Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, pp 558-565, ACM, July 1978.

[19] J.E.B. Moss, "Nested transactions: An Approach to Reliable Distributed Computing," Ph.D. Thesis, Laboratory for Computer Science, MIT, Cambridge, MA, April 1981.

[20] D.A. Menasce and R.R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *IEEE Trans. on Soft. Engg.*, Vol. SE-5, No. 3, May 1979.

[21] D. P. Mitchell, M. J. Merritt, "Distributed Algorithm for Deadlock Detection and Resolution," *ACM 3rd Proceedings on Distributed Computing*, pp 282-284, ACM, Aug. 1984.

[22] N. Natarajan, "A Distributed Scheme for Detecting Communication Deadlocks," *IEEE Trans. on Soft. Engg.*, Vol. SE-12, No. 4, April 86.

[23] R. Obermarck, "Distributed Deadlock Detection Algorithm," *ACM Trans. on Database Systems*, Vol. 7, pp. 187-208, June 1982.

[24] M. Roesler, W.Q. Burkhard, K.B. Cooper, "Efficient Deadlock Resolution for Lock-Based Concurrency Control Schemes," *Proc. 8-th Conf. on Distr. Comp. Systems*, pp. 224-233, June 1988.

[25] K.C. Sevcik, "Comparison of Concurrency Algorithms Using Analytical Models," *Information Processing*, pp. 847-858, 1983.

[26] M.K. Sinha and N. Natarajan, "A Priority based Distributed Deadlock Detection Algorithm," *IEEE Trans. on Soft. Engg.*, Vol. SE-11, No. 1, pp. 67-80, Jan. 1985.

[27] A. Sekino, K. Moritani, T. Masai, T. Tasaki and K. Goto, "The DCS - A New Approach to Multisystem Data-Sharing," *Proc. National Computer Conference*, Las Vegas, NV, July 1984.

