Detection and Resolution of Deadlocks in Distributed Database Systems

Kia Makki
Department of Computer Science
University of Nevada, Las Vegas
Las Vegas, Nevada 89154
kia@unlv.edu

Niki Pissinou
The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
pissinou@cacs.usl.edu

Abstract

Deadlock detection and resolution is one of the major component of a successful distributed database management system. In this paper, we discuss deadlock detection and resolution strategies and present two approaches for detecting and resolving deadlocks in both general distributed database systems and in distributed real-time database systems. Our first approach is to collect information on connectivity of nodes of the overall Transaction Wait-For Graph (TWFG) of the distributed database system and then use these connectivities information to build a local TWFG at each node of the overall TWGF. We then detect the deadlocks by locating the cycles in each local TWFG. To resolve the deadlocks the nodes involved in those cycles in each local TWFG, are removed until there is no cycle in the local TWFGs. Our second approach continuously checks for the occurrences of a deadlock between different transaction trees. As soon as it detects a deadlock it resolves it by aborting one of the transaction tree which has been initiated more recently. Some of the advantages of our approaches over the approaches which are using Probe messages are: (1) no extra storage required to store different probe messages, (2) no false (Phantom) deadlocks are reported, (3) detects and resolve all deadlocks. In addition, our approaches use less messages and time to detect and resolve all deadlocks in the existing TWFG of the distributed database system.

1 Introduction

One of the major problems in the design of a distributed database system is the detection and resolution of deadlocks. The deadlock problem is intrinsic to a distributed database system where locking is used as a mean of supporting concurrency control strategy. In distributed database systems, users access the data objects of the database by executing transactions. A transaction can be viewed as a process that performs a sequence of reads and writes on the data objects. The objective of concurrency control is to allow concurrent execution of transactions and at the same time maintaining the consistency of the database. In such an environment a transaction must lock a data object before accessing it. The use of locking for concurrency control introduces the possibility that one transaction, may be suspended because it requests a lock held by another transaction. One of the commonly used concurrency control mechanism which guarantees consistency is based on two phase locking scheme [Bernstein and Goodman]. The two phase locking imposes a constraint on lock request and lock release actions of a transaction. In such a system a deadlock occurs when a set of transactions are circularly waiting for each other to release resources.

Detecting deadlocks in distributed database systems is a difficult task since no site has a complete and up to date information about the entire system. Over the past several years many distributed deadlock detection and resolution algorithms have been proposed [2, 4, 6, 7, 8, 9, 10, 12, 13, 14, 15, 17, 18, 19, 20, 21, 23, 27, 28, 29, 30]. A good survey of some of these distributed deadlock detection algorithms can be found in [11, 16, 25, 26].

In this paper, we discuss some deadlock detection and resolution strategies, and present two approaches for detecting and resolving deadlocks in both general distributed database systems and in distributed real-time database systems. Our first approach ,collects information on connectivity of nodes of the overall Transaction Wait-For Graph (TWFG) of the distributed database system and then uses the information of these connectivities to build a local TWFG at each node of the overall TWGF. It then detects the deadlocks, by locating the cycles in each local TWFG. To resolve the deadlocks the nodes involved in those cycles in each local TWFG are removed until there is no cycle in the local TWFGs. Our second approach, continuously checks for the occurrences a deadlock between different transaction trees. As soon as it detects a deadlock it resolves it by aborting one of the transaction tree which has been initiated more recently. Some of the advantages of our approaches over the approaches which are using Probe messages are: (1) no extra storage required to store different probe messages, (2) no false deadlocks are reported, (3) detects and resolve all deadlocks. In addition, our approaches use less messages and time to detect and resolve all deadlocks in the existing TWFG of the distributed database system.

The remainder of this paper is structured as follows. In the next section we present an overview of the existing deadlock detection and resolution algorithms for distributed database systems. In section 3, we provide our distributed database model and the assumptions that are necessary for describing our deadlock detection and resolution approaches. Section 4, presents our first approach for

CIKM '95, Baltimore MD USA

detecting and resolving deadlocks in distributed database systems. Section 5, presents our second approach and its application to distributed real time database systems. We conclude with some final remarks.

