공학 연구논문집 제24권 제1호 pp. 167~189, 울산대학교
Journal of Engineering Research Vol. 24, No. 1, pp. 167~189, 1993.
University of Ulsan.

Performance Analysis of Distributed Deadlock Detection and Resolution Algorithms based on a Hybrid Wait-for Graph and Probe Generation Scheme†

Young Chul Park* · Yang Soo Park* · Young Phil Cheung**
Dept. of Computer Science* · Computer Engineering**

Abstract

We present a continuous algorithm for deadlock detection and resolution in distributed database systems. our algorithm maintains at each local site an augmented transaction wait-for graph and uses a modified priority-based probe generation scheme in order to minimize the number of inter-site messages sent. The augmented transaction-wait for graph contains, in addition to lock-wait informatin, also information about message-wait relationships among the agents of a transaction, received probes and transitive wait-for relationships among transactions. Global deadlocks are declared whenerver a probe or transitive-wait relationship is propagated to an agent of the transaction that initiated the probe. In order to minimize the probability of false deadlocks, two extensions to the original algorithm are presented which make use of an additional validation procedure. We report on the results of simulation experiments that compare the performance of the three proposed schemes for deadlock detection and resolution.

대기 그래프와 프로브 생성에 기초한 교착상태의 분산 검출 및 회복 알고리즘의 성능분석†

박영철* · 박양수* · 정영필**
전자계산학과* · 컴퓨터 공학과**

† This paper was supported in part by FACULTY RESEARCH FUND, University of Ulsan, 1992.

요 약

본 논문은 분산 데이타베이스 시스템에서의 교착상태 검출 및 회복을 위한 연속 알고리즘을 제시한다. 이 알고리즘은 각 지역 사이트에 확장된 트랜잭션 대기 그래프를 유지하며 각 사이트간 메세지 전달의 수효를 최소화하기 위해 우선순위에 기초한 프로브 생성 기법을 변형하여 사용한다. 확장된 트랜잭션 대기 그래프는 로크-대기 정보 뿐만 아니라 트랜잭션 에이전트들 간의 메세지-대기 정보, 수신한 프로브들, 그리고 트랜잭션들 사이의 이행-대기 관계들에 관한 정보도 포함한다. 전역 교착상태는 어떤 프로브나 이행-대기 관계가 그 프로브를 개시한 트랜잭션의 어떤 에이전트로 전파될 때마다 선언된다. 교착상태의 거짓선언 확률을 최소화하기 위하여 제시한 알고리즘에 대하여 두가지 확장을 가했다. 이들은 교착상태 선언시 이의 정당성을 검증하는 부가적인 절차를 추가한 것이다. 세가지 제안한 방법의 성능을 시뮬레이션을 통하여 비교분석 하였다.

1. Introduction

A distributed database system consists of a collection of sites interconnected through a communication network, each of which maintains a local database system. Users interact with the system via transactions. A transaction is an execution of a program that accesses a shared database. Each site is able to process local transactions, which access data only in the single site. In addition, a site may participate in the execution of global transactions, which access data in several sites. Execution of global transactions requires communications among sites.

A deadlock occurs when a set of transactions are circularly waiting for each other to release resources. When the circular wait occurs at a single site only it is referred to as a local deadlock, while a circular wait involving transactions executing at multiple sites is called a global deadlock. The difficulty in dealing with deadlocks in a distributed database system is due to detection of global deadlocks. Global deadlocks in the system access only resources local to their originating sites. Therefore, a distributed deadlock detection approach has been adopted most widely [2], [4], [7], [12], [17], [19], [21] instead of a centralized or hierarchical approach [5], [8], [11], [20].

In distributed deadlock detection, global deadlocks are detected by sending inter-site deadlock detection messages. Detection schemes for global deadlocks can be classified into two categories depending upon the type of graph they construct, which is either an actual graph or a condensed graph. Schemes that construct an *actual graph*[3], [7], [12] are based upon transmission of detection messages, which convey string of transactions of an arbitrary length. In the string, the first transaction is a global transaction waited by some other site; each transaction - global or local - in the string waits for the completion of the immediately following transaction in the string; and the last transaction is a global transaction that is waiting for either a response or a new requset from another site. The number of messages sent can be reduced in half by assigning priorities to transactions and transmitting only those messages where the priority of the first transaction in the string is higher than the priority of the last transaction in the string[12]. In either case, the detection message is sent to the site where the last transaction in the string is waiting and used to construct an actual deadlock detection graph at that site. This scheme requires the trans-mission of a large number of messages, expecially when shared locks are permitted, and is not very practical when the number of transactions is large.

On the other hand, in schemes that are based on the construction of a *condensed graph*[6], [11], deadlock detection messages contain only two transactions, (T_i, T_j), where T_i transitively waits for the completion of T_j. The message is sent either to the site of origin of transaction T_i(backward trans-mission) or to the site of origin of transaction T_j(forward transmission). Backward trans-missions result in a very large number of messages transmitted when thate are no deadlocks in the system. In order to alleviate this problem, this method has been modified by making also use of a priority scheme such that a message, called a *probe*, is sent when a transaction with a higher priority transitively waits for another transaction with a lower prority [4], [14], [17], [19]. While priority based probe schemes avoid the transmission of backward messages, they also have a problem of sending messages to transaction managers and also to resource managers. This problem causes the number of messages to be doubled compared with schemes that do not send messages to resource managers. However, an interesting feature of the modified probe scheme is that once a probe is received it is stored and forwarded until no more paths are found for delivering the probe or a compensating message for the probe, called an *antiprobe*, is received. This feature eliminates the need to retransmit the same probe a number of times and speeds up the detection and resolution of future deadlocks.

In this paper we introduce a new deadlock detection and resolution scheme that reduces significantly the overall number of deadlock detection messages sent. Our scheme uses a hybrid combination of a transaction wait-for graph (TWFG) and a probe generation mechanism. A local TWFG is maintained at each site to detect local deadlocks and to avoid the inter-transaction messages. Probes, each of which represents the fact that a global transaction with a higher priority transitively waits for another global transaction with a lower priority, are sent to remote sites to construct condensed graphs. Global deadlocks are detected by using the local TWFG and the probes received. In order to compensate for the probes that have been sent already antiprobes are sent.

The rest of this paper is organized as follows. In Seciton 2 our new distributed deadlock detection and resolution algorithm is presented. Section 3 presents two validation algorithms for the one given in Section 2. These two algorithms will reduce the chance of false deadlocks. In order to evaluate the performance of our proposed algorithms, intensive simulations have been conducted. Section 4 presents these simulation results in detail. Concluding remarks are given in Section 5.

2. A Distributed Deadlock Detection and Resolution Algorithm

Transaction operations are carried out by processes generated at each site. When a process needs a resource at another site, another process is created at that remote site to represent the parent process at that site. Each child process will carry the transaction identifier of its parent process. Processes of a transaction are called *agents or cohorts* of the transaction and they constitute a tree structure that is denoted as a *process tree* [9], [10]. Because multiple agents of the same transaction may exist on one site, we define a process P_x of transaction T_i at site S_r to be $T_iP_xS_r$ for clarity. All the agents of a transaction can run in parallel. However when a transaction has multiple agents running of its behalf at the same site, we assume that only one of them is executable by blocking the others as in [9], [22]. For the rest of this paper, let the transaction identifier of an agent W be TID(W). For example, TID $(T_iP_xS_r)$ is T_i.

