Deadlock Detection techniques for Distributed Systems

Praveen Mahadevanna
Department of Computer Science and Engineering
The University of Texas at Arlington (UTA)

Abstract

Deadlock detection and resolution is one among the major challenges faced by a Distributed System. In this paper, we discuss deadlock detection techniques and present two approaches for detecting deadlocks in Distributed Systems. Our first approach uses a Hybrid combination of a transaction wait-for graph construction and a probe generation mechanism. Wherein a local transaction wait-for graph is maintained at each site to detect local deadlocks without transmitting any intra-site deadlock detection messages, in addition of using Probes. The second approach for deadlock detection uses the concept of a Lock History which is carried by each transaction, the notion of intention locks, and three-staged hierarchical approach to deadlock detection, with each stage, or level of detection activity being more complex than the preceding one.

Overview

We begin our discussion with an introduction to the topic on hand in *section1*. In *section2* we present several approaches used for the deadlock detection in distributed systems. Approaches for deadlock detection in distributed systems are explained, the first using the *'hybrid'* scheme is described in *section3* followed by another approach in *section4* that uses the *'Lock History'* and *section*$_5$ finally presents the conclusion and discussion in the end.

1. Introduction

A Distributed system consists of a collection of sites that are interconnected through a communication network each maintaining a local database system. Transactions are the units of interaction with the system that is a set of atomic operations and transforms the database from one consistent state to another. Each process requests at most one lock at a time, and the process is blocked if in case the requested resource is not available immediately until the resource is granted or the transaction itself is aborted. A Deadlock results when a set of transactions are waiting circularly for each other to release resources. Deadlocks are usually characterized in terms of a *transaction wait for graph (TWFG)* that is a directed graph wherein each vertex represents a transaction. E.g. $T_i \longrightarrow T_j$ means that T_i is waiting for the completion of T_j. It is a proved fact that a deadlock exists if and only if there is a cycle in *TWFG* [10].

Transactions (*atomic* units of operation) may be either local/global wherein the global transactions require the communication among participating sites. Deadlock occurring at a single site is called as *local deadlock* whereas those involving transactions executing at multiple sites is called a *global deadlock*. In distributed deadlock detection, global deadlocks are detected by sending inter-site deadlock detection messages. Detection schemes for global deadlocks are classified into two categories depending upon the type of graph they construct, which is either an ***actual graph*** or a ***condensed graph*** [10].

Actual graph detection schemes are based upon transmission of detection messages conveying strings of transactions of an arbitrary length in which each transaction (global/local) waits for the completion of the immediately following transaction in the string, the first transaction being the global one being waited by some other site and the last transaction being the

global transaction that is waiting for either a response or a new request from another site. The deadlock detection message (=> string of transactions) is sent to the site for which the last transaction in the string is waiting and is used to construct an actual deadlock detection graph at the receiving site of the message. The messages thus transmitted can be reduced into 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 [10][1].

In the ***condensed graph*** detection scheme the deadlock messages contain only two transactions say (T_i ,T_j) where transaction T_i transitively waits for the completion of transaction T_j. The message is sent either to the originating site of transaction T_i (backward transmission) or to the originating site of transaction T_j (forward transmission) , in order to prevent the large number of message transfers *probe* is sent when a transaction with a higher priority transitively waits for another transaction with a lower priority which avoid the transmission of backward messages [10]. An interesting feature in 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 *antiprobe* is received. However it is not free from drawbacks in the sense that they send messages to transaction managers and also to resource managers (resulting in the transmission of a lot of deadlock detection and resolution messages even when only local transactions are concerned in a site) and they treat local and global transactions equally[10][1].

2. Various approaches for deadlock detection in distributed systems [11] [1]

2.1 Path-pushing algorithms

The basic idea underlying this class of algorithms is to build some simplified form of global WFG at each site. For this purpose each site sends its local WFG to a number of neighboring sites every time a deadlock computation is performed. After the local data structure of each site is updated, this updated WFG is then passed along, and the procedure is repeated until some site has sufficiently complete picture of the global situation to announce deadlock or to establish that no deadlocks are present. The main features of this scheme, namely, to send around paths of the global WFG, has led to the term *path-pushing algorithms* [11][1].

2.2 Edge-chasing algorithms

The presence of a cycle in a distributed graph structure can be verified by propagating special messages called ***probes*** along the edges of the graph. Probes are assumed to be distinct from resource request and grant messages. When the initiator of such a probe computation receives a matching probe, it knows that it is in cycle in the graph. A nice feature of this approach is that executing processes can simply discard any probes they receive. Blocked processes propagate the probe along their outgoing edges [11][1].

3. Deadlock detection in distributed systems using a 'hybrid technique' of a 'Transaction wait-for graph construction' and a 'probe generation mechanism'. [10]

A local transaction wait-for graph is maintained at each site to detect local deadlocks without transmitting any intra-site deadlock detection 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 both the transaction wait-for graph and the probes received. In order to compensate for the probes that have been sent already, ***antiprobes*** are sent.

