Deadlock Detection Views of Distributed Database

B.M. Monjurul Alom, Frans Henskens, Michael Hannaford
School Of Electrical Engineering and Computer Science, University Of Newcastle,
Callaghan, NSW 2308, Australia

Abstract
Deadlock detection is very difficult in a distributed database system because no controller has complete and current information about the system and data dependencies. The deadlock problem is intrinsic to a distributed database system which employs locking as its concurrency control algorithm. This paper attempts a comprehensive study of deadlock detection in distributed database systems. Afterwards, a deadlock detection algorithm is presented. The algorithm is based on creating Linear Transaction Structure (LTS), Distributed Transaction Structure (DTS), finding local and global cycle, deciding priority Id of the transaction and local-global abortion. The proposed algorithm does not detect any false deadlock or exclude any really existing deadlocks. In this technique global deadlock is not dependent on the local deadlock.

Key Words: Deadlock Cycle, Priority_Id, Transaction Queue (TQ), TWFG, Transaction Manager (TM).

1. Introduction
During the last decade computing systems have undergone substantial development, which has greatly impacted on distributed database systems. While commercial systems are gradually maturing, new challenges are imposed by the world-wide interconnection of computer systems [1]. This creates an ever growing need for large scale enterprise-wide distributed solutions. In the future, distributed database systems will have to support hundreds or even thousands of sites and millions of clients and, therefore, will face tremendous scalability challenges with regard to performance, availability and administration [1]. Deadlocks can arise in each database system that permits concurrent execution of transactions using locking protocols, which is the case in most of today’s (distributed) database systems.

In modern computer systems, several transactions may compete for a finite number of resources [2]. Upon requesting a resource, a transaction enters a wait state if the request is not granted due to non-availability of the resource. A situation may arise wherein waiting transactions may not ever get a chance to change their states. This can occur if the requested resources are held by other similarly waiting transaction. This situation is called deadlock.

In a distributed database system, data access by concurrent transactions are synchronized in order to preserve database consistency [3]. This synchronization can be achieved using concurrency control algorithms such as two phase locking (2PL), timestamp ordering [4], optimistic concurrency control [5] or a variation of these basic algorithms. In practice, the most commonly used concurrency control algorithm is 2PL. However, if locking is used, a group of transactions may become involved in a deadlock [3]. Consequently, some form of deadlock resolution must accompany 2PL.

There are many algorithms implemented in centralized database systems for deadlock detection. All of them are based on finding cycles in a transaction wait for graph (TWFG) in which the nodes of the graph are used to represent transactions, with the directed edges indicating for which transactions a given transaction is waiting [6]. When cycles are detected in TWFG, they are broken by choosing a transaction that is involved in the cycle causing the transaction to fail (usually allowing the transaction to restart with the original input). This operation becomes more complex when TWFG is distributed among multiple sites of a distributed database.

In a distributed database system, although a transaction may perform all of its actions at the site in which it originates, it may also perform actions (or actions may be performed on behalf of it) at other than the original site. If this happens, an agent [7] is created at the remote site to represent the transaction at that site. This agent becomes part of the original transaction for concurrency control and recovery purposes.

Many algorithms have been proposed to detect deadlocks in distributed database systems [6, 8-11]. Some methods are based on transmitting probes between sites. Probes are special messages used to detect deadlocks. Probes (these messages) follow the edges of the wait-for graph without constructing a separate representation of the graph [8, 9, 11]. The advantage of this approach is that probe algorithms are more efficient than wait-for-graphs. The disadvantage of the probe approach is that after deadlock is detected, the constituents of the cycle remain to be discovered.

In this paper we describe an algorithm that is based on creating Linear Transaction Structure (LTS) to find local
cycles for each site of distributed database systems (DDBS). To find the global deadlock cycle Distributed Transaction Structure (DTS) is used for DDBS. In each site, each transaction has a unique priority id assigned by the transaction manager (TM); priority id is used to find the youngest transaction which caused a deadlock cycle. Transaction Queues (TQ) are used to store the transaction’s priority id which forms cycles in LTS and DTS. The proposed technique is efficient as it does not detect any false deadlock.