Each transaction has a globally unique identifier that consists of two fields: one denoting the site at which the transaction is originated and the other containing the value of the olcal clock at that site when the transaction sas generated. Based on this, a unique priority can be assigned to each transaction as follows. For two transactions T_i and T_j, T_i hs *higher priority* than T_j(or T_i is *younger* than T_j), denoted as $T_i > T_j$ if either loacal clock of T_i is greater than that of T_j or local clock of T_i is equal to the local clock of T_j but the site identifier of T_i is greater than the site identifier of T_j. Throughout the rest of this paper, we assume that $T_i > T_j$ if i > j: for instance, $T_5 > T_3$.

Each site maintains a local TWFG, which is a directed graph whose nodes are transaction processes running at that site. Let us assume that N is the maximum number of transactions that are executable at site S_r. The local TWFG at site S_r, denoted as $TWFG_r$, is represented as an array [1...N] of records with the following fields: TID(transaction identifier), global (boolean), LWS(lock waits), MWS(message waits), TAWS(transitive-antagonistic waits), and PBS(probes). We assume that the network guarantees that messages are error-free and are able to arrive at their destination in finite time and in the same order as they were sent.

Definition 2.1 A process *waiter lock-waits* for another process *waitee*, denoted as LW (*waiter, waitee*), if *waiter* is waiting for *waitee* to release a resource needed by *waiter*.

Definition 2.2 A process *sender* of a transaction *message-waits* for another process *receiver* of the transaction, denoted a MW (*sender, receiver*), if *sender* is waiting to receive a message (either an answer or a new request) from *receiver*.

Definition 2.3 A transaction T_i is antagonistic with another transaction T_j if T_i is a global transaction and either $T_i > T_j$ or T_j is a local transaction.

Definition 2.4 A path from a transaction process $T_iP_xS_x$ to another transaction process $T_jP_yS_y$ in the global TWFG is an antagonistic path if T_i is antagonistic with every transaction on the path except for the first one.

When there exists an antagonistic path from an agent of T_i to another agent of T_j, we say that T_i waits for the agent of T_j *transitively and antagonistically*. A transitive-antagonistic wait (TAW) record, denoted a TAW(*initiator, terminus, count*), means that some processes of transaction *initiator* wait for a transaction process *terminus* transitively and antagonistically and *count* different incoming edges(lock-wait edges and/or message-wait edges) to terminus are related to antagonistic paths connecting some processes of initiator to terminus. Consider a partial TWFG presented in Figure 2.1. In the figure *a* through *h* represent transaction processes and directed edges represent either lock-wait edges or message-wait edges. Assume that transaction TID(*a*) is antagonistic with other transactions in the figure except for transaction TID(*e*). Note that e cannot be a terminus of any TAW record whose initiator is TID(*a*) because TID(*a*) is not antagonistic with TID(e). The count of TAW(TID(*a*), *f*, 2) is 2 since *f* has two incoming edges that are related to the antagonistic paths from *a*.

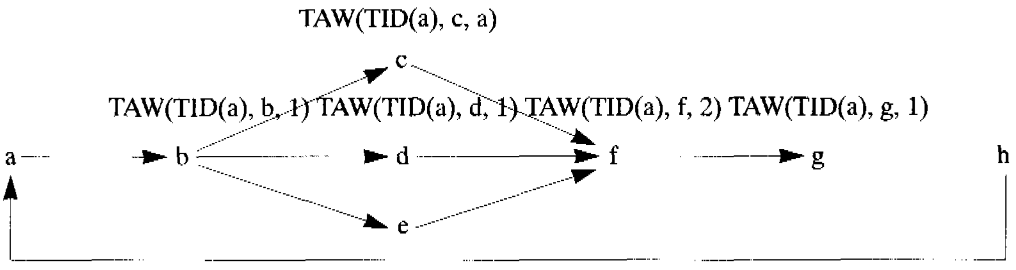

Figure 2.1 Propagation of transitive-antagonistic waits

We declare that there is a *potential deadlock* in the system when a transaction T_i transitively waits for a process of transaction T_j and there are some paths from T_j to T_i. Suppose that an edge from *g* to *h* is added into Figure 2.1. Then we declare that there is a potential deadlock in the system since TAW (TID(*a*), *g*, 1) holds and there is a path from TID(*g*) to TID(*a*).

In order to facilitate the construction of TAWs, inter-site messages called *probes* are sent as in [4], [7], [16], [17], [19], [21]. A probe, denoted as PB(*initiator, sender, receiver*), is a message from the site of *agent sender* to the site of agent *receiver* to inform *receiver* that the transaction *initiator* has been transitively and antagonistically waiting for *sender* at the site of *sender*. A probe PB (initiator, sender, receiver) is sent when one of the following holds: (1) TAW(initiator, sender, count) exists and MW(sender, receiver) is newly added. (2) MW(sender, receiver) exists and TAW(initiator, sender, 1) is newly adde.

To construct the condensed TWFG at a site, upon receiving a probe and upon adding a lockwait edge, the following procedure TAW_propagation is called to represent that a transaction *initiator* waits for a transaction process *terminus* transitively and antagonistically through an incomng edge to terminus and also to propagate that effect down to lock-wait edges and message-wait edges that are outgoing from terminus.

```
Procedure TAW_propagation(initiator, terminus)
begin
  if initiator < TID(terminus) and TWFG[TID(terminus)]. global = TRUE
  then return;
  if TAW(initiator, terminus, count) is in TWFG[TID(terminus)]. TAWS
  then increase count by 1 and return;
  add TAW(initiator, terminus, 1) into TWFG[TID(terminus)]. TAWS;
  for each MW(V, W) in TWFG[TID(terminus)]. MWS do
    if V = terminus
    then send probe PB(initiator, V, W) to the site of agent W;
  for each LW(V, W) in TWFG[TID(terminus)]. LWS do
    if V = terminus
    then TAW_propagation(initiator, W)
end;
```

To compensate for probes sent, inter-site messages called *antiprobes* [16], [17] are introduced. An antiprobe, denoted as AP (*initiator, sender, receiver*), is a message from the site of agent *sender* to the site of agent *receiver* to inform receiver that sender is no longer waited by transaction *initiator*. An antiprobe AP(initiator, sender, receiver) is sent only when TAW(initiator, sender, count) is deleted in the presence of MW (sender, receiver).

