Efficient Algorithms for Detection and Resolution of Distributed Deadlocks (*Extended Abstract*) *

Shigang Chen, Yi Deng, Wei Sun and Naphtali Rishe
School of Computer Science
Florida International University
University Park, Miami, FL 33199
{csg,deng,weisun,rishen}@fiu.edu, Phone: (305)348-2744, Fax: (305) 348-3549

Abstract

We present a simple and efficient distributed algorithm for detecting generalized-deadlocks in distributed systems. Unlike previous algorithms, which are all based on the idea of distributed snapshot, and require multiple rounds of message transfers along the edges of the global wait-for graph (WFG), the proposed algorithm uses a novel approach that incrementally constructs an "image" of the WFG at an initiator node. The algorithm has time complexity of $d+1$ and message complexity of $e+n$, where n is the number of nodes, d the diameter, and e the number of edges of the WFG. Compared with the best existing algorithm, our algorithm notably reduces both time and message complexities. Correctness proof and performance analysis for the algorithm are provided. In addition, the new approach simplifies deadlock resolution. An extension to the algorithm is presented to handle generalized-deadlock resolution with only a slight increase to the message complexity.

1 Introduction

Deadlock detection and resolution have long been considered as a fundamental problem in distributed systems. A deadlock is a system state in which every process in some subset of processes in the system waits indefinitely for at least one other process in this subset to respond to a request for some resource. In the past decade, a number of algorithms have been proposed to provide a solution to the problem. These algorithms can be grouped into several classes based on their underlying computation (resource request) models, such as the AND [10, 12, 13], OR [2, 11], AND-OR [5], and p-out-of-q models [1, 8, 14].

A system is said to be based on the p-out-of-q model if a process in the system issues a request for q resources and remains blocked until at least p out of the q requested resources are granted. The p-out-of-q model is also called the *generalized resource request model* because both the AND and OR models can be considered as its special cases. It also includes the AND-OR model [6]. Consequently, a deadlock in the p-of-out-q model is called a *generalized-deadlock*. An example of p-out-of-q requests arises with replicated files when quorum-based replica control algorithms are used [4]. To preserve data consistency, a process that wants to read (write) a replicated data item, must read (write) r (w) copies out of the n copies of the data item such that $r+w>n$ and $2w>n$. To read or write a copy, a process must request and obtain a lock on the copy. Thus, reading or writing a replicated file generates "p-out-of-q" requests for locks.

Distributed generalized-deadlock detection is much more difficult than deadlock detection in the simpler AND (or OR) model. This is because it requires the detection of a complex topology (*generalized tie*, see Definition 2), instead of a simple cycle (or a knot), in the global *wait-for graph* (WFG). Among the existing distributed deadlock detection algorithms, only [1, 8, 14] address generalized-deadlocks. To the best of our knowledge, the only algorithm addressing the resolution of distributed generalized-deadlocks is in [7], which is an extension of [8]. All three algorithms in [1, 8, 14] follow the same approach based on the idea of distributed snapshots. They have either two distinct phases [1, 14] or one phase consisting of two overlapped sweeps [8] of message transmission. In the first phase (or the outward sweep), they record a WFG, which

*This work was supported in part by the Rome Laboratory, U.S. Air Force under contract No. F30602-93-C-0247, by NASA under grant No. NAGW-4080 and by NSF under grant No. CDA-9313624.

is distributed among all the involved processes in the system, by taking a snapshot of the system state. In the second phase (or the inward sweep), they reduce the recorded WFG by simulating the unblocking of those processes whose requests can be granted.

