A One-Phase Algorithm to Detect Distributed Deadlocks in Replicated Databases

Ajay D. Kshemkalyani, *Senior Member, IEEE*, and Mukesh Singhal, *Senior Member, IEEE*

Abstract—Replicated databases that use quorum-consensus algorithms to perform majority voting are prone to deadlocks. Due to the P-out-of-Q nature of quorum requests, deadlocks that arise are generalized deadlocks and are hard to detect. We present an efficient distributed algorithm to detect generalized deadlocks in replicated databases. The algorithm performs reduction of a distributed wait-for-graph (WFG) to determine the existence of a deadlock. If sufficient information to decide the reducibility of a node is not available at that node, the algorithm attempts reduction later in a lazy manner. We prove the correctness of the algorithm. The algorithm has a message complexity of $2e$ messages and a worst-case time complexity of $2d+2$ hops, where e is the number of edges and d is the diameter of the WFG. The algorithm is shown to perform significantly better in both time and message complexity than the best known existing algorithms. We conjecture that this is an optimal algorithm, in time and message complexity, to detect generalized deadlocks if no transaction has complete knowledge of the topology of the WFG or the system and the deadlock detection is to be carried out in a distributed manner.

Index Terms—Distributed database, replicated database, quorum consensus, generalized deadlock, graph reduction.

1 Introduction

In database systems, a deadlock occurs when some transactions wait indefinitely on each other for their requests to be satisfied. A deadlock hampers the progress of transactions in a database and lowers the resource availability; therefore, all deadlocks must be promptly detected and eliminated [18], [19], [26]. Although deadlock detection has been extensively studied in traditional distributed databases, e.g., [15], [17], [19], [24], [27], it has not received sufficient attention in the context of replicated (distributed) databases [4], [5], [10].

In a replicated database, data items are replicated at different sites to increase availability (i.e., fault tolerance) and responsiveness to read requests. A write to a data item requires that the value should be written to all the replicas of that item. A read of a data item can be satisfied by reading any copy. Quorum algorithms are used to serialize concurrent read and write operations from different transactions for concurrency control [4], [5], [12]. Several quorum-consensus algorithms have been proposed and they require some form of voting by the replicas of the data item to be read/written, to reach a consensus on whether a read/write can proceed without violating serializability [1], [4], [5], [10], [12], [25]. A typical quorum request is a P-out-of-Q request where the requesting transaction is waiting for at least P votes out of the total of Q votes distributed among the replicas. Once a replica of a data item grants its vote, that replica gets locked and cannot revote until it is unlocked. Replicated databases that use quorum-consensus algorithms are prone to deadlocks because transactions which are waiting for a quorum to be satisfied may be involved in an indefinite wait. Requests in replicated databases fall under the P-out-of-Q request model and the resulting deadlocks are generalized deadlocks.

For the purpose of modeling deadlocks, interaction between transactions is modeled by a directed graph, called a wait-for graph (WFG) [18]. Nodes in a WFG represent transactions and an edge from node i to node j indicates that transaction i has requested a resource from transaction j and transaction j has not granted the resource to transaction i. A deadlock is characterized by topological properties of the WFG that depend upon the underlying transaction request model. For example, in the simplest request model, called the single-request model, as well as in the AND request model, the presence of a cycle in the graph implies a deadlock. In the OR request model, the presence of a knot is a necessary and sufficient condition for a deadlock to exist.

In the P-out-of-Q request model [6], also called the generalized request model, a transaction makes requests for Q resources and remains blocked until it is granted any P out of the Q resources. The P-out-of-Q request model is equivalent to the AND-OR request model [14] in which the condition for a blocked transaction to get unblocked is expressed as a predicate on the requested resources using AND and OR operators. For example, predicate $i \wedge (j \vee k)$ denotes that the transaction is waiting for a resource from i and for a resource from either j or k. The P-out-of-Q and the AND-OR models are equivalent because a predicate in the

- *A.D. Kshemkalyani is with the Electrical Engineering and Computer Science Department, University of Illinois at Chicago, 1120 Science and Engineering Offices, 851 S. Morgan St., Chicago, IL 60607-7053. E mail: ajayk@eecs.uic.edu.*
- *M. Singhal is with the Department of Computer and Information Science, Ohio State University, 2015 Neil Ave., Columbus, OH 43210. E-mail: singhal@cis.ohio-state.edu.*

Manuscript received 9 June 1996; revised 2 Sept. 1999. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104370.

TABLE 1
Comparison of Worst-Case Performance Complexities

Criterion	Bracha-Toueg [6]	Wang et al. [28]	Kshemkalyani-Singhal [21]	Brzezinski et al. [7]	Proposed algorithm
Phases	2	2	1	S	1
Delay	$4d$	$3d+1$	$2d+2$	S^2	$2d$
Messages	$4e$	$6e$	$4e-2n+2l\ (\leq 4e)$	S^2	$2e$

Note : S = number of nodes in the system. Given a WFG (N, E), $n = |N|$, l = number of its sink nodes, $e = |E|$, d = its diameter. Also, $n, d \ll S$.

AND-OR model can be expressed as a disjunction of P-out-of-Q type requests and vice-versa. A generalized deadlock corresponds to a deadlock in the P-out-of-Q (or AND-OR) request model. AND and OR models are special cases of the generalized deadlock model.

Although the problem of deadlock detection has been well explored in the single-request and the AND request models in database systems, e.g., [8], [9], [11], [15], [17], [18], [20], [22], [23], [24], [27], [29], much remains to be done toward the detection of generalized deadlocks in replicated databases. Generalized deadlocks also arise in other domains such as resource allocation in distributed operating systems, store-and-forward communication networks, and communicating processes. Therefore, efficient detection of generalized deadlocks is an important problem.

1.1 Previous Work on Generalized Deadlock Detection

Detecting generalized deadlocks in a distributed database system is a difficult problem because it requires detection of a complex topology in the global WFG; the topology is determined by the conditions that need to be satisfied for each of the blocked nodes in the WFG to unblock. A cycle in the WFG is a necessary but not sufficient condition, whereas a knot in the WFG is a sufficient but not necessary condition for a generalized deadlock.

Traditionally, ad hoc methods like timeout have been used to handle deadlocks. A major drawback of this simple approach is that oftentimes, deadlock is falsely detected when it does not exist, thus causing a wastage of system resources when the transaction is rolled back and then redone. The timeout based methods are proving to be less viable as we strive for high performance computing systems where applications and end-users' expectations of quick response times, continuous availability, and minimal failures are being increasingly market-driven. Precise algorithms are needed to detect deadlocks.

Design of correct and efficient distributed algorithms to detect generalized distributed deadlocks is a difficult problem and very few distributed algorithms exist to detect generalized distributed deadlocks [6], [7], [21], [28]. We do not consider centralized algorithms such as [2], [14] in which a snapshot of the WFG is collected by some process and then examined by that process for deadlock. The algorithms in [6], [21], [28] are based on the "record and reduce" principle; that is, the distributed WFG is recorded and reduced to determine if there is a deadlock. Reduction of the WFG simulates the granting of requests and is a general technique to detect deadlocks [18]. Algorithms that follow the "record and reduce" principle must have a way to detect the termination of the reduction process so that a correct conclusion about the existence of a deadlock may be drawn. The algorithm in [7] is based on the principle of detection of weak termination of a distributed computation.

The algorithm by Bracha and Toueg [6] consists of two phases. In the first phase, the algorithm records a snapshot of a distributed WFG and in the second phase, the algorithm reduces the graph to check for generalized deadlocks. The second phase is nested within the first phase. Therefore, the first phase terminates after the second phase has terminated. In the two-phase algorithm of Wang et al. [28], the first phase records a snapshot of a distributed WFG. The end of the first phase is detected using an explicit termination detection technique, after which the second phase is initiated to reduce the recorded WFG to detect a deadlock. Termination of this phase is also detected using an explicit termination detection technique. In the one-phase Kshemkalyani-Singhal algorithm [21], the recording of a snapshot of the distributed, dynamically changing WFG and reduction of the recorded WFG is done in two concurrent sweeps of the WFG. The algorithm deals with the complications introduced because the reduction of a node in the inward sweep can begin before the state of all WFG edges incident at that node have been recorded in the outward sweep. The termination detection of the "reduce" sweep is merged with the termination detection of the "record" sweep and achieved by a single invocation of an explicit termination detection algorithm. Brzezinski et al. [7] define a generalized deadlock in terms of weak termination of a distributed computation and develop an algorithm that detects weak termination. Nodes are logically arranged as a ring and a token circulates on the ring to monitor the states of the nodes. The token keeps circulating until the monitored states of the nodes are the same in two consecutive rounds. Table 1 compares the performance of the above distributed algorithms.