To compensate for TAW_propagation, upon receiving an antiprobe and upon deleting a lock-wait edge, the following procedure TAW_contraction is called to represent that the former TAW relationship from initiator to terminus through an incoming edge to terminus does not hold anymore and also to propagate that effect down to lock-wait edges and message-wait edges that are outgoing from terminus.

```
Procedure TAW_contraction(initiator, terminus)
begin
  if initiator < TID(terminus) and TWFG[TID(terminus)]. global = TRUE
  then return;
```

```
    if TAW(initiator, terminus, count) is in TWFG[TID(terminus)]. TAWS
    then begin
        if count > 1
        then decrease count by 1 and return;
        remove TAW(initiator, terminus, count) from TWFG[TID(terminus)]. TAWS
        for each MW(V, W) in TWFG[TID(terminus)]. MWS do
            if V = terminus
            then send antiprobe AP(initiator, V,W) to the site of agent W;
        for each LW(V, W) in TWFG[TID(terminus)]. LWS do
            if V = terminus
            then TAW-contraction(initiator, W)
    end
end;
```

Deadlocks are detected upon adding a lock-wait edge and upon receiving a probe. Once a potential deadlock is declared by detecting that transaction T_i transitively waits for transaction T_j and there are paths from T_j to T_i, for each path from T_j to T_i, the youngest transaction on the path is selected as a victim and it is added into victim_set. Checking paths and selecting the youngest transaction on a path in a depth-first way can be found in [12], [13]. After checking every path, if either T_i or T_j is one of the victim, victim_set is replaced by T_i or T_j respectively. For each victim in victim_set, before aborting the victim, if the victim is a global transaction, all PBs and TAWs having the victim as their initiator are removed from the local TWFG.

```
Procedure Deadlock_detection(s, t, victim_set)
/* check paths from transaction s to transaction t */
begin
    victim_set = {};
    if t does not have an entry in TWFG
    then return;
    for each path from s to t in TWFG do
        slect the youngest transaction on the path and add it into victim_set.
    if either s or t is in victim_set
    then set victim_set as that transaction.
    for each victim in victim_set do begin
        if TWFG[victim]. global + TRUE then begin
            for i = 1 to N do begin
                for each PB(initiator , V, W) in TWFG[i]. PBS do
```

```
                  if initiator  = victim
                  then remove PB(initiator , V, W) from TWFG[I]. PBS;
                for each TAW(initiator , W, count) in TWFG[I]. TAWS do
                  if initiator  = victim then begin
                    remove TAW(initiator , W, count) from TWFG[I]. TAWS.
                    for each MW(X, Y) in TWFG[i]. MWS do
                      if X = W
                      then send antiprobe AP(initiator , X, Y) to the site of agent Y;
                  end
              end
          end;
          abort victim.
      end;
    end;
```

Upon receiving probe PB(initiator, sender, receiver) at the site of receiver, procedure PB_propagation(initiator, sender, receiver) is called. In the procedure if TID(receiver) does not have an entry in the local TWFG or MW (receiver, sender) is in the local TWFG, the message is neglected. Otherwise, we detect and resolve global deadlocks by checking paths from TID(receiver) to initiator in the local TWFG. If initiator is not aborted in the deadlock resolution, the probe is stored in the local TWFG and TAW_propagation follows. Note that TID(receiver) does not have an entry in the local TWFG if the transaction has been aborted already. Having a message-wait edge MW(receiver, sender) means that thout of date. We do keep the received probe for the correct construction of TAWs.

```
Procedure PB_propagation(initiator, sender, receiver)
begin
  if TID(receiver) does not have an entry in TWFG or
    MW(receiver, sender) is in TWFG[TID(receiver)]. MWS
  then return;
  Deadlock_detection(TID(receiver), initiator, victim_set);
  if initiator ∈ victim_set
  then return;
  insert PB(initiator, sender, receiver) into TWFG[TID(receiver)]. PBS;
  TAW_propagation(initiator, receiver);
end;
```

Upon receiving antiprobe AP(initiator, sender, receiver), procedure AP_propagation (initiator, sender, receiver) is called. In the procedure if TID(receiver) does not have an entry in TWFG or PB(initiator, sender, receiver) is not in TWFG, the message is neglected. Otherwise, we remove PB(initiator, sender, receiver) and TAW_contraction follows.

```
Procedure AP_propagation(initiator, sender, receiver)
begin
   if TID(receiver) does not have an entry in TWFG or
     PB(initiator, sender, receiver) is not in TWFG[TID(receiver)]. PBS
   then return;
   remove PB(initiator, sender, receiver) from TWFG[TID(receiver)]. PBS;
  TAW_contraction(initiator, receiver)
end;
```

When a lock-wait LW(waiter, waitee) occurs, procedure LW_addition(waiter, waitee) is called. The procedure first detects local deadlocks and then detects global deadlocks for each TAW record that has an agent of transaction TID(waiter) as its terminus. If either TID(waiter) or TID(waitee) is aborted, we do nothing more for the lock-wait. Otherwise, after adding LW(waiter, waitee) into the local TWFG, for every possible antagonistic path extension and also for a new antagonistic path made by the addition of the lock-wait edge, TAW_propagation follows.

```
Procedure LW_addition(waiter, waitee)
begin
   Deadlock_detection(TID(waitee), TID(waiter), victim_set);
   if TID(waitee) ∈ victim_set or TID(waiter) ∈ victim_set.
   then return;
   For each TAW(initiator, V, count) in TWFG[TID(waiter)].TAWS do begin
      Deadlock_detection(TID(waitee), initiator, victim_set);
      if TID(waitee) ∈ victim_set
      then return
   end;
   add LW(waiter, waitee) into TWFG[TID(waiter)]. LWS.
   For each TAW(initiator, V, count) in TWFG[TID(waiter)]. TAWS do
      if V = waiter
      then TAW_propagation(initiator, waitee);
   if TWFG[TID(waiter)]. global = TRUE
```

```
    then TAW_propagation(TID(waiter), waitee)
end;
```

When a lock-wait LW(waiter, waitee) does not hold anymore, the following procedure LW_deletion(waiter, waitee) is called to delete LW(waiter, waitee) and also to propagate the effect of the destruction of some antagonistic paths caused by the deletion of the lock-wait edge.

```
Procedure LW_deletion(waiter, waitee)
begin
    delete LW(waiter, waitee) from TWFG[TID(waiter)]. LWS.
    For each TAW(initiator, V, count) in TWFG[TID(waiter)]. TAWS do
        if V = waiter
        then TAW_contraction(initiator, waitee);
    if TWFG[TID(waiter)]. global = TRUE
    then TAW_contraction(TID(waiter), waitee)
end;
```

When a message-wait MW(sender, receiver) occurs, procedure MW_addition(sender, receiver) is called. In the procedure, TWFG [TID(sender)]. global is set as TRUE and if PB (initiator, receiver, sender) is in TWFG[TID (sender)]. PBS, the BP record is deleted and TAW_contraction(initiator, sender) follows. We now add MW(sender, receiver) and then for each TAW(initiator, sender, count), send probe PB(initiator, sender, receiver) to the site of agent receiver.

```
Procedure MW_addition(sender, receiver)
begin
    TWFG[TID(sender)]. global = TRUE;
    for each PB(initiator, V, W) in TWFG[TID(sender)]. PBS do
        if V = receiver and W = sender
        then begin
            delete PB(initiator, V, W) from TWFG[TID(sender)]. PBS;
            TAW_contraction(initiator, sender)
        end;
    add MW(sender, receiver) into TWFG[TID(sender)]. MWS;
    for each TAW(initiator, V, count) in TWFG[TID(sender)]. TAWS do
        if V = sender
        then send probe PB(initiator, sender, receiver) to the site of agent receiver
end;
```

When a transaction process sender receives a new request or an answer from its chort reciver, a message-wait MW(sender, receiver) does not hold anymore. In this case, we simply delete MW(sender, receiver) from TWFG[TID(sender)]. MWS and do nothing else.