2 Related Research

During the past several years, a significant amount of effort has been focused on the development of the correct and efficient deadlock detection and resolution algorithms for distributed database systems. One brute force way to abort and resolve deadlocks in distributed database systems, is to use timeouts. In this approach, each transaction is given a specified time interval to accomplish its task. A transaction will be aborted if it has waited for more than its time interval, after issuing a lock request. Since finding 'a correct timeout interval is very difficult in such an environment, this strategy is not very practical. Furthermore, this strategy may cause a transaction to be aborted and restarted repeatedly. To avoid this kind of situation we need to make sure that we abort only those transactions which are directly involved in a deadlock. This has led many researchers to discover different deadlock detection algorithms.

In general, deadlock detection algorithms for distributed database systems can be classified into three groups [16, 26]: Centralized, Hierarchical and Distributed.

The simplest way to detect and resolve deadlocks in distributed database systems is to appoint one site as a Central site which will be responsible for constructing the global TWFG, and then to search it for possible cycles [13, 15]. The central site may maintain the global TWFG constantly, or it may construct it whenever there is a need for deadlock detection. In such a centralized system all sites request and release resources by sending request/release resource messages to the central site. When the central site receives a request/release resource message it updates its TWFG. The central site continuously checks for deadlocks in the TWFG by searching for cycles in TWFG. Although a centralized deadlock detection algorithm is simple and easy to implement (and may be practical and efficient for local networks), it may impose a very large communications cost in geographically distributed systems. An obvious drawback of this algorithm is that the central site is unfairly burdened with responsibility and with message traffic. Also, the reliability is poor because if the central site which has all the status information fails, the entire system comes to halt.

In the hierarchical deadlock detection algorithms [15], sites are arranged in a hierarchical fashion (or tree), and a site detects deadlocks involving only its descendant sites. When these hierarchical deadlock detection algorithms are used, sites are grouped as clusters based on resource access patterns, and clusters are organized in a hierarchical fashion. In each cluster, a designated site is responsible for detecting deadlock within the cluster using a centralized algorithm. Unlike centralized algorithms the entire system does not come to halt if a single site fails. Also, sites that do not belong to clusters involved in deadlocks are not bothered with the deadlock detection activities. The hierarchical deadlock detection algorithms would not be very effective, if most deadlocks involved many clusters.

Distributed deadlock is harder to detect, since each site has only a local view of the whole system, and therefore collaboration of the sites is required to detect deadlocks involving more than one site. In distributed deadlock detection algorithms, all sites collectively cooperate and equally contribute in detecting and resolving deadlocks. Unlike centralized deadlock detection and resolution algorithms, distributed ones are not vulnerable to a single point of failure. Also no site is swamped with deadlock detection activity.

Due to the lack of a globally shared memory, distributed deadlock detection algorithms are difficult to design. A distributed deadlock detection algorithm can be initiated whenever a transaction is forced to wait. Over the past several years many distributed deadlock detection and resolution algorithms have been proposed [6, 7, 8, 23].

There are two main approaches for detecting deadlocks in distributed databases. The first approach is to construct a global system state [20, 23]. The second approach is to send a special message called a Probe [7]. A Probe travels along the edges of the global TWFG, and a deadlock is detected when a probe message returns to its initiating process.

One of the earliest distributed deadlock detection algorithm is by Menasce and Muntz [20]. They have utilized the idea of of a *Transaction Wait-For Graph*for a centralized case. The basic idea underlying this algorithm, is to build a local Transaction Wait-For Graph at each site. During a deadlock detection process each site sends its local TWFG to a number of neighboring sites. Each site after updating its local TWFG, sends its local TWFG to its neighboring sites. This process continues until eventually some site develops a complete TWFG of the entire distributed database system, and then checks the TWFG for existence of deadlocks. Obermarck's algorithm [23] improves upon Menasce and Muntz's algorithm by reducing the number of messages required for passing the local TWFG around in order to construct the global TWFG and decreasing deadlock detection overhead. This improvement is achieved by providing total ordering for transactions, and by allowing each site to extract the nonlocal portion of the global TWFG and then send it to its neighboring sites for constructing the global TWFG. One of the advantages of these kind of algorithms is that they do not require that the global TWFG be built and maintained in order for deadlocks to be detected. In the literature, these kind of algorithms are referred to as a *Path-Pushing algorithms.*

Unfortunately, both algorithms have been shown to be incorrect. Menasce and Muntz's algorithm has been shown to be incorrect by Gliger and Shattuck [12], and Obermarck's algorithm has been shown to be incorrect by Knapp [16] and Elmagarmid [11]. Specifically, they shat these algorithms can not detect all the deadlocks, or that they detect *Phantom* deadlocks (deadlocks that are non-existent). The reseason for this is that the local TWFGs that are propagated to other sites may not collectively represent the global TWFG.