1.2 Paper Objectives

We present an efficient one-phase distributed algorithm for detecting generalized distributed deadlocks and prove its correctness. The algorithm initiated by an initiator consists of two concurrent sweeps—an outward sweep that records the WFG and an inward sweep that reduces the WFG to detect a deadlock. The outward sweep induces a spanning tree in the WFG. Reduction is performed by sending replies backward on cross-edges of the WFG and backward on

spanning tree edges. During this distributed reduction, even if sufficient information to decide the reducibility of a node is not available at that node, appropriate replies are sent, and the algorithm attempts reduction later in a lazy manner at an up-tree node in the spanning tree. The initiator receives replies on all its outgoing spanning tree edges and cross-edges because the sending of replies is never delayed, and detects termination of the distributed reduction when all such replies are received. Thus, no explicit termination detection algorithm is used. At termination, the existence of a deadlock, if any, is detected.

The proposed algorithm performs better than the existing algorithms to detect generalized deadlocks in terms of the worst-case time complexity and the worst-case message complexity [6], [7], [21], [28]; the algorithm has a message complexity of $2e$ messages and a worst-case time complexity of $2d + 2$ hops, where e is the number of edges and d is the diameter of the WFG. We conjecture that this algorithm is optimal in the number of messages and in time delay if detection of generalized deadlocks is to be carried out under the following framework: 1) no node has complete knowledge of the topology of the WFG or the system, and 2) the deadlock detection is to be carried out in a distributed manner. If the initiator is deadlocked, it has all the necessary information to adequately resolve the deadlock, unlike the algorithms in [6], [7], [21], [28].

The rest of the paper is organized as follows: In Section 2, we discuss the system model and give a precise problem description. In Section 3, we describe the idea behind the algorithm and use an illustrative example. In Section 4, we present the algorithm. In Section 5, we prove the algorithm's correctness. In Section 6, we analyze the performance of the algorithm, and compare it with that of previous algorithms. Section 7 contains a discussion and concluding remarks.

2 System Model

A distributed database system contains databases at various sites. Each data item may be replicated in the databases at various sites. Each transaction in the distributed database system runs at a single site and may access different data items at various different sites. Each transaction is managed by a Transaction Manager, and each replica of a data item is managed by a Data Manager.

The various sites are connected by communication channels so that a logical channel can be set up between each pair of sites. There is no shared memory in the system and sites communicate solely by sending messages over the channels. The messages are reliably delivered with finite but unpredictable delays, and in the order in which they were sent on a channel. If the logical channels deliver messages out of order, then a simple message numbering scheme can help the receiver to process the messages arriving on a logical channel in the correct order.

We make the following assumptions about the system model:

1. It has been shown in [20] that during the execution of a transaction, a data item replica (managed by a Data Manager) behaves like a transaction (managed by a Transaction Manager) which makes requests only in the single-request model, in the following sense. A transaction (respectively, a data item) blocks when it is waiting for a reply for lock requests (respectively, waiting to be unlocked) and cannot release locks (respectively, cannot be locked by another entity) while it is blocked. Thus, a transaction blocks when it makes a P-out-of-Q request to lock a data item, whereas a data item replica blocks when it grants its lock and implicitly makes a single request that it be unlocked. Henceforth, we will not distinguish between "transaction" and "data item" to provide a uniform treatment for both transactions and data items and simplify the presentation of the deadlock detection algorithm. Moreover, the resulting algorithm is directly applicable to resource deadlocks that occur in databases, as well as to communication deadlocks.
2. A deadlock detection algorithm is run by the Transaction Managers and Data Managers. As far as the algorithm is concerned, these Managers are synonymous with the corresponding transactions and data item replicas; therefore, we will refer only to transactions and data item replicas.

As a result of the above assumptions, each transaction and data item replica can henceforth be referred to as a node. The WFG now models both transactions and data item replicas. A node in the WFG is a transaction or a data item replica; a WFG edge from node i to node j denotes one of the following: 1) transaction i has requested a lock on data item (replica) j and j has not granted the lock request to i. 2) data item (replica) i is locked by transaction j and j has not released the lock on i.

We now formalize the blocking and unblocking of nodes. When a node i makes a generalized request and blocks (i.e., goes from active to idle state), the unblocking condition of its request is denoted as f_i. The domain of f_i is the set of all nodes which are referenced in f_i. Function f_i is evaluated in the following manner: substitute *true* for a node id in f_i if i has received a reply, indicating granting of that request, from that node; otherwise, substitute *false* for it. Then evaluate the function.

The node unblocks (goes from idle to active state) when a sufficient number and combination of its requests to make f_i true are granted. When the node unblocks, it withdraws the remaining requests it had sent but are not yet granted.

The following two axioms describe the blocking and unblocking of nodes:

Axiom 1. *A node blocks when it makes a generalized request and does not send any computation messages until it gets unblocked.*

Axiom 2. *A blocked node gets unblocked if and only if its requests are satisfied without any intervention in the computation.*

Note that Axiom 2 describes the normal way in which a node can get unblocked. A node can get unblocked abnormally if it spontaneously withdraws its requests or its requests are satisfied due to the resolution of a deadlock

of which it is a part [20]. There is a risk of false deadlocks being reported if a node unblocks abnormally. Detection of such false deadlocks can be eliminated by using a time-stamping mechanism to consider the dynamically changing WFG along the latest observable state [20]. We do not allow a node to unblock abnormally for simplicity.

The interaction between transactions and data items is modeled by a directed AND-OR wait-for graph denoted by (N, E), where N is the set of nodes and E is the set of directed edges between nodes. Typically, the number of nodes in a WFG is small compared to the number of nodes in the system, i.e., $|N| << |S|$, where S is the set of all nodes in the system.

A node i keeps the following variables to keep track of its portion of the directed AND-OR WFG:

IN_i: **set of node ids** $\leftarrow \emptyset$; /*set of nodes which are directly blocked on node i. It denotes the direct predecessors of node i in the WFG. */

OUT_i: **set of node ids** $\leftarrow \emptyset$; /*set of nodes on which node i is blocked. It denotes the set of nodes that are direct successors of node i in the WFG. */

f_i: **AND-OR expression** $\leftarrow \perp$; /*the condition for unblocking.*/

OUT_i gives the domain of function f_i. The transitive closure of OUT_i, denoted by OUT_i^+, gives the reachability set of i. The transitive closure of IN_i, denoted by IN_i^+, is the set of nodes whose reachability set contains i.

2.1 Problem Statement

A generalized deadlock exists in the system iff a certain complex topology, identified next, exists in the global WFG.

Definition 1. *A generalized deadlock is a subgraph (D, K) of a WFG (N, E) where: 1) each $i \in D(\neq \emptyset)$ is blocked on a function $f_i(OUT_i)$ which evaluates to false when each variable is instantiated as follows:*

$$(\forall j \in D, j \text{ is set to } false) \wedge (\forall j \in OUT_i \setminus D, j \text{ is set to } true),$$

and 2) K is the projection of the edges in (N, E) on the nodes in D.

From Axioms 1 and 2, it follows that none of the nodes in D will ever get unblocked. All nodes in D thus remain blocked forever. All the nodes in the WFG that do not belong to any D have a sufficient number of edges to nodes in $OUT_i \setminus D$, i.e., $f_i(OUT_i)$ evaluates to *true* when each variable is instantiated as follows:

$$(\forall j \in D, j \text{ is set to } false) \wedge (\forall j \in OUT_i \setminus D, j \text{ is set to } true).$$

All these nodes that are not in any D are not deadlocked because their requests can be satisfied.