The remainder of this paper is organized as follows: Existing Algorithms are described in section 2. The distributed database system model and formal model of transaction processing are presented in section 3. The proposed algorithm is described in 4. Section 5 presents the explanation of the algorithm with an example. The paper concludes with a discussion and final remarks in section 6.

2. Related Work

The distributed deadlock detection algorithms that have been proposed are divided into two categories. Algorithms that belong to the first category pass information about transaction requests to maintain a global wait-for-graph. In the algorithms in the second category, simpler messages are sent among transactions; no global wait-for-graph is explicitly constructed. However, a cycle in the graph will ultimately cause messages to return to the initiator of the deadlock detection message, signaling the existence of deadlock.

2.1. Chandy & Mishra Algorithm [9]

This algorithm uses transaction wait for graphs (TWFG) to represent the status of transactions at the local sites and uses probes to detect global deadlocks. The algorithm by which a transaction Ti determines if it is deadlocked is called a probe computation. A probe is issued if a transaction begins to wait on another transaction and gets propagated from one site to another based on the status of the transaction that received the probe. The probes are meant only for deadlock detection and are distinct from requests and replies. A transaction sends at most one probe in any probe computation. If the initiator of the probe computation gets back the probe, then it is involved in a deadlock. This scheme does not suffer from false deadlock detection even if the transactions do not obey the two-phase locking protocol.

2.2. Sinha’s Scheme [11]

This algorithm, an extension to Chandy’s scheme [9] is based on priorities of transactions. Using priorities, the number of messages required for deadlock detection is reduced considerably. An advantage of the scheme is that the number of messages in the best and worst cases can be easily determined.

The author’s model consists of transactions and data managers that are responsible for granting and releasing locks. A transaction’s request for a lock on a data item is sent to the data manager for the item. If the request can not be granted, the data manager initiates deadlock computation by sending a probe to the transaction that holds a lock on the data item, if the priority of the holder is greater than that of the requestor. The transaction inserts this probe in a probe-q that it maintains. The probe is then sent to the data manager of the data item it is waiting for. At this stage of deadlock computation, priorities of transaction are used to decide whether to propagate or not. The probe is propagated only if the priority of the holder of the data item it manages is greater than that of the initiator. When a transaction begins to wait for a lock, all the probes from its queue are propagated. When a data manager gets back the probe it initiated, deadlock is detected. Since the probe contains the priority of the youngest transaction in the cycle, the youngest transaction is aborted.

2.3. Obermack’s Algorithm [6]

This algorithm at each site builds and analyzes directed TWFG and uses a distinguished node at each site. This node is called “external” and is used to represent the portion of TWFG that is external (unknown) to the site. This algorithm does not work correctly; it detects false deadlocks because the wait-for graphs constructed do not represent a snap-shot of the global TWFG at any instant.

The detection algorithm at each site performs the following steps:

- Build TWFG.
- Obtain and add information received as strings from other sites to the TWFG.
- Create wait-for edges from “external” to each node representing agent of transaction that is expected to send on communication link.
- Create wait-for edges to “external” to each node representing agent of transaction that is waiting to receive from communication link.
- Analyze the TWFG listing all elementary cycles.
- Select a victim to break each cycle that does not contain the node “external.” As each victim is chosen, remove all cycles that include victim.
- For each cycle Ex--> T1--> T2 .....TX-->Ex containing the node “external”, send a string Ex, T1, T2 ......TX to the site TX is waiting for to receive, if the transaction id of T1 is greater than that of TX.
This algorithm was the first to use a condensed 
transaction-wait-for graph (TWFG) in which the vertices 
represent transactions and edges indicate dependencies 
between transactions. This algorithm can fail to detect 
some deadlocks and may discover false deadlocks. The 
algorithm is described by the following rules:

**Rule 1:**
Event: Transaction T requests for a resource rd at site Sk, 
and rd is currently held by transactions T1, T2, ……Tn.
Action: An edge is added from the node denoting T, to 
each of the transactions T1, T2, ……Tn. If this action 
causes a cycle in TWF (k), then a deadlock exists.

**Rule 2:**
Event: A blocking pair (T, T') is received at site Sk.
Action: An edge is added from T to T' in TWF (K). If a 
cycle results, then a deadlock is detected.