Example 2.1 Consider a snap shot of a global TWFG in Figure 2.2. There are 3 sites and 5 transactions. Among them T_6, T_7, and T_9 are global transactions.

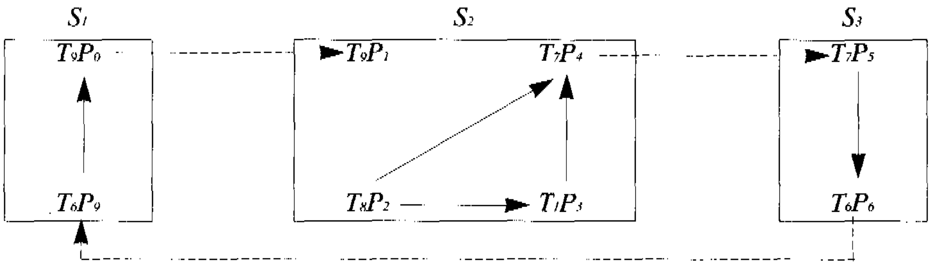

Figure 2.2 Snap shot of a global TWFG

At S_1 and S_2, since there is not any antagonistic lock-wait edges, no TAWs are constructed and thus no probes are generated. At site S_3, there exists one antagonistic lock-wait edge LW($T_7P_5S_3$, $T_6P_6S_3$). As a result of LW addition and TAW propagation, TAW(T_7, $T_6P_6S_3$, 1) is constructed and probe PB(T_7, $T_6P_6S_3$, $T_6P_9S_1$) is sent S_1. At site S_1, upon receiving the probe sent from S_3, as a result of PB propagation and TAW propagation, the probe is stored and TAW(T_7, $T_6P_9S_1$, 1) is constructed in $TWFG_1$. Note that $T_9P_0S_1$ cannot have a TAW record whose initiator is T_7 since T_7 is not antagonistic with T_9.

Assume that T_9P_1 in S_2 requests a resource that is held by T_8P_2 in a conflicting lock mode. Then the lock request is blocked and it initiates LW addition procedure. Since the lock-wait edge is antagonistic, according to TAW propagation, TAW(T_9, $T_8P_2S_2$, 1), TAW [T_9, $T_1P_3S_2$, 1) and TAW(T_9, $T_7P_4S_2$, 2) are constructed, and probe PB(T_9, $T_7P_4S_2$, $T_7P_5S_3$) is sent to S_3. At site S_3, upon receiving the probe sent from S_2, as a result of PB propagation and TAW propagation, the probe is stored at $TWFG_3$, TAW(T_9, $T_7P_5S_3$, 1) and TAW(T_9, $T_6P_6S_3$, 1) are constructed, and probe PB(T_9, $T_6P_6S_3$, $T_6P_9S_1$) is sent to S_1.

At site S_1, upon receiving the probe sent from S_3, as a result of PB propagation and deadlock detection procedures, a deadlock is declared, T_9 is aborted and all PBs and TAWs whose initiator is T_9 are removed from $TWFG_1$ since the initiator of the probe is T_9, there is a local path from LW($T_9P_1S_2$, $T_8P_2S_2$) is deleted, as a result of LW deletion and TAW contraction, an antiprobe AP(T_9, $T_7P_4S_2$, $T_7P_5S_3$) is sent to S_3. At site S_3, upon receiving the antiprobe sent from S_2, as a result of PB contraction and TAW contraction, the corresponding PB is deleted, TAW(T_9, $T_7P_5S_3$, 1) and TAW(T_9, $T_6P_6S_3$, 1) are also deleted, and antiprobe AP(T_9, $T_6P_6S_3$, $T_6P_9S_1$) is sent to S_1. At site S_1, upon receiving the antiprobe sent from S_3, since there is not the corresponding PB record in $TWFG_1$, the propagation of antiprobe stops at this point. □

3. Extensions to the original algorithm

While the algorithm described in Section 2 is guaranteed to detect all real global deadlocks [15], like most distributed deadlock detection algorithms it suffers from the problem of false deadlock detection. Due to the propagation delay of the antiprobe, the information gathered at each site may not reflect accurately the transaction wait-for relationships. The unnecessary abortion of transactons due to false deadlocks may degrade the system performance and increases the recovery costs. However, avoiding all false deadlocks requires tracing all the sits where the corresponding probes went through as in [4], [21], and this process contributes also substantially to the deterioration in system performance.

Remember that a global deadlock is detected when there is an antagonistic path starting from a process of transaction T_i and ending at a process of transaction T_j, at site S_r, suh that the latter receives a probe or contains a TAW record whose initiator is T_i, and in addition there is a path from some process of T_j to some process of T_i in the local TWFG at Sr. A false deadlock could be called only if the meantime this antagonistic path has been broken but no corresponding antiprobe has been received. The broken links can be the result of local deadlock resolution. If we were to choose the current blocker strategy for victim selection in the resolution of local deadlocks, then the probability of broken links would be reduced. However, due to considerations of fairness, possibility of deadlock and increased recovery costs, we do not take this approach and opt to stay with the strategy of selecting the youngest transaction in a deadlock cycle as the victim for resolving local (and global) deadlocks.

In order to reduce the probability of false deadlocks we propose two extensions to the original algorithm proposed in Section 2 that perform additional validation procedures to check the status of the global transactions that may cause a global deadlock. We denote the original algorithm based on the Hybrid Wait-for graph as HW and the two extensions using the Validation procedures as HWV1 and HWV2 respectively. However, we restrict transactions in HWV1 and HWV2 to be processed sequentillay. In other words, parallel execution of global transactions are not permitted. More specifically, an agent of a transaction cannot proceed with its computation after sending a request to an agent of the transaction at remote site until the latter sends response back to the former.

3.1 Deadlock Detection with HWV1

The basic idea behind the validation procedure is to revalidate each received message to insure that it is not stale and to check the status of global transactions that causes a potential global deadlock. We say that a global transaction at a given site is in *intersite-waiting* state at that site if either some agent of the transaction is in message-waiting state or there is an antagonistic path, in the local TWFG of the site, that starts from an agent of the transaction to an agent of another transaction that isin message-waiting state. Thus, it is clear that a local site can eliminate those potential global deadlocks caused by global transactions that are not in intersite-waiting state at that site. In addition, to facilitate the validation of messages a global block count is associated with every transaction. The *global block count* of a transaction, denoted by *gbc*, gives the number of times that the transaction has been blocked for lock requests. Each entry in the TWFG contains also its gbc value; in addition the gbc is also included in the TAW records, probes and antiprobes. Thus these records and messages have now the following formats: TAW(*initiator, terminus, count, gbc*), PB(*initiator, sender, receiver, gbc*) and AP(*initiator, sender, receiver, gbc*), where *gbc* is the global block count of transaction *initiator*.