A distributed deadlock detection algorithm should satisfy the following two correctness conditions:

Liveness: If a deadlock exists, it is detected by the algorithm within a finite time.

Safety: If a deadlock is declared, the deadlock exists in the system.

At the time that a node blocks or within a system-tuned timeout period during which the node has remained blocked, the node initiates a deadlock detection algorithm. Note that only the nodes that are reachable from a node in the WFG can be involved in a deadlock with that node. Thus, the complete WFG is not examined to determine if a node is deadlocked; only the part of the WFG which is reachable from that node needs to be examined. The deadlock detection algorithm is presented for a static WFG. In Section 7, we explain how to extend the algorithm to handle a dynamically changing WFG.

3 Basic Idea

No node has the knowledge of the complete topology of the WFG or the system; therefore, the initiator node determines the reachable part of the WFG and attempts to sense its topology by diffusing FLOOD messages. To initiate deadlock detection, the initiator node sends FLOOD messages to all of its successor nodes. When a node receives the first FLOOD message, it propagates it to all of its successor nodes, and so on. The edges of the WFG on which the first FLOOD message is received by each node induce a directed spanning tree (DST) in the WFG.

Deadlock detection as well as detecting termination of the algorithm are performed by echoing the FLOOD messages at "terminating" nodes and reducing the graph when an appropriate condition at a node in the echo phase is satisfied. A *terminating node* in the graph is either a sink node or a nonsink node that has already received a FLOOD message. Since a sink node is active (and thus, is already reduced), it responds to all FLOOD messages by ECHO messages. By sending an ECHO message, a node informs that it has been reduced. When a nonsink node in the graph receives the second or a subsequent FLOOD message, it responds with an ECHO message provided it has been reduced by then. However, a dilemma arises if a nonsink node in the graph has not been reduced when it receives a second or subsequent FLOOD message. The state of such a node is presently indeterminate and may eventually become reduced after a sufficient number of ECHO messages have been generated and moved up in the graph. Such a node cannot immediately respond to a FLOOD with an ECHO message and, if it waits to see if it is later reduced, the algorithm may deadlock! This dilemma is solved in the algorithm using lazy evaluation as follows.

3.1 Lazy Evaluation

If a nonsink node in the graph has not been reduced when it receives the second or a subsequent FLOOD message, it immediately responds to such a FLOOD message with a Position Indeterminate Packet (PIP) message. A PIP message conveys the indeterminate state of the node. In contrast, an ECHO message conveys the fact that the sender node is reduced. A node attempts a reduction whenever it receives an ECHO. If a node is reduced after it has received

a response to all the FLOOD messages that it sent (we call this "local reduction" of the node), it sends an ECHO message to its parent in the DST. Otherwise, it sends a PIP message to its parent in the DST. Note that if the node was not reduced at this instant, it does not mean that it is not reducible. This is because some of its successor nodes that sent a PIP, might have gotten reduced later and reduction of these nodes might have been sufficient to reduce this node, had it waited long enough. To take care of such conditions, 1) the reduced status of nodes that previously sent a PIP message is propagated upward in the DST toward the initiator node. Also, 2) when an (unreduced) node sends a PIP message to its parent node, the message contains the unsatisfied portion of the unblocking function, called the *residual function*, of the sender node. For example, if the unblocking function of a node is $x \wedge (y \vee z)$ and the node has received an ECHO from y, then the residual function is x. Ancestor nodes of the unreduced node gather both these pieces of information and make an attempt to determine if the node can be reduced.

The information about nodes that sent a PIP but were later reduced is propagated in the following manner. A node i keeps a set of node ids, denoted by R_i, that contains the ids of nodes in OUT_i^+ that sent a PIP, but were reduced later. When a node i sends an ECHO or a PIP message, the current value of R_i is sent in the message. When a node i receives an ECHO or a PIP message, it adds the contents of the received R set to R_i. This is eager dissemination of reduced node information. The eager dissemination is sufficient but not necessary for lazy evaluation. It is necessary for node i only to send R_i on the ECHO or PIP response sent to the parent after the last ECHO or PIP response to its FLOOD messages has been received. However, the size of the set of residual functions (discussed next) at nodes in the WFG is likely to be larger with this modification.

A node j keeps a set of residual functions, denoted by Z_j, that contains tuples of the form $\langle k, f_k \rangle$, where f_k denotes the residual function of node k. The information about the residual function of nodes is propagated in the following manner: When a node sends an ECHO or a PIP message to its parent in the DST (this happens when the node has received a response from all of its successor nodes), the message contains the residual function set of the sender node. An ECHO or a PIP message sent to a nonparent node carries null as the value of the residual function set. When a node j receives an ECHO or a PIP message with a nonnull value of the residual function set, it adds the received residual function set to Z_j. This retarded collection of residual functions is necessary and sufficient for evaluation of the unblocking function at nodes. This is how the information about the residual unblocking function of nodes and the information that a node that sent a PIP was eventually reduced is propagated upward in the tree.

A node j evaluates its unblocking function f_j whenever it receives an ECHO message. In addition, after a node j has received a response to all FLOOD messages it had sent, it evaluates every residual function in the set Z_j as follows:

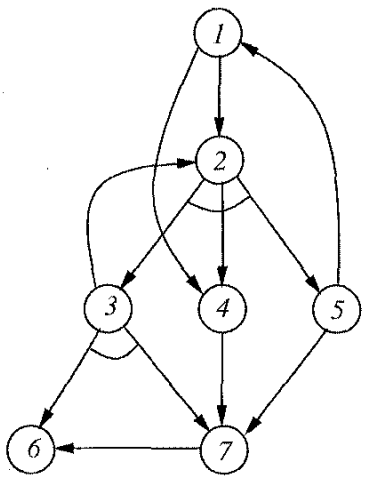

Fig. 1. An example of a wait-for graph (WFG).

select a tuple $\langle k, f_k \rangle$ from Z_j and check if entries in R_j are sufficient to reduce f_k. If a node j succeeds in reducing node k's residual function f_k, we say that node k has been *remotely reduced* (at node j). In such a situation, node j adds k to R_j and deletes tuple $\langle k, f_k \rangle$ from Z_j. This is done repeatedly until no more entries in Z_j can be reduced.

Thus, a node j uses information in R_j about its successor nodes that sent PIP but got reduced or that were remotely reduced, to attempt to reduce the residual function of as yet unreduced descendants in Z_j. As a residual function traverses up the DST, it can progressively strengthen because more reduced node information gets collected by lazy evaluation further up the DST.

The initiator node is deadlocked if it is not reduced after receiving responses to all of its FLOOD messages because no further lazy evaluation can occur. Otherwise, it is not deadlocked. The message complexity, time complexity, local computational complexity, and the size of messages for this algorithm are analyzed in Section 6.

3.2 An Example

We now illustrate the basic idea behind the algorithm with the help of an example. Fig. 1 shows a distributed WFG that spans seven nodes numbered 1 through 7. All nodes except node 6 are blocked. The unblocking functions at these nodes are given next using an oversimplified notation illustrated by the following example:

$$f_1 = 4 \vee 2, f_2 = 3 \wedge 4 \wedge 5, f_3 = 2 \vee (6 \wedge 7), f_4 = 7, f_5 = 1 \vee 7, f_6 = \mathit{true}, f_7 = 6.$$

$f_1 = 4 \vee 2$ denotes that node 1 needs a reply from node 2 or node 4 to unblock.

Suppose node 1 initiates deadlock detection and sends out FLOOD messages to nodes 2 and 4. Fig. 2 shows the diffusion of FLOOD messages through the WFG. The thicker edges of the graph denote the edges along which nodes received their first FLOOD message and define the DST.

