공학 연구논문집 제24권 제1호 pp. 133~166, 울산대학교
Journal of Engineering Research Vol. 24, No. 1, pp. 133~166, 1993.
University of Ulsan.

A Distributed Deadlock Detection and Resolution Algorithm Based on a Hybrid Graph Construction and Probe Generation Scheme†

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

Abstract

We present a periodic algorithm for deadlock detection and resolution in distributed database systems, which allows for early detection of explicit and implicit deadlocks. Our algorithm uses an augmented transaction wait-for graph (TWFG) at each site, which contains in addition to lock-wait information, also information about message-wait and master-slave relationships among agents of a transaction, and status of transactions with respect to the commitment protocol. In order to detect and resolve global deadlocks with a minimal number of inter-site messages we adopt a modified version of the probe generation scheme such that two pools are maintained at each site. One pool keeps probes received, while the other keeps the receipts of probes sent. Periodically, we check for deadlocks by using the information contained in the local TWFGs and the probes received. The receipts of probes sent are used to avoid retransmission of the same probe and also to transmit antiprobes. Our algorithm allows for parallel execution of transactions at multiple sites and supports multiple modes of locks and lock conversion.

대기 그래프와 프로브 생성에 기초한 교착 상태의 분산 검출 및 회복 알고리즘†

박영철* · 박양수* · 정영필**
전자계산학과* · 컴퓨터공학과**

† 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 주기적 알고리즘을 제시한다. 이 알고리즘은 명시적, 암시적 교착 상태들을 초기에 검출할 수 있도록 하며 이를 위해 각 사이트에 확장된 트랜잭션 대기 그래프(TWFG)를 유지한다. 확장된 트랜잭션 대기 그래프는 로크 정보 뿐만아니라 트랜잭션 에이전트들간의 메시지 대기 정보 및 주종 관계에 대한 정보, 그리고 트랜잭션 완료 프로토콜에 관련된 트랜잭션들의 상태정보들도 포함한다. 각 사이트 간 메시지의 수효를 최소로 가진 전역 교착상태를 검출하고 회복하기 위해서는 프로브 생성 기법을 변형하여 사용하며 제시된 알고리즘은 각 사이트에 두개의 메모리 풀을 유지한다. 하나의 풀은 수신한 프로브들을 유지하며, 다른 하나의 풀은 보낸 프로브들의 수신에 대한 정보를 유지한다. 각 지역 사이트의 트랜잭션 대기 그래프에 포함된 정보와 수령한 프로브드들을 사용하여 주기적으로 교착상태를 검사한다. 송신한 프로브에 대한 수신 정보는 동일한 프로브의 전송을 피하기 위해 사용하며 또한 대항 프로브들을 전송하기 위해서도 사용한다. 이 알고리즘은 다중 사이트에서 트랜잭션들의 병렬수행을 허가하며 다중 로크 모드 및 로크 변환을 지원한다.

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. Each site is able to process local transactions, which access data only in that 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 distributed database systems involve in most cases only a few sites since most transactions in the system access only resources local to their originating sites. Therefore, a distributed deadlock detection approach has been adopted most widely [7], [10], [20], [29], [34], [35], [37] instead of a centralized or hierarchical approach [12], [21], [27], [36].

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* [9], [20], [29] 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 request 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 [29]. 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 transmission of a large number of messages, especially when shared locks are permitted, and is not very practical when the number of transactions is large.

One the other hand, in schemes that are based on the construction of a *condensed graph* [16], [27]. 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* transmission) or to the site of origin of transaction T_j(*forward* transmission). Backward transmissions result in a very large number of messages transmitted when there 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 priority [10], [33]-[35]. 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. In our probe mechanism each site maintains two pools of probes: one for probes received and one for receipts of probes sent. Since majority of transactions are local and distributed deadlocks are known to occur infrequently [17], [29], [36], our algorithm constructs periodically local TWFG at each site to detect local deadlocks without any deadlock detection message transmission as in [7], [9], [12], [13], [29], [38]. Global deadlocks are detected by using the local TWFG and the probes received. Then 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. 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 Section 2 the transaction model and the modes of locks under consideration are explained. Our systematic scheduling policy is discussed in Section 3. Section 4 introduces our graph construction mechanism in detail for detection of deadlocks in our system. Our message transmission mechanism for sending probes and antiprobes are given in Section 5 and our new distributed deadlock detection and resolution algorithm is presented in Section 6. The performance characteristics of the proposed scheme and a refinement for the minimal cost victim selection are illustrated in Section 7 and Section 8 respectively. Concluding remarks are given in Section 9.

2. Preliminaries

A transaction in this paper is a set of database operations, which has four well known ACID properties: atomicity, consistency, isolation, and durability [19]. In order to perform functions of a transaction at different sites, the transaction has one or more processes executing of its behalf at each of the corresponding sites. We call these processes the *agents* of the transaction. The agent that starts a transaction is called the *root agent* of the transaction, and the site of the root agent is called the *site of origin* of the transaction. An agent can initiate other agents for its own purpose. In that case, the former is called a *master agent* and the latter is called *slave agents*. An agent can have at most one master agent(of course, the root agent of the transaction cannot have its master agent) and at the same time can have several slave agents of its own. In other words, all the agents of a transaction constitute a tree structure that is denoted as a process tree [17]-[19], [25], [26]. An agent carries a transaction identifier, an identifier of the process that executes the part of the transaction, and a site identifier that denotes the site where the process is running. 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 executing by blocking the others as in [25], [39].

For ensuring serializability, our model uses strict *two-phase locking* [5], [11], [24], which requires that a transaction has to lock a resource before it accesses the resource and all locks of a transaction are held until the transaction terminates. We support multiple modes of locks(IS, IX, S, SIX, and X locks) [5], [15], [17]. Our model is upward compatible with the *multiple granularity locking* protocol [15], [17] in the sense that it integrates without changes into a system that supports a resource hierarchy.

Two lock requests for the same resource by two different transactions are said to be *compatible* if they can be granted concurrently. The *compatibility matrix*, say *Comp*, is shown in Table 1 where NL means No Lock. For short hand notation, we shall also use a function with the same name as the matrix to indicate the truth table. Comp[lock1, lock2] is true (false) if lock1 and lock2 are compatible (incompatible, respectively). For example, Comp[S, IS] is true but Comp[IX, SIX] is false.

	NL	IS	IX	SIX	S	X
NL	t	t	t	t	t	t
IS	t	t	t	f	t	f
IX	t	t	t	f	f	f
SIX	t	t	f	f	f	f
S	t	t	f	f	t	f
X	t	f	f	f	f	f

Table. 1 Compatibility Matrix

	NL	IS	IX	SIX	S	X
NL	NL	IS	IX	SIX	S	X
IS	IS	IS	IX	SIX	S	X
IX	IX	IX	IX	SIX	SIX	X
SIX	SIX	SIX	SIX	SIX	SIX	X
S	S	S	SIX	SIX	S	X
X	X	X	X	X	X	X

Table. 2 Conversion Matrix

A transaction that holds a resource might request a more exclusive lock mode on the same resource, which is denoted as *a lock conversion* or *a lock upgradation*. When a request turns out to be a conversion, the holding lock mode and the requesting lock mode of the transaction are used to compute a new lock mode that the transaction eventually wants to hold for the resource by use of a *conversion matrix*, say *Conv*. It is represented in Table 2 with the holding lock mode as rows and the requesting lock mode as columns. For example, when a transaction holds an IX lock on a resource and re-requests an S lock for the resource, the transaction eventually wants to hold an SIX lock (Conv[IX, S]) for the resource.

3. Scheduling Policy

In this section we review the scheduling policy that we introduced in [31] with some modifications. Our scheduling policy honors lock requests on a first-in-first-out basis except for lock conversion. In our scheme the *lock manager* of a site maintains a *lock table* that holds the following information for each resource being locked in the site: a holder list, a queue, a *total mode* of the holders (*tm_h*), and a *total mode* of requests in the queue (*tm_q*). Each holder in a holder list takes three attributes: a transaction identifier (*tid*), a granted lock mode (*gm*), and a blocked lock mode (*bm*), i.e. (*tid*, *gm*, *bm*); each request in a queue takes two attributes: a transaction identifier (*tid*) and a blocked lock mode (*bm*), i.e. (*tid*, *bm*); the total mode of the holders is defined as Conv[...Conv[Conv[Conv[gm_1, bm_1], gm_2], bm_2], ..., bm_n] assuming that n requests are in the holder list and each one takes (T_i, gm_i, bm_i) where $1 \leq i \leq n$; and the total mode of requests in the queue is defined as Conv[...Conv[Conv[bm_1, bm_2], bm_3], ..., bm_n] assuming that n requests are in the queue and each one takes (T_i, bm_i) where $1 \leq i \leq n$.

The notion of the total mode is introduced in order to allow for a more efficient checking of lock grantability as compared with the *group mode* concept of [17]. The reader should notice that the group mode of the holders corresponds in our notation to Conv[... Conv[Conv[gm_1, gm_2], gm_3], ..., gm_n].

A transaction can request resources while running. If a resource that a transaction wants to hold a lock is held by other transactions with conflicting modes of locks or the total mode of

the corresponding queue is not compatible with the requesting lock mode, then the transaction is suspended, i.e. blocked, until the request is granted. More specifically, when a request from transaction T_i asking lock L_i for resource R_x arrives, the holder list of R_x is checked first to see if the request is a lock conversion (i.e. T_i already holds a lock). In the case that the request is not a lock conversion, if Comp[tm_h, L_i] and Comp[tm_q, L_i], then T_i is notified that the request is granted after placing (T_i, L_i, NL) behind all the blocked holders in the holder list and updating *tm_h* of R_x by Conv[*tm_h*, L_i]. Otherwise, the request is not granted by appending an entry (T_i, L_i) at the end of the queue and updating *tm_q* of R_x by Conv[*tm_q*, L_i]. When a request turns out a lock conversion, i.e. there is an entry (T_i, gm_i, bm_i) in the holder list where bm_i is NL, a new lock mode is computed by Conv[gm_i, L_i]. If the new lock mode is compatible with the granted lock mode of every other holders in the holder list, then T_i is notified that the request is granted after substituting the new lock mode for gm_i, and updating *tm_h* of R_x by Conv [*tm_h*, L_i]. Otherwise, bm_i is replaced with the new lock mode, *tm_h* of R_x is updated by Conv [*tm_h*, L_i], the entry (T_i, gm_i, bm_i) is repositioned in the holder list according to the upgrader positioning rule that will be explained below, and T_i is blocked until the request is granted.