In this paper, we first present a new distributed algorithm for generalized-deadlock detection. The algorithm performs better than [1, 8, 14] in terms of both time and message complexities (see Table 1). An extension to the algorithm is also presented to handle generalized-deadlock resolution, whose performance is also better than [7]. The improvements are the result of a novel approach behind the proposed algorithms, that differs from all the above algorithms. Instead of recording a distributed WFG of the system and reducing it in a distributed way as described above, which requires two or more rounds of message transfer along all wait-for edges of the WFG, the proposed algorithm incrementally constructs a WFG, which is an "image" of the system state, at a single process (the initiator of the algorithm). By doing so, our algorithm requires only one diffusion of messages outward from the initiator process along the edges of the WFG to all the involved processes. Consequently, our algorithm notably reduces both time and message complexities, compared with [8]. In particular, we conjecture that our time complexity $(d+1)$ is the optimal worst-case complexity that any distributed generalized-deadlock detection algorithm can achieve. In addition, our approach also simplifies the deadlock resolution because the information about the deadlock is readily available to the initiator.

In the rest of the paper, the underlying computation model is defined in Section 2. In Section 3, an informal description of the deadlock detection algorithm is provided. The correctness and performance of the algorithm are proved and analyzed in Sections 4 and 5, respectively. In Section 6, we discuss deadlock resolution. Finally, we conclude the paper in Section 7.

2 Model of computation

A distributed system is composed of n nodes, each of which represents a process and has a system-wide unique identity. Each pair of nodes is connected by a logical channel. There is no shared memory in the system. Nodes communicate by message passing, and message delays on a channel are arbitrary but finite. A destination node receives messages in the same order as they are sent by a source node. Messages are neither lost nor duplicated, and are transmitted error-free.

The following data structure is used at a node $i, i = 1...n$, to keep track of its current state. We assume that the logical time at each process is maintained as specified in [9].

t_i:	the current logical time at i,
t_block_i:	the logical time at which i last blocked,
out_i:	the set of nodes for which i is waiting,
in_i:	the set of tuples $< k, t\_block_k >$, where k is a node waiting for i and t_block_k is the logical time at which k sent a request to i,
p_i:	the number of replies required for i to unblock.

There are two types of events associated with each node, namely, *computation events* and *control events*. A computation event is caused by the underlying computation of a process; and a control event is caused by the execution of the deadlock detection algorithm. The messages generated by these two types of events are called *computation messages* and *control messages*, respectively. The computation messages include REQUEST, REPLY, CANCEL, and ACK messages.

Each node is either *active* or *blocked*. An active node can send both computation and control messages. A blocked node, however, can only send control messages or ACK messages, i.e., its underlying computation is suspended. A node i becomes blocked after it sends a p_i-out-of-q_i request (via REQUEST messages) to q_i other nodes. It records these nodes in out_i. When a node j receives a REQUEST message from node i, it records $< i, t\_block_i >$ in in_j and immediately sends an ACK back to node i to acknowledge the receipt of the request [1]. A REPLY message denotes the granting of a request. When j sends a REPLY to i, $< i, t\_block_i >$ is removed from in_j. The node i becomes unblocked (goes from blocked to active state) *only* when any p_i out of the q_i requests are granted, namely, i receives REPLY messages from at least p_i out of the q_i nodes. When i unblocks, it sends CANCEL messages to withdraw the remaining $q_i - p_i$ requests it had sent.

Each REQUEST, REPLY, or ACK message is timestamped with the requester's logical clock value [9] at which it blocked, so that an ACK or a REPLY can be matched with its corresponding request. ACK or REPLY messages with unmatched timestamps are discarded.

[1] Notice that this ACK can be the same acknowledgement used by the underlying network to guarantee reliable communication channels. There is no need to send a separate message to deliver such information.

Comparing factor	Bracha-Toueg [1]	Wang et al. [14]	Kshem.-Singhal [8]	Our Algorithm
Phase	2	2	1 phase, 2 sweep	1 phase, 1 sweep
Delay	$4d$	$3d+1$	$2d$	$d+1$
No. Messages	$4e$	$6e$	$4e-2n+2l$	$e+n$
Total Size of Data Sent	$4e$	$6e$	$4e-2n+2l$	$2e+n$
Support Resolution	no	no	yes	yes

Table 1: Performance comparison between our algorithm and the existing algorithms. Given a WFG, n = number of the nodes, l = number of the leaf nodes, e = number of the edges, d = diameter of the WFG.