Fig. 3 shows how various nodes respond to FLOOD messages they receive. Since node 6 is active, it responds

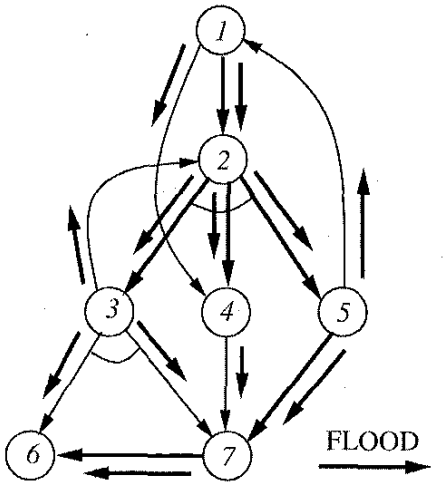

Fig. 2. Diffusion of FLOOD messages.

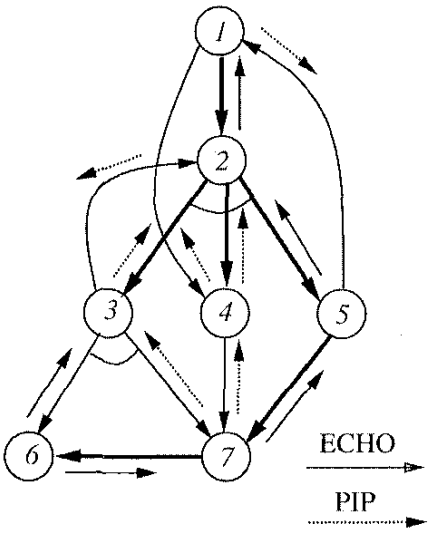

Fig. 3. Propagation of ECHO/PIP messages.

to the FLOOD messages from nodes 3 and 7 by ECHO[1] (6, 1, $\emptyset$, $\emptyset$) messages. Before node 7 receives the ECHO message, it receives FLOOD messages from nodes 3 and 4. Node 7 responds to these FLOOD messages by PIP(7, 1, $\emptyset$, $\emptyset$) messages because the state of node 7 is indeterminate at these instants. On the receipt of an ECHO(6, 1, $\emptyset$, $\emptyset$) message, node 7 succeeds in reducing itself and sends an ECHO(7, 1, {7}, $\emptyset$) message to node 5, its parent in the DST.

After receiving an ECHO message from node 7, node 5 gets reduced and sets R_5 to {7}. On receipt of a PIP message from node 1, node 5 sends ECHO(5, 1, {7}, $\emptyset$) message to node 2.

Node 4 receives a FLOOD from node 1 before it receives PIP from node 7. Consequently, it responds to the FLOOD with a PIP(4, 1, $\emptyset$, $\emptyset$). Node 4 is not reduced after it has received PIP from node 7 and thus sends a PIP(4, 1, $\emptyset$, $\{\langle 4,7\rangle\}$) to its parent in the DST (node 2).

Node 2 sends a PIP(2, 1, $\emptyset$, $\emptyset$) message to node 3 in response to the FLOOD it receives from node 3. Node 3 is not reduced after it has received ECHO from node 6 and PIP messages from nodes 2 and 7. However, its residual function is $2 \vee 7$. Therefore, it sends PIP$(3, 1, \emptyset, \{\langle 3, 2 \vee 7\rangle\})$ message to node 2.

On the receipt of PIP$(4, 1, \emptyset, \{\langle 4,7\rangle\})$ from node 4 and PIP$(3, 1, \emptyset, \{\langle 3, 2 \vee 7\rangle\})$ from node 3, Z_2 at node 2 becomes $\{\langle 3, 2 \vee 7\rangle, \langle 4,7\rangle\}$. On the receipt of ECHO(5, 1, {7}, $\emptyset$) message from node 5, node 2 sets R_2 to {7}. It adds its residual function $\langle 2,\ 3 \wedge 4\rangle$ to Z_2 and succeeds in reducing all three residual functions in Z_2 using R_2. Consequently, R_2 becomes {3, 4, 7}. Since node 2 is reduced, it sends $ECHO(2, 1, \{3, 4, 7\}, \emptyset)$, to node 1. On receipt of this message, node 1 is reduced and declares "no deadlock."

1. The first parameter of an ECHO or a PIP message is the sender's id, the second parameter is the initiator node id, the third parameter is the R set of the sender, and the fourth parameter is the Z set of the sender node.

4 A Distributed Deadlock Detection Algorithm

The pseudocode for the algorithm uses the symbol $\leftarrow$ for the assignment operator, and the CSP-like symbol $\Box$ for the selection operator. We choose the CSP-like notation because it expresses concurrency more explicitly. The notation $[GC_1\Box\ GC_2\Box\ \ldots\ \Box GC_n]$ is an alternative guarded command, where a guarded command GC_i has the form "$a \longrightarrow b$" with the semantics "if a then b else $skip$". A node i has variables OUT_i, IN_i, and f_i which describe the WFG locally. Different invocations of the algorithm by the same initiator are differentiated by timestamps, which are not shown for simplicity. The deadlock detection algorithm is given in Fig. 4a and Fig. 4b. The processing when a FLOOD, ECHO, or PIP is received is done atomically.

5 Correctness Proof

We prove that the initiator of the deadlock detection algorithm declares deadlock iff it is deadlocked. The proof uses several observations (Observations 1-7) and lemmas (Lemmas 1-11) about the properties of the algorithm.

The FLOOD messages induce a directed spanning tree (DST) in the WFG. The root of the tree is the initiator and the parent of each node i in the tree, denoted by $parent_i$, is the node from which i received its first FLOOD. The transitive closure of $parent_i$, denoted by $parent_i^+$, is the set of all ancestors of i. The children of node i in the DST, denoted by $offspring_i$, are the nodes k such that $parent_k = i$. The transitive closure of $offspring_i$, denoted by $offspring_i^+$, is the set of nodes in the subtree rooted at i.

Assertion: FLOOD messages are diffused through the entire reachable WFG of the initiator.

Data Structures
$parent_i$: **integer** $\leftarrow 0$; /*node id of parent of node i. */
out_i: **set of integer** $\leftarrow OUT_i$; /*nodes for which node i is waiting. */
R_i : **set of integer** $\leftarrow \emptyset$; /* nodes in this subtree which sent PIPs, */
/*and which subsequently got reduced.*/
pip_sent_i: **boolean** $\leftarrow false$; /* indicates if i sent PIP to other nodes. */
X_i : **AND-OR expression** $\leftarrow f_i$; /* unblocking function for i. */
define struct FNRES $\{id$:**integer;** /* node identifier. */
uc:**AND-OR expression;**$\}$ /* residual unblocking function. */
str: **FNRES** $\leftarrow \perp$; /* local residual function. */
Z_i : **set of FNRES** $\leftarrow \emptyset$; /* residual functions of unreduced nodes in subtree. */

initiate algorithm
/*Executed by node i to detect whether it is deadlocked. */
$init \leftarrow i$;
$parent_i \leftarrow i$;
send $FLOOD(i, i)$ to each j in out_i.

receive FLOOD(k, $init$)
/*Executed by node i on receiving a FLOOD message from k. */
[
/* FLOOD for new invocation (detected by timestamps, unshown).*/ /*Case F1.*/
$out_i = \perp \longrightarrow$
$parent_i \leftarrow k$; $out_i \leftarrow OUT_i$; $R_i, Z_i \leftarrow \emptyset$; $X_i \leftarrow f_i$; $pip\_sent_i \leftarrow false$;
$f_i = true \longrightarrow$ /* i is unblocked. Case F1-A. */
send $ECHO(i, init, \emptyset, \emptyset)$ to k;
$f_i = false \longrightarrow$ /* i is blocked. Case F1-B. */
send $FLOOD(i, init)$ to each $j \in out_i$;
□
/* FLOOD received before all expected PIPs/ECHOs received. */ /* Case F2. */
$out_i \neq \emptyset \longrightarrow$
$X_i = true \longrightarrow$ /* i is unblocked. Case F2-A. */
send $ECHO(i, init, R_i, \emptyset)$ to k;
$X_i = false \longrightarrow$ /* i is blocked. Case F2-B. */
send $PIP(i, init, R_i, \emptyset)$ to k;
$pip\_sent_i \leftarrow true$;
□
/* FLOOD received after all expected ECHOs/PIPs received. */ /* Case F3. */
$out_i = \emptyset \longrightarrow$
$X_i = true \longrightarrow$ /* i is unblocked. Case F3-A. */
send $ECHO(i, init, R_i, \emptyset)$ to k;
$X_i = false \longrightarrow$ /* i is blocked. Case F3-B. */
send $PIP(i, init, R_i, \emptyset)$ to k.
]

receive ECHO(j, $init$, R, Z)
/*Executed by node i on receiving an ECHO from j. */
$X_i = false \longrightarrow$ /* if i is blocked, try reducing it by instantiating all instances of j */
$X_i \leftarrow X_i(OUT_i \mid_{j=true})$; /* in OUT_i by *true* and then evaluating X_i. Step E1. */
$X_i = true \longrightarrow$
$init = i \longrightarrow$ NO deadlock; exit;