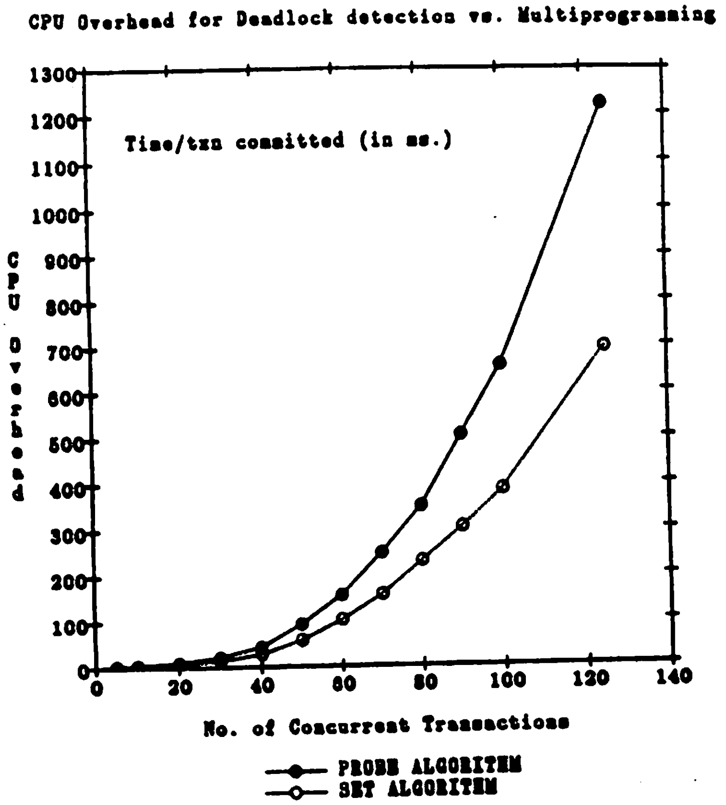

Figure 10: CPU Overhead vs. Concurrency Level

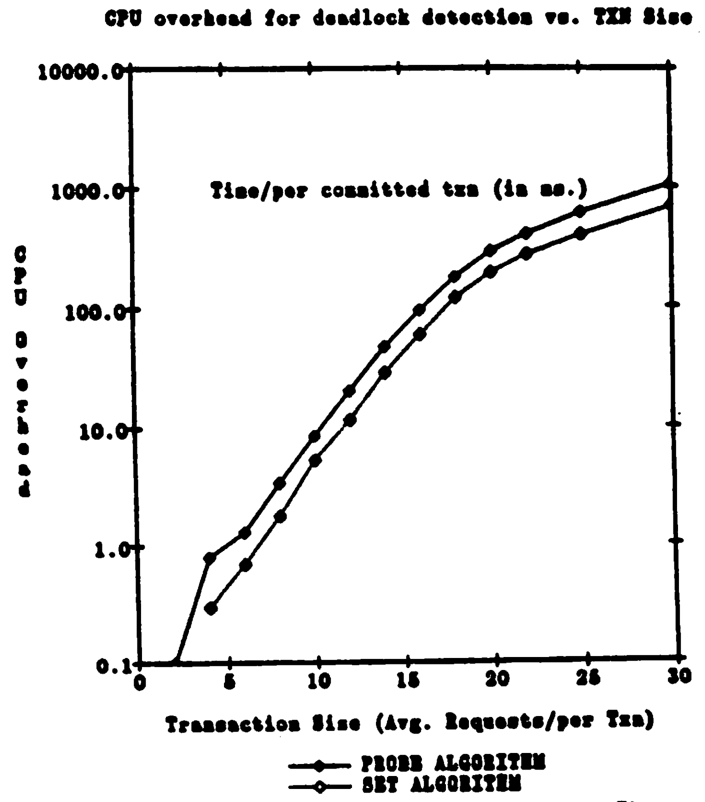

Figure 11: CPU Overhead vs. Avg. Txn. Size

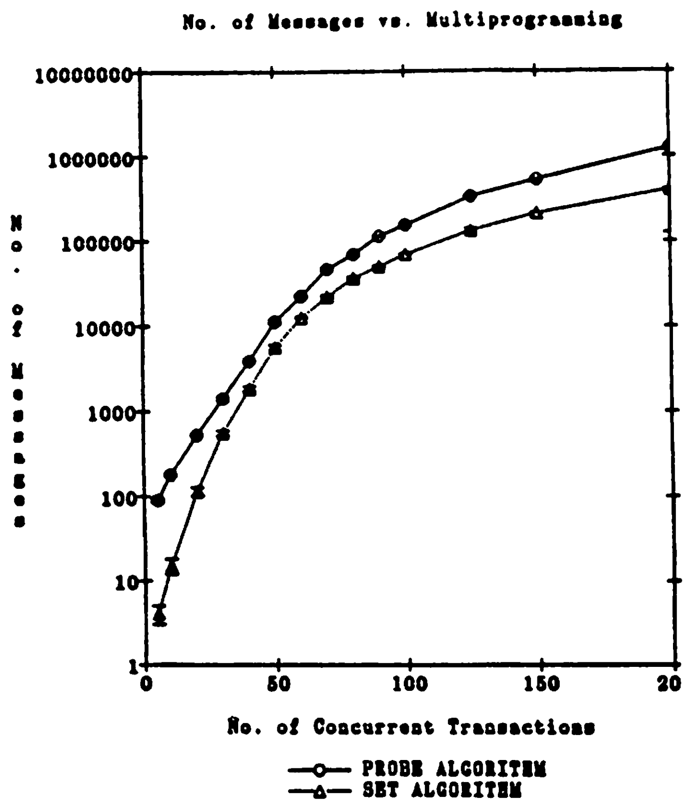