3.1 Description of the Basic idea [10] [1]

Transaction operations are carried out by processes. When a transaction is initiated, it is executed by a process. When the process needs resources at other sites, one process, which is call as a ***child***

process is created at each corresponding remote site to represent the parent process at that site which carries the ***transaction identifier*** of its parent process and it can also be a parent by creating its own child processes. Processes of a transaction are called ***agents*** or ***cohorts*** of the transaction and they constitute a tree structure represented as a ***process tree***. Since multiple agents of the same transaction may exist on one site, we define a process P_x of transaction at site S_r to be $T_iP_xS_r$. All the agents of a transaction can run in parallel. However when a transaction T_i has multiple agents running at the same site, we assume that only one of them executable by blocking the others. Thus the transaction identifier of an agent W can be represented as TID(W), e.g. TID($T_iP_xS_r$) is T_i. Each transaction has a globally unique identifier that consists of two fields i.e. the *site identifier* 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, a unique priority can be assigned to each transaction as follows. If T_i and T_j are two transactions, T_i has a higher priority than T_j denoted as $T_i > T_j$, if either the local 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. Hence we assume that $T_i > T_j$ if i>j, for instance $T_4 > T_2$.

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

A process called ***sender*** of a transaction ***message-waits*** for another process, called ***receiver*** of the same transaction, denoted as ***MW(sender, receiver)***, if sender is waiting to receive a message(either an answer for a previous/new request) from the receiver.

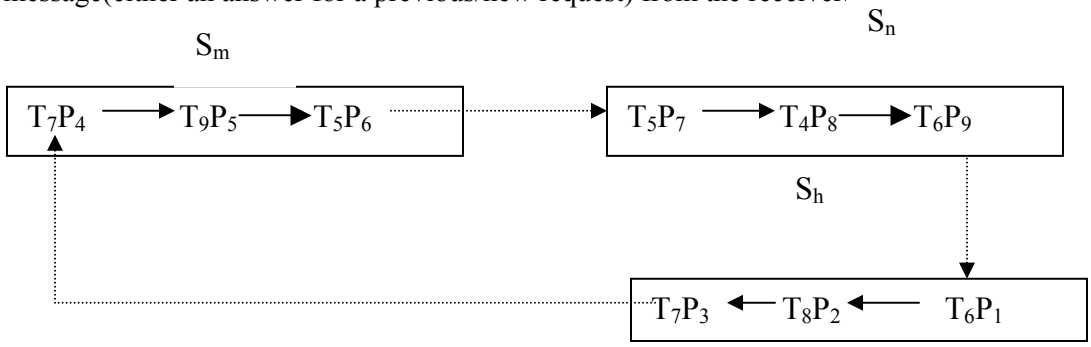

Fig1. Detection of a global deadlock[10]

The basic idea can be conveyed as follows. Assuming there is an existence of a global deadlock as indicated in Fig1 and assuming that T_7, T_6 and T_5 are global transactions, where $T_7 > T_6 > T_5$; T_9, T_8, and T_4 are local transactions, and solid arrows represent lock-waits and dotted arrows represent message-waits. Suppose that site S_m sends a message *"T_7 transitively waits for T_5"* to site S_n, due to the reasons cited (1) due to the presence of the local path indicated by $T_7P_4S_m \longrightarrow T_9P_5S_m \longrightarrow T_5P_6S_m$ in site S_m, (2) $T_7 > T_5$ and (3) MW($T_5P_6S_m$, $T_5P_7S_n$). Upon receipt of this message at site S_n, if site S_n sends a message *"T_7 transitively waits for T_6"* to site S_h, since (1) that message condenses the received message and the local path $T_5P_8S_n \longrightarrow T_4P_8S_n \longrightarrow T_6P_9S_n$ in site S_n, (2) $T_7 > T_6 > T_5$ and (3) MW($T_6P_9S_n$, $T_6P_1S_h$). Note that *"T_7 transitively waits for T_6"* is a condensed message for the global path ($T_7P_4S_m \longrightarrow T_4P_8S_n \longrightarrow T_6P_9S_n$). Once the message *"$T_7$ transitively waits for T_6"* is received at site S_h, since there is a local path $T_6P_1S_h \longrightarrow T_8P_2S_h \longrightarrow T_7P_3S_h$ in site S_h, a cycle can be formulated and thus a deadlock is declared.

3.2 Description of the algorithm [10]

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 a n array

[0…N-1] of records and each entry of the array has the following fields: TID(the transaction identifier of the transaction for this entry), global(boolean variable denoting whether the transaction is either global/local), LWS(a pointer to a set of lock-wait edges), MWS(a pointer to a set of message-waits), TAWS(a pointer to a set of transitive-antagonistic waits), and PBS(a pointer to a set of probes received). We assume that the underlying network guarantees the error-free and finite time arrival of the messages to their destination maintaining the same order in which they were sent.