Fig. 4a. The deadlock detection algorithm (continued on next page).

```
        pip_sent_i = true ⟶ R_i ← R_i ⋃ {i};        /* update R_i to indicate i sent PIPs. */
common_reply_processing;                            /* perform processing common to PIPs and ECHOs. */

receive PIP(j, init, R, Z)
/*Executed by node i on receiving a PIP from j. */
common_reply_processing;                            /* No special actions unique to PIP are needed. */

common_reply_processing
/*Executed by node i to do common actions when either a PIP or ECHO is received. */
out_i ← out_i \ {j};                                /* update local variables out_i, R_i, Z_i. Step EP1. */
R_i ← R_i ⋃ R;
Z_i ← Z_i ⋃ Z;
out_i = ∅ ⟶                                         /* all expected replies are received. Step EP2. */
    X_i = false ⟶                                   /* i is not yet reduced. Add to Z_i. Step EP2.1. */
        str.id ← i;
        str.uc ← X_i;
        Z_i ← Z_i ⋃ {str};
    eval;                                           /* use R_i to evaluate unreduced nodes in Z_i. Step EP2.2. */
    (X_i = true ⋀ pip_sent_i) ⟶                     /* examine X_i using R_i which was updated in eval. */
        R_i ← R_i ⋃ {i};                            /* if i sent PIP, update R_i to indicate so. Step EP2.3. */
    X_i = true ⟶                                    /* i is locally reduced using updated R_i. Step EP2.4. */
        init = i ⟶ NO deadlock; exit;
        send ECHO(i, init, R_i, Z_i) to parent_i;
    X_i = false ⟶                                   /* i is not locally reduced using updated R_i. Step EP2.5. */
        init = i ⟶ deadlock; exit;
        send PIP(i, init, R_i, Z_i) to parent_i;

eval
/*Executed by node i to evaluate Z using the data that nodes in R are unblocked. */
tempR : set of integer ← R_i;                       /* working variable for R_i. */

repeat
    For every r ∈ tempR do par
        for every z ∈ Z_i do par instantiate each occurrence of r in z.uc by true ;
        rap od;
        tempR ← tempR \ {r};
    rap od;
    for every z ∈ Z_i do par
        z.uc = true ⟶
            tempR ← tempR ⋃ {z.id};
            z.id ≠ i ⟶ R_i ← R_i ⋃ {z.id};          /* if z.id = i, then X_i will also be true. */
            Z_i ← Z_i \ {z};
    rap od;
until tempR = ∅.
```

Fig. 4b. The deadlock detection algorithm (continued from previous page).

The initiator *init* sends FLOOD messages to all nodes in its OUT_{init}. When a node receives the first FLOOD message, it sends FLOOD messages to all its direct successor nodes (case F1-B) and so on. From induction, FLOOD messages are diffused through the entire reachable WFG of the initiator.

Definition 2. *A node i is locally terminated iff it has processed all the PIPs and ECHOs it expected in response to the FLOODs it sent, i.e., $out_i = \emptyset$.*

From Theorems 2 and 3 on time and message complexity (see Section 6), it follows that the algorithm terminates in finite time.

Definition 3. *Node i is* locally reduced *iff it receives a sufficient number of ECHOs so that $X_i = true$ when $out_i = \emptyset$.*

Definition 4. *Node i is remotely reduced at*

$$j \in IN_i^+ \; iff \; \exists j \in IN_i^+, \; \exists z \in Z_j$$

such that $z.id = i$ and there are enough elements in R_j (these have all been reduced either locally or remotely) by the time $out_j = \emptyset$ to satisfy i's residual unblocking condition $z.uc$ at j.

This reduction is remote and i is not aware of it. Note that i's residual unblocking condition in Z_j may be stronger than X_i, indeed it may even be *true*. The presence of a sufficient number of elements in R_j indicates that i's requests as represented in X_i are satisfiable.

The Boolean variable $reduce^i$ will be used to indicate whether node i was reduced. The Boolean variable $reduce_j^i$ will be used to indicate whether node i is reduced at node j. $reduce_i^i$ indicates that node i was locally reduced. $reduce_{j \neq i}^i$ indicates that node i was remotely reduced at node j.

Observation 1. *The Z parameter sent by i on a PIP or an ECHO to $parent_i$ is the value of Z_i when $out_i = \emptyset$. The Z parameter sent by i on a PIP or ECHO to other nodes is $\emptyset$.*

Observation 2. *z (where $z.id = i$) can belong to Z_j in two ways:*

1. *if $j \neq i$, j received an ECHO/PIP that contained z and added z to Z_j (step EP1), or*
2. *if $i = j$, j added z to Z_j because $reduce_j^j$ was* false *when $out_j = \emptyset$ (step EP2).*

Lemma 1 states that the residual function for a node i, i.e., z, where $z.id = i$, does not exist at a node that is not an ancestor of i in the DST nor is it sent in ECHOs or PIPs by such a node.

Lemma 1. *$\forall i \in N, \forall j \in N \setminus (parent_i^+ \bigcup \{i\})$, Z_j does not contain z such that $z.id = i$, nor does node j send or receive an ECHO or PIP in which the Z parameter contains this z.*

Proof. Assume that Z_j contains z (where $z.id = i$). This can happen only by case 1 of Observation 2. j may receive an ECHO or PIP only from some k, $k \in N \setminus (parent_i^+ \bigcup \{i\})$, containing such a z (follows from Observation 1). k must have received such a z in an ECHO or PIP to include it in Z_k and send it to j. Using an inductive argument and noting that exactly one ECHO or PIP is sent on an edge (Theorem 3), there must be a node h, $h \in N \setminus (parent_i^+ \bigcup \{i\})$, that locally inserted i in Z_h by case 2 of Observation 2, implying $h = i$. This contradicts Observation 1. Hence, j does not receive such a z or send such a z in the Z parameter in an ECHO or PIP, and Z_j cannot contain z (where $z.id = i$). □

Lemma 2 states that if node i is not reduced at local termination, then its residual function, i.e., z, where $z.id = i$, may exist only at nodes that are ancestors of i.

Lemma 2. *$\neg reduce_i^i \Longrightarrow z$ (where $z.id = i$) may belong only to Z_j, where $j \in parent_i^+$.*

Proof. The Z parameter on all messages other than to $parent_i$ sent by i is $\emptyset$ (Observation 1).

If $\neg reduce_i^i$ then at the time $out_i = \emptyset$, i adds z (where $z.id = i$) to Z_i and z remains in Z_i after i executes *eval*. The Z parameter sent to $parent_i$ is Z_i. $parent_i (= j)$ will add the received Z parameter to Z_j. In turn, if $\neg reduce_j^i$, then j will forward Z_j which contains the z (where $z.id = i$) only to $parent_j$. By induction, the z value may belong to $parent_i^+$.

From Lemma 1, no other node k receives the z value under consideration in any message, or inserts it in Z_k. The lemma follows. □

Lemma 3 states that if the residual function of node i, i.e., z, where $z.id = i$, exists at another node j, then the residual function was created at i because node i was not reduced at local termination.