Moss [22] was first to use special messages called Probes in order to detect deadlocks in distributed database systems. Moss's algorithm propagates special messages, called Probes, along the edges of the TWFG in order to search for cycles in TWFG. Probe messages are concerned exclusively with deadlock detection and are distinct from resource requests/releases. A Probe is a triplet (i, j, k) denoting that it belongs to a deadlock detection initiated for process P_i, and it is being sent from process P_j on one site to process P_k on another site [26]. In Moss's algorithm search for deadlocks is initiated whenever a transaction becomes blocked and waits for another transaction. A transaction does not maintain any information regarding transactions that wait for it. Hence, the algorithm requires transactions to initiate deadlock detection activities periodically.

Moss's algorithm was later improved by Chandy and Misra [6], Mitchell and Merritt [21] and Sinha and Natarajan [27]. In Mitchell and Merritt's algorithm, Probes are sent

in the opposite directions of the edges of the TWFG. When the Probe comes back to its initiator, the initiator declares deadlock. Sinha and Natarajan's algorithm, improves these algorithms by using priorities for transactions to minimize the number of messages initiated for detecting deadlocks. In their algorithm a deadlock detection is initiated only when there is an antagonistic conflict (an antagonistic conflict is said to occur when a transaction waits for a data object that is locked by a lower priority transaction[26]).

In the literature [16,26], these kind of algorithms are referred to as an *Edge-Chasing algorithms*. The problem with most of the algorithms that use Probes [7,8], is that 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 message traffic in the system.

3 Distributed Database Model and Assumptions

A distributed database system consists of a collection of N database sites, S_1, S_2, ..., S_N, (each of which constitutes a centralized database system) which do not share a common memory or clock. The database sites are interconnected through a communication network, and the sites communicate with each other only by sending messages. No assumption is made regarding the underlying network topology. However, it is assumed that the underlying network is reliable and sites do not crash. It is further assumed that messages send by any site arrive sequentially and in finite time.

In such system, users access the data objects of the data base by executing transactions. A transaction can be viewed as a process that performs a sequence of operations (such as read, write, lock, or unlock) on the data objects. The data objects of a database can be viewed as resources that are requested/ released by transactions. A transaction may consists of several subtransactions, that normally execute at different sites. We assume that if a single transaction runs by itself in a distributed database system, it will terminate in finite time and release all resources. We also assume that there are K transactions denoted by, T_1, T_2, ..., T_K running simultaneously on the distributed database.

A transaction can be in one of two states: *active* or *blocked*. If a transaction has a lock request pending then it is in the bloked state, otherwise it is active. A transaction changes its state from active to *wait* if its lock request can not be granted. We assume that distributed transactions are represented by a group of processes which act on behalf of the transactions.

Most deadlock detection algorithms for distributed data base systems, detect deadlocks by finding cycles in a Transaction Wait-For Graph (TWFG), in which each node represents a transaction, and a directed edge from one transaction T_i to another transaction T_j indicates that T_i is waiting for a data object locked by transaction T_j. Deadlocks can be resolved by aborting one or more transactions involved in the cycles of the TWFG. Finding cycles in a distributed Transaction Wait-For Graph where no single site knows the entire graph, has been considered as one of the challenges in this area.

4 Deadlock Detection Algorithm A

In this section we describe our first approach for detecting and resolving deadlocks in a distributed database system. The approach is to collect information on connectivity of nodes of the overall Transaction Wait-For Graph (TWFG) of the distributed database system, and then to use the information of these connectivities to build a local TWFG at each node of the overall TWGF. Deadlocks are detected by locating the cycles in each local TWFG. To resolve the deadlocks the nodes involved in those cycles in each local TWFG are removed until there is no cycle in the local TWFGs.