2.5. Ho’s Algorithm [12]

According to this [12] approach, the transaction 
table at each site maintains information regarding 
resources held and waited on by local transactions. The 
resources table at each of the sites maintains information 
regarding the transactions holding and waiting for local 
resources. Periodically, a site is chosen as a central 
controller responsible for performing deadlock detection. 
The drawback of this scheme is that it requires 4n 
messages, where n is the number of sites in the system.

2.6. Kawazu’s Algorithm [13]

The algorithm is based on two phases. In the first 
phase local deadlocks are detected, and in the second 
phase global deadlocks are detected in the absence of 
local deadlocks. This scheme suffers from phantom 
deadlocks, because each local wait-for graph is not 
collected at the same time due to communication delays. 
Also, in case a transaction simultaneously waits for more 
than one resource, some global deadlocks may go 
undetected since the global deadlock detection is initiated 
only if no local deadlock is detected. Also Bracha’s [14], 
Mitchell’s [15] and Krivokapic’s [1] algorithms have 
been described to detect deadlock in distributed database.

3. Distributed Database System Model and 
Formal Model of Transaction Processing

A distributed database is a collection of data objects 
spread across a number of sites which communicate with 
each other via messages. Each site has data objects, 
controllers (Schedulers), and data managers. A distributed 
database model [4] is shown in figure 1. Each node has 
one or more of the following modules: a Transaction 
Manager (TM), a Data Manager (DM), a scheduler (S), 
and a Transaction Process (T). The scheduler at each site 
synchronizes the transaction requests and performs 
deadlock detection. A transaction may request multiple 
data objects simultaneously. In this case, all objects must 
be granted before the transaction can continue.

A transaction can be initiated in one site and it may 
then initiate one or more remote transactions at other 
sites. In this case the original transaction is referred to as 
the master transaction and the remote transactions as slave 
transactions. A slave can become a master by creating its 
own slave transactions.

Figure 1. Distributed database system model.

The database is considered to be distributed 
among n sites, S1, S2 … Sn, of a computer network. 
Users interact with the database via transactions [10]. A 
transaction is a sequence of actions which can be read, 
write, lock, or unlock operations. If the actions of a 
transaction involve data at a single site, the transaction is 
called local, as opposed to a distributed transaction which 
involves resources at several sites. It is assumed that 
distributed transactions are implemented as a collection of 
processes which act on behalf of the transaction. Those 
processes are called transaction incarnations. There may 
be one or more incarnations of the same transaction at 
each participating site. A transaction incarnation is 
responsible for things such as acquiring, using, and 
releasing resources to the site at which it is executing as 
needed by the transaction.

A transaction can be in two different states, 
namely, active and blocked. A transaction is blocked if its 
execution cannot proceed because a needed resource is
being held by another transaction, and the transaction is active otherwise. Figure 2, shows an example of a transaction_wait_for graph for a network with three sites S1, S2, S3. This graph shows five deadlock cycles. Three of them are local deadlocks at site {S2, S3}. The others are global deadlock cycles between {S2, S3} and {S1, S2}.

Figure 2. TWFG for a network with three sites S1, S2 and S3.

4. Proposed Distributed Deadlock Detection Technique

The proposed technique is based on calculating the following:

- Linear Transaction Structure (LTS) for each local site.
- Distributed Transaction Structure (DTS) for global resource transaction communication.
- Priority Id for each transaction in each of the sites.
- The local and global cycles.
- Abort of the victim transaction based on the cycles.

In this technique, a Linear Transaction Structure (LTS) is maintained for the transactions of each local site of the database systems and Distributed Transaction Structure’s (DTS) are used to handle the global deadlock among the distributed sites. Each transaction is assigned a unique Priority Id from the local Transaction Manager (TM) of the systems. All the local TM’s are controlled by Global TM (GTM). GTM also manages the TQ for DTS. The existence of the cycle from in local LTS represents a local deadlock and the existence of a cycle in the DTS represents a global deadlock. The proposed technique uses a graph (TWFG) to represent the transaction’s data request; that is maintained by Global Scheduler (GS), GS also collects the request (information) of the transaction from each local scheduler (S).