Lemma 3. *$z \in Z_j$ where $z.id = i \Longrightarrow i$ added z to Z_i because $\neg reduce_i^i$ when $out_i = \emptyset$.*

Proof. z (where $z.id = i$) can belong to Z_j in two ways given in Observation 2. The lemma holds if $i = j$ (case 2). For case 1, assume that j received an ECHO or PIP from some k containing such a z. Either $k = i$, or $k \neq i$ and k must have received such a z in an ECHO or PIP to include it in Z_k and send it to j. Using an inductive argument and noting that exactly one ECHO or PIP is sent on an edge, there must be a node h that locally inserted i in Z_h by case 2, implying $h = i$. Node i includes the z (where $z.id = i$) in Z_i only if $\neg reduce_i^i$ when $out_i = \emptyset$. □

Lemma 4 states that i may be reduced at most at one node in $\{i\} \bigcup parent_i^+$.

Lemma 4.

$$reduce^i \Longrightarrow reduce_i^i \oplus (reduce_j^i, where \; (j \in parent_i^+ \bigwedge (\nexists \, k \in parent_i^+ | (k \neq j \bigwedge reduce_k^i)))).$$

Proof. If $reduce_i^i$, then no message sent by i will have z (where $z.id = i$) in the Z parameter (Observation 1). From Lemma 3, for any other node k, Z_k will not contain this z variable. $reduce_{k \neq i}^i$ cannot happen because z (where $z.id = i$) $\notin Z_k$.

If $reduce_{j \neq i}^i$, assume without loss of generality that j is a node such that $\neg reduce_k^i$, where j, $k \in parent_i^+$ and $j \in parent_k^+$. From Lemma 2, z (where $z.id = i$) may belong only to Z_j, where $j \in parent_i^+$. As $reduce_j^i$, where $j \in parent_i^+$, when i gets reduced in procedure *eval* at j, the element z (where $z.id = i$) is deleted from Z_j. From Observation 1 and Lemma 1, it follows that no node in $parent_j^+$ will contain this z (where $z.id = i$); hence i cannot get reduced (again) at a node in $parent_j^+$. The lemma follows. □

Observation 3. *The R parameter that node i sends to $parent_i$ in an ECHO or a PIP is a superset of the union of the R parameters in ECHOs and PIPs that i received (step EP1).*

Lemma 5 states that if i was remotely reduced at node k, then at local termination, i belongs to R_j for every ancestor j of k.

Lemma 5. $reduce^i_{k \neq i} \Longrightarrow \forall j \in parent^+_k \bigcup \{k\}$ *when* $out_j = \emptyset$, $i \in R_j$.

Proof. If i was remotely reduced at node k, then i is placed in R_k during procedure *eval*. The lemma follows from Observation 3, the observation that the value of the R parameter on any PIP or ECHO received by j is set-added to R_j, and the observation that an ECHO or PIP is sent to $parent_j$ only after j has received an ECHO or PIP from each node in OUT_j (step EP2). □

Lemma 6 states that if a node j that sent a PIP gets reduced before local termination, the element j is contained in R_i for every ancestor i of j at the time i locally terminates.

Lemma 6. *(Node* j *sent a PIP and then* $reduce^j_j) \Longrightarrow \forall i \in parent^+_j \bigcup \{j\}$ *at the time* $out_i = \emptyset$, $j \in R_i$.

Proof. When node j sends a PIP and $out_j \neq \emptyset$, it sets pip_sent_j to *true* (case F2-B). In the other cases when j sends a PIP, the following holds: $out_j = \emptyset$ and $reduce^j_j = false$ (steps F3-B, EP2.5). Note that $out_j \neq \emptyset$ before $reduce^j_j$ if j sent a PIP and then $reduce^j_j$. Subsequently, at the time $reduce^j_j$, j is added to R_j (step E1 or EP2.3) and, henceforth, j does not send PIPs. When $out_j = \emptyset$, R_j is sent to $parent_j$ in an ECHO (step EP2.4). When $parent_j$ receives the ECHO from j, the R parameter contains j and is set-added to R_{parent_j}. Subsequently, when $out_{parent_j} = \emptyset$, $parent_j$ sends an ECHO/PIP to its parent, and the R parameter on this message contains j (Observation 3, steps EP2.4, EP2.5). Using an inductive argument, the R parameter containing j is sent in ECHOs and PIPs (steps EP2.4 and EP2.5) sent to nodes in $parent^+_j$. Each $i \in parent^+_j$ will have j in R_i at the time $out_i = \emptyset$ (step EP1). □

Lemma 7 states that if $i \in R_j$, then i was already reduced at some node l before local termination ($l = i$) or was remotely reduced at some node $l \in OUT^+_j \bigcup \{j\}$.

Lemma 7. $i \in R_j \Longrightarrow reduce^i_l, l \in OUT^+_j \bigcup \{j\}$ *and* $reduce^i_l$ *happened before* i *was placed in* R_j.

Proof. i may belong to R_j only under one of the following conditions:

1. If $reduce^i_i$ holds and $j = i = l$. This denotes local reduction (step E1).
2. When j invokes procedure *eval*, i gets reduced, i.e., $reduce^i_{j \neq i}$ and $j = l$. This is remote reduction (step EP2.2).
3. Whenever a PIP or ECHO is received by j, i is contained in the R set parameter on the received message, and this R parameter is set-added to R_j (step EP1).

If $i \in R_j$, then i got reduced at j (items 1 and 2 above) or a successor $k \in OUT_j$ sent j a PIP or ECHO containing i in the R parameter (item 3 above). For item 3 above, we show by an inductive argument that i got reduced at some node $h \in OUT^+_j$ by items 1 or 2. Note that only one PIP or ECHO is sent on a WFG edge (proved independently in Theorem 3). Therefore, there must exist a finite sequence $\langle j, k, \ldots, h \rangle$ of at most c nodes in OUT^+_j where:

- all the nodes except h added i to their local variable R by item 3, and then sent an ECHO/PIP whose R parameter contained j to the previous node in the sequence.
- h added i to R_h by items 1 or 2, and then sent an ECHO/PIP whose R parameter contained j to the previous node in the sequence.

It follows that if $i \in R_j$, then in all cases, $reduce^i_l$, where $l \in OUT^+_j \bigcup \{j\}$, and $reduce^i_l$ happened before i was placed in R_j. From Lemma 4, note that $l \in parent^+_i \bigcup \{i\}$. □

Observation 4. *If node* i *belongs to the* R *parameter in some ECHO or PIP received by* j, *then* $reduce^i_k$ *where* $k \in OUT^+_j$.

Lemma 8 states that if node i sends a PIP (either before it is locally reduced or because it is not reduced at local termination), then at local termination at each ancestor node j of i, $[(z \in Z_j, \text{where } z.id = i) \oplus (i \in R_j)]$.

Lemma 8.

$$(i \text{ sends a } PIP \bigwedge reduce^i_i) \bigvee \neg reduce^i_i \text{ when } out_i = \emptyset$$
$$\Longleftrightarrow \forall j \in (parent^+_i \bigcup \{i\}) \text{when } out_j = \emptyset,$$
$$[(z \in Z_j, where\ z.id = i) \oplus (i \in R_j)].$$

Proof. ($\Longrightarrow$:) If i sends a PIP and $reduce^i_i$, then i is inserted in R_i because $pip\_sent_i = true$ (steps E1, EP2.3), and R_i is sent to $parent_i$ in the R parameter in ECHO. From Observation 3, $i \in R_j$, $\forall j \in parent^+_i$. Z_i does not contain z, where $z.id = i$, at the time $out_i = \emptyset$ because $reduce^i_i$. From Lemma 3, it follows that no node j has $z \in Z_j$, where $z.id = i$.