Example 3.1 Assume that resource R_1 is held by two transactions (T_1 with an IS lock and T_2 with an IX lock) and at the same time is waited by two other transactions (T_3 with an S lock and T_4 with an X lock) in the queue. According to our definition, *tm_h* and *tm_q* of R_1 become an IX lock and an X lock respectively. We shall use a schematic notation where for a given resource R_i we indicate R_i[tm_h] followed by the holder list and [tm_q] followed by the elements in the queue. Thus the present configuration corresponds to:

R1[IX]: Holder((T_1, IS, NL)(T_2, IX, NL))
[X]: Queue((T_3, S)(T_4, X))

Suppose that T_1 re-requests and S lock for R_1. Since an S lock (Conv[IS, S]: the new lock mode that T_1 wants to hold) is not compatible with an IX lock (the granted lock mode of T_2), T_i is blocked and bm_i and tm_h are replaced with the new lock mode and an SIX lock (Conv[IX, S]) respectively. The situation becomes as follows:

R1[SIX]: Holder((T_1, IS, S)(T_2, IX, NL))
[X]: Queue((T_3, S)(T_4, X)) □

There are two cases where we need to check whether there are some blocked requests that can be granted and if they are grantable we need to grant those requests at this point. The first case is that a member of the holder list is forced out due to commitment or abortion. The second case is that some member of the queue leaves the system due to abortion. When any one of the two cases takes place, we want to check the blocked requests in a systematic and efficient way. This is enabled in our scheme by repositioning the entries in the holder list, which denote lock conversion requests that

cannot be granted. More specially, when a request of transaction T_i for a resource is found to be a lock conversion and is blocked, we replace the bm_i value in its holder entry with the value Conv[gm_i, requesting lock mode], replace tm_h with Conv[tm_h, requesting lock mode], delete the entry (T_i, gm_i, bm_i) temporarily from the holder list and then we position the entry in the holder list according to the following *upgrader positioning rule*(UPR):

Starting from the beginning of the holder list, find the first three requests, if any, denoted as T_a, T_b, and T_c that satisfy the conditions below:

(T_a, gm_a, bm_a), where $bm_a \neq$NL and Comp[bm_a, bm_i].

(T_b, gm_b, bm_b), where Comp[gm_b, bm_i] and Not Comp[bm_b, gm_i].

(T_c, gm_c, bm_c), where bm_c=NL

(UPR-1) If T_a is found then place (T_i, gm_i, bm_i) as an immediate predecessor of (T_a, gm_a, bm_a) in the holder list.

(UPR-2) If T_a is not found but T_b is found then place (T_i, gm_i, bm_i) as an immediate predecessor of (T_b, gm_b, bm_b) in the holder list.

(UPR-3) If T_a and T_b are not found then place (T_i, gm_i, bm_i) as an immediate predecessor of (T_c, gm_c, bm_c) in the holder list.

Example 3.2 Assume the following situation of a resource R1. For the explanation of the upgrader positioning rule, only the holder list is shown.

R1[IX]: Holder((T_1, IX, NL)(T_2, IS, NL)
(T_3, IX, NL)(T_4, IS, NL))

When T_2 re-requests R_1 with an S lock, the request cannot be granted and its entry becomes (T_2, IS, S). According to UPR-3 the entry is rearranged to the beginning of the holder list.

R1[SIX]: Holder((T_2, IS, S)(T_1, IX, NL)
(T_3, IX, NL)(T_4, IS, NL))

When T_3 re-requests R_1 with an S lock, the request cannot be granted and its entry becomes (T_3, IX, SIX). According to UPR-2 the entry is rearranged to be the immediate predecessor of T_2' s request.

R1[SIX]: Holder((T_3, IX, SIX)(T_2, IS, S)
(T_1, IX, NL)(T_4, IS, NL))

When T_4 re-requests R_1 with an S lock, the request cannot be granted and its entry becomes (T_4, IS, S). According to UPR-1 the entry is rearranged to be the immediate predecessor of T_2' s request.

R1[SIX]: Holder((T_3, IX, SIX)(T_4, IS, S)
(T_2, IS, S)(T_1, IX, NL))

In fact T_2 and T_4 cannot be granted in the presence of T_1 and T_3. When T_1 completes, T_3

can be granted even though T_2 and T_4 cannot be granted. □

The following lemmas and theorem follow as a consequence of the upgrader positioning rule. For the proofs see [31].

Lemma 3.1 Let (T_i, gm_i, bm_i) and (T_j, gm_j, bm_j) be two requests in the holder list of a resource where $bm_i \neq$ NL. If bm_i is compatible with gm_j, then gm_j is an IS lock.

Lemma 3.2 Let (T_i, gm_i, bm_i) and (T_j, gm_j, bm_j) be two blocked requests in the holder list of a resource. If bm_i is compatible with bm_j, then gm_i=gm_j is an IS lock and bm_i=bm_j is either an IX lock or an S lock.

Lemma 3.3 Let (T_i, gm_i, bm_i) and (T_j, gm_j, bm_j) be two blocked requests in the holder list of a resource. If gm_i is compatible with bm_j and bm_i is not compatible with gm_j, then (T_i, gm_i, bm_i) cannot precede (T_j, gm_j, bm_j) in the holder list.

Theorem 3.1 Let (T_i, gm_i, bm_i) and (T_j, gm_j, bm_j) be two blocked requests in the holder list of a resource where (T_i, gm_i, bm_i) precedes (T_j, gm_j, bm_j). When we grant blocked requests from the beginning of the holder list, once the request of T_i cannot be granted, the request of T_j cannot be granted either.

As a consequence of Theorem 3.1 it follows that when a holder is removed from the holder list of a resource R_x, tm_h of R_x is re-computed and the process of granting blocked requests can start from the beginning of the holder list and stops searching the holder list when it encounters an entry whose blocked request cannot be granted or when a non-blocked entry is found. Note that a blocked request can be granted only when its blocked lock mode is compatible with the granted lock modes of every other holders in the holder list. All the newly granted requests are put behind all the blocked holders after substituting the blocked lock mode for the granted lock mode and NL for the blocked lock mode. In addition to that, if the queue of R_x is not empty, after setting tm_q of R_x to NL, each request (T_i, bm_i) in the queue is checked starting from the first waiter up to the last one in the queue. Once bm_i is compatible with tm_h of R_x and also compatible with tm_q of R_x, the request is granted by removing the request from the queue, placing (T_i, bm_i, NL) behind all the blocked holders in the holder list and updating tm_h of R_x by Conv[tm_h, bm_i]. Otherwise, tm_q of R_x is replaced with Conv[tm_q, bm_i].

When some member of the queue leaves the system due to abortion, after removing the request from the queue, the same process of granting blocked requests as we did for the case of a holder deletion is carried out for the queue.

4. Transaction Wait-For Graph

We make use of an augmented transaction wait-for graph (TWFG) in order to detect deadlocks in the system. Our TWFG captures additional information in comparison with

other such graphs reported in literature [7], [9], [12], [29], [37], [38], namely message-wait relationships, hierarchy among agents of global transactions, and transaction state with regard to commitment protocol. Periodically, each site constructs its own up-to-date local TWFG by making use of its lock table plus other run-time information about its transactions as will be explained below. Thus the TWFG of the entire system is the union of the individual local TWFGs constructed at the corresponding sites.

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), LWS (lock-wait edges), MWS (message-wait edges), State (state of transaction), MTS (master agents), TAS (transitively antagonistic waits), ancestor (integer), and current (lock-wait edges). TAS is explained in Section 5, and *ancestor* and *current* are explained in Section 6 when cyles in TWFG are searched. We will explain LWS, MWS, State, and MTS in this section.

We assume that all the transactions running at a site can have their entries in the local TWFG. For notational convenience, when transaction T_i is running at site S_r, T_i has an entry in $TWFG_r$ and it is represented as $T_i \in TWFG_r$. Otherwise, it is shown as $T_i \notin TWFG_r$. Each entry of $TWFG_r$ can be accessed by a regular hashing method for the corresponding transaction identifier. Throughout the remainder of this paper, once $T_i \in TWFG_r$, we assume that T_i has its entry in the i^{th} position of $TWFG_r$. In other words, we assume that $TWFG_r[T_i]$ and $TWFG_r[i]$ are interchangeable. In addition to that the terminology *vertex* will be used to represent a transaction when it is appropriate for the explanation of our graph structure.

4.1 Lock-Wait Edges

In distributed database systems due to the fact that transactions can be executed at multiple sites concurrently, special attention needs to be devoted to the representation of the system with respect to deadlock. In the case of parallel transaction processing where a transaction can have multiple outstanding lock requests, it is possible that a transaction is not currently blocked, yet it may become deadlocked later in time if the agent that currently executes terminates successfully and the remaining agents are blocked.

Different definitions of deadlock have appeared in literature: one definition declares a deadlock when an agent transitively waits for itself [7], [12], [13], [20], [38], while another declares when a transaction transitively waits for itself [4], [10], [29], [33], [35]. The first definition is useful when nested transactions [28] are allowed, but when only conventional transactions are considered, the latter definition allows for earlier deadlock detection. We shall adopt this definition of a deadlock. More precisely, a deadlock occurs currently or will occur in the future if there exists a set of transactions $\{T_0, T_1, ..., T_{n-1}\}$ such that for any two transactions T_i and T_j in the set, T_i is waiting for the completion fo T_j, where $0 \leq i < n$, $j=(i+1) \bmod n$, and $n \geq 1$. We call the set of

transactions *a deadlock set.*

We classify deadlocks into two types: explicit deadlocks and implicit deadlocks. An *explicit deadlock* is a current deadlock due to the fact that for every transaction T_i in a deadlock set, there exists a transaction T_j in the deadlock set such that T_j holds a resource and T_i transitively waits for T_j at this resource. An implicit deadlock is a deadlock that will occur in the future due to the fact that there exists at least one transaction T_i in a deadlock set such that for every resource that T_i is waiting to be granted, the holders of the resource that are transitively awaited by T_i are not included in the deadlock set. In our scheme we detect not only explicit deadlock but also implicit deadlocks in the system. The reason might become clear through the following example.

Example 4.1 Assume the following situation at two different sites S_r and S_s.

At S_r,
R1[X]: Holder (T_1, X, NL))
[X]: Queue ((T_2, X)(T_3, X))
At S_s,
R2[X]: Holder ((T_3, X, NL))
[X]: Queue ((T_2, X)(T_1, X))

According to our definition of deadlock and our scheduling policy, there is an explicit deadlock between T_1 and T_3 because T_1 and T_3 wait for each other and they are holders of R_1 and R_2 respectively. In addition to the explicit deadlock, there are two implicit deadlocks: one between T_2 and T_3 (because T_2 waits for the completion of T_3 at R_2, T_3 waits for the completion of T_2 at R_1, and T_2 is not a holder of R_1) and the other between T_1 and T_2 (becausee T_1 waits for the completion of T_2 at R_2, T_2 waits for the completion of T_1 at R_1, and T_2 is not a holder of R_2). Assume that T_1 is aborted by some reason. Then according to our scheduling policy, T_2 becomes a holder of R_1 and we obtain an explicit deadlock between T_2 and T_3.