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. For example, at site S_m in Fig1, T_7 is antagonistic with T_9 and T_5. However, T_9 is not antagonistic with T_5.

A path from a transaction process ($T_iP_xS_s$) to another transaction process ($T_jP_yS_r$) in the global TWFG is an ***antagonistic path*** if T_i is antagonistic with ever transaction on the path except for T_i itself.

If there is an existence of an antagonistic path in the global TWFG from an agent of T_i to an agent of T_j, we say that T_i waits for the agent of T_j ***transitively and antagonistically.*** For example in Fig1, there exists an antagonistic path $T_7P_4S_m \longrightarrow T_9P_5S_m \longrightarrow T_5P_6S_m$ in site S_m; where in T_7 waits for $T_9P_5S_m$ and $T_5P_6S_m$ transitively and antagonistically. A transitive-antagonistic wait, denoted as ***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 Fig2. In Fig2 *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 except for transaction TIE(e). Hence *e* cannot be a terminus of any TAW having TID(a) as its initiator because TID(a) is not antagonistic with TID(e). The count field of *TAW(TID(a),f,2),* for instance, is two since *f* has two incoming edges that are related to the antagonistic paths from *a*.

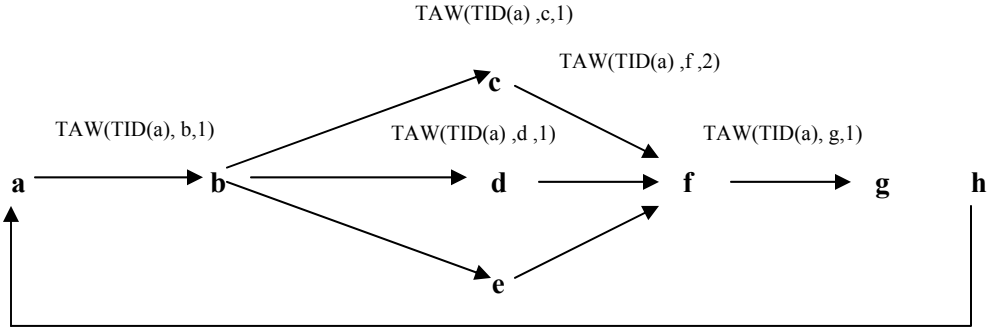

Fig2. Propagation of Transitive-antagonistic waits.[10]

We declare that there is a deadlock in the system hen transaction T_i transitively waits for a process of transaction T_j and there are some paths from T_j to T_i. Suppose if an edge is added from *g to h* then we declare there is a deadlock in the system since *TAW(TID(a), g,1)* holds and there is a path from *g to a.* To facilitate the construction of TAWs, ***inter-site messages*** called ***probes*** are sent. A probe is denoted as ***PB (initiator, sender, receiver),***
is a message from the site of agent sender to the site of agent receiver to inform the receiver that the transaction initiator has been transitively and antagonistically waiting for he 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 or (2)*MW(sender, receiver)* exists and TAW(initiator, sender,1) is newly added. In fact a probe is a condensed

message, which signifies that at the interface of a site or a set of sites, a global transaction initiator waits for another global transaction *TID(sender*) transitively and antagonistically. For example in Fig1, the probe sent from site S_m to site S_n is PB(T_7,$T_5p_6S_m$,T_5,P_7S_n), the probe sent from site S_n to site S_h is PB(T_7,$T_6P_9S_n$,T_6,P_1,S_h).

