A Distributed Algorithm for Resource Deadlock Detection Using Time Stamping

Himanshi Grover

Faculty of Engineering and technology

Manav Rachna International University

Faridabad, India

ABSTRACT

Deadlock is one of the most serious problems in multitasking concurrent programming systems. The deadlock problem becomes further complicated when the underlying system is Distributed. Deadlock detection and optimization is very difficult in distributed systems. The deadlock problem is intrinsic to a distributed database systems which employs locking at its concurrency control algorithm. Till now various techniques have been introduced to detect and prevent deadlocks

This paper presents a simple algorithm to detect resource deadlocks in distributed databases. The proposed algorithm is an improvement over the algorithm by B.M Jhonston. In the original algorithm there were no priority criteria to decide that which transaction needs to be aborted. But the proposed algorithm makes this decision by using the timestamps of transactions. Accordingly the youngest transaction is aborted. The algorithm ensures that only one process in the deadlock cycle will detect it, thus simplifying the resolution problem. All true deadlocks are detected in finite time and no false deadlocks are reported. An informal proof of correctness of the algorithm and an example are also presented.

Keywords

Deadlocks, Distributed database systems, Distributed database management systems

1. INTRODUCTION

A deadlock occurs if each of two transactions (for example, A and B) needs exclusive use of some resource (for example, a particular record in a data set) that the other already holds. Thus a Transaction A waits for the resource to become available to it which is being withheld by B. However, if transaction B is not in a position to release it because it, in turn, is waiting on some resource held by A, both are therefore deadlocked and the only way of breaking the deadlock is to cancel one of the transactions, thus releasing its resources.

The deadlocks occur when several queries/transactions are being attempted by same or different users by accessing the same data. These have to be sequenced to prevent or resolve the deadlocks.

A distributed system can be visualized as a set of sites, each site consisting of a number of independent transactions. A distributed database is a database in which storage devices are not all attached to a common CPU. It may be stored in multiple computers located in the same physical location, or may be dispersed over a network of interconnected computers.

Collections of data can be distributed across multiple physical locations. A distributed database can reside on network servers on the Internet, on corporate intranets or extranets, or on other company networks.

A distributed database management system ('DDBMS') is a software system that permits the management of a distributed database and makes the distribution transparent to the users.DDBMS is software for managing databases stored on multiple computers in a network.

No transaction knows the global state of the whole system. The transactions communicate through messages. The communication is asynchronous and a message may take an arbitrary but finite time. Several deadlock detection algorithms have been published. Messages sent from process A to process B are received by process B in the same order as they were sent. To detect the presence of deadlock in the proposed algorithm we do not use probe messages. Instead we use a update message, one of its function being to check for the occurrence of deadlock.

2. RELATED WORK

As mentioned earlier deadlock occurs whenever transactions need access simultaneously to data that has been by each other and none is able to complete the transaction. In the literature various techniques have been discussed both in stand alone systems as well as in distributed systems. In fact, the distributed Computing environment provides a tightly coupled facilities which are targeted towards a common goal, whereas stands alone system is more of an end-to-end architecture which also facilitates enveloping a distributed computing environment.

We shall discuss about techniques which can detect local as well as global deadlocks. Whenever a cycle is created i.e. when the message initiated by the initiator again comes back to the transaction initiating it and thus we conclude that deadlock has occurred.

Chandy & Mishra [6] has presented an 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. 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

G.S Ho and **C. V. Ramamurthy [9]** have presented a new approach in which 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.

Chim-fu young[12] has proposed an algorithm which is an improvement over Chandy , Mishra and Haas algorithm in which Deadlock is found by passing special messages called probe messages along the edges of a wait-for graph .As compared to chandy's algorithm :

- ✓ This algorithm is error free
- ✓ It suffers very little performance degradation as compared to the original one .
- ✓ Even for large values of multiprogramming Level, the probe based algorithm can Outperform time –out .
- ✓ The rate of probe initiation is a dominant Factor in determining system's performance

In this algorithm we focus on clearing the dependency table regularly. If the dependency tables are cleared periodically then the number of probe messages will be great as many probes needed to be propagated. As the controller initiates the probes periodically, the number of probe messages could be used to estimate the total blocking time period of a process.