Each transaction is given a global block count of zero when it first enters the system. Each time an agent of the transaction is denied a lock request, its gbc is increased by one. When an agent of a transaction requests a resource at other site, its gbc is sent together with its request to that site, and it becomes the initial value for the gbc of its cohort at that site. Similarly, when a response to a request is sent an updated gbc value is included in the answer. For example, assume that transaction T_1 is first created at site S_1. Then the global block count of T_1 at site S_1 is initialized to 0. If T_1 requests a resource that is currently held by T_2 with a conflicting lock mode, T_1 is blocked and its global block count is increased by 1. Later, it obtains the resource and wants to request another resource S_2. It would send its global block count together with its request to S_2. An agent of T_1 with a global block count of 1 will be created at S_2. If the agent cannot obtain the resource immediately, the request is blocked and its global block count is increased to 2. When the agent returns the result to S_1, the updated global block count is also returned to S_1. From then on, T_1 would proceed with its global block count of 2.

In addition to request initiation and response, the gbc information of a transaction and that of PB and TAW records need to be updated whenever probes and antiprobes are received and when TAW_propagation and TAW_contraction occur.

Definition 3.1 Let the largest one among global block count values associated with a global transaction T_i at a site be *gbc*'. A global block count value gbc associated with T_i is a stale one at the site if one of the following is true: (1) $gbc<gbc'$. (2) $gbc = gbc'$, T_i has an agent at the site and T_i is not in intersite-waiting at the site.

Upon receiving probe PB(initiator, sender, receiver, *gbc*) or antiprobe AP(initiator, sender, receiver, gbc), if initiator has an entry in TWFG and gbc < TWFG[initiator]. gbc, then gbc is declared stale and the message is discarded. However, if initiator has an entry in the local TWFG and gbc > TWFG [initiator]. gbc, then TWFG[initiator]. gbc is replaced by gbc. In the case of receiving a probe, if probe record PB(initiator, sender, receiver, gbc')is in the local TWFG and gbc' < gbc, PB(initiator, sender, receiver, gbc') is replaced by PB(initiator, sender, receiver, gbc) and TAW_propagation (initiator, receiver, gbc) follows. In the case of receiving an antiprobe, corresponding probe record is deleted from the local TWFG and TAW_contraction(initiator, receiver, gbc) follows.

Upon adding or deleting lock-wait edge LW (waiter, waitee), for each TAW record TAW (initiator, V, count, gbc') in TWFG[TID (waiter)]. TAWS if initiator has an entry in TWFG and gbc' < TWFG[initiator]. gbc, then the TAW record is a stale one and it is not considered for global deadlock detection (in the case of LW_addition) and TAW_contraction (in the case of LW_deletion). Procedures MW_addition(sender, receiver) and MW_deletion(sender, receiver) are the same as in the vanilla algorithm HW.

In procedure TAW_propagation(initiator, terminus, gbc) if TAW(initiator, terminus, count, gbc') is in the local TWFG, the following three cases need to be checked: (1) if gbc = gbc' then increase count by 1 and return. (2) if gbc < gbc' then return. (3) if gbc > gbc' then replace TAW(initiator, terminus, count, gbc') by TAW(initiator, terminus, 1, gbc) and propagate this new TAW record down to every lock-wait edge (by calling TAW_propagation) and message-wait edge (by sending probes) that are outgoing from terminus.

In procedure TAW_contraction(initiator, terminus, gbc) if TAW(initiaor, terminus, count, gbc') is in the local TWFG, the following three cases need to be checked: (1) if gbc = gbc' and count>1 then decrease count by 1 and return. (2) if gbc < gbc' then return. (3) if gbc = gbc' and count = 1 or gbc > gbc' then delete the TAW record and propagate the deletion down to every lock-wait edge.(by calling TAW_contraction) and message-wait edge (by sending antiprobes) that are outgoing from terminus.

Local deadlock detection is done exactly the same as in the algorithm HW. For global deadlock detection, a validation procedure is applid to eliminate (some) false deadlocks. If the global deadlock detection procedure is initiated by transaction *initiator'* , with *gbc'* as its global block count, of a probe or a TAW record and *initiator'* is not in inter-site waiting state, then the potential global deadlock is a false deadlock. Otherwise, we proceed to resolve the global deadlock as in the algorithm HW. Once a potential deadlock is found to be a false deadlock, all PBs and TAWs whose initiator is *initiator'* and at the same time whose global block count is less than or equal to *gbc'* are deleted from the local TWFG. In this process, some antiprobes might be sent owing to the deletion of TAW records.

Example 3.1 Consider a snap shot of a global TWFG in Figure 3.1 at time t_1 assuming that the gbc value of T_9 at site S_2 is 0; the gbc value of T_7 at site S_3 is 0; and LW ($T_9P_1S_2$, $T_8P_2S_2$) and LW($T_7P_5S_3$, $T_6P_6S_3$) are the last added lock-wait edges at S_2 and S_3 respectively. At site S_2, as a result of LW addition and TAW propagation, probe PB(T_9,

$T_7P_4S_2$, $T_7P_5S_3$, 1) is sent to S_3. At site S_3, a result of LW addition and TAW propagation, probe PB(T_7, $T_6P_6S_3$, $T_6P_9S_1$, 1) is sent to S_1. At site S_3, upon receiving the probe sent from S_2, as a result of PB propaation and TAW propagation, probe PB(T_9, $T_6P_6S_3$, $T_6P_9S_1$, 1) is sent to S_1. At site S_1, upon receiving those two probes sent from S_3, as a result of PB propagation and TAW propagation, $T_6P_9S_1$ contains PB(T_7, $T_6P_6S_3$, $T_6P_9S_1$, 1) PB(T_9, $T_6P_6S_3$, $T_6P_9S_1$, 1), TAW(T_7, $T_6P_9S_3$, 1) amd TAW(T_9, $T_6P_9S_3$, 1)

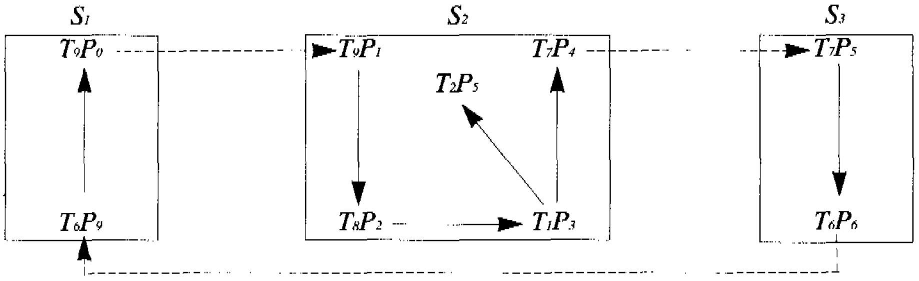

Figure 3.1 Global TWFG at time t_1

At time t_2, assume that T_2 in S_2 requests a resource that is held by T_8 in a conflicting lock mode as shown in Figure 3.2. This results in a local deadlock at S_2, and so the youngest transaction in the cycle, i.e. T_8, is aborted. The abortion of T_8 makes the lock-wait edge LW($T_8P_2S_2$, $T_1P_3S_2$) to be deleted. As a result of LW deletion and TAW contraction, antiprobe AP(T_9, $T_7P_4S_2$, $T_7P_5S_3$, 1) is sent to S_3.