When a probe is received or a lock-wait occurs, the function *TAW-propagation(initiator, terminus)* is invoked to construct a condensed TWFG at the site. The function represents that a transaction initiator waits for a transaction process terminus transitively and antagonistically through an incoming edge to terminus and also propagates that effect down to lock-wait edges and message-wait edges that are outgoing from terminus. *TAW-propagation( )* is defined as follows.

```
TAW_propagation(initiator, terminus)
{
  if(initiator < TID (terminus)
    and TWFG[TID(terminus)].global==True
    return;
  if  (TAW(initiator, terminus, count) is
    in TWFG[TID(terminus)].TAWS) {
     increase count field of TAW by 1;
     return;
   }
  add TAW(initiator, terminus, 1) into
  into TWFG[TID(terminus)].TAWS;
   for each message-wait MW(V,W)
      in TWFG[TID(terminus)].MWS do {
        if (V==terminus)
          send probe PB(initiator, V,W)
          to the site of  agent W;
   }
 for each lock-wait LW(V,W)
     in TWFG[TID(terminus)].LWS do {
       if(V==terminus)
            TAW_propagation(initiator,W)
    }
}
```

```
TAW_contraction(initiator, terminus)
{
  if(initiator < TID (terminus)
    and TWFG[TID(terminus)].global==True
    return;
  if  (TAW(initiator, terminus, count) is
    in TWFG[TID(terminus)].TAWS) {
      if(count>1){
        increase count field of TAW by 1;
        return;
       }
  delete TAW(initiator, terminus, count)
  from TWFG[TID(terminus)].TAWS;
   for each message-wait MW(V,W)
      in TWFG[TID(terminus)].MWS do {
        if (V==terminus)
          send antiprobe AP(initiator, V,W)
          to the site of  agent W;
   }
 for each lock-wait LW(V,W)
     in TWFG[TID(terminus)].LWS do {
       if(V==terminus)
            TAW_contraction (initiator, W)
    }
}
```

```
Deadlock_detection(s, t, victim_set)
{
  victim_set={};
  if (t does not have an entry in  TWFG)
     return;

  for each path from s to t in  TWFG do {
    select the youngest transaction on the   path;
    add that transaction into victim_set;
   }
 if (either s or t is in victim_set)
    set victim_set as that transaction;
 for each victim in the victim_set do {
   if(TWFG[victim].global==TRUE) {
     for i=0 to N-1 do {
       for each probe PB(initiator, V, W) in
        TWFG[I].PBS do{
          If (initiator==victim)
            Delete PB(initiator, V,W)
            From TWFG[I].PBS;
          }
       for each TAW(initiator, W, count)
        in  TWFG[I].TAWS do {
         if(initiator==victim) {
           delete TAW(initiator, W, count)
            from  TWFG[I].TAWS;
            for each message-wait MW(X,Y)
             in TWFG[I].MWS  do {
               if(X==W)
               send antiprobe AP(initiator, X,Y)
               to the site of agent Y;
        }}}}} abort victim.
     }}
```

To compensate for probes sent, ***inter-site messages*** called ***antiprobes*** 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, count)* is sent only when *TAW(initiator, sender, count)* is deleted from a local TWFG in the presence of *MW(sender, receiver)* at that site.

The function ***TAW-contraction(initiator, terminus)*** is used to compensate for *TAW-propagation(initiator, terminus)* and it indicates that the former TAW relationship fro 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. That function is invoked when one of the following occurs: an antiprobe is received a lock-wait does not hold anymore, or a message-wait occurs. *TAW-contraction( )* is defined as shown above.

A deadlock is declared at a site if and only if it is found that a transaction T_i transitively waits for transaction T_j and if there exists paths from an agent of T_j to an agent of T_i in the local

TWFG of the site. Once it is declared for each local 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. For each victim in the victim_set, before the victim is aborted, If the victim is a global transaction, all stored PBs and TAWs having the victim as their initiator are deleted from the local TWFG of the site. The Deadlock detection function is defined as shown above.

On receiving the probe *PB(initiator, sender, receiver)* at the site of agent receiver, function ***PB_propagation(initiator, sender, receiver)*** is invoked. In the function, if *TID(receiver)* does not have an entry in the local TWFG or *MW(receiver, sender*) exists in the local TWFG of the site, the message is neglected. Otherwise we detect and resolve the global deadlock by checking paths from *TID(receiver)* to initiator in the local TWFG.
PB_propagation( ) is defined as follows:

```
PB_propagation(inititor, sender, receiver)
{
 if (TID (receiver) does not have an entry
    in  TWFG or MW(receiver,sender) is
    in TWFG[TID(receiver)].MWS)
       return;

 Deadlock_detection(TID(receiver),
 initiator, victim_set);
 if(initiator is in victim_set)
     return;
 insert PB(initiator, sender, receiver) into
 into TWFG[TID(receiver)].PBS;
 TAW_propagation(initiator, receiver);
}
```

```
AP_propagation(inititor, sender, receiver)
{
 if (TID (receiver) does not have an entry
    in TWFG or
    PB(initiator, sender, receiver) is not in
   TWFG[TID(receiver)].PBS)
      Return;
 Delete PB(initiator, sender, receiver)
 from TWFG[TID(receiver)].PBS;
 TAW_contraction(initiator, receiver);
}
```

```
LW_addition(waiter, waitee)
{
 Deadlock_detection(TID(waitee),
    TID(waiter), victim_set);
 If(TID(waitee) is in victim_set or
 TID(waiter) is in victim_set)
      return;
 for each TAW(initiator, V, count)
 in TWFG[TID(waiter)].TAWS do {
    Deadlock_detection(TID(waitee),
        Initiator, victim_set);
    If(TID(waitee) is in victim_set)
         return;
 }
add LW(waiter, waitee) into
TWFG[TID(waiter)].LWS;
For each TAW(initiator, V,count) in
TWFG[TID(waiter)].TAWS do {
   If(V==waiter)
     TAW_propagation(initiator, waitee);
}
if(TWFG[TID(waiter)].global==TRUE)
 TAW_propagation(TID(waiter), waitee);
}
```

```
MW_addition(sender, receiver)
{
 TWFG[TID(sender)].global=TRUE;
 For each PB(initiator, V, W) in
  TWFG [TID(sender)].PBS do {
  If(V==receiver and W==sender) {
    delete PB(initiator, V, W) from
   TWFG [TID(sender)].PBS;
   TAW_contraction(initiator, sender);
  }
}
add message-wait MW(sendr, receiver)
 into TWFG[TID(sender)].MWS;
 for each TAW(initiator, V, count) in
 TWFG [TID(sender)].TAWS do {
  If(V==sender) send probe
 PB(initiator, sender, receiver) to the
 to the site of agent receiver; }
}
```

```
LW_deletion (waiter, waitee)
{
 delete LW(waiter, waitee) from
TWFG [TID(waiter)].LWS;
for each TAW(initiator, V,count) in
TWFG[TID(waiter)].TAWS do {
   If(V==waiter)
     TAW_contraction(initiator, waitee);
}
if(TWFG[TID(waiter)].global==TRUE)
  TAW_contraction(TID(waiter), waitee);
}
```