Definition 1 A wait-for graph (N, E) is a directed graph, where a node in N models a process, and an edge in E models a dependence relation between two processes in N. A directed edge from node P_1 to node P_2 indicates that P_1 is blocked and is waiting for P_2 to release and grant some resource.

The *dependent set* of a node i is the set of nodes in out_i. A node j is said to be *reachable* from i iff there is a directed path in the WFG from i to j.

Definition 2 A *generalized tie* (*tie* in short) is a graph (N_t, E_t), where N_t is a nonempty set of nodes that are blocked on p-out-of-q requests, and E_t is the set of wait-for edges between the nodes in N_t, such that each $i \in N_t$ has at least $q_i - p_i + 1$ outgoing edges in E_t.

Definition 3 A *generalized-deadlock* exists in a system if and only if there exists a generalized tie in the WFG of the system.

3 A Simple and Efficient Deadlock Detection Algorithm

The algorithm can be initiated by any node (which is called an *initiator*) when it blocks on a p-out-of-q request. When multiple nodes in the system block simultaneously, they will each initiate the deadlock detection algorithm. Each invocation to the algorithm is called an instance of the algorithm. Every instance of the algorithm is treated independently and identified by the initiator's identity and the logical time at which the initiator blocked. In the ensuing discussion, we will focus on a single instance of the deadlock detection initiated by a node i.

When a node i blocks on a p_i-out-of-q_i request, it initiates an instance of deadlock detection. A *precondition* for i to invoke the algorithm is that i has received an ACK message from every node j in its dependent set. The precondition is to ensure that $< i, t\_block_i >$ has already been recorded in in_j before the algorithm is invoked, which is necessary to guarantee that the proposed algorithm will detect every deadlock in the system.

There are two types of control messages: FORWARD and BACKWARD message. A major difference between the proposed algorithm and [1, 8, 14] is that, in the process of deadlock detection, a WFG (denoted as WFG_i), which is an "image" of the system state, is incrementally constructed at the initiator i. WFG_i consists of the nodes which are reachable from i and the wait-for edges between these nodes. The detection of generalized-deadlocks (searching for ties) is performed on WFG_i.

The initialization of the instance of deadlock detection is composed of two steps:

1. Create WFG_i at the initiator containing only one vertex i; and
2. the initiator sends a FORWARD message along every outgoing wait-for edges (i.e. to every node in out_i). (The purpose of a FORWARD message is to inform a node of its involvement in the deadlock detection.)

Definition 4 Suppose j sends a FORWARD message to k along a wait-for edge (j, k), which represents a resource request R_j. The edge (j, k) is an *E-edge* if k has not sent a REPLY message for R_j to j before receiving the FORWARD. □

When a node k receives a FORWARD message from a node j along the wait-for edge (j, k), it takes one of the following two actions:

1. If (j, k) is an E-edge and the message is the first FORWARD message received by k along an E-edge, then (1) k sends a BACKWARD message (t_block_k, in_k, out_k, p_k) to the initiator i and (2) propagates the FORWARD message along its outgoing wait-for edges (to the nodes in out_k) if it is blocked. (The purpose of the BACKWARD message is to report the node's current state information to i.)

2. If (j, k) is not an E-edge [2] or the message is not the first FORWARD message received by k along an E-edge, the received FORWARD message is discarded.

As the initiator i receives BCKWARD messages from other nodes in the system, WFG_i is incrementally constructed. We now define the data structure used to store WFG_i. This data structure is solely for deadlock detection, and plays a different role from the data structure defined in Section 2, which is used by each node in the system to keep its state information. To distinguish a node in WFG_i from the node in the underlying system which it represents, we will call a node in WFG_i a *vertex*. A vertex k in WFG_i is represented by a tuple $(k.t, k.in, k.out, k.p)$. The rule for constructing WFG_i is as follows:

When the initiator i receives a BACKWARD message (t_block_k, in_k, out_k, p_k) from a node k, it inserts a new vertex k into WFG_i, where $k.t := t_k$, $k.in := in_k$, $k.out := out_k$, and $k.p := p_k$.