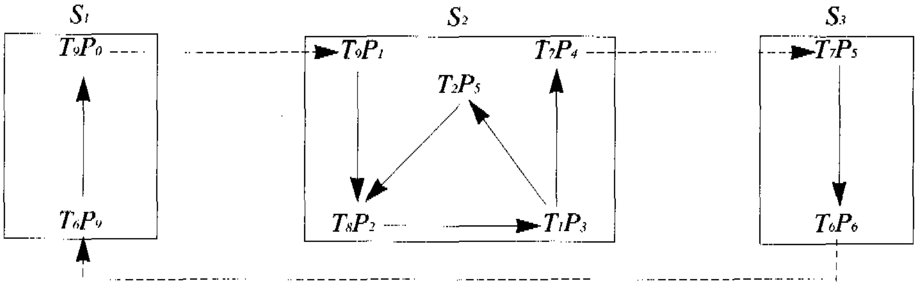

Figure 3.2 Global TWFG at time t_2

At time t_3, T_9P_1 at site S_2 finishes its job and the result is sent back to S_1 with a global block count having a value of 1. After that T_6P_9 at S_1 requests a resource that is held by T_9P_0 in a conflicting lock mode. The resulting configuration is shown in Figure 3.3. Algorithm HW declares a global deadlock, even through this is a false deadlock. The problem occurs since it does not check the status of the initiator(T_9 in the above case) before declaration of global deadlocks. In HWV1 algorithm however, since T_9 at S1 is not in intersite-waiting, the global block count 1 in TAW(T_9, $T_6P_9S_1$, 1, 1) becomes a stale one. According to this, a global deadlock is not declared but PB(T_9, $T_6P_6S_3$, $T_6P_9S_1$, 1) and TAW(T_9, $T_6P_9S_1$, 1, 1) are deleted from $TWFG_1$.

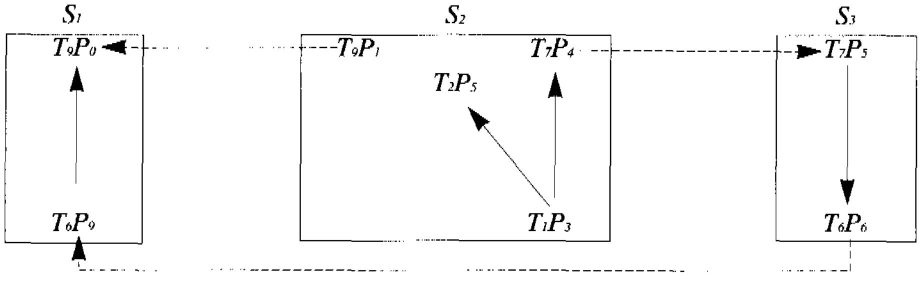

Figure 3.3 Global TWFG at time t_3

At time t_4, assume that $T_9P_0S_1$ requests a resource that is being currently held by T_7P_5 at site S_3 and that this request arrives before the antiprobe AP(T_9, $T_7P_4S_2$, $T_7P_5S_3$, 1). A cohort $T_9P_2S_3$ is created and a lock-wait edge LW($T_9P_2S_3$, $T_7P_5S_3$) is added into $TWFG_3$. In the local TWFG the gbc of T_9 becomes 2. This situation is illustrated in Figure 3.4.

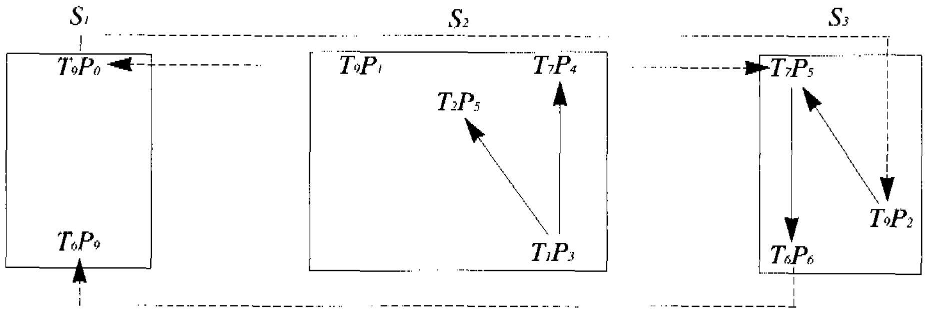

Figure 3.4 A global deadlock at S_1

As a result of LW addition and TAW propagation, the TAW record of $T_7P_5S_3$ is replaced by TAW(T_9, $T_7P_5S_3$, 1, 2), the TAW record of $T_6P_6S_3$ is replaced by TAW(T_9, $T_6P_6S_3$, 1, 2) and probe PB(T_9, $T_6P_6S_3$, $T_6P_9S_1$, 2) is sent to S_1. Note that there is no need to send antiprobe AP(T_9, $T_6P_6S_3$, $T_6P_9S_1$, 1), since the updated global block count field in the probe conveys the same message. Upon receiving the probe at S_1, a global deadlock is declared ant T_9 is aborted. When the antiprobe AP(T_9, $T_7P_4S_2$, $T_7P_5S_3$, 1) sent from S_2 arrives at S_3, that message is neglected since its global block count is less than $TWFG_3[T_9]$.gbc. □

3.2 Deadlock detection with HWV2

When a potential global deadlock is declared, validity of a global block count can be checked one more time to further reduce the possibility of false deadlocks without causing too much overhead. In HWV2 algorithm, global block counts, TAW, PB, AP informations are processed exactly the same as in the HWV1 algorithm. When a potential global deadlock is found at site S_r by a probe or TAW record whose initiator is T_k and whose global block count is gbc, gbc is checked first whether it is a false one or not. If it is a false one, the found deadlock is a false deadlock, and so all PB and TAW informations whose initiator is T_k and whose global block count is less than or equal to gbc are deleted from S_r. Otherwise, we do the following If T_k does not have any message-wait edge, we declare a global deadlock and do the corresponding deadlock resolution operations. However, when T_k has a message-wait edge, say MW($T_kP_xS_r$, $T_lP_yS_s$), one more validation checking of gbc is done as follows:

(1) A validation request message CHECK ($T_kP_xS_r$, $T_lP_yS_s$, gbc) is sent to S_s.

(2) At S_s, upon receiving a validation request message CHECK($T_kP_xS_r$, $T_lP_yS_s$, gbc), two validation checks are performed. One is checking whether gbc is a stale one or not and the other is checking whether there exists a message-wait edge MW($T_lP_yS_s$, $T_kP_xS_r$) in $TWFG_s[T_k]$. MWS. If either gbc is a stale one or there exists the message-wait edge, the validation fails and a negative response NEG(T_k, $TWFG_s[T_k]$.gbc) is sent to site S_r. Otherwise, a positive respond $POS(T_k$, gbc) is sent to site S_r.

(3) At S_r, upon receiving a positive response POS(T_k, gbc), all PB and TAW informations whose initiator is T_k are deleted from TWFGr and T_k is aborted. However, upon receiving a negative response NEG (T_k, gbc′), if gbc′ is less than or equal to $TWFG_r[T_k]$. gbc, nothing is doen. Otherwise, all PB and TAW informations whose initiator is T_k and whose global block count is less than gbc′ are deleted from $TWFG_r$ and $TWFG_r[T_k]$. gbc is replaced by gbc′.