Brian M. Johnston[13] ,et al. has presented an algorithm to detect the presence of deadlock in the proposed algorithm we do not use probe messages. Instead we use a update message The function of update message is two fold: first to modify the Wait-for variables and second to check the occurrence of deadlock. As compared to many recent algorithms in this field, the proposed algorithm can detect the most frequent deadlocks with minimum message passing. With message complexity defined as the number of messages transmitted between initiating a update message and detecting the cycle. In the worst case, in a system of n transactions, the overall message complexity of the proposed algorithm is O (n)

3. THE PROPOSED ALGORITHM

Each site in the network carries a unique site identifier called site_ID. Within the network a site maintains a certain portion of the database. Each site owns some data objects and maintains **a** few transactions. Each data object is identified by a unique identifier denoted by Data-obj. Every data object controlled by a site has a variable called Locked-by. The variable Locked-by determines the current state of the data object. If the data object is not locked by any transaction, Locked-by will store nil, else, it stores the identification of the locking transaction.

Each transaction has a unique site identifier denoted by T-ID. Each transaction has a timestamp value (TS) associated with it, which tells us when the transaction has entered into the system. A transaction can use data objects within its own site or make explicit requests for a data object in another site. As each site has a unique Site_ID, and every transaction within a site has a unique T_ID, the T-ID can be considered to be unique throughout network.

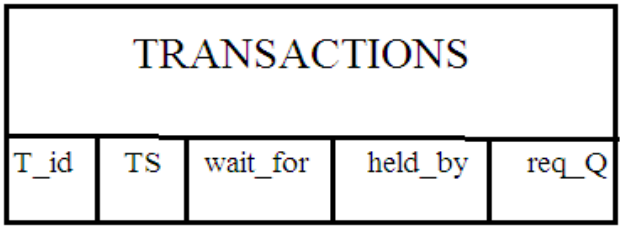

Figure 1: Structure of each transaction

Each transaction Ti at site Si has the following data structure: a variable called Wait-for (Ti), a variable called Held-by (Ti), and a queue of requesting transactions Request-Q (Ti). If the current transaction is not waiting for any other transaction then Wait-for(Ti) is set to nil, else, it denotes which transaction is at the head of the locked data object. If Held_by (Ti) is set to nil if the current transaction is executing, else it stores the transaction that is holding the data object required by the current transaction. Request-Q (Ti) contains **a**ll **o**utstanding requests for data objects which are being held by the transaction Ti. Each element in the Request-Q(Ti) is a tuple (Tj,Di), where Tj is the requesting transaction and **Di** is the particular data object held by Ti..

Suppose a transaction Ti makes **a** lock request for a data object **Dj.** If **Dj** is free then Dj is granted to TI and Locked_by (Dj) is set to Ti. If Dj is not free then Dj sends **a** not granted message to TI along with the transaction identifier locking Dj (henceforth, called Ti). Ti joins the Request-Q(Tj) and sets its Wait-for equal to Wait-for(T,). Now Ti initiates a update message to modify all the Wait-for variables which are affected by the changes in Locked_by variable of the data objects. Update message is **a** recursive function call that will continue updating all elements of every Request-Q in the chain. When a transaction Tj receives the update message it checks if its Wait-for value is the same as the new Wait-for value. If it is not the same then the value is modified. Now, **a** check for deadlock is performed.

If a deadlock is not detected then the update message is forwarded, else deadlock is declared and deadlock resolution is initiated.

Our algorithm is divided into two phases .The first phase deals with the detection of deadlocks and the second phase deals with its resolution. The first phase is similar to the one presented by B.M Jhonston [13] and is mentioned above but the technique used for resolution in his algorithm does not have any particular criteria for deciding the priority that which transaction needs to be aborted and hence a new resolution technique using time stamping has been presented here.

The proposed resolution technique works as follows: The timestamps of the transaction detecting the deadlock and the transaction which is the result of intersection of wait_for and request_Q of the detecting transactions are compared. The younger transaction out of the two is aborted. This chosen transaction sends a clear message to the transaction holding its requested data object. It also allocates every data object it held to the first requester in its Request-Q and enqueue remaining requesters to the new transaction. The transaction receiving the clear message purges the tuple in its Request-Q having the aborting transaction as an element

DEADLOCK RESOLUTION

{initiate deadlock resolution as follows}
{Compare timestamps of transaction Ti and Tj}
If (TS (Ti) < TS (Tj))
Then abort Tj;
Else abort Ti;
[let the aborted transaction be Tx]
{The aborted transaction releases all the data objects it holds}
Send clear (Tx, Held-by (Tx));
allocate each data object Di held by Tx to
the first requester T k in Request-Q (Tx);
for every transaction Ti in Request-Q (Tx)
requesting data object **Di** held by Tx
Enqueue (Tr, Request-Q (Tk));
end;
end;