Suppose that all the lock-wait information in S_r and S_s are gathered at site S_t. For each request (T_i, X) in a queue of a resource, lock-wait edges can be constructed according to the following three different schemes as far as only X locks are concerned:

Scheme-1: Construct an edge from T_i to the transaction in the holder list.

Scheme-2: If (T_i, X) is the first request in the queue, construct an edge from T_i to the transaction in the holder list. Otherwise, construct an edge from T_i to the transaction whose request immediately precedes (T_i, X) in the queue.

Scheme-3: Construct edges from T_i to the transaction in the holder list and to every transaction whose request precedes (T_i, X) in the queue.

Depending on the edge construction scheme, we can have three different graphs as shown in Figure 4.1, where label on each edge is put to show at which site the edge is constructed (R for S_r and S for S_s).

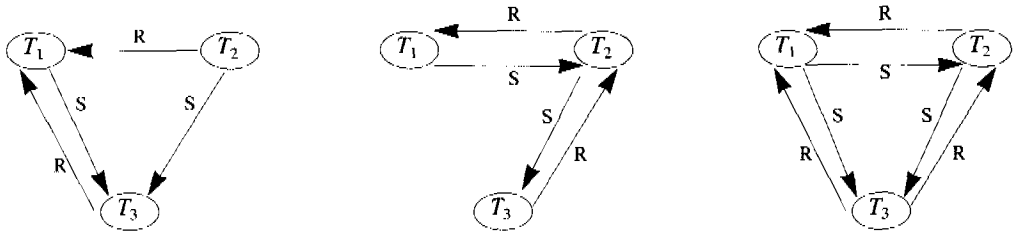

Figure 4.1 Three different TWFGs

Scheme-1, which is taken by [13], [16], [38], represents only one cycle (an explicit deadlock). When we abort either T_1 or T_3 as a victim, one deadlock still remains. For the detection of the remaining deadlock we have to wait until the graph is modified to reflect the system state correctly. In Scheme-2, which is used in [2], [3], when we select T_2 as a victim, a deadlock still remains between T_1 and T_3. For the resolution of the remaining deadlock we have to wait as in Scheme-1. Scheme-3 represents the exact status of the system by showing all deadlocks: one explicit deadlock and two implicit deadlocks. We will now show that Scheme-3 can be formally defined for a system that allows multiple modes of locks and supports lock conversion. □

Definition 4.1 A transaction T_i lock-waits for another transaction T_j at site S_r, denoted as LW_r (T_i, T_j), if a request of T_i for a resource at site S_r cannot be granted under the presence of a request of T_j for the resource.

When $LW_r(T_i, T_j)$ holds, a lock-wait edge from a vertex representing T_i to a vertex of T_j can be added into $TWFG_r$. Lock-wait edges for $TWFG_r$ are constructed by the following lock-wait edge construction rule:

(1) Let two requests in the holder list of a resource at site S_r be (T_i, gm_i, bm_i) and (T_j, gm_j, bm_j) such that (T_i, gm_i, bm_i) precedes (T_j, gm_j, bm_j). If Not Comp[gm_i, bm_j] or Not Comp[bm_i, bm_j], then $LW_r(T_j, T_i)$ holds. If Not Comp[gm_j, bm_i], then $LW_r(T_i, T_j)$ holds.

(2) For each request (T_i, gm_i, bm_i) in the holder list of a resource at site S_r and for each request (T_j, bm_j) in the queue of the resource, if bm_j is not compatible with either gm_i or bm_i, then $LW_r(T_j, T_i)$ holds.

(3) Let two requests in the queue of a resource at site S_r be (T_i, bm_i) and (T_j, bm_j) such that (T_i, bm_i) precedes (T_j, bm_j). If bm_i is not compatible with bm_j, then $LW_r(T_j, T_i)$ holds.

Lock-wait edges from requests in the queue to other requests in the queue or to the holders would suffice if no lock conversion is allowed. Rule (1) is added to account for lock-wait edges due to lock conversion. According to our upgrader positioning rule and scheduling policy the holders currently granted are at the end of holder list and have a blocking mode equal to NL, while the blocked holders are arranged at the beginning of list such that if T_i

precedes T_j this implies that T_j cannot be granted before T_i. Hence, rule (1) has two parts to account for the lock-wait edges going between blocked holders and unblocked ones.

Example 4.2 Assume the following situation at site S_r.

R1[SIX]: Holder ((T_1, IX, SIX)(T_2, IS, S)
(T_3, IX, NL)(T_4, IS,NL))

[SIX]: Queue ((T_5, IX)(T_6, S)(T_7, IX))

According to rule 1 of lock-wait edge construction rule, $LW_r(T_1, T_3)$, $LW_r(T_2, T_1)$, and $LW_r(T_2, T_3)$ hold. From rule 2, $LW_r(T_5, T_1)$, $LW_r(T_5, T_2)$, $LW_r(T_6, T_1)$, $LW_r(T_6, T_3)$, $LW_r(T_7, T_1)$, and $LW_r(T_7, T_2)$ hold. From rule 3, $LW_r(T_6, T_5)$ and $LW_r(T_7, T_6)$ hold. The resulting graph is shown in Figure 4.2. □

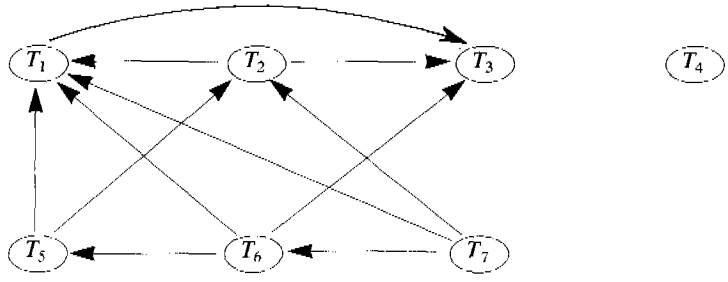

Figure 4.2 Lock-wait edges for Example 4.2

When some transactions are aborted during deadlock resolution all the locks they held should be released and some transactions that are waiting for these locks can be granted. In the algorithms based on the lock-wait edge construction rules corresponding to Schemes 1 and 2 discussed in Example 4.1 some new lock-wait edges might have to be added and deadlock detection has to be performed again; this process is repeated until no more deadlocks are found [12], [13]. Using our lock-wait edge construction rule once lock-wait edges are constructed at deadlock detection and resolution time, no new lock-wait edges need to be included in the graph during the time. This amounts to only one invocation of the deadlock detection and resolution process and hence is more time efficient.

4.2 Transaction Status

When a transaction process lock-waits for another transaction process, the process is blocked until the request can be granted. In order to enable parallelism in distributed database systems, an agent of a transaction can proceed with its computation after sending some requests to agents of the transaction at remote sites. The fact that an agent is waiting for a reply from another agent at a remote site and the latter is owing a reply to the former are represented by message-wait relationships. Unlike lock-wait relationships, message-wait relationships are not exclusively blocking. The concept of message-wait is introduced by the following two definitions.

Definition 4.2 An agent of T_i in S_s message-waits for another agent of T_i in S_r, denoted as $MW_s(S_s, S_r, T_i)$, if the former has sent a request or a response to the latter and has not received a response or a new request respectively from the latter yet.

Definition 4.3 An agent of T_i in S_r owes a message-wait to another agent of T_i in S_s, denoted as $MW_r(S_s, S_r, T_i)$, if the former has received a request or a response from the latter and has not sent a response or a new request respectively to the latter yet.

The message-wait in Definition 4.2 is denoted as an *outgoing* message-wait from the viewpoint of the waiting agent. On the other hand, the message-wait in Definition 4.3 is denoted as an *incoming* message-wait from the viewpoint of the awaited agent. These message-wait relationships are used for two purposes: one for conveying the deadlock detection messages from the site of the waiting agent to the site of the awaited agent and the other for checking whether a transaction is a global transaction or not. Note that we do not represent identity of agents in a message-wait relationship. A slave agent message-waits for its master agent only when the former finished its assignment from the latter. However, a master agent message-waits for its slave agents after it gave new assignments until they send the results back to the master agent.

To enable early detection of some deadlocks in an environment where parallel execution of transactions are permitted, we represent master-slave relationships among agents of a transaction. When an agent of T_i in site S_r is a slave agent of another agent of T_i in site S_s, that relationship is denoted as $MT_r(S_s, T_i)$. Note that as in the case of message-wait relationship, we do not care about identity of the agents in a master-slave relationship. In other words, when some agents of a transaction T_i in site S_A are initiated by some agents of T_i in sites S_B and S_C, that relationships are represented by $MT_A(S_B, T_5)$ and $MT_A(S_C, T_5)$. Note also that this information is represented only at the site where the slave agent resides since it is used for conveying the deadlock detection messages from the site where the slave agent resides to the site where the master agent resides.

We will use message-wait relationships (master-slave relationships) and message-wait edges (master-slave edges, respectively) interchangeably, since they are used for sending deadlock detection messages. Figure 4.3 shows lock-wait, message-wait, and master-slave edges among transactions in three sites S_A, S_B, and S_C. Solid arrows in a site represent lock-wait edges. Message-wait edges are shown near the corresponding transactions and dotted arrows connecting transactions in two different sites are added to make them more readable. For instance, the dotted arrow from T_4 in S_A to T_4 S_C signifies $MW_A(S_A, S_C, T_4)$ and $MW_C(S_A, S_C, T_4)$. The sites of master agents are indicated near the corresponding transactions and solid arrows connecting transactions in two different sites are added to make them more readable. For example, the solid arrow from T_4 in S_B to T_4 in S_A signifies $MT_C(S_A, T_4)$.