Example 3.2 Consider again the situation of Example 3.1 at time t_3. Assume that the lock-wait LW($T_6P_9S_1$, $T_9P_0S_1$) occurs before $T_9P_1S_2$ sends the result to $T_9P_0S_1$. That is, $T_9P_0S_1$ is still message-waiting for $T_9P_1S_2$. In the case of the HWV1 algorithm, a global deadlock is declared and T_9 is aborted unnecessarily. In HWV2 algorithm however, a validation request message CHECK($T_9P_0S_1$, $T_9P_1S_2$, 1) is sent to S_2. At S_2, upon receiving the validation request message, the validation fails and a negative response NEG(T_9, 1) is sent to S_1 since MW($T_9P_1S_2$, $T_9P_0S_1$) exists in TWFG2[T_9]. MWS. At S_1, upon receiving the negative response from S_2, nothing needs to be done since its global block count 1 is the same as TWFG1[T_9].gbc. In this case, even though two more messages are required to be sent, a false global deadlock is prevented. □

4. Performance Analysis

All three algorithms presented in the previous section were evaluated using a simulation based on a closed queueing model[15]. The simulation model is similar to the one used in [1] with a few extensions added for modeling the distributed system concepts. Our simulation program was implemented in CSIM[18], a process-oriented discrete-event simulation package based on the programming language C. In order to simplify the simulation we made the following assumptions: (1) only exclusive locks and shared locks are allowed, (2) each site can have at most one agent for each global transaction, (3) all data items have the same granularity level and have the equal probability of request, and (4) the Concurrency Control Unit (CC) of each site supervises the transaction scheduling, maintains the local TWFG and lock table and performs the deadlock detection and resolution procedure.

Initially, all transactions are generated at their originating sites and are put into their local Ready-Queues. A transaction then enters different phases depending upon the type of requests it issues. A transaction T_i leaves the Ready-Queue either to wait at the CC queue if it requests a local data item, or to wait at the Communication Manager (CM) queue if it issues a remote access request. If T_i requested the CC services, it will wait in the CC queue until the CC unit is available and then issue its first (next) request. If the request is granted, T_i releases the CC unit and enters the data accessing phase. If the requested data item is held by another transaction having a conflicting lock mode, T_i is added to the waiting queue of the requested data item and releases the CC unit which consequently checks the local TWFG for possible local or global deadlocks. If a transaction is selected as a victim to resolve some deadlocks, it is aborted. An aborted local transaction will be restarted after a preset restart-wait time and put into the back of the Ready-Queue to start over agin. If the victim is a global transaction, different actions will be taken. Assume that the abortion is occurred at site S_r and the aborted transaction is T_i. If T_i has been originatied at S_r, abort messages will be sent to al of its accessing sites. Otherwise, T_i' s agent at S_r is aborted and an abort message is sent to T_i' s originating site.

Each site contains a fixed number of data items and allows 100 processes as its maximum degree of multiprogramming. An important parameter that we varied in our simulation studies is the GlobalRatio, which is the percentage of non-local agents allowed at each site. For example, if the GlobalRatio is 0. 2, then the maximum number of non-local agents allowed at each site is 20. Transactions, i.e., agents are waiting in the Read-Queue if the current number of global agents in the CC queue reached the maximum allowed. Each local transaction makes at least 1 and at most 6 requests. Each global transaction accesses at least two different sites and makes 2 to 6 requests when the system contains 3 or 5 sites and at most 10 requests when the system contains 10 sites. Each transaction uses a random number generator to decide the number of requests, lock modes and the site to access. Each simulation run will terminate after a preset simulation time (6000 simulation time units). We compare the average value of overall throughput, global throughput, global abortion rates, the overall recovery cost and the global recovery cost under different number of sites and GlobalRatios. The throughput is the number of transactions committed per simulation run. The global abortion rate is measured as the number of global aborts per committed global transac-tions for each simulation run. Global throughput is the number of committed global transactions per simulation run.

We now procced to analyze the simulation results for the vanilla algorithm HW and the variants with validation procedures, HWV1 and HWV2. Figure 4.1 shows the percetage of false deadlocks declared by the vanilla algorithm HW for 5 sites when we varied the maximum number of requests per global transactions. The false global deadlocks reported here are only those that can be verified as such in the HWV1 and HWV2 algorithms. Thus, the actual number of false deadlocks maybe larger. As expected, the probability of false deadlocks is low when the global transactions make only a few requests and increases superlinearly with the maximum number of requests per global transaction.

min-max global req/trans.	false deadlock found	total global abort	%
2-4	1	495	0.2%
2-6	15	758	1.9%
2-8	43	897	4.7%
2-10	119	946	12.6%

Figure 4.1 False global deadlocks found in simulation runs

Figure 4.2 shows the number of overall committed transactions produced by these three algorithms under different GlobalRatio values. It indicates that the throughput is largely influenced by GlobalRatio. When GlobalRatio is low, the overall throughput is high and vice versa. This is because global transactions access non-local data items and need more time to complete their requests. As a consequence, the average data holding time is also increased, i.e. a transcation has to wait longer for its blocked request to be granted. This means a longer average transaction response time and therefore a poorer throughput for a system that has a large amount of global transactions. Besides, global transactions are more likely to be aborted because of global deadlocks especially when the data contention is very high (as in our simulation model).

Figure 4.3 shows the number of committed global transactions for these three algorithms. It shows that there is not much difference on throughput when GlobalRatio is low although two validation algorithms use more system resources for the validation routine. However, the global throughput of the HWV2 algorithm is 21.6% less than that of the vanilla algorithm HW when GlobalRatio equals to 1.0. This is not a surprise. The HWV1 algorithm only checkstransaction status locally. Its validation routine can be efficiently executed without using much system resources. On the other hand, the validation routine of the HWV2 algorithm not only checks locally but also checks the transaction status of the other site. Although its validation routine reduces the chance of false global deadlocks, it also delays detection of real global deadlocks and increases data item holding time and the average transaction response time. Thus the throughput of the HWV2 algorithm is also significantly reduced. Figure 4.4 shows the response time of the three algorithms when GlobalRatio is 1.0.

Number of sites	3			5			10		
Global Ratio	0.2	0.5	1.0	0.2	0.5	1.0	0.2	0.5	1.0
HW	8408	6342	4666	10427	7659	5500	13130	9002	3251
HWV1	8048	6237	5309	10543	7513	5309	15694	9327	3256
HWV2	8715	5944	3924	10049	6245	4419	14704	6834	2625

Figure 4.2 Comparison of the overall throughput

Number of sites	3			5			10		
Global Ratio	0.2	0.5	1.0	0.2	0.5	1.0	0.2	0.5	1.0
HW	554	1225	4666	464	815	5500	127	232	3251
HWV1	562	1188	5309	461	781	5309	165	248	3256
HWV2	526	1145	3924	426	763	4419	160	233	2625

Figure 4.3 Comparison of the throughput for global transactions

Number of sites	3	5	10
HW	20.39	17.61	29.33
HWV1	20.42	18.51	29.32
HWV2	24.97	22.32	46.10

Figure 4.4 Comparison of the average transaction response time

The advantages of these two validation algorithms are the low global abortion rate and the low recovery cost for aborted global transactions. The average recovery cost for aborted global transactions and the global abortion rate of the three algorithms are illustrated in Figure 4.5 and Figure 4.6 respectively. As expected, the original algorithm has the highest global abortion rates and the highest recovery cost for aborted global transactions. The HWV1 algorithm performs slightly better than the vanilla algorithm HW does. The HWV2 algorithm produces the best results in all cases especially when GlobalRatio is low and there are more sites in the system.