4.1 Description of the Algorithm A

In this algorithm, we assume that each node (process) in the overall Transaction Wait-For Graph (TWFG) has a unique label denoted by a_1, a_2, ..., a_m where m is the number of nodes in the overall TWFG. We define a partial path to be a sequence of labels of some sites. We further assume that each node in the overall TWFG has a local TWFG which is initially set to empty and as nodes receive the partial paths through their incoming edges they gradually build their local TWFGs.

Our deadlock detection and resolution algorithm gets initiated when a blocked process in TWFG, initiates a deadlock detection message after waiting for its resource request for some pre-specified period of time. This deadlock detection message will carry a First-In-First-Out (FIFO) Queue of node labels. Initially, the initiator node appends its own label to the FIFO Queue of this deadlock detection message. It then sends this deadlock detection message along with its FIFO Queue to all of its immediate neighbors which are connected to this node through its outgoing edges.

In this algorithm, whenever a node in the overall TWFG receives such a deadlock detection message along with its queue, it first checks to see if it has any outgoing edges. If the node does not have any outgoing edges (i.e. it is a sink node which is an active node) then the deadlock detection message will remain in that node. Otherwise the following steps will take place at that node.

Step 1 It appends its own label to the end of the partial path given in the FIFO Queue of the deadlock detection message.

Step 2 It adds the partial path given in the FIFO Queue to its partially developed local TWFG.

Step 3 It checks the partial path to see if its label has occurred in the partial path string twice (i.e. to see if this partial path has been at this node once before). If this partial path has been at this node once before then it does nothing, otherwise it forwards the received deadlock detection message along with its FIFO Queue to all of its immediate neighbors, which are connected to this node through its outgoing edges.

This process will continue until all the deadlock detection messages propagate through the edges of the overall TWFG and stop at some nodes of the overall TWFG which find their labels occurring in the partial path in the FIFO Queue of the message twice.

When all deadlock detection messages have arrived at their destinations, and have been settled at some nodes of the overall TWFG (i.e. no deadlock detection messages are in transit), then each node of the overall TWFG uses its own local TWFG which has been constructed during deadlock detection message passings to detect possible deadlocks in the distributed database system.

We sequentially check each local TWFG for possible cycles starting for example from site S_1. This process can be

done by using one of the graph traversal algorithms such as Breadth-First-Search (BFS) or Depth-First-Search (DFS).

The process starts from site S_i for $i = 1$ as follows:

1. If there is no cycle in site S_i's local TWFG, and i is not equal to n then go to site S_{i+1} and go to step 1. Otherwise, terminate the deadlock detection algorithm. If there are some cycles in site S_i's local TWFG then go to step 2.

2. Choose a node, (as a victim) for abortion which has created maximum number of cycles in that local TWFG. After the chosen node is aborted, site S_i updates its local TWFG and then it informs the sites which are using this node in their local TWFGs to eliminate this node and its incident edges from their local TWFGs. Then go to step 2. (the victim selection and abortion will be continued on the local TWFG at site S_i until there no more cycle can be found in that local TWFG).

3. If $i = n$ then terminate the deadlock detection algorithm. Otherwise, repeat steps 1 and 2 respectively for next site in sequence (i.e. site S_{i+1}).

4.2 Correctness of the Algorithm A

In this section, we show that the deadlock detection and resolution algorithm presented in the previous section is correct. To show that the algorithm A is correct, we have to prove that: (1) It terminates after a finite amount of time; (2) It does not report false (Phantom) deadlocks; (3) It does detect all deadlocks in the overall TWFG that are reachable from the node that initiates deadlock detection algorithm. The following theorems, show that the algorithm A terminates in finite amount of time, detects all deadlocks, and it does not report Phantom deadlocks.

Theorem 1: Algorithm A terminates after a finite amount of time.

Proof. In order to show that algorithm A terminates in a finite amount of time, we need to show that all the deadlock detection messages which have been sent through the overall Transaction Wait-For Graph's nodes will eventually rest at some nodes in the overall TWFG. The following two cases show that this is indeed the case.

Case 1: A deadlock detection message eventually reaches a node in the overall TWFG which does not have any outgoing edges (i.e. it is an active node). In this case, the deadlock detection message will rest at that node and no further propagation of that message takes place.

Case 2: A deadlock detection message eventually reaches a node in the overall TWFG with a partial path that contains the label of this node twice. At this point according to the algorithm the deadlock detection message will rest at that node without further propagation through the overall TWFG. □

Theorem 2: Algorithm A does not report false (Phantom) deadlocks.