When a local TWFG is constructed during our periodic deadlock detection time, our dead-

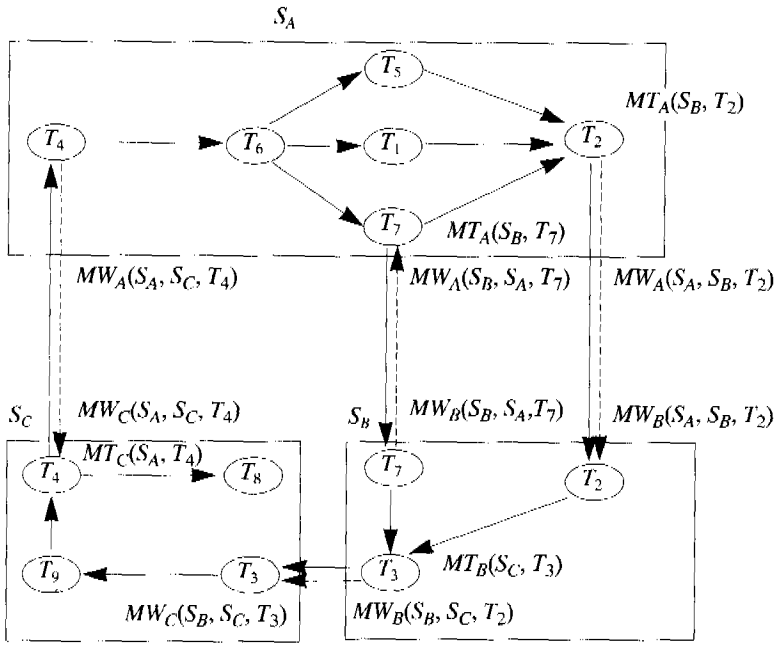

Figure 4.3 TWFGs in three sites

lock detector augments the graph also with information regarding the transactions status with regard to the execution of the commitment protocol. We assume that the two-phase commit (2PC) protocol [5], [6], [17], [25] is employed. After a global transaction completes its execution, we enter the voting phase of the protocol that starts by having an agent, called the coordinator, polls the other agents, called participants, by sending them a VOTE-REQ message. A participant responds by sending a YES or NO message depending upon whether it is ready to commit; if it responds NO then it aborts the transaction. In the decision phase of the protocol, the coordinator gathers all the votes from the participants. If they all YES vote he sends a COMMIT message to all participants; otherwise the coordinator sends an ABORT message to all participants that voted YES.

Definition 4.4 A global transaction in a site can be in one of three different states defined below in connection with 2PC protocol.

(1) When any agent of a transaction in a site has not received or not sent any message related to 2PC (either VOTE-REQ, ABORT, or COMMIT message), the transaction in the site is in *active* state.

(2) When some agent of a transaction in a site has decided to abort the transaction (received or sent either ABORT message or NO vote), the transaction in the site is in *abort* state.

(3) When some agent of a transaction in a site has received or sent some messages related to 2PC (either VOTE-REQ or COMMIT message) but no agent of the transaction in the site has decided to abort the transaction, the transaction in the site is in *pending* state.

When a transaction is in *pending* state, the transaction cannot be involved in any deadlock because the transaction has completed its execution such that all deadlock detection messages related to the transaction can be neglected. Once a transaction is in *abort* state however, the transaction has to be aborted at all sites for the atomicity of global transactions and all deadlock detection messages related to the transaction have to be canceled. A local transaction can be in either *active* or *abort* state.

We discuss now how these additional relationships can be reflected in the local TWFG at site S_r. For each lock-wait tuple $LW_r(T_i, T_j)$ a record of the form (T_j, link) is added to the list $TWFG_r[T_i]$.LWS. For each master-slave tuple $MT_r(S_m, T_i)$ a record of the form (S_m, link) is added to the list $TWFG_r[T_i]$.MTS. Similarly, for each message-wait tuple $MW_r(S_m, S_n, T_i)$ a record of the form (S_m, S_n, link) is added to the list $TWFG_r[T_i]$.MWS. Thus, in our context a transaction T_i that executes at site S_r is declared as a global transaction if $TWFG_r[T_i]$.MWS is not null. Otherwise, it is declared as a local transaction.

5. Message Transmission

In order to facilitate the detection of global deadlocks, inter-site messages called probes are sent as in [10], [20], [33]-[35], [37]. A probe, denoted as PB(*initiator*, *terminus*, *sender*), is a message from the *sender* site to the receiving site to inform it that the transaction called *initiator* has been transitively waiting for the transaction *terminus* at the sender site. For each probe sent, a site keeps a receipt of the probe in order to avoid retransmission of the probe. The receipt of the probe is denoted as PB(initiator, terminus, receiver). Correspondingly, each site maintains two pools: one pool of probes received (RPP) and one pool of receipts of probes sent (SPP). Before we present the generation of probes and the maintenance of those two pools of probes in detail, some preliminary explanations are given first.

Until now we didn' t say anything about how to assign transaction identifiers. In order to guarantee uniqueness of a transaction identifier we assume that each identifier consists of two fields: one denoting the site at which the transaction is originated and the other containing the value of the local clock at that site when the transaction was generated. Based on this assumption, a unique priority can be assigned to each transaction as follows. For two transactions T_i and T_j, T_i has higher priority than T_j, denoted as $T_i \rangle T_j$, if either local clock of T_i is greater than that of T_j or local clock of T_i equals 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 $T_i \rangle T_j$ if i $\rangle$ j: for instance, $T_5 \rangle T_3$.

During deadlock detection time, we declare that there is a deadlock in the system when either a cycle in a local TWFG is found or there exists a probe PB(initiator, terminus, sender) in RPP and also there is a path from the terminus to the initiator in the local TWFG. Once a deadlock is found, a transaction with

the highest priority in the cycle or on the path is selected as a victim to be aborted and the state of the victim becomes abort.

Definition 5.1 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 5.2 A transaction T_i waits for another transaction T_j *transitively and antagonisitically* at site S_s, denoted as $TA_s(T_i, T_j)$, if all of the following hold:

(1) T_i is antagonistic with T_j.

(2) If $T_i \in TWFG_s$, then $TWFG_s[T_i]$.State is *active*.

(3) $T_j \in TWFG_s$ and $TWFG_s[T_j]$.State is *active*.

(4) At least one of the following holds.

4-a) $LWs(T_i, T_j)$.

4-b) for some site S_r, $PB(T_i, T_j, S_r) \in RPP_s$.

4-c) for some transaction T_k, $TA_s(T_i, T_k)$ and $LW_s(T_k, T_j)$.

When the condition 4-a of Definition 5.2 is satified while satisfying conditions (1) - (3), we call the lock-wait an *antagonistic* lock-wait. Conditions 4-a and 4-b above are called TA *triggering conditions* and condition 4-c is called TA *propagation condition*. TAs are constructed during our periodic deadlock detection time and at that time for each tuple $TA_s(T_i\ T_j)$ a record of the form (T_i, link) is added to the list $TWFG_s[T_j]$.TAS.

The generation of probes and the corresponding actions taken are governed by the following *PB_Transmission Rule*.

PB-1. During deadlock detection time at site S_s, probe $PB(T_i\ T_j, S_s)$ is sent from site S_s to another site S_r and a receipt $PB(T_i, T_j, S_r)$ is stored at SPP_s if all of the following hold:

(1) $TA_s(T_i, T_j)$

(2) $PB(T_i, T_j, S_r) \notin RPP_s \cup SPP_s$.

(3) either $MW_s(S_s, S_r, T_j)$ or $MT_s(S_r, T_j)$.

PB-2. Upon receiving probe $PB(T_i, T_j, S_s)$ at site S_r, it is stored at RPP_r.

Since we store all probes received in RPP and the receipts of probes sent in SPP we can avoid retransmission of the same probe by checking whether the receipt of the probe is already in SPP. The reason why we check RPP in condition (2) of PB-1 is to guarantee that at least at the time of the probe generation $TA_r(T_i, T_j)$ does not hold at site S_r. According to condition (3) of PB-1, we send probes from the site of the waiting agent to the sites where the awaited agents of the waiting agent reside. While this condition is included also in other probe generation scheme [37], we also send a probe to the site where the master agent of the agent of the terminus transaction resides. This enables us to detect some deadlocks earlier in an environment where we have parallel execution of transactions.

Example 5.1 In Figure 4.3, assume that all the probe pools in three sites are empty. When deadlocks are checked at S_A, there is not any deadlock but there exist two anatgonistic lock-waits: $LW_A(T_4, T_6)$ and $LW_A(T_7, T_2)$. Based on this, the following TAs can be constructed: $TA_A(T_4, T_6)$, $TA_A(T_4, T_5)$, $TA_A(T_4, T_1)$, $TA_A(T_4, T_2)$, and $TA_A(T_7, T_2)$. Note that $TA_A(T_4, T_7)$ does not

hold since T_4 and T_7 are global transactions and $T_4 \langle T_7$. According to PB_Transmission Rule, probes PB(T_4, T_2, S_A) and PB(T_7, T_2, S_A) are sent to S_B, and probe receipts PB(T_4, T_2, S_B) and PB(T_7, T_2, S_B) are stored at SPP_A.

Assume that deadlocks are checked at S_B before those two probes sent from S_A are stored at RPP_B. Then no deadlocks are found and owing to the antagonistic lock-wait edge LW_B (T_7, T_3) and an outgoing message-wait edge $MW_B(S_B, S_C, T_3)$, probe PB(T_7, T_3, S_B) is sent to S_C and probe receipt PB(T_7, T_3, S_C) is stored SPP_B.

When deadlocks are checked at S_B again after those two probes sent from S_A are stored at RPP_B, since $TA_B(T_4, T_3)$ and $MW_B(S_B, S_C, T_3)$ hold, probe BB(T_4, T_3, S_B) is sent to S_C and probe receipt PB(T_4, T_3, S_C) is stored at SPP_B. Even though $TA_B(T_7, T_3)$ and $MW_B(S_B, S_C, T_3)$ hold, we do not send PB(T_7, T_3, S_B) to S_C because probe receipt PB(T_7, T_3, S_C) is already in SPP_B. Assuming that those two probes sent from S_B are stored at RPP_C, probes and probe receipts stored up to now are shown in Figure 5.1.

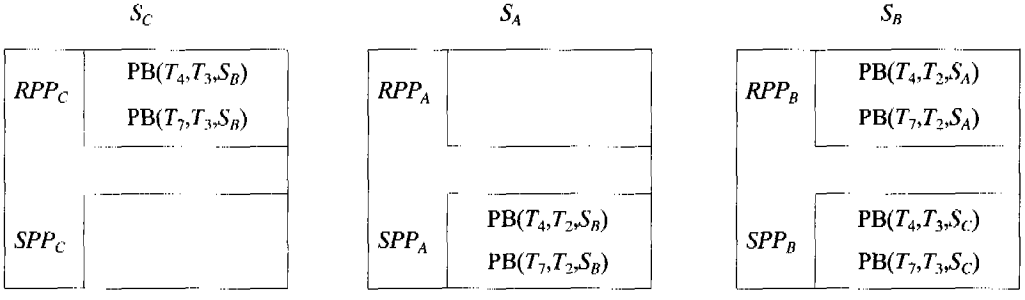

Figure 5.1 Probe pools with an optimal solution

When deadlocks are checked at S_C after receiving both of the two probes, since there is a probe PB(T_4, T_3, S_B) in RPP_C and also there is a path from T_3 to T_4 in $TWFG_C$, a deadlock ($T_4 \rightarrow T_3 \rightarrow T_9 \rightarrow T_4$) is found and T_9 that has the highest priority on the path is aborted. Note that the cycle is not an actual cycle. In fact it is a condensed one for two cycles in the system: one with $T_4 \rightarrow T_6 \rightarrow T_5 \rightarrow T_2 \rightarrow T_3 \rightarrow T_9 \rightarrow T_4$ and another with $T_4 \rightarrow T_6 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_9 \rightarrow T_4$. Wait-for relationships on site S_A are condensed by the probe PB(T_4, T_2, S_A) in RPP_B and those of site S_A and site S_B are condensed by the probe PB(T_4, T_3, S_B) in RPP_C. By aborting T_9 other two deadlock cycles in the system (one with $T_4 \rightarrow T_6 \rightarrow T_7 \rightarrow T_2 \rightarrow T_3 \rightarrow T_9 \rightarrow T_4$ and another with $T_4 \rightarrow T_6 \rightarrow T_7 \rightarrow T_3 \rightarrow T_9 \rightarrow T_4$) are also resolved. In this case, 4 messages(4 probes) are sent for detection and resolutoin of deadlocks in the system. As far as current situation is concerned an optimal solution is obtained, i.e. all deadlocks in the system are resolved by aborting only one transaction T_9 that is the transaction with the highest priority

in every deadlock cycle in the system. □

Definition 5.3 Probe PB(T_i, T_j, S_s) in RPP_r is a stale probe if one of the following holds:

(1) $T_i \in TWFG_r$ and $TWFG_r[T_i]$.State ≠ active.