Number of sites	3			5			10		
Global Ratio	0.2	0.5	1.0	0.2	0.5	1.0	0.2	0.5	1.0
HW	134	738	1517	329	870	1389	575	1378	2632
HWV1	171	666	1493	342	890	1327	590	1309	2435
HWV2	146	627	1393	258	829	1077	433	1015	2060

Figure 4.5 Comparison of the recovery costs for global transactions

Number of sites	3	5	10
HW	0.254	0.364	1.265
HWV1	0.277	0.370	1.125
HWV2	0.220	0.340	0.941

Figure 4.6 Comparison of the average abortion rates

Based on those experimental results we draw the following conclusions:

(1) When the ratio of global transactions is small and they make only few global requests the chance of false deadlock is rare. Consequently, the vanilla algorithm HW is the best choice since it minimizes the response time and all other system metrics are fairly close to those of the HWV1 and HWV2 algorithms.

(2) In the environment of the sequential transaction processing and when the rate of global transactions are considerable, the HWV1 algorithm can always be used to reduce the chance of false global deadlocks while still maintaining a good throughput.

(3) For systems that give higher priority to global transactions or are sensitive to abortion of global transactions, the HWV2 algorithm would be a good choice.

5. Conclusion

We proposed a new distributed deadlock detection and resolution algorithm that allows for parallel execution of transactions at multiple sites and for the possiblity of multiple lock modes. Local deadlocks are dected by a regular cycle detection procedure in the augmented local TWFGs, while global deadlocks are detected by sending probes and antiprobes which enable us to construct a condensed global TWFG at each site. Probes and antiprobes are only sent from global transactions with higher priorities to transactions with lower priorities and they indicate that a transaction with a higher priority transitively waits for another one with a lower priority.

In order to reduce the probability of false deadlocks, we presented two extensions to our original algorithm which can be used in an environment of sequential transaction processing. These extensions associate a global block count with each transaction, in order to keep track of the number of times a given global ransaction has been blocked. This information is then used in a validation procedure which decides whether a given received probe or transitive-antagonistic wait is stale or not. We analyzed and compared the preformance of the three deadlock detection algorithms in terms of throughput, response time, abortion rate and recovery costs. We are currently working on extending the vanilla algorithm HW for periodic deadlock detection and resolution.

References

1. R.Agrawal, M.J.Carey and L.W.McVoy, "The performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems," *IEEE Trans. Software Eng.*, Vol.SE-13, No.12, pp. 1348-1363, December 1987.
2. K.M.Chandy and J.Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems," *in ACM Symposium on Principles of Distributed Computing*, pp.157-164, 1982.
3. A.K.Choudhary, "Cost of Distributed Deadlock Detection: A Performance Study," in *Proc. Sixth Int. Conf. on Data Engineering*, pp.174-181, February 1990.
4. 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 Trans. Software Eng.*, Vol.SE-15, No.1, pp.10-17, January 1989.
5. A.K. Elmagarmid A.P.Sheth, and M.T.Liu, "Deadlock Detection Algorithms in Distributed Database Systems," in *Proc. Second Int. Conf. on Data Engineering*, pp.556-564, February 1986.
6. V.D.Gligor and S.H.Shattuck, "On Deadlock Detection in Distributed Systems," *IEEE Trans. Software Eng.*, Vol.SE-6, No. 5, pp.435-439, September 1980.
7. L.H.Hass and C.Mohan, "A Distributed Deadlock Detection Algorithm for a Resource-Based System," Rep.RJ3765, *IBM Research Lab.*, San Jose, California, January 1983.
8. G.S.Ho and C.V. Ramamoorthy, "Protocols for Deadlock Detection in Distributed Database Systems," *IEEE Trans. Software Eng.*, Vol.SE-8, No.6, pp.554-557, November 1982.
9. B.G.Lindsay, L.H.Haas, C.Mohan, P.F. Wilms, and R.A. Yost, "Computation and Communication in R*: A Distributed Database Manager," *ACM Trans. Computer Systems*, Vol. 2, No.1, pp.24-38, Feb. 1984.
10 L.Manteiman, "The birth of OSI TP: A new way to link OLTP networks," Data Communications, November 1989.
11. D.A. Menasce and R.R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *IEEE Trans. Software Eng.*, Vol. SE-5, No.3, pp.195-202, May 1979.
12. R. Obermarck, "The Distributed Deadlock Detection Algorith," *ACM Trans. Database Systems*, Vol.7, No.2, pp.197-

208, June 1982.

13. Y.C.Park and P.Scheuermann, "A Deadlock Detection and Resolution Algorithm For Sequential Transaction Processing with Multiple Lock Modes," in *Proc. 15th Int. Computer Software and Applications Conf.*, pp.70-77, September 1991.
14. Y.C.Park, P.Scheuermann, and S.H.Lee, "A Periodic Deadlock Detection and Resolution Algorithm with a New Graph Model for Sequential Transaction Processing," in *Proc. 8the Int. Conf. on Data Engineering*, pp.202-209, February 1992.
15. Y.C.Park, P.Scheuermann, and H.L.Tung, "Distributed Deadlock Detection with Hybrid Watit-for Graphs and Anti-Probe Validation," Technical Report, University of Ulsan, Korea, 1993.
16. M.Roesler and W.A. Burkhard, "Deadlock Resolution and Semantic Lock Models in Object-Oriented Distributed Systems," in *Proc. 1988 ACM-SIGMOD Int. Conf. on Management of Data*, pp. 361-370, June 1988.
17. M.Roesler and W.A.Burkhard, "Resolution of Deadlocks in Object-Oriented Distributed Systems," *IEEE Ttrans. Computer*, Vol. 38, No.8, pp.1212-1224, August 1989.
18. H.Schwetman, "CSIM User' s Guide," *MCC Technical Report*, ACT-126-90, March, 1990.
19. M.K. Sinha and N.Natarjan, "A Priority Based Distributed Deadlock Detection Algorithm" *IEEE Trans. Software Eng.*, Vol.SE-11, No.1, pp.67-80, January 1985.
20. M.Stonebraker, "Concurrency Control and Consitency of Multiple Copies of Data in Distributed INGRES," *IEEE Trans. Software Eng.*, Vol.SE-5, No.3, pp.188-194, May 1979.
21. K. Sugihara, T.Kikuno, N.Yoshida, and M.Ogata, "A Distributed Algorithm for Deadlock Detection and Resolution," in *Proc. Fourth Int. Conf. on Distributed Computing Systems*, pp.169-176, 1984.
22. X/Open Company Ltd, "Interim Reference Model for Distributed Transaction processing," Transaction Processing Working Group, X/Open Company Limited, July 1989.