Proof. In order to show that the algorithm A does not report false deadlocks, we need to show that the algorithm does not report a false cycle. This is trivial, since according to the algorithm A there must be an actual cycle in the overall TWFG in order for algorithm to report it. Hence, no false cycle can be reported and therefore no false (Phantom) deadlocks can be reported by algorithm A. □

Theorem 3: Algorithm A detects all the deadlocks in the overall TWFG that are reachable from the node that initiates deadlock detection algorithm.

Proof. In order to show that algorithm A detects all the deadlocks in the overall TWFG that are reachable from the node that initiates deadlock detection algorithm, we need to show that algorithm A finds all the possible cycles in the overall TWFG that are reachable from the node that initiates the deadlock detection algorithm. Assume the contrary, and that there is at least one cycle in the overall TWFG which has not been picked up by the algorithm. We show that this is not possible. Lets assume that the cycle consists of nodes a_l, a_m, ..., a_x, a_l. Since this cycle is included in the overall TWFG that is reachable from the node that initiates a deadlock detection message, one of the deadlock detection messages must arrive at one of the nodes in this cycle. Otherwise this cycle does not belong to this overall TWFG. It belongs to some other independent TWFG. So if one of the deadlock detection messages must arrive at one of the node in that cycle, then assume node a_p is that node. Since node a_p belongs to the cycle a_l, a_m, ..., a_l, then according to the algorithm one the deadlock detection messages must be leaving the node a_p through one of its outgoing edges to another node in that cycle and so on. Therefore, eventually a deadlock detection message which has a_p as part of its partial path will arrive at a_p (again since a_p is on that cycle). Hence, that cycle will be detected by the algorithm A. □

5 Deadlock Detection Algorithm B

In this section we describe our second approach for detecting and resolving deadlocks in a distributed database system. This approach continuously checks for the occurrences a deadlock between different transaction trees and as soon as it detects a deadlock it resolves it by aborting one of the transaction trees which has been initiated more recently.

5.1 Description of the Algorithm B

In this algorithm, we assume that each original transaction has a unique global identifier and all the nodes in the transaction tree that are generated by a given transaction will have the same identifiers as their originator's identifier. Also, we assume that each node in the transaction tree knows who is its predecessor node. In addition, we assume that when two transaction trees will join for the first time (via a request made from one node in one transaction tree to a node in the second transaction tree), the transaction tree that has been initiated earlier will get the ID of the other transaction tree and store it in its root. This information is needed for detecting potential deadlocks when such transaction trees try to connect again.

Initially, we assume each transaction starts its transaction tree independent of other transactions. However, as these transaction trees expand and branch out over the distributed database system, they may colide with one another, over some resources which may cause distributed deadlocks.

Algorithm B continuously checks for the occurrences a deadlocks between different transaction trees using the following steps:

Step 1 Whenever a node from one transaction tree requests for a resource hold by a node from another transaction tree for the first time, let this request be made successfully by joining the two transaction trees via the directed edge initiating from the node which has made the request to the node which is holding the requested resource. Then store the ID of the transaction tree which has more recent initiation time among the two transaction trees, in the root of the other transaction tree.

Step 1 If these two transaction trees try to connect again via some other request, then before the connection can take place, check to see if this new connection will cause a deadlock. If it does, one of the transaction trees has to be aborted. In this case, the transaction tree which has been initiated more recently is aborted.

For implementing step 1 and 2, we use an efficient implementation of the classical *Union* and *Find* operations. In step 1 of algorithm B, in order to find out if two transaction trees are trying to connect for the first time or not, we need to apply the Find operation to see if these two transaction trees have different root IDs. If they have, we know that the two transaction trees are trying to connect for the first time, and thatthe algorithm allows the connection to take place. However, in step 2 the algorithm uses the Find operation to discover that the two transaction trees have already been connected at some nodes. So in that step we need to check for a potential deadlock.

In step two the algorithm checks for a potential deadlock by sending a reverse Probe message along the reverse path of the transaction tree which has requested the resource. The reverse Probe message initially starts from the node requesting for resource from a node in the other transaction tree. The reverse Probe message carries the ID of the transaction tree which has requested the resource. In order to avoid overloading the system with unwanted reverse Probe messages propagating through the overall TWFG, we restrict the reverse Probe to only travel through the edges with oposite directions. Therefore, the reverse Probe can not travel through forward (outgoing) edges. This restriction allows the reverse Probe to check for the cycle between the two transaction trees in a very efficient way without creating too many unneccessary Probe messages.