(2) $T_j \notin TWFG_r$.

(3) $T_j \in TWFG_r$ and $TWFG_r[T_j]$.State ≠ *active*.

Definition 5.4 Probe receipt PB(T_i, T_j, S_r) in SPP_s is a stale probe receipt if one of the following holds:

(1) $T_j \notin TWFG_s$.

(2) $T_j \in TWFG_s$ and $TWFG_s[T_j]$.State ≠ active.

(3) $T_j \in$ TWFG, $TWFG_s[T_j]$.State=*active*, and $TA_s(T_i, T_j)$ does not hold.

The terminus transaction of a probe or a probe receipt does not exist in the corresponding site when the transaction has been either committed or aborted already. We call those probes and probe receipts that are not stale the active probes and probe receipts respectively. Checking whether a probe or a probe receipt is a stale one is done at three different times: (1) during the deadlock detection time, (2) upon receiving a compensating message, and (3) when a global transaction is aborted. Once a probe (receipt) is found to be a stale one, it is removed from the corresponding pool. When condition (3) of Definition 5.4 holds, in addition to the deletion of the probe receipt from SPP, a compensating message has to be sent against the corresponding probe sent. There is no need to send any compensating message in other conditions of Definition 5.4 since any agent of the terminus transaction at a different site is guaranteed to be in the same destiny, i.e. either committed or aborted due to the atomicity of distributed transactions.

To compensate for probes sent, inter-site messages called *antiprobes* [33], [34] are introduced. An antiprobe, denoted as AP (*initiator*, *terminus*, *sender*, *status*), is a message from the *sender* site to inform the receiving site of the message that the transaction *terminus* in the *sender* site is no longer waited by the transaction *initiator* by some reason marked at *status* that is either *abort* or *active*. The generation of antiprobes and the corresponding actions taken are governed by the following *AP_Transmission Rule*.

AP-1. During deadlock detection time at site S_s, for each probe receipt PB(T_i, T_j, S_r) in SPP_s, if $T_j \in$ TWFG, $TWFG_s[T_i]$. State is active, and $TA_s(T_i\ T_j)$ does not hold, then the receipt is removed from SPP_s and antiprobe AP(T_i, T_j, S_s, status) is sent to site S_r. If $T_i \in TWFG_s$ and $TWFG_s[T_i]$.State is *abort*, then the status of the antiprobe becomes *abort*. Otherwise, it becomes *active*.

AP-2. Upon receiving antiprobe AP(T_i, T_j, S_s, *active*) at site S_r, if probe PB(T_i, T_j, S_s) exists in RPP_r, then the probe is removed from the pool.

AP-3. Upon receiving antiprobe AP(T_i, T_j, S_s, abort) at site S_r or when a global transaction T_i at site S_r is aborted, we do both of the following:

(1) for each probe PB(T_k, T_m, S_p) in

RPP_r, if either T_k or T_m is equal to T_i, then the probe is deleted from RPP_r.

(2) for each probe receipt PB(T_k, T_m, S_p) in SPP_r, if either T_k or T_m is equal to T_i, then the receipt is deleted from SPP_r and only when T_k is equal to T_i, antiprobe AP(T_i, T_m, S_r, *abort*) is sent to site S_p.

Note that even though deadlock detection is done periodically, actions taken when a global transaction is aborted or upon receiving probes and antiprobes are done continuously.

Example 5.2 Consider Figure 4.3 again assuming that all the probe pools in three sites are empty and deadlocks are checked at S_A and S_B as in Example 5.1 such that probes PB(T_4, T_2, S_A) and PB(T_7, T_2, S_A) are sent from S_A to S_B; and probes PB(T_4, T_3, S_B) and PB(T_7, T_3, S_B) are sent from S_B to S_C. Different from Example 5.1, assume that deadlocks are checked at S_C right after probe PB(T_7, T_3, S_B) is stored at RPP_C but before probe PB(T_4, T_3, S_B) is received. then $TA_C(T_7, T_4)$ holds since PB(T_7, T_3, S_B) is in RPP_C and there is a path from T_3 to T_4 in $TWFG_C$. According to that and $MT_C(S_A, T_4)$, probe PB(T_7, T_4, S_C) is sent to S_A and probe receipt(T_7, T_4, S_A) is stored at SPP_C. Assuming that all probes sent up to now are received at their destination sites, probes and probe receipts stored are shown in Figure 5.2.

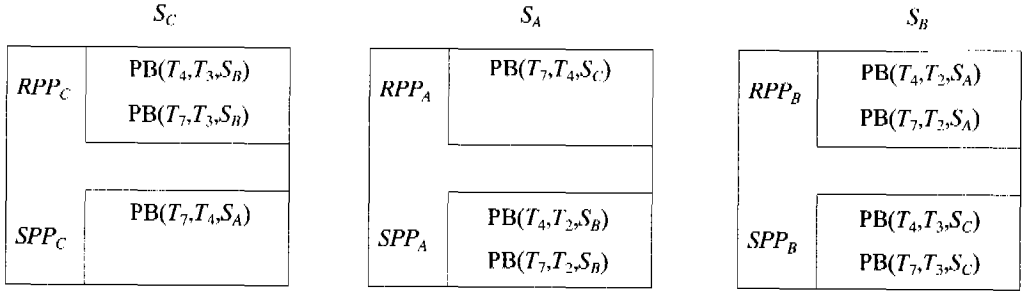

Figure 5.2 Probe pools before any transaction abortion

When deadlocks are checked again at S_C after receiving probe PB(T_4, T_3, S_B), a cycle ($T_4 \rightarrow T_3 \rightarrow T_9 \rightarrow T_4$) is found and T_9 that has the highest priority in the cycle is aborted. In this situation probe receipt PB(T_7, T_4, S_A) in SPP_C becomes a stale one because $TA_C(T_7, T_4)$ does not hold any more. According to AP_Transmission Rule the probe receipt is deleted and antiprobe AP(T_7, T_4, S_C, *active*) is sent to S_A. Depending on the arrival time of the antiprobe and the deadlock detection time at S_A, the following two cases are possible:

(Case 1) If the antiprobe sent from S_C arrives before another periodic deadlock detection time comes at S_A, probe PB(T_7, T_4, S_C) in RPP_A is removed and no more message transmission is required. In this case, 6 messages (5 probes and 1 antiprobe) are sent for the detection and resolutioin of deadlocks. The remaining probes and probe receipts in each probe pool are shown in Figure 5.3.

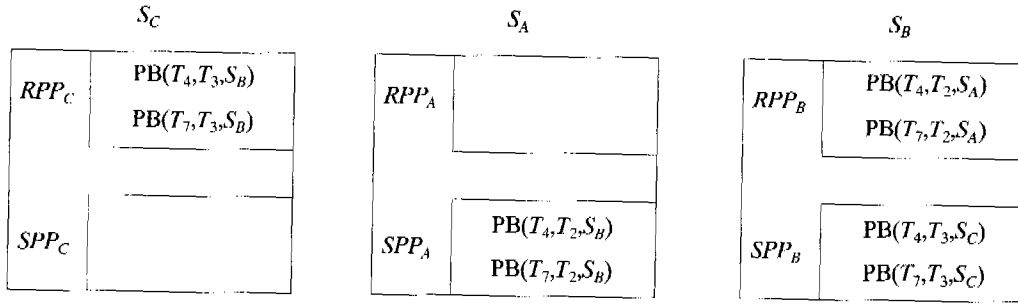

Figure 5.3 Probe pools after abortion of T_9

(Case 2) Assume that deadlock detection at S_A is done right after probe PB(T_7, T_4, S_C) is received but before the antiprobe from S_C arrives. By considering RPP_A and $TWFG_A$, a deadlock ($T_7 \rightarrow T_4 \rightarrow T_6 \rightarrow T_7$) is found and T_7 is aborted. According to this the state of transaction T_7 in $TWFG_A$ becomes abort and $TA_A(T_7, T_2)$ does not hold anymore. Because T_7 is selected as a victim, PB(T_7, T_4, S_C) in RPP_A and PB(T_7, T_2, S_B) in SPP_A become stale and are removed from their corresponding pools. In addition to that antiprobe AP(T_7, T_2, S_A, *abort*) is sent to S_B. Actually this is a false deadlock because there is not any more deadlock in th system once T_9 is aborted at S_C. Later when the antiprobe from S_C arrives, it is simply neglected because there is not any matching probe in RPP_A. At S_B, upon receiving the antiprobe sent from S_A, because the state of the antiprobe is abort, we do all of the following three actions: PB(T_7, T_2, S_A) in RPP_B is removed, PB(T_7, T_3, S_C) in SPP_B is removed, and antiprobe AP(T_7, T_3, S_B, *abort*) is sent to S_C. At S_C, upon receiving the antiprobe sent from S_B, PB(T_7, T_3, S_B) in RPP_C is removed and no more message transmission follows. In this case, 8 messages (5 probes and 3 anti-probes) are sent for detection and resolutioin of deadlocks. The remaining probes in each probe pool are shown in Figure 5.4.