Figure 12: No. of Intersite Messages vs. Concurrency Level

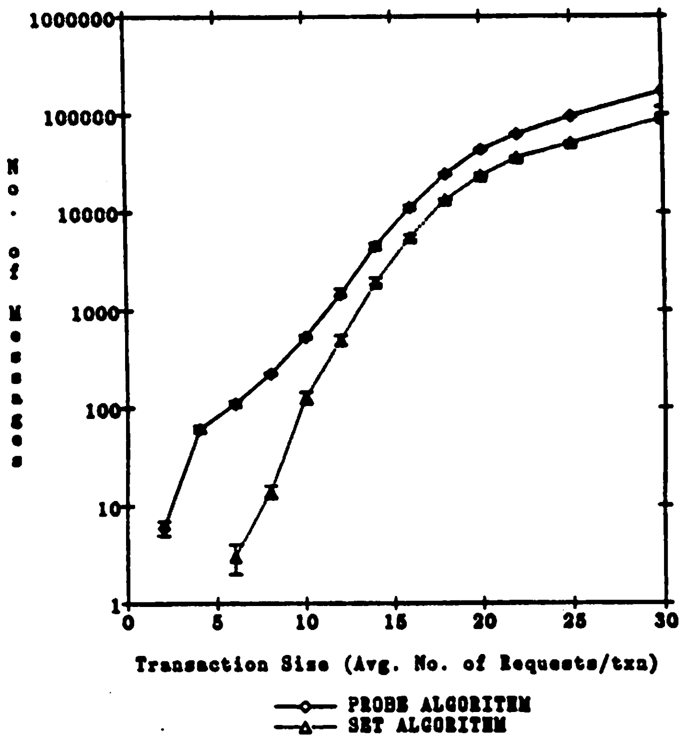

Figure 13: No. of Intersite Messages vs. Avg. Txn. Size

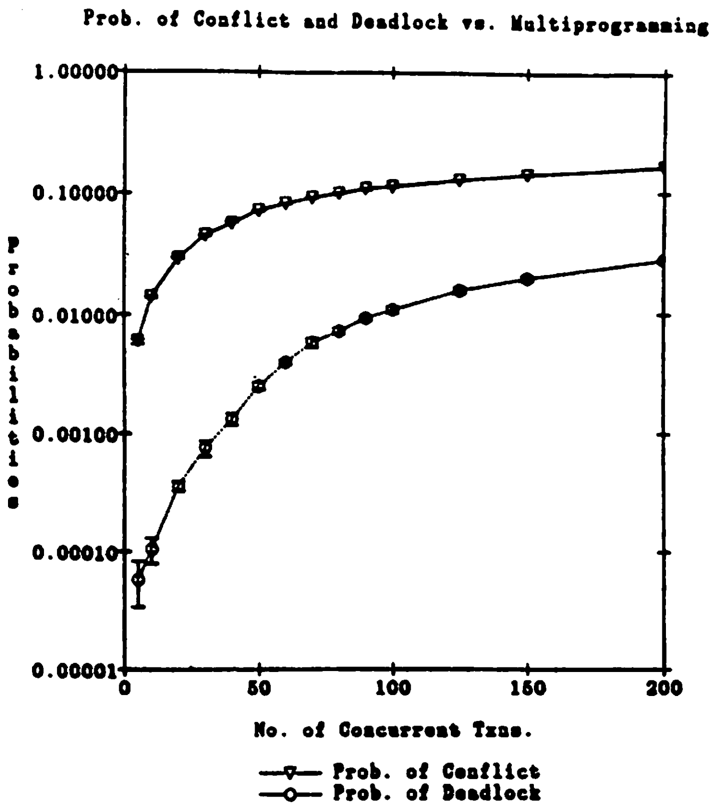

Figure 14: $P_c$ and $P_d$ vs. Concurrency Level