{Transaction Tk receiving a clear(Tj, Tk)
message}
begin
purge the tuple having Tj as the requesting
transaction from Request-Q(Tk);
end:

4. LLUSTRATIVE EXAMPLE

4.1 EXAMPLE 1

Consider a distributed database with seven transactions as shown in Figure 2**.** The state of each transaction is also shown in the figure. However, it does not necessarily imply that each transaction resides in the same site. Figure 2 shows the state of the system when the deadlock has occurred. When Transaction To makes a request to Transaction T3, a cycle is created. Then as shown in Table 1, T0 **joins** the Request-Q of T3. TO will update its Wait-for to reflect the current state and will propagate the update message to all elements in its own Request-Q. This continues until T3 discovers that Wait-for (T3) intersected with Request-Q(T3) is not nil. Now, T3 declares deadlock and is chosen as the transaction to be aborted. T3 sends a clear message to T2 , so that T3 is purged from Request-Q(T2). T3 also releases the data objects it held, thus making transactions T4 and T6 to start executing.

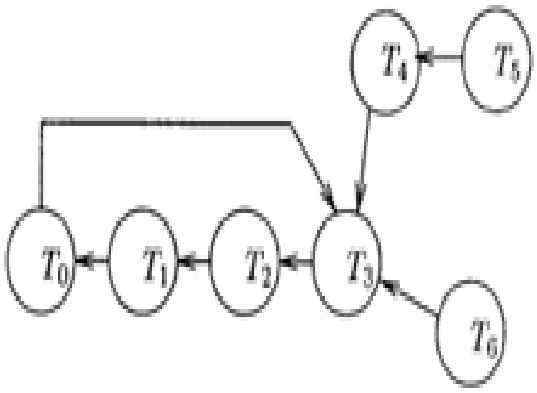

Figure 2: Single site with a number of transactions

We compare the timestamps of transactions T0 and T3

Table 1: Transaction structure for Figure 2

T_ID	TS	Wait_for	Held_by	Request_Q
T0	4	T0	T3	T1
T1	7	T0	T0	T2
T2	6	T0	T1	T3
T3	1	T0	T2	T4,T6,T0
T4	2	T0	T3	T5
T5	3	T0	T4	NIL
T6	5	T0	T3	NIL

According to this table, the intersection values are as follows:

Wait_for(T0) ∩ Request_Q (T0) = nil

Wait_for(T1) ∩ Request_Q (T1) = nil

Wait_for(T2) ∩ Request_Q (T2) = nil

Wait_for(T3) ∩ Request_Q (T3) = T0

Wait_for(T4) ∩ Request_Q (T4) = nil

Wait_for(T5) ∩ Request_Q (T5) = nil

Wait_for(T6) ∩ Request_Q (T6) = nil

Timestamp of transaction T0 [TS (T0) = 4] is greater than timestamp of transaction T3 [TS (T3) = 1]. This means that transaction T0 is younger as compared to transaction T3. And hence we will abort transaction T0 and it would release all its resources.

4.2 Example 2

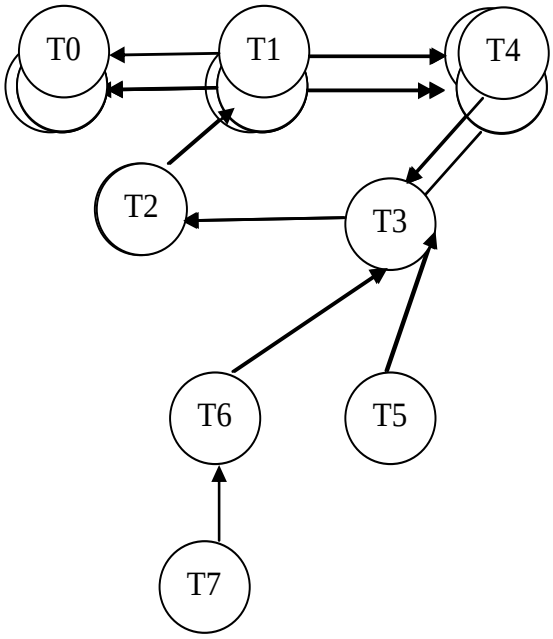

Figure 3: A site having a Eight transactions

As per the directed edges shown in figure 3, we will fill up the corresponding variable values in Table 2.