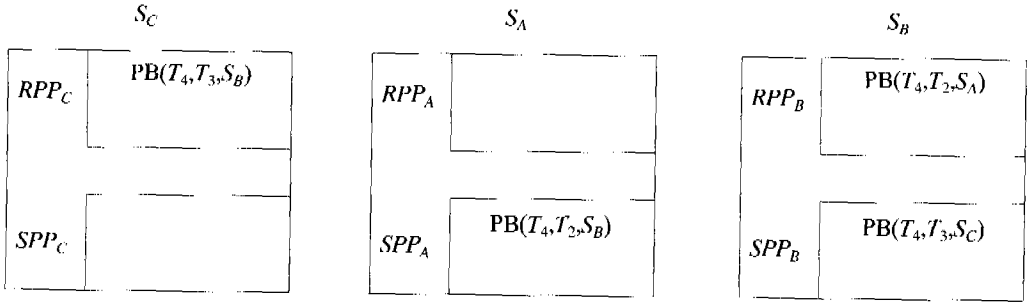

Figure 5.4 Probe pools after abortion of T_9 and T_7

For the rest of this example, we will continue deadlock detection and resolution based on the result of Case 1 above, i.e. Figure 5.3. Suppose that transaction T_3 at S_C requests a lock for a

resource that T_4 holds with a conflicting lock mode. When a next deadlock detection time comes up, owing to PB(T_4, T_3, S_B) in RPP_C and $LW_C(T_3, T_4)$, a cycle ($T_4 \rightarrow T_3 \rightarrow T_4$) is found, T_4 is aborted, and PB(T_4, T_3, S_B) in RPP_C is removed. Once T_4 is aborted at S_C, it is also aborted at S_A for the atomicity of a global transaction. When T_4 is aborted at S_A, according to AP-3 of AP_Transmission Rule, probe receipt PB(T_4, T_2, S_B) in SPP_A is removed and antiprobe AP(T_4, T_2, S_A, *abort*) is sent to S_B. Upon receiving the antiprobe at S_B, because the state of the antiprobe is avort, PB(T_4, T_2, S_A) in RPP_B and PB(T_4, T_3, S_C) in SPP_B are removed and antiprobe AP(T_4, T_3, S_B, *abort*) is sent to S_C. Upon receiving the antiprobe at S_C, because there is not any probe in RPP_C whose initiator is the same as that of the antiprobe, the antiprobe is simply neglected. The result of this situation is given in Figure 5.5.

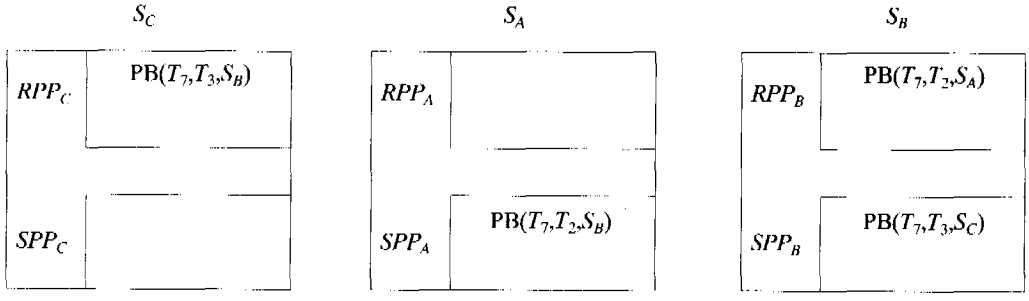

Figure 5.5 Probe pools after abortion of T_9 and T_4

For the completeness of our example, assume that transaction T_3 has completed its job and committed. Based on the situation in Figure 5.5, when deadlock detection time comes up at S_B and S_C, PB(T_7, T_3, S_C) in SPP_B and PB(T_7, T_3, S_B) in RPP_C become stale and are removed because T_3 does not have its entry in $TWFG_B$ and $TWFG_C$. Note that we do not send any antiprobes for the removal of those probes and probe receipts. The same concept can be applied to other transactions such that eventually all the pools of probes and probe receipts in the system can be cleared. □

As can be seen from the above example, our scheme is not immune from detecting false deadlocks. Trying not to detect any false deadlock in the distributed deadlock detection environment might cause some serious performance degradation as in [37]. If the probability of false deadlock detection might be so low, it can be a good idea to assume that whenever a deadlock is declared it is a real one.

Actually our probe generation scheme is very similar to the one used in [37] (also used for the hierarchical deadlock detection in [27]) where the detection message is sent when a global transaction transitively waits for another global transaction. However, our scheme differs from them by considering relationships among agents of global transactions, status of transactions in each site, and priorities of

transactions for the formulation of probes and antiprobes.

6. Deadlock Detection and Resolution

Deadlock detection and resolution in our scheme is done periodically and each site may or may not have the same period. In the remainder of this section we assume that our deadlock detection and resolution scheme is explained for site S_r without loss of generality. In other words all subscripts representing the site identifier will be omitted when it does not cause any confusion such that $TWFG_r$, RPP_r, SPP_r, LW_r, MW_r, TA_r, and MT_r are represented by TWFG, RPP, SPP, LW, MW, TA, and MT respectively. The overall process of our periodic deadlock detection and resolution scheme can be summarized as the following five steps. Each step will be refined shortly.

Algorithm **periodic_detection_resolution**
Step 1: Construct TWFG.
Step 2: Remove stale probes and reflect some active probes into TWFG.
Step 3: Detect deadlocks and select victims.
Step 4: Remove stale probes and construct TAs.
Step 5: Remove stale probe receipts, send probes and antiprobes, and abort victims

In Step 1 TWFG is constructed based on the schemes given in Section 4. Lock-wait edges are added to TWFG by applying lock-wait edge construction rule for each resource in the lock table. State of transactions, master-slave edges, and message-wait edges are constructed into TWFG by interogating all the transactions running at the current site.

Our algorithm employs two variables *ancestor* and *current* for each entry in TWFG. The *ancestor* that is initialized to 0 is used to mark traversed vertices in the current subgraph as well as is used for backtracking of the depth-first-search algorithm [14]. The *current* of a vertex that is initialized to *LWS* of the vertex indicates the next lock-wait edge to be searched. Note that once *State* of a transaction is marked as either *pending* or *abort*, *MTS* and *MWS* for it are set to null because probe sending is not related to that transaction. The whole process of Step 1 is refined below.

```
Step 1: /* Construct TWFG */
   abortion_set := { };
   for v :=1 to N do begin
      TWFG[v].TID := null; TWFG[v].LWS := null; TWFG[v].MWS := null;
      TWFG[v].TAS := null; TWFG[v].MTS := null; TWFG[v].State := active;
      TWFG[v].ancestor := 0;
   end;
   for each transaction Ti in the current site do begin
      TWFG[i].TID := Ti;
      construct message-wait edges (MWS), master-slave edges (MTS), and
```

```
    state of the transaction (State).
  end;
  for each resource entry in the lock table do
    construct lock-wait edges.
  for v := 1 to N do begin
    TWFG[v].current := TWFG[v].LWS;
    if TWFG[v].State ≠ active then begin
      TWFG[v].MTS := null; TWFG[v].MWS := null
    end
  end;
```

In Step 2 probes in RPP are examined. For each probe PB(T_i, T_j, S_s) in RPP, if the probe is a stale one, it is removed. If the probe is active and T_i and T_j habe entries in TWFG, we add LW(T_i, T_j) to TWFG. This makes detection of deadlocks (actually cycles) simple. The whole process of Step 2 is refined below.

```
Step 2: /* Remove stale probes and reflect some active probes into TWFG */
  for each probe PB(Ti, Tj, Ss) in RPP do begin
    if (Ti ∈ TWFG and TWFG[Ti].State ≠ active) or
      (Tj ∉ TWFG) or
      (Tj ∈ TWFG and TWFG[Tj].State ≠ active)
    then remove the probe from RPP
    else
      if Ti ∈ TWFG and LW(Ti, Tj) ∉ TWFG[Ti].LWS
        then add LW(Ti, Tj) to TWFG[Ti].LWS
  end_for;
```

In Step 3 we do a directed walk for each vertex in TWFG. A directed walk in TWFG starting from vertex v (v becomes a root vertex for this directed walk) always either terminates at v or takes us back to a vertex marked a non-zero ancestor. The first condition corresponds to the case in which there is no more cycle reachable from v in the following senses: (1) There is no cycle in that subgraph at the beginning, (2) There was a cycle that had been already resolved by previous another directed walk, or (3) The current walk has already resolved a cycle, if it exists, so at this point there is no more cycle reachable from the root vertex. The second condition is satisfied when there is a cycle in that subgraph. During the directed walk, we backtrack to the ancestor and then proceed to the next incident vertex when we cannot move forward. That condition is expressed in our internal structure in such a

way the *current* value of a vertex is null. That internal condition takes place when all reachable cycles, if any, have been resolved by selecting some vertices as victims. The whole process of Step 3 is refined below.

```
Step 3: /* Cycle detection and victim selection */
  for v := 1 to N do begin
    TWFG[v].ancestor := -1;
    while v ≠ -1 do begin
      if TWFG[v].current = null then begin
        w := TWFG[v].ancestor; TWFG[v].ancestor:=0 ; v := w;
      end else begin
        Let the lock-wait edge pointed to by TWFG[v].current be (w, link).
        if TWFG[w].current = null
        then TWFG[v].current := link
        else
          if TWFG[w].ancestor ≠ 0 then begin
            victim_selection(v, w); v := w;
          end else begin
            TWFG[w].ancestor :=v ; v := w;
          end
      end
    end_while
  end_for;
```

The process of victim_selection in Step 3 can be explained as follows. Let an edge $v \rightarrow w$ be the one that detects a cycle in our directed walk. Starting from v, we backtrack until w is visited again while selecting a transaction with the highest priority in the cycle. During backtracking, the *ancestor* of each backtracked vertex is cleared except for w. On finishing backtracking, *current*, *LWS*, *MWS* and *TAS* of the victim are set to null and *State* of it is set to *abort*.