This technique assures that global deadlock is not dependent on local deadlock. On the other words, local deadlock does not cause the global deadlock. The victim transactions are (youngest transactions) decided based on priority id (from Transaction Queue (TQ)) from the detected cycles. This technique stores the id (number) of each transaction to their corresponding LTS and DTS whilst finding the local and global deadlock cycles. According to this approach, no false deadlock is detected or does not exclude any really deadlocked cycle.

If any transaction T_p requests a data item that is held by another transaction T_q, this technique stores the values of p and q to the linear transaction structure (LTS), where p and q represents their corresponding transaction number. Each row of the LTS stores a pair of values (p, q). Each site must have its own LTS. It is assumed that the total no of distributed sites are n, total number of transaction pairs in each LTS are N.

Distributed Transaction Structure (DTS) generally stores all the transactions which are interconnected (requests for data item from other sites) from one site to another site. DTS also records the transaction’s (those are connected to other sites) intra requests (connectivity). DTS is managed by Data Manager (DM). Each transaction should have a unique priority transaction id named \( PT_{id} \) from transaction manager (TM). Transaction queue (TQ) is used to store the \( PT_{id} \) for the abortion of the youngest (victim) transaction from a deadlocked cycle.

To find local deadlock, DM starts storing transaction’s requests for data item in LTS, from any transaction in any site. TM records priority transaction id in TQ for those transactions which form cycles in LTS; it is recommended that (any) starting transaction in the cycle has the highest priority Id (but the starting transaction in LTS could be any transaction). Whilst detecting global deadlock, at first Data Manager (DM) starts storing the intra connected transaction’s (those are connected to other sites) request in DTS from any site. After then, DM records the transaction requests from site to site. This is to provide less priority id for the transaction’s data request from one site to another site, as in general, global deadlock cycles become free from deadlock after aborting the transaction’s data request from one site to another site. Similarly GTM records priority transaction id in TQ for those transactions which form cycles in DTS. Generally in TQ, the priority id in the first position has the leading priority and the priority id in the last position has lowest priority. The priority id which has lowest priority is the youngest transaction. The following definitions are necessary to implement the technique or algorithm:
Definition 1: Each Local deadlock cycle LD_i is detected from the values of LTS_i. A global deadlock cycle GDC is calculated from a set of { DTS_1, DTS_{i+1}, DTS_{i+2}, \ldots \ldots DTS_n } . GDC is not dependent on the set of { LD_1, LD_{i+1}, LD_{i+2} \ldots \ldots LD_n }.

Definition 2: \{( \forall q \in LTS_i \cup \exists LD_i; \text{iff (LTS}_i[q_k]=LTS_i[q_{k+1}] \vee LTS_i[q_{k+2}] \vee \ldots \ldots LTS_i[q_{k+n}] ) \wedge \quad (LTS_i(q)=LTS_{i+1}[q_{p+1}] \wedge LTS_i(q_{k})=LTS_{i+2}[q_{p+2}] \wedge \ldots \ldots \wedge LTS_i(q_{k+n})=LTS_{i+n}[q_{p+n}]) \wedge \ldots \ldots LTS_i[q_{p+1}]) \text{ (LTS}_i[q],LTS_i[q_{j+1}], \ldots \ldots LTS_i[q_{p+2}] ) \ldots \ldots LTS_i[q_{p+1}])\} \ 

Definition 3: \exists yvictim(T_d(i)); TQ\{PT_{id}(1),PT_{id}(2),PT_{id}(n)\} \rightarrow \text{iff (PT}_{id}(1)>PT_{id}(2)>\ldots \ldots PT_{id}(n) ) \rightarrow \text{ abort the request of yvictim from LDi;}

Definition 4: Abort the request from yvictim (youngest victim) transaction; from each calculated LD_i through all distributed sites of the database systems.