Table 2: Transaction structure for Figure 3

T_ID	TS	Wait_for	Held_by	Request_Q
T0	4	NIL	NIL	T1
T1	8	T0,T4	T0,T4	T2
T2	1	T0,T4	T1	T3
T3	3	T0,T4	T2	T4 , T5, T6
T4	2	T0,T4	T3	NIL

T5	5	T0,T4	T3	NIL
T6	7	T0,T4	T3	T7
T7	6	T0,T4	T6	NIL

According to the above table, the intersection values are as follows:

Wait_for(T0) ∩ Request_Q (T0) = nil

Wait_for(T1) ∩ Request_Q (T1) = nil

Wait_for(T2) ∩ Request_Q (T2) = nil

Wait_for(T3) ∩ Request_Q (T3) = T4

Wait_for(T4) ∩ Request_Q (T4) = nil

Wait_for(T5) ∩ Request_Q (T5) = nil

Wait_for(T6) ∩ Request_Q (T6) = nil

Wait_for(T7) ∩ Request_Q (T7) = nil

From the above values of intersection we can see that the intersection values of T3 is not nil rather it is T4 .And hence we say that deadlock has been detected . Now we will compare the timestamps of transactions T3 and T4 . TS (T3) = 3 and TS (T4) =4

Since TS (T3) < TS (T4) ,so we can say that transaction T4 is younger transaction . And hence we will abort T4 . So that it releases all the resources held by it..

Figure 4 shows a deadlock free TWFG of figure 3

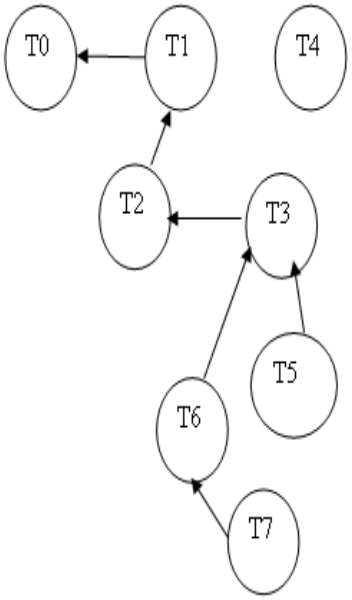

Figure 4 : Deadlock free TWFG of figure 3

4.3 EXAMPLE 3

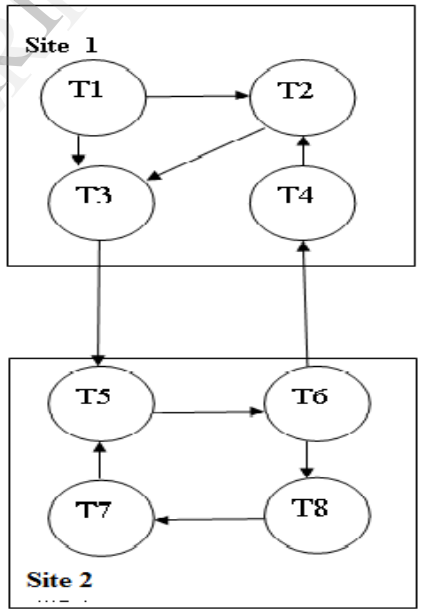

Figure 5 : A distributed environment having 2 sites

Table 3 : Transaction structure for site 1 in figure 5

T_ID	TS	Wait_for	Held _by	Requst_Q
T1	2	T2	T2 ,T3	NIL
T2	1	T2	T3	T4
T3	3	NIL	NIL	T1 , T2
T4	4	T2	T2	NIL

Wait_for(T0) ∩ Request_Q (T0) = nil

Wait_for(T1) ∩ Request_Q (T1) = nil

Wait_for(T2) ∩ Request_Q (T2) = nil

Wait_for(T3) ∩ Request_Q (T3) = nil

Since all the intersection values are nil So we conclude that there exists no deadlock.

Table 4 : Transaction structure for site 2 in figure 5

T_ID	TS	Wait_for	Held_by	Request_Q
T5	**3**	**T6**	**T6**	**T7**
T6	**1**	**T6**	**T8**	**T5**
T7	**2**	**T6**	**T5**	**T8**
T8	**4**	**T6**	**T8**	**T6**

Wait_for(T5) ∩ Request_Q (T5) = nil

Wait_for(T6) ∩ Request_Q (T6) = nil

Wait_for(T7) ∩ Request_Q (T7) = nil

Wait_for(T8) ∩ Request_Q (T8 = T6

Since intersection values for transaction T8 is not nil. We will compare timestamps of both T8 and T6 and we find that the TS (T6) < TS (T8). So we will abort T8 as it is the younger transaction.