During our directed walk, once a cycle is detected with an edge $(v \rightarrow w)$ and resolved appropriately with backtracking, the walk resumes at the vertex w. It should be noted that we may traverse more than once a subset of transactions in the cycle that has been resolved just before to check undetected cycles reachable from the root. The whole process of victim_selection is refined below.

```
Procedure victim_selection (s, t)
/* Starting from s we do backtracking until t is visited while selecting a victim. */
begin
```

```
    victim :=t ; v := s;
    while v ≠ t do begin
       if v > victim
       then victim := v;
       w := TWFG[v].ancestor; TWFG[v].ancestor := 0; v := w
    end_while;
    TWFG[victim].State := abort; TWFG[victim].LWS := null;
    TWFG[victim].current := null;  TWFG[victim].MWS := null;
    TWFG[victim].TAS := null;  abortion_set:= abortion_set + {victim};
  end;

  /* Consider antagonistic lock-wait edges */
  for v: = 1 to N do
     if (TWFG[v].MWS ≠ null)
     then for each LW(v, x) in TWFG[v].LWS do
          TA_Propagation(v, x);
```

The process of TA_Propagation in Step 4 can be explained as follows. Let v and w be two transactions such that v transitively waits for w. We first check whether v is antagonistic with w. Once it is true and TA(v, w) is not in TWFG[w].TAS, TA(v, w) is added to TWFG [w].TAS and for each lock-wait LW(w, x) in TWFG[w].LWS, we call TA_Propagation (v, x) recursively. The whole process of TA_Propagation it refined below.

```
Procedure TA_Propagation (v, w)
begin
   if (TWFG[w].State = active) and
     ((TWFG[w].MWS≠null and v > w) or (TWFG[w].MWS = null)) and
     (TA(v, w) ∉ TWFG[w].TAS)
   then begin
      add TA(v, w) to TWFG[w].TAS;
      for each LW(w, x) in TWFG[w].LWS do
        TA_Propagation(v, x)
   end
end;
```

In Step 5 we remove stale probe receipts from SPP, send antiprobes and probes, and abort victims in the abortion_set. Note that when we consider message-wait edges for sending probes, only outgoing message-waits are considered. Incoming message-waits are

not used for this purpose. The whole process of Step 5 is refined below.

```
Step 5: /* Remove stale probe receipts, send probes and antiprobes, and abort victims */
  /* Remove stale probe receipts and send antiprobes */
  for each probe receipt PB(T_i, T_j, S_s) in SPP do begin
     if (T_j ∉ TWFG) or (T_j ∈ TWFG and TWFG[T_j].State ≠ active)
     then remove the probe receipt
     else if (TA(T_i, T_j) ∉ TWFG[T_j].TAS) then begin
           remove the probe receipt from SPP
           if (T_i ∈ TWFG) and (TWFG[T_i].State = abort)
           then send an antiprobe AP(T_i, T_j, S_r, abort) to S_s
           else send an antiprobe AP((T_i, T_j, S_r, active) to S_s
     end
  end_for;

  /* Send probes */
  for v := 1 to N do
     for each TA(T_i, v) in TWFG[v].TAS do begin
        for each MT(S_s, v) in TWFG[v].MTS do
           if PB(T_i, v, S_s) ∉ RPP ∪ SPP
           then begin
              send a probe PB(T_i, v, S_r) to S_s;
              add a probe receipt PB(T_i, v, S_s) into SPP
           end;
        for each MW(S_p, S_s, x) in TWFG[x].MWS do
           if S_p = S_r and PB(T_i, v, S_s) ∉ RPP ∪ SPP
           then begin
              send a probe PB(T_i, v, S_r) to S_s;
              add a probe receipt PB(T_i, v, S_s) into SPP
           end
        end_for;

  /* Abort victims */
  for each transaction v in abortion_set do
     abort v;
```

In our scheme the same deadlock can be found at multiple sites due to the parallelism of global transactions. However, when the parallelism is not permitted as in [29], [33], [34] our algorithm also detects a global deadlock only at one site. For example in Figure 6.7, there is a global deadlock($T_6 \rightarrow T_2 \rightarrow T_3 \rightarrow T_6$). In this case however, the same transaction T_6 is selected as a victim at S_B and S_C according to our deadlock resolution algorithm.

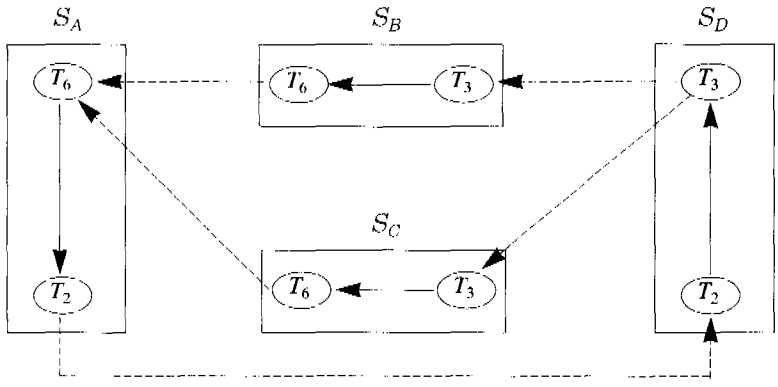

Figure 6.7 Deadlock detection at two different sites

The fact that our algorithm detects all global deadlocks in the system can be probed by the following lemma and theorem.

Lemma 6.1 If $TA_v(T_i, T_j)$ and $LW_w(T_j, T_k)$ hold for two different sites S_v and S_w, then there will be PB(T_i, T_j, S_n) in RPP_w eventually for some site S_n.

proof Without loss of generality, assume that there are five agents V, M, X, N and W of transaction T_j in S_v, S_m, S_x, S_n and S_w respectively such that M is the master agent of V, X is the master agent of M, N is a slave agent of X, and W is a slave agent of N. In that sense, agent X is the lowest common ancestor of agents V and W. Because M ans X are master agents of V and M respectively, MT_v (S_m, T_j) and $MT_m(S_x, T_j)$ hold. In addition to that since X and N are master agents of N and W respectively and also agent W has not finished its assignment (because W is in lock-waint), $MW_x(S_x, S_n, T_j)$ and $MW_n(S_n, S_w, T_j)$ hold.

When deadlocks are checked at S_v, because $TA_v(T_i, T_j)$ and $MT_v(S_m, T_j)$ hold, a probe PB(T_i, T_j, S_v) is sent to S_m. At S_m, when deadlocks are checked after receiving the probe sent from S_v, owing to PB(T_i T_j, S_v) in RPP_m and $MT_m(S_x, T_j)$, a probe PB(T_i, T_j, S_m) is sent to S_x. Up to now probes are sent through the ancestor path of agent V in the process tree of T_j. Now, the direction is reversed. i.e. probes are sent to descendents of X in the process tree until a probe in our concern is sent to S_w by the following reasons. At S_x, when deadlocks are

checked after receiving the probe sent from S_m, owing to PB(T_i, T_j, S_m) in RPP_x and $MW_x(S_x, S_n, T_j)$, a probe PB(T_i, T_j, S_x) is sent to S_n. When deadlocks are checked at S_n after recciving the probe sent from S_x, owing to PB (T_i, T_j, S_x) in RPP_n and $MW_n(S_n, S_w, T_j)$, a probe PB(T_i, T_j, S_n) is sent to S_w and when the probe is received at S_w, it is stored in RPP_w. QED

Theorem 6.1 All global deadlocks in the system will be detected and resolved.

proof Once there is a global deadlock in the system, there exists a set of transactions (a deadlock set) $\{T_1, T_2, T_3, ..., T_n\}$ such that each transaction in the set transitively waits for the completion of all other transactions in the set. Without loss of generality, we assume that each transaction in the set lock-waits for only one transaction in the set at exactly one site such that the lock-wait information makes exactly one cycle and each transaction in the set is a member of the cycle.

Once there exists a global deadlock in the system, there is a unique global transaction T_h in the deadlock set such that T_h is antagonistic with all other transactions in the set. According to our assumption for the deadlock set, there is another global transction T_l such that there exists a local path from T_l to T_h in $TWFG_r$ for some site S_r. Suppose that T_h lock-waits for some transaction in the set at site S_s such that there is a local path from T_h to another global transaction T_{x1} that is in the deadlock set and also is not in the lock-wait state at S_s. When deadlocks are checked at S_s, $TA_s(T_h, T_{x1})$ holds. Suppose T_{x1} lock-waits for some transaction in the set at site S_a that is different from S_s. According to Lemma 6.1, RPP_a can contain a probe PB(T_h, T_{x1}, S_t) eventually for some site S_t. Once again suppose that there is a local path from T_{x1} to another global transaction T_{x2} that is in the deadlock set and also is not in the lock-wait state at S_a. When deadlocks are checked at S_a, $TA_a(T_h, T_{x2})$ holds. Without loss of generality, assume that T_{x2} is T_l. Because T_l lock-waits for some transaction in the set at site S_r, RPP_r can contain a probe BP(T_h, T_l, S_q) eventually for some site S_q according to Lemma 6.1 again. When a period comes up for deadlock detection at S_r, a deadlock is declared because of PB(T_h, T_l, S_q) in RPP_r and local path T_l to T_h in $TWFG_r$. QED

7. Performance Characteristics

One of the performance measures for distributed deadlock detection and resolution algorithms is the number of messages transmitted for detection and resolution of deadlocks. For the measurement of its number of messages, assume that n transactions constitute a global deadlock and n transactions among them are global transactions. The number of probes and antiprobes sent for the detection and resolution of global deadlocks depends on the number of global transactions, distribution of global transactions, and the number of message-wait and master-slave edges. According to the placement of these parameters, we can easily specify the best situation and the worst situation for the number of messages transmitted to detect and resolve a global deadlock cycle in the system.

Since a master-slave edge can convey exactly the same number of messages that an outgoing message-wait edge can do in these extreme cases, we will not consider master-slave edges in this analysis. Let e be the number of edges (lock-wait edges and message-wait edges) and e' be the number of outgoing message-wait edges in the cycle.

1) The best situation

This case can be occurred when only one global transaction can initiate probes, which cause at most e'-1 probes. For the resolution of a global deadlock detected, no antiprobes are sent if the selected victim is a local transaction. Therefore, the number of inter-site messages to detect and resolve a global deadlock becomes at most e'-1.

For example in Figure 7.1, 5 transactions are involved in a global deadlock with 4 global transactions and 5 outgoing message-wait edges.

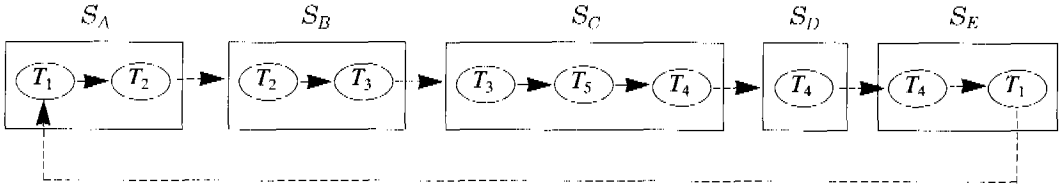

Figure 7.1 The best acase for the number of messages

Among those global transactions only T_4 at S_E can initiate a probe and the probe propagates to S_A, S_B, and S_C. At S_C, a global deadlock is declared and a local transaction T_5 is aborted. In this case, only 3 probes are transmitted without any antiprobe transmission.

2) The worst situation

This case can be occurred whtn n'-1 global transactions initiate probes. In this case, an outgoing message-wait edge might convey at most n'-1 probes and at most e'-1 outgoing message-wait edges can be involved in a global deadlock detection and resolution such that the number of probes transmitted becomes at most $(n'\text{-}1)*(e'\text{-}1)$. After the global deadlock is detected and a victim is aborted, at most e'-1 antiprobes can be transmitted. Therefore, the total number of messages to detect and resolve a global deadlock in the worst case requires at most $n'*(e'\text{-}1)$ inter-site messages.

For example in Figure 7.2, 5 transactions are involved in a global deadlock cycle with 4 global transactions and 5 outgoing message-wait edges as in the case of the best situation but different edge directions and transaction placement.