Definition 5: \{ \forall q \in LTS_i \cup \exists LD_i; \text{iff (LTS}_i[q_k]=LTS_i[q_{k+1}] \vee LTS_i[q_{k+2}] \vee \ldots \ldots LTS_i[q_{k+n}] ) \wedge \quad (LTS_i(q)=LTS_{i+1}[q_{p+1}] \wedge LTS_i(q_{k})=LTS_{i+2}[q_{p+2}] \wedge \ldots \ldots \wedge LTS_i(q_{k+n})=LTS_{i+n}[q_{p+n}]) \wedge \ldots \ldots LTS_i[q_{p+1}])\}

Definition 6: if LTS_i \leftrightarrow LTS_{i+1} \quad ; \text{at first, DM starts storing the transaction’s (those are connected to other sites.) intra requests (connectivity) into DTS. Data Manager (DM) then records the request of transactions between LTS_i, LTS_{i+1} into DTS. GDC is detected applying definition 2 to all of the calculated DTS from the whole distributed database system (DDBS), yvictimpair is aborted from each DTS applying definition 3, to free from (GDC) global deadlock cycle.

The distributed deadlock detection algorithm is presented in Figure.3.

Algorithm Distributed-DD()
Begin
//Pindex which represents the index of the inserted
//transaction pair in LTS
Local_Deadlock_Detection();
Global_Deadlock_Detection();
End

Local_Deadlock_Detection()

Begin
For i=1 to site n do
Pindex=1;
Begin
While (not finish all transaction requests in site, ) do
If (Transaction T_p requests for data item on T_q) then
Begin
LTS_i[pindex][1]=p;
LTS_i[pindex++][2]=q;
End // for
End // while
End // for i
End // Local deadlock

Global_Deadlock_Detection();

Begin
// applying definition 5
For i=1 to n do
Begin
DTS_i=Record (Calculate) different transactions request communication among different sites;
// Apply local Deadlock Detection Techniques in each
// DTS_i
Local_deadlock_detection (DTS_i);
End
Figure 3. Algorithm distributed-dd

4.1. Explanation of the Proposed Technique

Linear transaction structures LDS1, LDS2, and LDS3 (comprising Tables 1, 2 and 3) have been created from the transaction wait for graph (TWFG) shown in Figure 2.

<table>
<thead>
<tr>
<th>Table 1: LTS1</th>
<th>Table 2: LTS2</th>
<th>Table 3: LTS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p q</td>
<td>p q</td>
<td>p q</td>
</tr>
<tr>
<td>1 2</td>
<td>3 5</td>
<td>8 7</td>
</tr>
<tr>
<td>3 1</td>
<td>5 6</td>
<td>7 10</td>
</tr>
<tr>
<td>4 1</td>
<td>6 3</td>
<td>10 9</td>
</tr>
<tr>
<td>4 3</td>
<td>5 4</td>
<td>9 7</td>
</tr>
<tr>
<td>3 4</td>
<td>6 4</td>
<td>8 9</td>
</tr>
</tbody>
</table>

According to definition 2, in LTS1 there is no deadlock cycle; in LTS2 the deadlocked cycle LD2 = \{3 \rightarrow 5, 5 \rightarrow 6, 6 \rightarrow 3\} and in LTS3 the deadlocked cycle LD3 = \{7 \rightarrow 10, 10 \rightarrow 9, 9 \rightarrow 7\}. In TQ (Transaction Queue comprising table 4 & 5), all the transaction’s priority id from LD2 and LD3 are recorded by TM. According to definition 3, the yvictim (youngest transaction) is 6 in LD2 and 9 in LD3. The data requests for these transactions are bolded in (LTS2) Table 2 and (LTS3) Table 3. According to definition 4, these transaction pairs \{6 \rightarrow 3\} and \{9 \rightarrow 7\} are aborted. The remaining transaction’s data requests in Table1 (LTS2) still have a cycle which is LD2 = \{10 \rightarrow 9, 9 \rightarrow 11, 11 \rightarrow 10\}. Similarly the yvictim transaction’s request \{11 \rightarrow 10\} in LD2 is aborted from LTS2. After aborting the victim transactions from LTS2, and LTS3; the remaining transaction’s data requests in Table 8 are deadlock free.