Definition 5 An *edge* (j, k) belongs to WFG_i if (1) both j and k are vertices in WFG_i, (2) $k \in j.out$ and (3) $< j, j.t > \in k.in$.

There is an edge (j, k) in WFG_i if and only if the waiting-for relation recorded at vertex j and the waited-by relation recorded at vertex k refers to the same request.

Whenever a new vertex is inserted into WFG_i, the algorithm tries to find whether there is a tie in WFG_i. If there is a tie, the algorithm reports a deadlock.

The algorithm terminates in two cases: (1) a deadlock is detected or (2) the initiator unblocks. In the later case, WFG_i is deleted and the memory space is released.

4 Sketch of Correctness Proof

We first introduce the notations and conventions used in the proof. For the purpose of correctness proof only, we introduce T as the global physical (or real) time of the system, in contrast to the logical time t used in the algorithm. It should be noted that our algorithm does not depend on a global (physical time) clock.

- T_j: the physical time at which node j sends a BACKWARD message to the initiator i.
- R_j: the last *p-out-of-q* request issued by node j before T_j.

Without losing generality, assume that i is the initiator. Due to the limited space, only a sketch of the proof is presented here. The detailed formal proof is provided in [3].

Lemma 1 If (j, k) is an edge in WFG_i, then no REPLY message for R_j is sent from node k to node j before T_k.

Proof: We establish the contrapositive of the lemma. Let $t\_block_j(R_j)$ denote the logical time at which j blocked on R_j. Since R_j is the last request issued by j before T_j, the variable $j.t$ will be set to $t\_block_j(R_j)$ upon receipt by i of the BACKWARD message sent by j (by construction of the algorithm). Now suppose k sends a reply for R_j before time T_k. Upon sending this reply, k removes $< j, t\_block_j(R_j) >$ from in_k. There are now two cases.

Case 1: k does not receive another request from j before T_k.
Then, the in_k field of the BACKWARD message that k sends to j will not contain a tuple of the form $< j, t >$ (for some value t). Upon receipt by i, $k.in$ will be assigned this value of in_k. Since $k.in$ and $j.t$ are assigned to exactly once during the construction of WFG_i, it follows, by definition 5, that WFG_i will never contain an edge (j, k).

Case 2: k receives another request from j before T_k. Call this request R'_j. Then, the in_k field of the BACKWARD message that k sends to j will contain the tuple $< j, t\_block_j(R'_j) >$, where $t\_block_j(R'_j)$ denotes the logical time at which j blocked on R'_j, and no other tuples with j as the first element. Upon receipt by i, $k.in$ will be assigned this value of in_k. Since the receipt by j of $k's$ reply to R_j occurs between the first time j blocked (on R_j) and the second time j blocked (on R'_j), we have $t\_block_j(R_j) < t\_block_j(R'_j)$, by Lamport's clock condition [9]. Since $j.t$ is set to $t\_block_j(R_j)$ (see above), and $t\_block_j(R_j) \neq t\_block_j(R'_j)$, we have, again by definition 5, that WFG_i will never contain an edge (j, k).

In both cases we have established the contrapositive, and so the lemma is established. □

Theorem 1 No false deadlock is detected by the proposed algorithm.