If $\neg\ reduce^i_i$, then i adds z, where $z.id = i$, to Z_i which is sent to $parent_i$ only (Observation 1). i is not inserted in R_i and is not sent in the R parameter to any node. If i is remotely reduced at j ($j \in parent^+_i$ by Lemma 4), then $\forall k \in parent^+_i$ that lie between i and j, $z \in Z_k$ and $i \notin R_k$. $z \in Z_k$ because a precondition for $reduce^i_j$ to occur is that z belongs to Z_j; from Lemma 1 and Observation 2, the only way this can happen is that all nodes k receive z in the Z parameter and pass it up the tree toward j. Also, $i \notin R_k$ because i gets reduced only at j (Lemma 4); this can happen only after ECHOs/PIPs containing z, where $z.id = i$ in the Z parameter reach from i to j through k, at which time j adds i to R_j. If $i \in R_k$, it follows from Lemma 7 that i is already reduced, a contradiction.

At j, z is removed from Z_j and atomically added to R_j at local termination. For all nodes h in $parent^+_i$ that lie between j and the initiator, the z does not belong to Z_h (see Observation 2) because: 1) the Z parameter in no message from a node not on the DST path from j

to h contains z (Lemma 1); and 2) neither does the PIP/ECHO from any node on the DST path between j and h because j removed z from Z_j (Observation 1). Also, from Lemma 5, $i \in R_h$ for all such h. Hence, the RHS holds.

($\Longleftarrow$:) If $z \in Z_j$, where $z.id = i$, then z must have been locally inserted in Z_i by i only (Lemma 3); this happens when $\neg reduce_i^i$ at the time $out_i = \emptyset$.

If $i \in R_j$, then from Lemma 7, i was remotely reduced ($\neg reduce_i^i$ when $out_i = \emptyset$) or i was locally reduced. Note that if i did not send a PIP and was locally reduced, then i is not inserted in R_i or in any R_j. It then follows that i must have already sent a PIP if it was locally reduced. The LHS follows. □

Observation 5. *For any i, the value of X_i which is represented in $z.uc$, where $z.id = i$ and z is in the parameter Z, progressively strengthens as it ascends up the spanning tree in PIP and ECHO messages.*

Lemma 9. *If a node i sends an ECHO or a PIP to node j, then node i must have received a FLOOD from node j.*

Proof. Node i sends an ECHO or PIP to node j only in these two cases:

- on the receipt of a FLOOD from j if either case F1-A, F2, or F3 holds, or
- on the receipt of the last ECHO or PIP (steps EP2.4, EP2.5), i sends an ECHO or a PIP to $parent_i$. $parent_i$ is initialized to j, so i must have received a FLOOD from the parent (case F1-B).

In both the situations, i must have received a FLOOD from j. □

Observation 6. *If node i receives an ECHO or a PIP from node j, node i has already sent a FLOOD to node j and $j \in out_i$. (Follows from Lemma 9 and case F1-B.)*

Before proving that reduction is performed correctly, we define a function h on the nodes in the WFG as follows: First, view the predicate f_i in disjunctive normal form (DNF), where each disjunct x is of type P-out-of-Q [14]. $OUT_{i,x}$ is the set of successors of i that are involved in disjunct x. A disjunct x at node i requires $p_{i,x}$ replies in response to its $q_{i,x}$ requests, where $p_{i,x} \leq q_{i,x}$, to unblock node i. For every disjunct at any node i in a deadlock (D, K), there are at least $q_{i,x} - p_{i,x} + 1$ outgoing WFG edges to other nodes in D.

$$h(i) \leftarrow \begin{cases} 0 & \text{if } i \text{ is a leaf in WFG} \\ \infty & \text{if } i \text{ is deadlocked} \\ 1 + \left(\min_x \left\{ \begin{array}{l} p_{i,x}^{th} \text{ smallest of} \\ \{h(j) | j \in OUT_{i,x}\} \end{array} \right\} \right) & \text{otherwise} \end{cases}$$

$h(i)$ indicates the shortest length over a sufficient number of paths that have to be traversed by replies to reach i, so as to unblock i. If node i were to get unblocked by receiving replies, at least one of them has to traverse a path of length $h(i)$. However, this does not preclude node i from getting unblocked by receiving a reply that has traversed a path of length greater than $h(i)$. This is because more than the required number of nodes may send replies, and these nodes need not lie on paths of length $\leq h(i)$.

A node that is not deadlocked has a finite value of h because it has a sufficient number of edges to other nodes which are not deadlocked and there are sequences of replies by which the node can get unblocked. A deadlocked node is assigned a value of ∞ for h because there are no sequences of replies by which the node can get unblocked. The length of the shortest path traversed by a series of replies to unblock a deadlocked node is ∞.

We now show that the algorithm performs reduction correctly.

Definition 5. *A node in the WFG performs reduction iff it gets reduced and behaves as follows:*

1. *node i sends a PIP (ECHO) to all nodes other than $parent_i$ from which FLOOD is received before (after) local reduction. The Z parameter is null on the PIP (ECHO) message.*
2. *node i sends an ECHO (PIP) to $parent_i$ when $out_i = \emptyset$ if $reduce_i^i (reduce_{j \neq i}^i)$. The R and Z parameters are set to R_i and Z_i, respectively, where:*

   R_i is set to:
   $\bigcup_{k \in OUT_i} R$ parameter received on ECHO/PIP from k
   $\bigcup \{i\}$ if (i has sent a PIP) and ($reduce_i^i$ when $out_i = \emptyset$))
   $\bigcup$ (nodes that i remotely reduces in proc. eval),

   Z_i is set to:
   ($\bigcup_{k \in OUT_i} Z$ parameter received in ECHO/PIP from k
   $\setminus \{k \mid k$ is (remotely) reduced in procedure eval$\}$)
   $\bigcup \{z \mid z.id = i, z.uc = X_i$, at local termination$\}$
   if $\neg reduce_i^i$ when $out_i = \emptyset$

Observation 7. *Node i does not send any ECHOs unless $reduce_i^i$.*

Lemma 10. *A node i for which $h(i) < \infty$ performs reduction.*

Proof. We prove the result by using induction on $h(i)$.

Base case $h(i) = 0$: We show that a node i for which $h(i) = 0$ performs reduction by noting that its following two properties satisfy Definition 5. 1) When such a node i receives the first FLOOD, it executes case F1-A and records itself as "active" ($X_i = true$). $reduce_i^i$ because i has received a sufficient number of ECHOs, which is zero (0) in this case. Such a node returns an ECHO for every FLOOD it receives (cases F1-A, F3-A). Cases F1-B, F2, F3-B, E*, and EP* do not occur at this node. Hence, it does not send any PIP. From Lemma 9, it follows that ECHOs it sends are only in response to FLOODs. 2) By steps EP1 and EP2, the parameters R and Z sent to $parent_i$ are set per Definition 5. Therefore, a node for which $h(i) = 0$ performs reduction.

$h(i) = x > 0$: Assume that a node i with $h(i) = x$ performs reduction.

$h(i) = x + 1$: It needs to be shown that a node i with $h(i) = x + 1$ performs reduction. At the time node i receives the first FLOOD, node i executes case F1-B and records $X_i = f_i$ (which evaluates to *false* because i is blocked). By definition, there are a sufficient number

of nodes in OUT_i to unblock i and these nodes have a value of h that is $\leq x$. Such nodes perform reduction by the induction hypothesis. Such a node, say $k \in OUT_i$, gets reduced only in the following ways:

1. $reduce_k^k = true$ before k receives the FLOOD from i (cases F2-A, F3-A).
2. $parent_k = i$ and $reduce_k^k = true$ (case F1, step PE2.4).
3. $reduce_i^k$ during *eval* at i (step PE2.4).
4. $reduce_{j \neq i}^k$, $j \in OUT_i^+$, and the reduction of k is learned by i through the R parameter in a received ECHO or PIP. This scenario includes the case where $reduce_k^k$ becomes *true* after k receives a FLOOD from i, where $i \neq parent_k$. Note that j may or may not be an offspring of i.
5. $reduce_j^k, j \notin OUT_i^+, j \neq i$. In this case, the R parameter containing k sent in ECHOs or PIPs by j does not reach i.
6. $reduce_{j \neq k}^k, j \in OUT_i^+$ and the R parameter containing k sent in ECHOs or PIPs by j does not reach i.
7. $reduce_k^k$ after receiving the FLOOD from i, $parent_k \neq i$ (case F2-B) and the R parameter containing k sent on ECHOs or PIPs by k does not reach i.