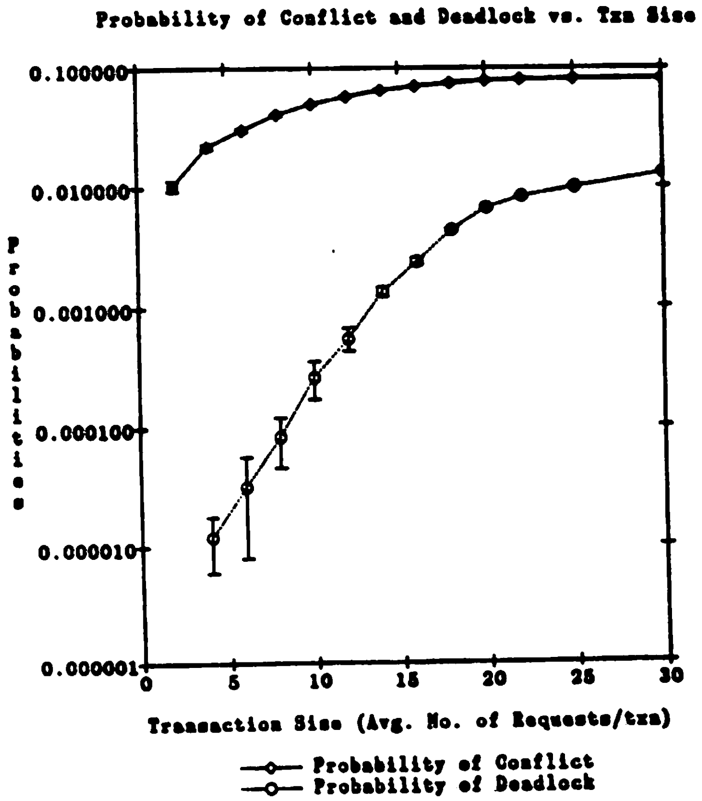

Figure 15: $P_c$ and $P_d$ vs. Avg. Txn. Size

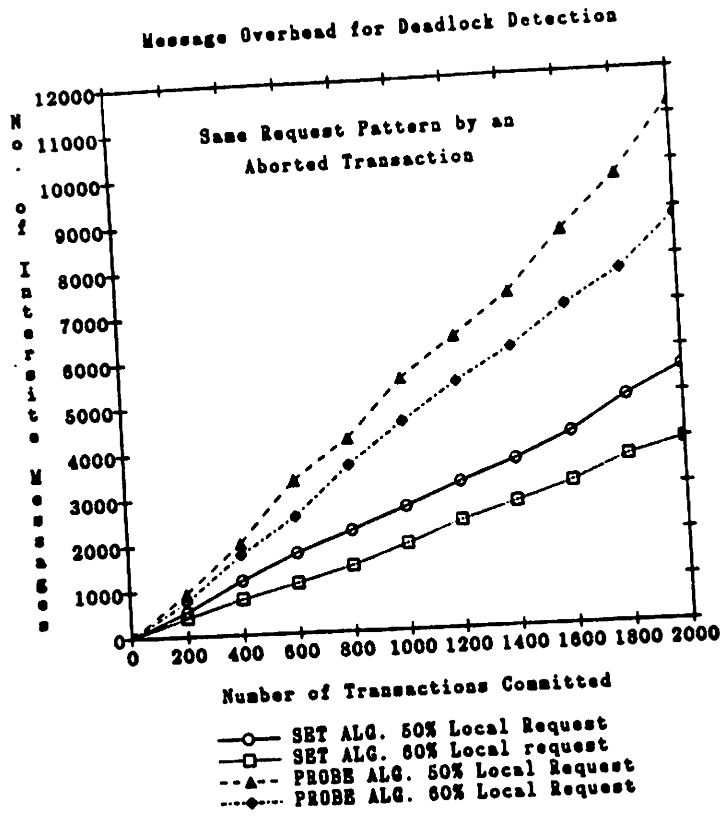

Figure 16: No. of Intersite Messages vs. Txn. Committed

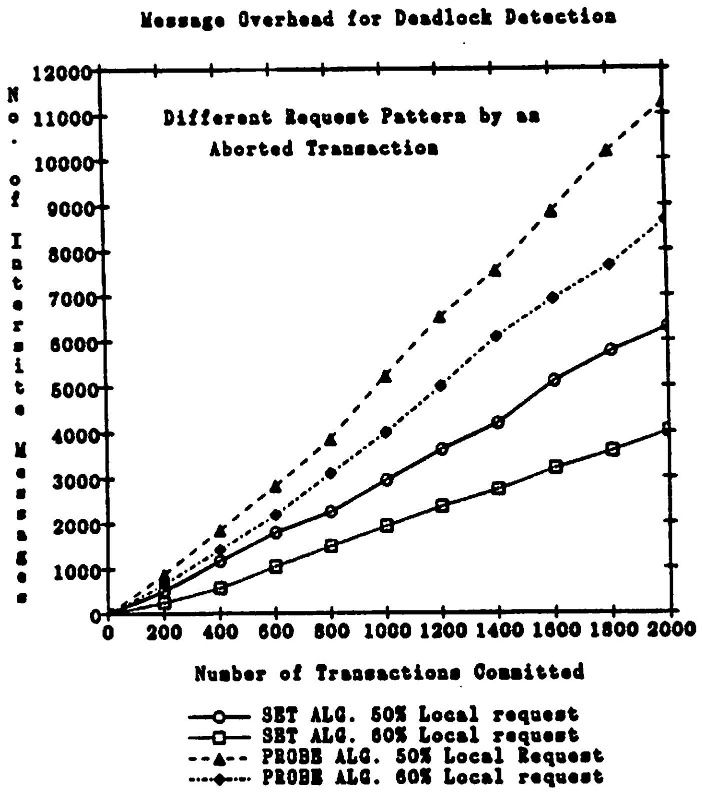