For example in Fig1 at site S_n upon receiving $PB(T_7,T_5p_6S_m,T_5,P_7S_n)$ from S_m, this probe is stored in TWFG[t_5].PBS and a new probe $PB(T_7,T_6P_9S_n,T_6,P_1S_h)$ is generated by invoking TAW_propagation($T_7,T_5P_7S_n$). Upon receiving the antiprobe *AP(initiator, sender, receiver),* function AP_propagation(initiator, sender, receiver) is invoked. In the function, if *TID(receiver)* does not have an entry in TWFG or *PB(initiator, sender, receiver)* is not in TWFG, the antiprobe message is neglected. Otherwise, we delete the corresponding probe *PB(initiator, sender, receiver)* and *TAW_contraction( )* is invoked. *AP_propagation( )* is defined as shown above[10].

When a *lock-wait LW(waiter, waitee) occurs,* the function *LW_addition(waiter, waitee)* is invoked which first detects the local deadlocks and then detects global deadlocks for each TAW record having an agent of transaction *TID(waiter)* as its terminus. If either *TID(waiter)* or

TID(waitee) is aborted nothing is done for that lock-wait, else, 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( )* is invoked. *LW_addition( )* is defined above.

When a *lock-wait LW(waiter, waitee) does not hold anymore* the function *LW_deletion(waiter, waitee)* is invoked 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. The function *LW_deletion( )* is defined as indicated above[10].

When a *message-wait MW(sender, receiver) occurs* the function *MW_addition(sender, receiver)* is invoked. In the function *TWFG[TID(sender)].global* is set to TRUE and if *PB(initiator, receiver, sender)* exists in *TSFG[TID(sender)].PBS*, the stored PB record is deleted and *TAW_contraction(initiator, sender)* is invoked. Add *MW(sender, receiver)* and then for each *TAW(initiator, sender, count)*, probe *PB(initiator, sender, receiver)* is sent to the site of agent *receiver*. The function *MW_addition ( )* is defined as shown above[10].

When a transaction process *sender* receives a new request or an answer for the previous request from its cohort receiver, a message-wait *MW(sender, receiver)* does not hold anymore. In this case, we simply delete *MW(sender, receiver)* from *TWFG[TID(sender)].MWS* .

4. Distributed deadlock detection algorithm based on 'lock history' [14]

This algorithm differs from the existing algorithms in that is uses the concept of a ***Lock History*** which each transaction carries with it, the notion of intention locks, and three-staged hierarchical approach to deadlock detection, with each stage, or level of detection activity being more complex than the preceding one.

This assumes a distributed model of transaction execution where each transaction has a site of origin (***Sorig***) which is the site at which it entered the system. Whenever a transaction requires a remote resource it migrates to the site where that resource is located which involves the creation of an agent at the new site and this agent may in turn create additional agents start, commit or abort actions or return execution to the site from which it migrated.

The transaction may have several active agents to allow concurrent execution.
Agents can in any of the three states: ***Active, blocked (waiting) or Inactive***. An *inactive agent* is one that had done its work at a site and has created an agent at another site, or the one that has returned execution to its creating site and is now awaiting further instructions such as commit, abort or become active again. A blocked transaction is one that has requested a resource that is locked by another transaction. An active agent is one that is not blocked or inactive. We assume that all transactions are well formed and $_2$phase meaning that any active agent can release a lock only after the transaction has locked all the resources it needs for its execution and only after it has terminated its execution and the active agent is notified to release the lock during the $_2$phase commit.