If the reverse Probe message eventually comes back through the opposite direction to the node which has initiated the reverse Probe, then there is a cycle between the two transaction trees and therefore there is a deadlock and we need to abort the most recent transaction. However, if after some pre-specified time interval the requested node does not hear from the reverse Probe message this should indicate that eventhough the two transaction trees are connected at some point already but the connection is a safe connection (i.e. is not creating a deadlock). Hence the requesting node can safely wait for its request and neither of the transaction trees are required to be aborted.

Our second approach can be very useful in distributed real-time datatbase systems, where there are ime constraints on the transactions (i.e. transactions must finish before their deadlines), and we can not afford to wait around too long for the detection and resolution of deadlocks.

5.2 Correctness of the Algorithm B

In this section, we show that the deadlock detection and resolution algorithm presented in the previous section is correct. To show that the algorithm B is correct, we need to prove the following theorem.

Theorem 4: The algorithm B (1) It terminates after a finite amount of time; (2) It does not report false (Phantom) deadlocks; (3) It does detect all deadlocks.

Proof. The detail proof of this theorem is provided in [19]. Here we try to justify only the first item in the theorem which is trivial because we have a pre-specified time interval for the reverse Probe to determine if there is a deadlock or not. Therefore, the algorithm B always terminates after a finite amount of time. □

6 Concluding Remarks

In this paper, we have discussed deadlock detection and resolution strategies, and proposed two approaches for detecting and resolving deadlocks in both general distributed database systems and in distributed real-time database systems. Our first approach is to collect information on connectivity of nodes of the overall Transaction Wait-For Graph (TWFG) of the distributed database system and then use these connectivities information to build a local TWFG at each node of the overall TWGF. We then detects the deadlocks by locating the cycles in each local TWFG. To resolve the deadlocks the nodes involved in those cycles in each local TWFG are removed until there is no cycle in the local TWFGs. Our second approach continuously checks for the occurrences a deadlock between different transaction trees and as soon as it detects a deadlock it resolves it by aborting one of the transaction tree, that has been initiated more recently. Some of the advantages of our approaches over the approaches which are using Probe messages are: (1) no extra storage required to store different probe messages, (2) no false (Phantom) deadlocks are reported, (3) detects and resolve all deadlocks. In addition, our approaches use less messages and time to detect and resolve all deadlocks in the existing TWFG of the distributed database system.

Our second approach can be very useful in the distributed real-time datatbase systems where there is a time constraints on the transactions (i.e. transactions must finish before their deadlines) and we can not afford to wait around too long for the detection and resolution of deadlocks.

References

[1] B. Awerbuch and S. Micali, "Dynamic Deadlock Detection Protocols," In *Proceedings of the Foundations of Computer Science*, IEEE, New York, October 1986.

[2] D. J. Badal "The Distributed Deadlock Detection Algorithm," *ACM Transaction on Computer Systems*, November 1986.

[3] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, pp. 185-221, June 1981.

[4] G. Bracha and S. Toueg, "A Distributed Algorithm for Generalized Deadlock Detection," In *Proceedings of the ACM Symposium on Principles of Distributed*

Computing, Vancouver, Canada, pp. 285-301, August 1984.

[5] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transaction on Programming Languages and Systems*, Vol. 3, No. 1, pp. 63-75, February 1985.

[6] K. M. Chandy and J. Misra, "A Distributed Algorithm for detecting Deadlocks in Distributed Systems," In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada, pp. 157-164, August 1982.

[7] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed Deadlock Detection," *ACM Transaction on Computer Systems*, Vol. 1, pp. 144-156, May 1983.

[8] A. L. Choudhary, W. H. Kohler, J. A. Stankovic, and D. Towsley, "A Modified Priority Based Probe Algorithm for Distributed Deadlock Detection and Resolution," *IEEE Transaction on Software Engineering*, Vol. 15, No. 1, pp. 10-17, January 1989.