Outgoing message-wait edges connecting S_E and S_D, S_D and S_C, S_C and S_B, S_B and S_A and S_A and S_E convey 0, 1, 2, 3, and 3 probes respectively. For instance, the outgoing message-wait edge connecting agents of T_2 between S_C and S_B conveys PB(T_4, T_2, S_C) and PB(T_3, T_2, S_C). At S_E, when deadlocks are checked after receiving probe PB(T_4, T_1, S_A), a

global deadlock is declared and T_4 is aborted. When T_4 at S_D is aborted, and antiprobe AP (T_4, T_3, S_D, *abort*) is sent to S_C. This message pro-pagates down the corresponding probe path until it cannot go further at site S_E. As a result, 13 messages (9 probes and 4 antiprobes) are transmitted.

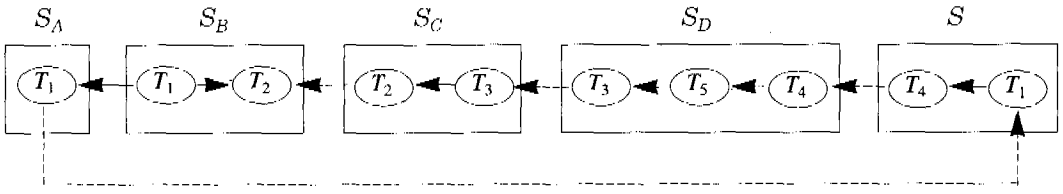

Figure 7.2 The worst case for the number of messages

8. The Youngest Victim Selection

When a deadlock is declared in our scheme, the deadlock cycle detected may not illustrate the actual situation because of the probe transmission mechanism. It is a condensed one. Therefore, when a victim is selected according to the algorithm given in Section 6, the victim may not be the transaction with the highest priority among transactions that constitute the deadlock. Under some circumstances this could have a significant impact on the fairness of the victim selection. However, as in the case of the local deadlock resolution, the youngest transaction (a transaction with the highest priority) in global deadlock can be selected as a victim by a modification of the original algorithm. Note that we are not saying the optimal cost victim selection in the whole system (it is found to be NP-hard [17]) but the selection of a victim in a deadlock cycle.

For the purpose of selecting the youngest transaction in a deadlock, we can treat a local transaction T_i as if it is a global transaction only when it is transitively waited by a global transaction T_j and at the same time the priority of T_i is greater than that of T_j. Once it is found to be true, T_i works as if it is a global transaction as far as deadlock detection and resolution is concerned. For this work we need to modify the definition of antagonisty.

Definition 8.1 (modification of Definition 5.1) A transaction T_i is *antagonistic* with another transaction T_j if $T_i \rangle T_j$ and either T_i is a global transaction or T_i is transitively waited by a global transaction.

Note that other definitions for the construction and treatment of TAs, probes, and antiprobes are exactly the same as the original scheme. Actually, we need to present the whole algorithm for the modified version of deadlock detection, but we show how they work through the following two examples.

Example 8.1 Consider Figure 7.2. When we follow the original algorithm given in Section

6, T_4 is aborted at site S_E. To get the minimal cost victim however, the modified algorithm works as follows. At S_D, because a global transaction T_4 lock-waits for a local transaction T_5 that has higher priority than T_4, T_5, works as if it is a global transaction as far as deadlock detection and resolution is concerned. According to this, probe PB(T_5, T_3, S_D) is sent to S_C. Note that $TA_D(T_4, T_3)$ does not hold anymore in this modified scheme. By the same way as before, probe PB(T_5, T_1, S_A) is stored at RPP_E and when deadlocks are checked at S_E, PB(T_5, T_4, S_E) is sent to S_D. When deadlocks are checked at S_D after receiving the probe sent from S_E, a cycle ($T_5 \rightarrow T_4 \rightarrow T_5$) is found and T_5 is selected as a victim. Once T_5 is aborted, probe PB(T_5, T_4, S_E) in RPP_D is removed and all probes and probe receipts caused by T_5 and stored in other sites are rmoved by sending antoprobes through the probe path. Including those probes whose initiators are T_3 and T_2, the number of probes sent is 10 and the number of antiprobes sent is 5. If we apply the modified scheme to Figure 7.1, the number of messages sent becomes 13 (8 probes and 5 antiprobes). □

Example 8.2 Consider Figure 4.3. The modified algorithm to get the minimal cost victim works as follows. At S_A, when deadlocks are checked, three probes whose initiators are T_7, T_6, and T_5 are sent to S_B because local transactions T_6 T_5 are transitively waited by global transation T_4 and they have higher priority than T_4 and T_2. When deadlocks are checked at S_B after receiving those probes from S_A, three probes whose initiators are T_7, T_6, and T_5 are sent to S_C. When deadlocks are checked at S_C only one probe whose initiator is T_9 is sent to S_A. The probe whose initiator is T_9 propagates to S_A, S_B, and finally to S_C where a deadlock ($T_9 \rightarrow T_3 \rightarrow T_9$) is found. When T_9 is avorted at S_C, an antiprobe that compensates for its corresponding probe is sent to S_A and it is propagated to S_B and then to S_C. As a result, the number of total messages is 12 (9 probes and 3 antiprobes). □

Even though the modification of the original algorithm for the selection of the minimal cost victim causes more message transmission compared to the original scheme (3 to 13 for Figure 7.1, 13 to 15 for Figure 7.2, and 4 (the best case) to 12 for Figure 4.3), when deadlocks are very rare and minimal cost victim selection in required, this modification makes sense.

9. Closing Remarks

A new distributed deadlock detection and resolution algorithm is presented in this paper. Global deadlocks are detected and resolved by constructing transaction wait-for graphs periodically and also by sending deadlock detection messages. We define how to extract lock-wait information among transactions in the environment of parallel execution of transactions at multiple sites. Multiple modes of locks and lock conversion are also considered.

To minimize the number of inter-site mess-

ages at the expense of the possibility of not selecting the youngest victims, condensed transaction wait-for graphs are constructed by sending probes and antiprobes. In sending probes and antiprobes we consider priorities among transactions, message-wait and master-slave relationships among agents of global transactions, and status of transactions with respect to the commitment protocol.

To further reduce the number of inter-site message transmission, each site maintains two pools of probes (a pool of probes received and another pool of receipts of probes sent) such that once probes are sent they are not sent again. Probes received are used for detection of deadlocks that exist currently or may occur later in time.

References

1. R. Agrawal, M. J. Carey and D. J. DeWitt, "Deadlock Detection is Cheap," *ACM SIGMOD Record*, Vol. 13, No. 2, pp. 19-34, January 1983.
2. R. Agrawal, M. J. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Trans. Database Systems*, Vol. 12, No. 4, pp. 609-654, December 1987.
3. 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.
4. C. Beeri and R. Obermarck, "A Resource Class Independent Deadlock Detection Algorithm," in *Proc. Seventh Int. Conf. on Very Large Databases*, pp. 166-178, 1981.
5. P. A. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database Systems," *Addison-Wesley*, 1987.
6. S. Ceri and G. Pelagatti, "Distributed Databases: Principles and Systems," *McGraw-Hill*, 1984.
7. 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.
8. W. N. Chin, "Some comments on 'Deadlock Detection is Cheap' in SIGMOD Record Jan. 83," *ACM SIGMOD Record*, pp. 61-63, March 1984.
9. A. K. Choudhary, "Cost of Distributed Deadlock Detection: A Performance Study," in *Proc. Sixth Int. Conf. on Data Engineering*, pp. 174-181, February 1990.
10. 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.
11. C. J. Cate, "An Introduction to Database Systems: Vol II," *Addison-Wesley*, 1983.
12. 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.

13. A. K. Elmagarmid and A. K. Datta, Two-Phase Deadlock Detection Algorithm," *IEEE Trans. Computers*, Vol. 37, No. 11, pp. 1454-1458, November 1988.
14. S. Even, "Graph Algorithms," *Computer Science Precess*, 1979.
15. J. F. Garza and W. Kim, "Transaction Management in an Object-Oriented Database System," in *Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data*, pp. 37-45, 1988.
16. 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.
17. J. Gray, "Notes on Database Operating Systems," in Lecture Notes in Computer Science 60, Advanced Course on Operating Systems, ed. G. Seegmuller, Springer Verlag, New York, pp. 393-481, 1978.
18. J. Gray and A. Reuter, "Transaction Processing," Version I of the Slides, 1987.
19. T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, Vol. 15, No. 4, pp. 287-317, 1983.
20. L. H. Hass and C. Mohan, "A Distributed Deadlock Detection Algorithm for a Resource-Based System," Rep. RJ3765, *IBM Research Lab.*, San Jose, California, Junuary 1983.
21. G. S. Ho and C.V. Ramamoorthy, "Protocols for Deadlock Detection in Distributed Databasc Systems," *IEEE Trans. Software Eng.*, Vol. SE-8, No. 6, pp. 554-557, November 1982.
22. B. Jiang, "Deadlock Detection is Really Cheap," *ACM SIGMOD Record*, Vol. 17, No. 2, pp. 2-13, June 1988.
23. D. B. johnson, "Finding all the Elementary Circuits of a Directed Graph," *SIAM J. Computing*, Vol. 4, No. 1, pp. 77-84, March 1975.
24. H. Korth and A. Silberschatz, "Database System Concepts," McGraw-Hill Book Company, 1986.
25. B. G. Lindsay, 1. 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.
26. L. Manteiman, "The birth of OSI TP: A new way to link OLTP networks," Data Communications, November 1989.
27. 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.
28. J. E. B. Moss, "Nested Transactions: An Approach to Reliable Computing," M.I.T. Report, M.I.T.-LCS-TR-260, *M.I.T., Laboratory of Computer Science*, 1981.
29. R. Obermarck, "The Distributed Deadlock Detection Algorithm," *ACM Trans. Database Systems*, Vol. 7, No. 2, pp. 197-208, June 1982.
30. 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.

31. 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. 8th Int. Conf. on Data Engineering*, pp. 202-209, February 1992.
32. K. H. Pun and G. G. Belford, "Performance Study of Two Phase Locking in Single-Site database Systems," *IEEE Trans. Software Eng.*, Vol. SE-13, No. 12, pp. 1313-1328, December 1987.
33. 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.
34. M. Roesler and W. A. Burkhard, "Resolution of Deadlocks in Object-Oriented Distributed Systems," *IEEE Trans. Computer*, Vol. 38, No. 8, pp. 1212-1224, August 1989.
35. 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.
36. 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.
37. 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.
38. G. T. Wuu and A. J. Bernstein, "False Deadlock Detection in Distributed Systems," *IEEE Trans. Software. Eng.*, Vol. SE-11, No. 8, pp. 820-821, August 1985.
39. X/Open company Ltd, "Interim Reference Model for Distributed Transaction processing," Transaction Processing Working Group, X/Open Company Limited, July 1989.