The lock table information for a resource consists of: Transaction/Agent ID of the transactions site of origin , the type of lock, (if possible) the resource that the transaction holding a lock intends to lock next. The *current lock field* is referred to as the *current* field of the lock table, and the field containing the future intentions of the transaction holding the current lock is called the ***next*** field, which identifies the site(s) to which the transaction migrated. The algorithm assumes two types of locks ***Exclusive write (W)*** and ***Shared read (R)***. In addition it also uses an intention lock ***(I)*** indicating that the transaction wishes to acquire a lock on a resource, either to modify it ***(IW)*** or to read it ***(IR)***. *The Intention locks are placed in a resource lock table* when an agent is created at a site of a locked resource that it requires or when a resource at the same site is requested which is already locked by another transaction. The intention locks are also placed in the lock table of the last locked resource(s) if the transaction can determine which resources(s) it intends to lock at a remote site in its next execution step[14].

An example of a lock table is LT (R_2B): $T\{W(R_2B), IW(R_3C)\}$; $T_2\{IW(R_2B)\}$.

The lock table for resource R_2 at site B shows that T_1 holds a write lock on R_2, and that T_2 has placed an intention write lock on R_2. T_1 has also indicated that it intends to place a write lock on resource R_3 at site C. Only a single transaction or agent may hold an exclusive write lock on a resource. Any number of intention locks (IW/IR) may be placed on a resource, which means that any number of transactions may wait for a resource. Each site must therefore have some method for determining which transaction will be given the resource when it becomes free. This algorithm uses the lock history (LH) of a transaction, which is a record of all types of locks on any resources that have been requested or are being held by that transaction. Each transaction carries its lock history during is execution. *An example for a lock history (LH) of a transaction T_1 is LH(T_1): {W(R_3C), W(R_2B), R(R_1A)}*. This LH shows that T_1 holds a write lock on resource R_3 at site C, a write lock on resource R_2 at site B, and the read lock on resource R_1 at site A. ***Lock history is used for the following reasons***: to avoid global deadlocks in some cases, to support the selection of victim transactions for rollback and to avoid detection of false deadlocks.

This algorithm detects deadlock either by construction a ***wait-for-graph (WFG)*** or directly from ***wait-for-strings (WFSs).*** A WFG can be constructed b the deadlock detection algorithm using the lock histories of transactions that are possible involved in deadlock cycle. WFG consists of two types of nodes: transactions (agents) and resources. A directed arc from a resource to a transaction node indicates that the transaction has a lock on the resource, while a directed arc from a transaction node to a resource indicates that the transaction has place and intention lock on that resource. *A cycle in the WFG indicates the existence of a deadlock.* The WFS is both a lost of transaction-waits-for-transaction strings (in which each transaction is waiting for the next transaction in the string), and a lock history for each transaction in the string. E.g. WFS[T_2{W(R_2A), IW(R_3B)}; T_4{W(R_3B)}] indicates that T_1 is waiting for T_4. Here we assume that each transaction or agent will have a globally unique identifier that indicates it s site of origin. The deadlock can be detected directly from WFSs without constructing the WFG by simply detecting whether any transaction recurs more than once in the WFS, which is equivalent of having a cycle in the WFG. Each site in the system has a distributed deadlock detector (copy of the same algorithm) that performs deadlock detection for transactions or agents at that site. Several sites can simultaneously be working on detection of any potential deadlock cycle. The proposed algorithm uses staged approach for deadlock detection since it has been found that cycles of length 2 occur more frequently than cycles of length 3 which occur more than cycles of length 4 and so on. We distinguish two types of deadlock cycles: the ones that can be detected using only the information available at one site and those that require inter-site messages to detect.

In the proposed algorithm, the first type has been divided into two levels of detection activity. Level one checks for possible deadlock cycles every time a remote resource is requested and another transaction is waiting for a resource held by the transaction making the remote resource request. Since level one involves data from the lock table of one resource, it should be fast and in expensive, if the requested resource is not available after X units of time then the probability of a deadlock has increased sufficiently to justify a ore complex and time consuming check in level two. Level two requires more time since it attempts to detect the deadlock by using the lock tables of all resources at the site. Level three is intended to detect all remaining deadlocks, that is , deadlocks that require inter-site communication.

Level one of detection activity an efficiently detect direct global deadlocks of cycle length 2. The global deadlock of 2 transactions T_1 and T_2 is *direct* when T_1 and T_2 deadlock at two sites that are also the last sites at which T_1 and T_2 are executed i.e. were not blocked. Indirect global deadlocks are those that are not direct! Thus, if T_1 and T_2 execute only at two sites, they can generate only direct global deadlocks. If they execute at more than two sites, they can also result in indirect global deadlocks.

Resources can be considered of two types: Type I includes resources whose intention lock can be determined from a remote site, that is, the transaction can determine the remote

resource lock granularity and its mode before migrating to the site of the remote site. Type I resources are usually those that have just one level of granularity, namely the whole resource.