[9] A. K. Elmagarmid, N. Soundarajan and M. T. Liu, "A Distributed Deadlock Detection and Resolution Algorithm and its Correctness," *IEEE Transaction on Software Engineering*, October 1988.

[10] A. K. Elmagarmid, "Deadlock Detection and Resolution in Distributed Processing Systems," *Ph.D. Dissertation, Dept. of Computer and Information Science, Ohio State University*, Columbus, Ohio 1985.

[11] A. K. Elmagarmid, "A Survey of Distributed Deadlock Detection Algorithms," *ACM SIGMOD RECORD* Vol. 15, No. 3, September 1986.

[12] V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Databases," *IEEE Transaction on Software Engineering*, Vol. 6, No. 5, September 1980. *ACM Computing Surveys*, Vol. 13, pp. 185-221, June 1981.

[13] J. N. Gray, "Notes on Database Operating Systems," *in Operating Systems: An Advanced Course*, Springer-Verlag, New York pp. 393-481, 1978.

[14] L. M. Haas and C. Mohan, "A Distributed Deadlock Detection Algorithm for Resource Based Systems," *IBM Research Laboratory*, Res. Rep. RJ 3765, San Jose California, 1981.

[15] G. S. Ho and C. V. Ramamoorthy, "Protocols for Deadlock Detection in Distributed Database Systems," *IEEE Transaction on Software Engineering*, Vol. 8, No. 6, November 1982.

[16] E. Knapp, "Deadlock Detection in Distributed Database Systems," *ACM Computing Surveys*, Vol. 19, No. 4, December 1987.

[17] A. D. Kshemkalyani and M. Singhal, "An Invariant-Based Verification of a Priority-Based Probe Algorithm for Distributed Deadlock Detection and Resolution," *IEEE Transaction on Software Engineering*, Vol. 17, No. 8, August 1991.

[18] A. D. Kshemkalyani and M. Singhal, "Efficient Detection and Resolution of Generalized Distributed Deadlocks," *Ohio State University Technical Report*, Dept. of CIS, Columbus, Ohio, July 1990.

[19] K. Makki and N. Pissinou, "On Deadlock Detection and Resolution in Distributed Database Systems," *Technical Report*, The Center For Advanced Computer Studies, The University of Southwestern Louisiana, Lafayette, LA, 1994.

[20] D. E. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *IEEE Transaction on Software Engineering*, Vol. 5, No. 3, May 1979.

[21] D. P. Mitchell and M. j. Merritt, "A Distributed Algorithm for Deadlock Detection and Resolution," In *Proceedings of the ACM Conference on Principles of Distributed Computing*, New York, pp. 282-284, August 1984.

[22] J.E.B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing," *Technical Report 260* Laboratory of Computer Science, Massachusetts Institute of Technology Cambridge, MA, April 1981 .

[23] R. Obermarck, "Distributed Deadlock Detection Algorithm," *ACM Transaction on Database Systems*, Vol. 7, No. 2, pp. 187-208, June 1982.

[24] P. K. Reddy and S. Bhalla, "Deadlock Prevention in a Distributed System," *ACM SIGMOD RECORD*, Vol. 22, No. 3, September 1993.

[25] M. Singhal, "Deadlock Detection in Distributed Systems," *IEEE Computer*, November 1989.

[26] M. Singhal and N.G. Shivaratri, "Advance Concepts in Operating Systems," *McGraw-Hills*, 1994.

[27] M. K Sinha and N. Natarajan, "A Priority Based Distributed Deadlock Detection Algorithm," *IEEE Transaction on Software Engineering*, Vol. 11, No. 1, January 1985.

[28] K. Sugihara, T. Kikuno, N. Yoshida and M. Ogata, "A Distributed Algorithm for Deadlock Detection and Resolution," In *Proceedings of the 4th Symposium on Reliability in Distributed Software ans Database Systems*, October 1984.

[29] C. F. Yeung, S. L. Hung and K. Y. Lam, "Performance Evaluation of a New Distributed Deadlock Detection Algorithm ," In *ACM SIGMOD RECORD*, Vol. 23, No. 3 pp. 21-26, September 1994.

[30] C. F. Yeung, S. L. Hung, K. Y. Lam and C.. K. Law, "A New Distributed Deadlock Detection Algorithm for Distributed Database Systems," In *Proceedings of 1994 IEEE TENCON*, pp. 506-510, 1994.