Proof Sketch: The proposed algorithm reports a deadlock only when there is a tie in WFG_i. We prove by

[2] *(j, k) is not an E-edge* means that it has already ceased to exist before k receives the FORWARD from j.

contradiction that, if a tie G is found in WFG_i, then every node involved in G never unblocks. Without losing generality, assume that j is the first node in G which unblocks on the recorded p_j-out-of-q_j request (R_j). Note that the assumption refers to a particular request, namely, the last request issued by the node before it sends its state information to the initiator. Let N_j be the set of vertices in G that j has edges outstanding at. Every vertex in N_j represents a node in the system for which j is waiting at time T_j. There are no less than $(q_j - p_j + 1)$ vertices in N_j because G is a tie. Hence, before node j unblocks on R_j, at least one node in N_j must send a REPLY message to it. Suppose k is such a node. By Lemma 1, node k sends the REPLY to node j after T_k. But node k blocks on R_k at T_k because $k \in G$. Before node k sends the REPLY, k has to unblock on R_k first. That is in contradiction with the assumption that j is the first node in G to unblock (on the recorded request, R_j). □

Theorem 2 Every deadlock in the system will be detected by the proposed algorithm within finite time.

Proof Sketch: Let G be a tie in the underlying system. For all $j \in G$, j blocks on a p_j-out-of-q_j request. Let i be the last node in G to initiate an instance of deadlock detection. We prove by contradiction that i detects a deadlock. Suppose i does not detect a tie in WFG_i. By the analysis in section 5, each instance of the algorithm transmits finite control messages. Hence, after finite time, no control messages are in transmission, which implies that WFG_i does not change any more. Such a stable WFG_i is denoted as WFG_i^f.

Consider an arbitrary vertex j of G. Since j is a member of a tie G, j will be blocked forever (after the physical time at which G was formed, and in the absence of deadlock resolution). This is easily seen by considering the first member j' of G to unblock, and noting that (by definition 2), there is at least one member j'' of G which must reply to j''s outstanding request before j' can unblock. Since j'' is itself blocked, j'' must unblock before replying, which contradicts the fact that j' is the first member of G to unblock.

Now if i is the last node in G to initiate deadlock detection, then G must already be formed when i initiates this instance, by virtue of our precondition (given on page 4) for invoking the deadlock detection algorithm. Since every node in G is blocked forever from this moment (in physical time) onwards, the state information about G, which is stored in variables (i.e., t_i, t_block_i, out_i, in_i) of all nodes in G, will remain unchanged. From this fact and the construction of our algorithm, we see that:

> If an arbitrary node j of G is a vertex of WFG_i^f, then so is k, for every $k \in G$ such that j waits for k. Furthermore, (j, k) is an edge in WFG_i^f. (*)

See [3] for the detailed proof of this assertion.

Suppose $G^f =< N^f, E^f >$ is a sub-graph of WFG_i^f, where N^f is the set of vertices in both WFG_i^f and G, and E^f is the set of edges between vertices in N^f. $N^f \neq \emptyset$ because $i \in N^f$. For all $j \in N^f$, since j is a node in the tie G, there are at least $(q_j - p_j + 1)$ nodes in G for which j is waiting. By (*) above, each of these nodes is a vertex in G^f, and the wait-for edges from node j to these nodes are also in G^f. That is, j has at least $(q_j - p_j + 1)$ edges outstanding at other vertices in G^f. Hence, G^f is a tie, which contradicts the assumption that there is no tie in WFG_i. Hence the theorem holds. □

5 Complexity Analysis

In this section, we consider the time, message, as well as space complexities. Let WFG_s be a system of n nodes and e edges, with a diameter of d. Let i be the initiator. Note that only the nodes reachable from i are involved in the deadlock detection. In computing the message complexity, we only consider logical message transfers. Based on the type of underlying communication network, a logical message may result in the transfer of a number of physical messages, which is not an issue here. As in [1, 8, 14], assume that the message delay on a logical channel (one hop) is 1 unit of time.

The Bracha-Toueg algorithm [1] has message complexity of $4e$ messages and time complexity of $4d$ hops; the Wang-Huang-Chen algorithm [14] has message complexity of $6e$ and time complexity of $3d + 1$; the Kshemkalyani-Singhal algorithm [8] has message complexity of $4e - 2n + 2l$ and time complexity of $2d$. As shown below, the performance of our algorithm is better than the existing results.