Type II consists of resources whose intention lock granularity and mode can be determined only after the transaction has migrated to the remote site. Type II can have locking on varying levels of granularity such as for e.g. Pages of a file in a distributed database system.

When a site deadlock detector receives WFS, it substitutes the latest lock histories for any transaction for which it has a later version (the longest lock history is the latest). It then constructs a new WFG or WFS and checks for cycles. If a cycle is found, then the deadlock exists. If any transactions are waiting for other transactions that have migrated to other sites, the current site must repeat the process of constructing and sending WFGs /WFSs to the sites to those sites. If these transactions are at this site and active, deadlock detection activity can cease else it will continue till a deadlock is found or it is discovered that there is no deadlock[14].

4.2 The algorithm [14]

Step1: {Remote resource R requested/anticipated by transaction or agent T}

A. If a type I remote resource is requested, place appropriate IL entry in *next* field of the lock table of the current resource (the last resource locked by T, if any) and in LH(T).
B. {Start *level one* detection activity at current site}. Construct a WFG/WFS from lock histories of all transactions holding and requesting R, and , if a type I remote resource is requested, check for deadlock.
C. If no deadlock is detected:
    - Have an agent created at the site of the requested resource and ship the WFS (generated at step 1B or step4A) there
    - stop.

Step2: {Local resource R requested}.

A. If resource R is available : {lock it}.
    (1) Place an appropriate lockin lock table of resource R and in LH(T).
    (2) Stop.
B. If the resource is not available:{Start level two detection activity}.
    (1) Place appropriate IL in lock table of resource R and in LH(T), and delay X time units
    (2) If the resource is now available:
        (a) Remove IL from lock able and LH(T).
        (b) Go to step 2A.

    (3) If the resource is not available: {continue level two activity}.
        (a) Construct a WFG /WFS using the lock histories of the transactions in the WFSs that have been sent from other sites and the lock histories of all blocked or inactive transactions at this site, and check for deadlock.
        (b) If any deadlock is found, resolve the deadlock.
        (c) If no deadlock is found, delay Y units.
        (d) If the requested resource is now available, go to step 2A.
        (e) If the transaction being awaited is at this site and active, stop.
        (f) If the resource is still not available, go to step 3{start level 3 detection activity}.

Step 3: {Wait for message generation}.

A. {Start level three detection activity}. Construct a new WFS either by condensing the latest WFG or by combining all WFSs.
B. Send the WFS to the site to which the transaction being awaited has gone if the awaited transaction in each substring has a smaller identifier than the first transaction in that sub-string and stop.

Step 4: {Wait-for message received}.

If wait-for message received DO:

A. {Start level 3 detection activity}. Construct a WFG/a new WFS from the lock histories of the transactions in he WFSs form other sites and from the lock histories of all blocked or inactive transactions at this site. (Use the latest WFS from each site) Check for a deadlock. If deadlock is found, resolve it.
B. If an awaited transaction has migrated to another site that is different from the one that sent the WFS message, go to step 3 {Repeat WFS generation}.
C. If the awaited transaction is active, stop.

4.1 Explanation of the algorithm [14]

Step1: This step is executed any time a transaction (or agent) T requests a remote resource, or when it determines that it will require a remote resource. The lock table of the resource that the transaction is currently using is checked to see whether any other transactions are waiting (to check whether other transactions have placed intention locks) for that resource. If so, the WFG is constructed by using the lock histories of all the transactions requesting and holding the resource and a check for cycles is made. If no cycles are found then a new WFS and causes an agent to be created at the site of the requested resource.

Step2: It is executed each time a local resource is requested. If the resource is available, appropriate locks are placed and the resource is granted. Intention locks are placed in the lock table of the requested resource and in the lock history of the requesting transaction in case if the resource is not available. If the resource is not available even after the *delay period* the chances of deadlock are higher so the algorithm shifts to another level of detection. It now uses the lock histories from each blocked or inactive transaction at that site, as well as from any WFSs from other sites that have been brought my migrating transactions. If there are no cycles in the new WFG or WFS, and the resource is still not available after a second delay(also tunable by the system users), the possibility of deadlock is again much higher, since the current site has insufficient information to detect the deadlock, hence the proposed algorithm progresses to the third level of detection(step 3)[14].

Step3: The wait-for message generated by this step consists of a collection of substrings each is a lost of transactions waiting for the next transaction in the substring which also lists the resources locked or intention locked by each transaction in the substring. This step includes the optimization that a WFS lexical ordering that the first transaction that has migrated has a lower lexical ordering than the first transaction in the substring. For example, for the WFG shown in Fig2, the WFS would be [T_2{W(R_2B), IW(R_3C)}; T_3{W(R_3C), IW(R_4D)}; T_4{W(R_4D), IW(R_1A)}]. T_4 has migrated to site A. The WFS would be sent to site A only if T_4's identifier is less than T_2's identifier[14].