Figure 17: No. of Intersite Messages vs. Txn. Committed

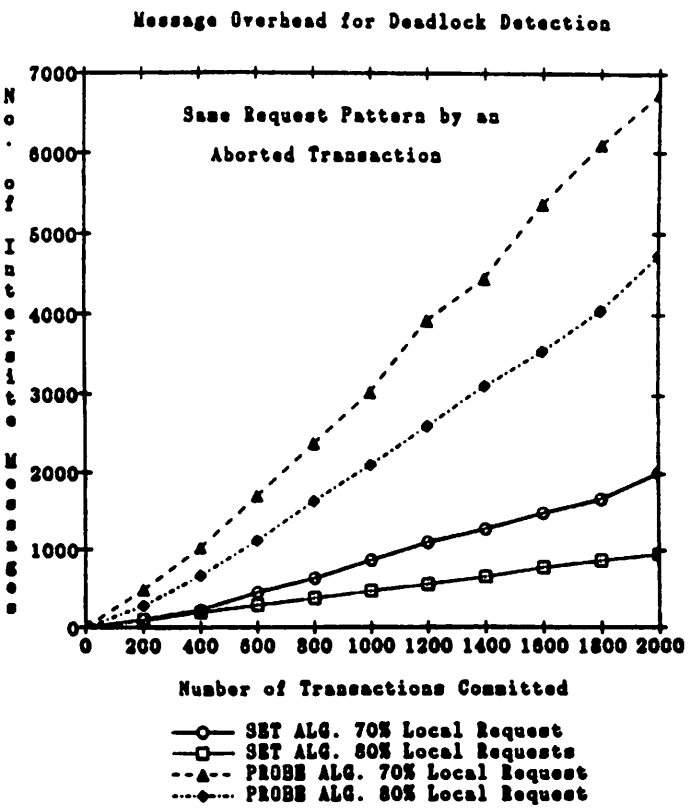

Figure 18: No. of Intersite Messages vs. Txn. Committed

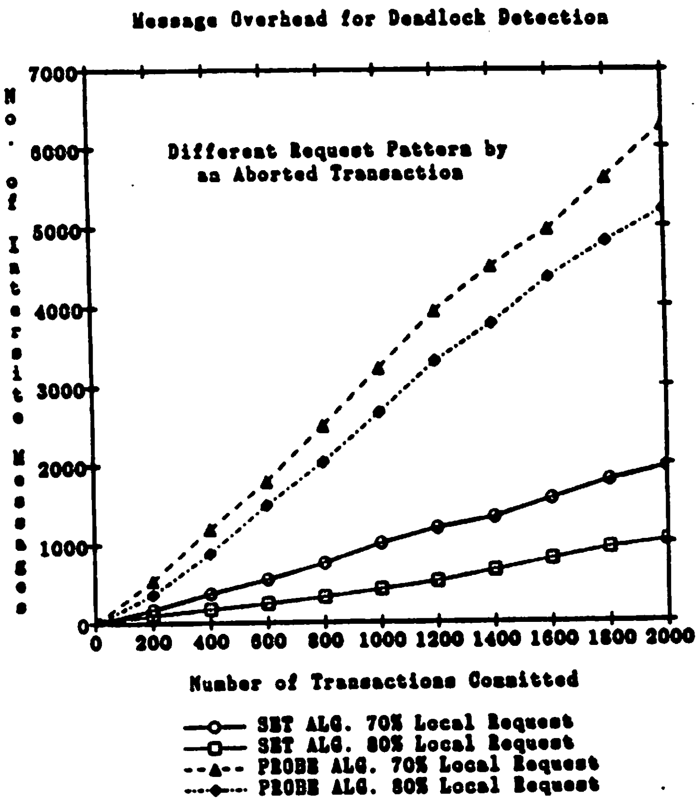