<table>
<thead>
<tr>
<th>Table 4: TQ2</th>
<th>Table 5: TQ3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_{no} TP ID</td>
<td>T_{no} TP ID</td>
</tr>
<tr>
<td>3 1</td>
<td>7 1</td>
</tr>
<tr>
<td>5 2</td>
<td>10 2</td>
</tr>
<tr>
<td>6 3</td>
<td>9 3</td>
</tr>
<tr>
<td>11 4</td>
<td></td>
</tr>
</tbody>
</table>

The remaining transaction’s data requests in Table1 (LTS3) still have a cycle which is LD3 = \{10 \rightarrow 9, 9 \rightarrow 11, 11 \rightarrow 10\}. Similarly the yvictim transaction’s request \{11 \rightarrow 10\} in LD2 is aborted from LTS3. After aborting the victim transactions from LTS2, and LTS3; the remaining transaction’s data requests are presented in Table 8 and Table 9 which are deadlock free.

According definition 6, site1 and site2 are interconnected through transaction requests; also site2 and site3 are interconnected through transaction requests. In site1 T3 requests data item that is also held by T3 in site2 and T4 in site1 is requested by T4 from site2. In site1, transaction T4 requests the data item that is held by T3. Data Manager (DM) records at first, the intra connected transaction’s request from any site (those are connected to other sites.) in DTS.

<table>
<thead>
<tr>
<th>Table 6: DTS1</th>
<th>Table 7: DTS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>p q</td>
<td>p q</td>
</tr>
<tr>
<td>4 3</td>
<td>6 4</td>
</tr>
<tr>
<td>3 3</td>
<td>4 8</td>
</tr>
<tr>
<td>3 4</td>
<td>8 7</td>
</tr>
<tr>
<td>4 4</td>
<td>7 6</td>
</tr>
</tbody>
</table>

After then, DM records the transaction’s request from site and site. Hence, distributed transaction structures DTS1 and DTS2 are also created (comprising Tables 6, 7) according to the definition 6. TQ for DTS1 and DTS2 are considered as it is created for LTS. Applying definition 2, 3 and 4 in DTS1 and DTS2, transaction pairs \{4 \rightarrow 4\}, and \{7 \rightarrow 6\} are selected as yvictimpair and aborted to free from the global deadlock cycle. The global deadlock free DTS1 and DTS2 are presented in Table 10, and Table 11.

<table>
<thead>
<tr>
<th>Table 8: D-lock free LTS2</th>
<th>Table 9: D-lock free LTS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>p q</td>
<td>p q</td>
</tr>
<tr>
<td>3 5</td>
<td>8 7</td>
</tr>
<tr>
<td>5 6</td>
<td>10 9</td>
</tr>
<tr>
<td>3 4</td>
<td>8 9</td>
</tr>
<tr>
<td>6 4</td>
<td>9 11</td>
</tr>
</tbody>
</table>

Considering the transaction requests from Table8, Table 9, Table 10, and Table 11, the transaction wait for graph (TWFG) is presented in Figure 4. According to this approach, the TWFG shown in figure 2, is presented in figure 4 without having any local or global deadlocks.
5. Conclusion

Handling deadlock involves two problems: deadlock detection and deadlock resolution. In a DBMS, deadlock resolution means that one or more of the participating transactions, the victim, is chosen to be aborted, thereby resolving the deadlock [1]. A deadlock detection algorithm or technique is correct if it satisfies two conditions: (1) every deadlock is eventually detected, and (2) every detected deadlock really exists, i.e., only genuine deadlocks are detected.

Obermack’s [6] algorithm does not work correctly; it detects false deadlocks because the wait-for graphs do not represent a snap-shot of the global TWFG at any instant. Menasce’s Scheme [10] can fail to detect some deadlocks and may discover false deadlocks. Ho’s Algorithm [12] uses the transaction table and resources table at each site to maintain information regarding resources; the drawback of this scheme is that it requires 4n messages, where n is the number of sites in the system. Kawazu’s Algorithm [13] suffers from phantom deadlocks, because each local wait-for graph is not collected at the same time due to communication delays also some global deadlocks may go undetected since the global deadlock detection is initiated only if no local deadlock is detected.

In this paper we have presented an approach to detect both local deadlocks and global deadlocks. This technique assures that global deadlock detection is not dependent on local deadlock detection. The proposed algorithm does not detect any false deadlocks and every detected deadlock really exists. The technique uses TQ (Transaction queue) to store the priority id for all transactions which are in local deadlock cycles or in global deadlock cycles. Based on the priority id, the youngest transactions are aborted to free the system from deadlock cycles.

6. References