Step4: This step is executed only after a wait-for message has been received. WFS/WFGs are generated using the lock histories of the transactions in the WFSs received previously from other sites and the lock histories of any blocked or inactive transactions(at the present site). If a deadlock is detected then it is resolved, else there is still insufficient information to detect a cycle and another iteration is performed by transferring control to step 3. The algorithm stops if the transaction being waited for is still active[14].

5. Discussion and conclusion [10] [1] [14]

The most important performance measures for distributed deadlock detection and resolution algorithms is the number of messages transmitted for detection and resolution of global deadlocks. 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 edges. Considering the approach of the 'hybrid' algorithm, which allows for parallel execution of transactions at multiple sites and for multiple modes of locks. Here, the Local deadlocks are detected by a regular cycle detection mechanism in the augmented local TWFGs without transmitting any intra-site deadlock detection messages, while that involving global deadlocks are detected by sending probes and antiprobes which enable us to construct a condensed global TWFG at each site. Probes are sent only when a global transaction with a higher priority transitively waits for another global transaction with a lower priority[10].

Assume that *n* transactions constitute a global deadlock and *n'* transactions are global. 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.

The best case results when only one global transaction can initiate probes causing at most e'-1 probes. To resolve the global deadlock no antiprobes are sent if the selected victim is a local transaction. Hence, the number of inter-site messages to detect and resolve a global deadlock becomes at most e'-1[10].

The Worst case results when n'-1 global transaction initiate probes which can cause the outgoing message-wait edge to 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'-1)*(e'-1). After detecting global deadlock and aborting the victim, at most e'-1 antiprobes will be transmitted. Hence the total number of messages to detect and resolve a global deadlock in the worst case results in the transmission of at most n'*(e'-1) inter-site messages[10].

In our second approach for deadlock detection based on the concept of ***Lock History*** the algorithm phases out in 3phases and the costly procedure of invoking deadlock detection is minimized in the sense that it processes deadlock detection in $_3$phases as explained. It also has an added advantage. It overcomes the problem that most of the algorithms that use Probes in which only those deadlocks in which the initiator is involved can be detected. If the initiator is waiting outside a deadlock, its probes are of no use in detecting the deadlock, it only adds up to the message traffic in the system[1] [14].

References

1. Kia Makki, Niki Pissinou, "*Detection and resolution of deadlocks in distributed database systems*", Proceedings of the 1995 conference on International conference on information and knowledge management, p.411-416.

2. Don P. Mitchell, Michael J. Merritt, "*A distributed algorithm for deadlock detection and resolution"*, Proceedings of the third annual ACM symposium on Principles of distributed computing, p.282-284, August 1984.

3. MARSLAND, T.A., AND ISLOOR, S.S.1980, "*Detection of deadlocks in distributed database systems*".

4. K.M.Chandy and J.Misra, *"A Distributed Algorithm for detecting Deadlocks in Distributed Systems*" pp. 157-164, August 1982.

5. M.Sfinghal, "*Deadlock detection in distributed systems*", Computer, v.22 n.11, Nov. 1989.

6. K.Sugihara, T.Kikuno, N.Yoshida and M.Ogata, "*A Distributed Algorithm for Deadlock Detection and Resolution*", October 1984.

7. "*The information structure of distributed mutual exclusion algorithms*", Beverly A. Sanders .

8. ACM Transactions on Computer Systems (TOCS) August 1987 ,Volume 5 Issue 3.

9. C.F.Yeung, S.L.Hung, K. Y. Lam and C..K. Law, "*A New Distributed Deadlock Detection Algorithm for Distributed Database Systems*", In Proceedzngs o/199,4 IEEE TENCON, pp. 506-510, 1994.

10. "*A distributed deadlock detection and resolution algorithm based on a hybrid wait-for graph and probe generation scheme*", Young Chul Park, Peter Scheuermann, Hsiang Lung Tung Proceedings of the 1995 conference on International conference on information and knowledge management December 1995.

11. "*Distributed deadlock detection algorithm*", Ron Obermarck, ACM Transactions on Database Systems (TODS) June 1982, Volume 7 Issue 2.

12. "*Local distributed deadlock detection by knot detection*", I Cidon, J.M. Jaffe, Proceedings of the ACM SIGCOMM conference on Communications architecture & protocols September 1986.

13. "*Deadlock detection in communicating finite state machines by even reachability analysis*", Wuxu Peng, Mobile Networks and Applications December 1997 ,Volume 2 Issue 3.

14. "*The Distributed Deadlock Detection Algorithm*", D.Z. Badal, Hewlett-Packard Laboratories.

15. "*A Petri net model for the performance analysis of transaction database systems with continuous deadlock detection*", Ing-Ray Chen, Rajakumar Betapudi, Proceedings of the 1994 ACM symposium on Applied computing April 1994.