Table 5 : Transaction structure for site 1 and 2 in figure 5

T_ID	TS	Wait_for	Held_by	Request_Q
T1	7	T2	T2, T3	NIL
T2	1	T2	T3	T1 , T4
T3	5	T2	T5	T1 , T2
T4	2	T2	T2	T6
T5	6	T6	T6	T7
T6	3	T6	T8	T5
T7	8	T6	T5	T8 NIL
T8	4	T6	T7	T6

Wait_for(T0) ∩ Request_Q (T0) = nil

Wait_for(T1) ∩ Request_Q (T1) = nil

Wait_for(T2) ∩ Request_Q (T2) = NIL

Wait_for(T3) ∩ Request_Q (T3) = T2

Wait_for(T5) ∩ Request_Q (T5) = nil

Wait_for(T6) ∩ Request_Q (T6) = nil

Wait_for(T7) ∩ Request_Q (T7) = nil

Now we will compare timestamps of transactions T2 and T3. TS(T2) < TS(T3) So we will abort transaction T3.

After aborting transaction T3, we will get a deadlock free TWFG in figure 6.

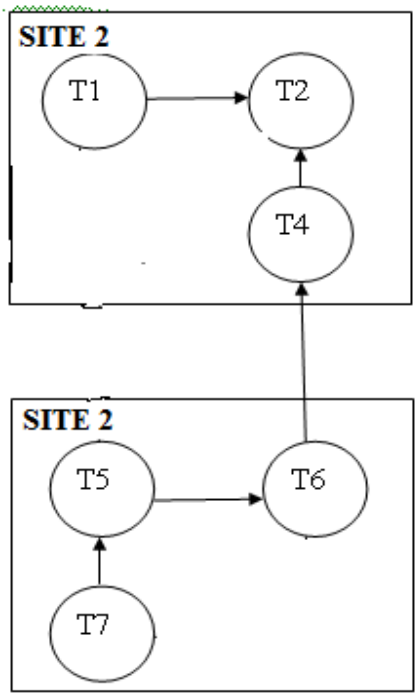

Figure 6: Deadlock free TWFG for figure 5

CONCLUSION

A simple algorithm for deadlock detection in distributed systems is presented. In the above proposed algorithm, we do not use probe messages to detect deadlock. However, we use the update message whose function is two fold: first to modify the Wait-for variables and second to check the occurrence of deadlock. Along with this update message we make use of time stamping technique which is a better way of deciding the priority of the transactions . And hence to decide that which transaction needs to be aborted.

5. REFERENCES

[1] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," ACM, vol. 13:2, pp. 186-221, 1981.

[2] H. T. Kung and J. T. Robinson, "Optimistic Methods for Concurrency Control," ACM Transaction on Database Systems, vol. 6, pp. 213-226, 1981.

[3] R. Obermarck, "Distributed Deadlock Detection Algorithm," ACM Transaction on Database Systems, vol. 7:2, pp. 187-208, 1982.

[4] J. N. Gray, "A discussion on distributed systems," IBM Research Division, 1979.

[5] K. M. Chandy, J. Misra, and L. M. Hass, "Distributed Deadlock Detection," ACM Transaction on Computer Systems, vol. 1:2, pp. 144-56, 1983

[6] X. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems " in First proceedings of ACM ,pp-157-164, 1982.

[7] D. A. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases" IEEE Transaction on Software Engineering, vol. 5:3, pp. 195-202, 1979.

[8] M. K. Sinha and N. Natarjan, "A Priority Based Distributed Deadlock Detection Algorithm" IEEE Transaction on Software Engineering, vol. 11:1, pp. 67-80, 1985.

[9] G. S. Ho and C. V. Ramamurthy, "Protocols for Deadlock Detection in Distributed Database Systems" IEEE Transaction on Software Engineering, vol. 8:6, pp. 554-557, 1982.

[10] S. Kawazu, S. Minami, K. Itoh, and K. Teranaka, "Two-Phase Deadlock Detection Algorithm in Distributed Databases " in 5th IEEE conference, 1979.

[11] H. Wu, W.-N. Chin, and J. Jaffar, "An Efficient Distributed Deadlock Avoidance Algorithm for the AND Model," IEEE Transactions on Software Engineering, vol. 28:1, pp-18-29, 2002.

[12] Chim_fu Yeung, "A new distributed deadlock detection algorithm for distributed databases systems", Department of Computer science, pp-506 -510, 1983.

[13] Brian M. Johnston, Ramesh Dutt Javagal: Ajoy Kumar Datta, Sukumar ghosh, "A Distributed Algorithm for Resource Deadlock Detection", Department of Computer Science , IEEE Transactions,vol 11, pp-252 -256 ,1991