Recall that only when a node j receives a FORWARD message along an E-edge for the first time, it propagates the FORWARD to the nodes in its dependent set. Therefore, a spanning tree composed of nodes reachable from i, with the initiator as the root, can be defined. A node j is the parent of a node k if and only if j sends the first FORWARD message received by k and (j, k) is an E-edge. The height of the spanning tree is at most d. Hence, it takes at most d hops to propagate FORWARDs to all the nodes reachable from i. When a node receives a FORWARD along

an E-edge for the first time, it sends a BACKWARD to the initiator, which takes 1 hop. Therefore, all control messages are transmitted within at most $d+1$ hops. We conjecture that $d+1$ is the optimal time complexity that can be achieved by any distributed generalized-deadlock detection algorithm.

By the construction of the algorithm, each node in WFG_s has at most one chance to send FORWARDs along its outgoing edges, which implies that there is at most one FORWARD message sent along each wait-for edge in WFG_s. Hence, the number of FORWARD messages can not be greater than e. Each node in WFG_s sends at most one BACKWARD message to the initiator. The number of the BACKWARD messages can not be greater than n. Therefore, the number of control messages transmitted is $e+n$ in the worst case. For simplicity, consider the length of a control message in [1, 8, 14], which consists of several integers, as 1 unit. The total size of data transmitted in the algorithms of [1, 8, 14] are $4e$, $6e$ and $4e-2n+2l$ units, respectively. The total size of data transferred in our algorithm is $2e+n$ units. The detailed analysis is provided in [3].

In algorithms [1, 8], the memory space needed by each node to store the snapshots for different instances of deadlock detection is $O(n^2)$ in the worst case. Therefore, the total space required is $O(n^3)$. In our algorithm, no snapshot needs to be stored at any node. The initiator requires at most $O(n^2)$ space to store the constructed WFG. In the worst case, when all the n nodes initiate deadlock detections simultaneously, $O(n^3)$ space in all is needed. Thus our algorithm is comparable to the other algorithms in its space complexity.

6 Deadlock Resolution

The proposed algorithm works under the assumption that the only way for a blocked node to unblock is to get enough REPLY messages. To resolve a deadlock, however, a node is allowed to abort. In such a system model, deadlocks are not stable and can be resolved by aborting a node or a set of nodes. The semantics of a node's aborting is the same as described in [7]. That is, when a node aborts, the process it represents is rolled back and a new one restarts using a higher timestamp. The resources held by the aborted node are released, and thus can be granted to the nodes requiring them.

To the best of our knowledge, the only algorithm addressing the problem of the distributed generalized-deadlock resolution is in [7]. In what follows, we present an extension to the proposed deadlock detection algorithm to handle the distributed generalized-deadlock resolution with only a slight increase to message complexity. We have proved in [3] that every deadlock in the system can be resolved by the extended algorithm.

Simply speaking, whenever a node i detects that it is involved in a tie, it aborts itself to resolve the deadlock. After i aborts, its previous state information that has been collected by other nodes becomes outdated. Node i then sends ABORT messages to these nodes to inform them of its aborting. When a node receives an ABORT message, it eliminates the outdated information.

More specifically, every node i ($i = 1, ..., n$) maintains a data structure named $Node\_Set_i$. $Node\_Set_i$ keeps track of the set of nodes, to which i sent its state information (via BACKWARD messages) after i issued the last (namely, the most recent) request. The rules for maintaining $Node\_Set_i$ are as follows: (1) Whenever i blocks, $Node\_Set_i$ is set empty; and (2) when i sends a BACKWARD message to j, $Node\_Set_i = Node\_Set_i + \{j\}$. Suppose i initiates a deadlock detection and detects a tie in WFG_i. If i is a vertex in the tie, i aborts itself and sends an ABORT message to every node in $Node\_Set_i$. When a node j receives an ABORT message from i, one of the following two actions is taken: (1) If j is the initiator of an instance of the deadlock resolution algorithm and i is a vertex in WFG_j, remove every outgoing edge of i from WFG_j; (2) otherwise, discard the ABORT message.