Figure 19: No. of Intersite Messages vs. Txn. Committed

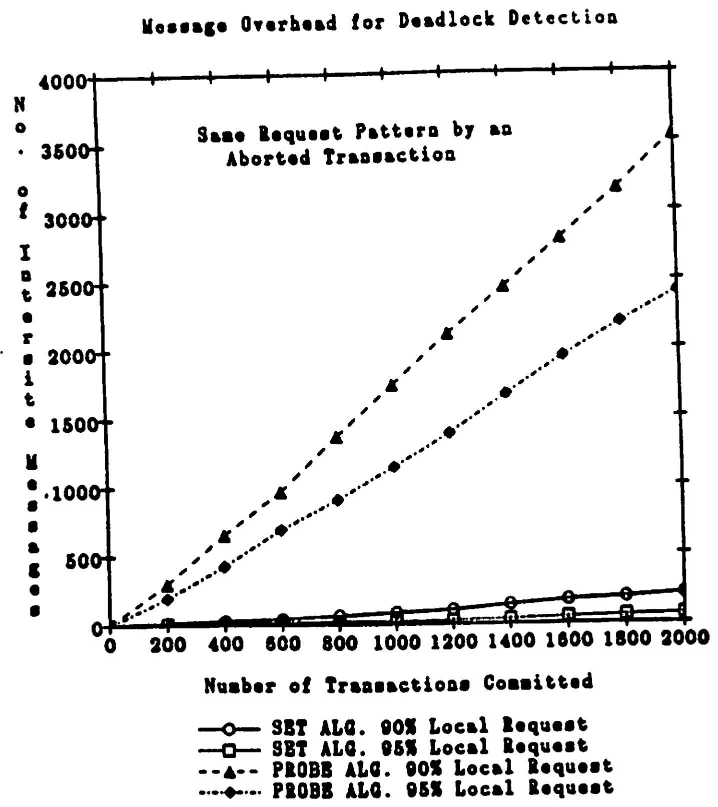

Figure 20: No. of Intersite Messages vs. Txn. Committed

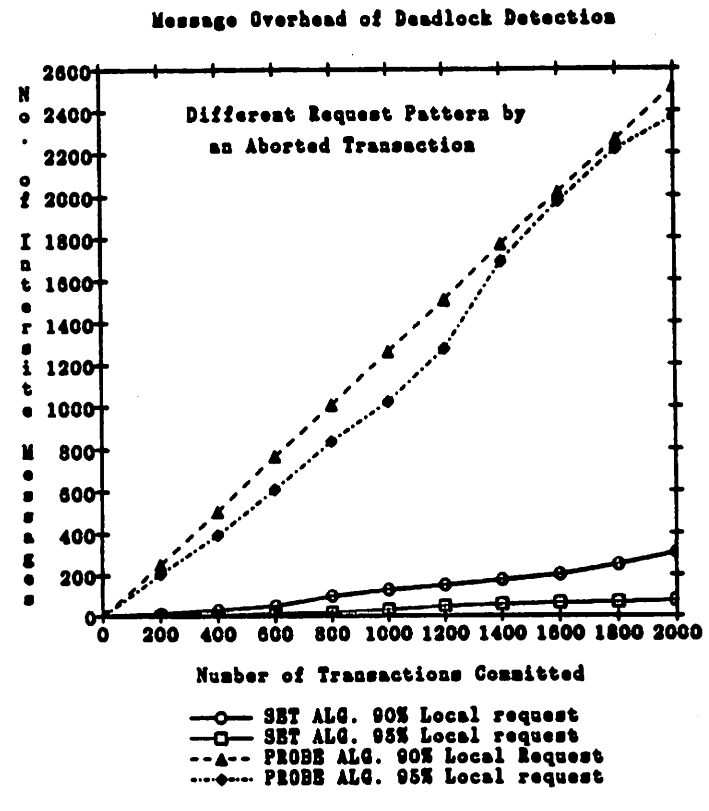

Figure 21: No. of Intersite Messages vs. Txn. Committed