The termination conditions of the deadlock resolution algorithm are different from those of the deadlock detection algorithm. The resolution algorithm terminates in one of the following two cases: (1) The initiator aborts itself; or (2) it unblocks. In either case, WFG_i is deleted and the memory space is released.

There are at most n ABORT messages in a single instance of the resolution algorithm. It takes one hop to transmit the ABORT messages. By the results in Section 5, the resolution algorithm has message complexity of $e+2n$ and time complexity of $d+2$. Our algorithm is much more efficient than [7], which has message complexity of $(5e-2n+2l)$ and time complexity of $2d$. Like [7], our algorithm may abort nodes that are not deadlocked at the moment they are aborted. However, in our algorithm, it takes only one hop of message transfer for the ABORT messages to reach the nodes in $Node\_Set$, in contrast to the sequence of message transfers for the information from the aborted node to reach the initiator in [7], which takes d hops

in the worst case. Hence, the outdated information caused by the abort events is eliminated more quickly in our algorithm, which implies less nodes are aborted unnecessarily in our algorithm than those in [7].

7 Conclusion

We have presented two distributed algorithms for the detection and resolution of generalized-deadlocks in distributed systems, which significantly improve the existing results. Correctness proof sketch and complexity analysis for the deadlock detection algorithm are also provided. Detailed formal correctness proof and performance analysis for the two algorithms are in [3].

The improvement achieved by the proposed algorithm is due to the novel approach behind the algorithms, which differs from the one used by all the existing algorithms for the detection and resolution of distributed deadlocks. It incrementally constructs a WFG at the initiator instead of recording a distributed snapshot, and thus the reduction of the WFG can be done locally rather than in a distributed fashion. Consequently, our algorithms are not only simpler but also much more efficient. We believe that this approach points out a new way to design distributed deadlock detection and resolution algorithms.

References

[1] G. Bracha and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2:127 – 138, 1987.

[2] K. M. Chandy and J. Misra. Distributed deadlock detection. *ACM Transactions on Computer System*, 1(2):144 – 156, May 1983.

[3] S. Chen and Y. Deng. Efficient algorithms for detection and resolution of distributed deadlocks. *Technical Report, School of Computer Science, Florida International University*, October 1994.

[4] D. G. Gifford. Weighted voting for replicated data. *Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150 – 163, 1979.

[5] T. Herman and K. M. Chandy. A distributed procedure to detect AND/OR deadlocks. *Department of Computer Science, Technical Report, TR-LCS-8301, University of Texas, Austin, TX*, February 1983.

[6] E. Knapp. Deadlock detection in distributed database. *ACM Computing Surveys*, 19(4):303 – 328, December 1987.

[7] A. D. Kshemkalyani. Characterization and correctness of distributed deadlock detection and resolution. *Ph.D. dissertation, Ohio State University*, August 1991.

[8] A. D. Kshemkalyani and M. Singhal. Efficient detection and resolution of generalized distributed deadlocks. *IEEE Transactions on Software Engineering*, 20(1):43 – 54, January 1994.

[9] L. Lamport. Time, clocks, and the order of events in a distributed system. *Communication of the ACM*, 21:558 – 565, July 1978.

[10] D. A. Menasce and R. R. Muntz. Locking and deadlock detection in distributed data base. *IEEE Transactions on Software Engineering*, SE-5(3):195 – 202, May 1979.

[11] N. Natarajan. A distributed scheme for detecting communication deadlocks. *IEEE Transactions on Software Engineering*, SE-12(4):531 – 537, April 1986.

[12] R. Obermarck. Distributed deadlock detection. *ACM Transactions on Database Systems*, 7(2):187 –208, June 1982.

[13] M. Roesler and W. A. Burkhard. Resolution of deadlocks in object-oriented distributed systems. *IEEE Transactions on Computers*, 38(8):1212 – 1224, August 1989.

[14] J. Wang, S. Huang, and N. Chen. A distributed algorithm for detecting generalized deadlocks. *Technical Report, Department of Computer Science, National Tsing-Hua University*, 1990.