Each node k that gets reduced by cases 1 and 2 above sends an ECHO to i. Each node k that gets reduced by cases 3 and 4 above, gets reduced at node i and at a successor of i, respectively.

If a sufficient number and combination of nodes k get reduced by cases 1–4, then i gets locally reduced, and behaves as follows: 1) After getting reduced, i sends an ECHO only in response to every FLOOD (cases F1-A, F2-A, F3-A, and step EP2.4 to respond to the FLOOD from $parent_i$) and does not send any PIP. Before i got reduced, i never sent a ECHO (Observation 7) and i sent a PIP only in response to every FLOOD (case F2-B), other than the FLOOD from $parent_i$. 2) By steps EP1 and EP2, the parameters R and Z sent to $parent_i$ are set per Definition 5. Hence, i performs reduction.

Each node k that gets reduced by cases 5-7 above sends a PIP to i in response to the FLOOD from i. If a sufficient number and combination of nodes k do not get reduced by cases 1-4, then they will get reduced by cases 5-7, but $\neg reduce_i^i$. We show that i will be remotely reduced. Denote the set of nodes k by A. For each node k in A, identify the node l in $parent_k^+$ where $reduce_l^k$, and l added k to R_l (Lemmas 5 and 6). Denote this set of nodes l as B. Clearly, B exists—it is $\{init\}$ in the degenerate case. Let j be the common DST ancestor of i and the nodes in B. Clearly, j exists—it is *init* in the degenerate case. Then, R_j contains all the elements in A (Lemmas 5 and 6) which are sufficient to reduce i in procedure *eval* at j, if i is not already in R_j.

Specifically, from Lemma 8,

$$[(z \in Z_j, \text{where } z.id = i) \oplus i \in R_j)].$$

If $(z \in Z_j$ where $z.id = i)$, then i gets reduced at j. If $i \in R_j$, then i was reduced somewhere along the branch from i to j (Lemmas 4 and 7). This was due to the eager dissemination of R on all PIPs and ECHOs. The residual unblocking predicate X_i, represented as z, where $z.id = i$, in the Z parameter of ECHOs/PIPs was transmitted toward j up the spanning tree edges through some combination of PIPs and ECHOs. This $z.uc$, where $z.id = i$, may have been strengthened along the way, (and might have even become *true*), when R at the intermediate nodes became big enough to satisfy some (or all) of $z.uc$ (Observation 5).

In addition to getting remotely reduced, i behaves as follows: 1) i sends a PIP to only all nodes other than $parent_i$ from which a FLOOD is received (cases F2-B, F3-B) and never sends ECHOs (Observation 7). i sends a PIP to $parent_i$ (step EP2.5). 2) By steps EP1, EP2, the parameters R and Z in the PIP sent to $parent_i$ are set per Definition 5.

Hence, a node whose $h = x + 1$ performs reduction. □

Lemma 11. *A node i for which $h(i) = \infty$ does not get reduced.*

Proof. By definition, all nodes whose h is ∞ form a deadlock (D, K) in the WFG. For any node $i \in D$, i does not have a sufficient number of edges to nodes in $OUT_i \setminus D$ to get unblocked. When node i receives the first FLOOD, $X_i = f_i$ (which is *false*) and i propagates FLOOD on its outward edges.

From case F1-B and Observation 6, node i may receive at most one ECHO only on an outgoing WFG edge. All the nodes in $OUT_i \setminus D$ have their $h < \infty$ and perform reduction. Each node in $OUT_i \setminus D$ may send an ECHO on its incoming edges but that is not sufficient to reduce i.

We show by contradiction that no node in D gets reduced. Assume that $i \in D$ is the first node in D to get reduced (locally or remotely). If i was locally reduced, then i received at least an ECHO from another node in D (contradicts the assumption, by Observation 7) or at least one $j \in D$ belongs to R_i, implying by Lemma 7 and Observation 4 that j was reduced before i got reduced (contradicts the assumption). If i was remotely reduced at some k, at least one $j \in D$ belongs to R_k, implying by Lemma 7 and Observation 4 that j was reduced (either local or remote to j) before i was reduced at k (contradicts the assumption). *Reductio ad absurdum.*

Thus, no $i \in D$ gets reduced. □

Theorem 1. *The initiator declares deadlock iff it is deadlocked.*

Proof. Reduction of the WFG is performed correctly from Lemmas 10 and 11. The order of reduction of nodes is unpredictable because of unpredictable message delays. However from Holt's result [18], the nodes can be reduced in any order without changing the final outcome. All deadlocked nodes are not reduced and all other nodes are reduced.

(Sufficiency:) If the initiator declares deadlock, it is not locally reduced (step EP2.5). The initiator cannot get remotely reduced. Therefore, it is not reduced. From Lemmas 10 and 11, it is deadlocked.

(Necessity:) If the initiator is deadlocked, it is not reduced (Lemma 11). So, on local termination, it declares deadlock in step EP2.5. □

5.1 Deadlock Resolution

If the initiator i finds that it is deadlocked, it can use Z_i to locally construct the topology of the deadlocked portion of the WFG. It can then use various strategies to choose a desirable set of nodes to abort to resolve the deadlock [18]. The algorithm considerably facilitates efficient and fast resolution of a detected deadlock, whereas the other algorithms [6], [7], [21], [28] require an additional round of messages to collect the information that is needed to resolve the deadlock.

6 Performance

We analyze the time complexity, message complexity, size of messages, and computational complexity for an invocation of the proposed deadlock detection algorithm on a WFG (N, E). The parameters used are d, the diameter of the WFG, and $e = |E|$.

Theorem 2. *The algorithm terminates in $2d + 2$ message hops.*

Proof. The algorithm terminates when the initiator receives a reply along each outgoing edge. Assume that each message hop takes one time unit.

The FLOODs initiated by the initiator induce a spanning tree in the WFG. When a node receives its first FLOOD, it sends FLOODs immediately. Let $d_{max} \leq d$ be the maximum distance of any node from the initiator in the WFG. The latest time that a FLOOD is in transit is $d_{max} + 1$ message hops.

A node i in the spanning tree immediately replies to its parent when it has received an ECHO or PIP from each node in OUT_i. FLOODs sent to nodes in $OUT_i \setminus offspring_i$ are immediately responded to by ECHOs or PIPs. At $d_{max} + 2$ time units, each node i has received an ECHO/PIP from each node in $OUT_i \setminus offspring_i$. A node at distance d_{max} will have received all expected replies at time $d_{max} + 2$, and sends a reply to its parent at distance $d_{max} - 1$. By induction, a node at depth x receives all expected replies by time $d_{max} + 2 + d_{max} - x$. The initiator which is at distance 0 receives all expected replies by time $2\,d_{max} + 2$ and terminates. □

Observe that the initiator can detect it is not deadlocked in fewer message hops as soon as it gets locally reduced. In the best case, this is only two message hops.

Theorem 3. *An invocation of the algorithm uses $2e$ messages.*

Proof. A node sends exactly one FLOOD on each outgoing edge once (case F1-B). Thus, e FLOODs are sent for an invocation of the algorithm.

A FLOOD from i sent to a node in $OUT_i \setminus offspring_i$ is responded to by a ECHO or a PIP (cases F2 and F3). A FLOOD sent by i to a node in $offspring_i$, where $offspring_i$ is a leaf node in the WFG is responded to by an ECHO (case F1-A). A FLOOD sent by i to a node in $offspring_i$, where $offspring_i$ is a nonleaf node in the WFG is responded to by exactly one PIP or ECHO (step EP2.5 or step EP2.4). Thus, there are exactly e PIPs or ECHOs, and the message complexity is exactly $2e$. □

Observe that PIP and ECHO messages are of variable length, and may be larger than those in earlier algorithms [6], [7], [21], [28]. However, message headers are usually large, so a slightly larger message body should not pose a problem. We now analyze the size of PIPs and ECHOs. A node j cannot belong to any R_i if it does not send any PIPs before it is locally reduced. In the best case, R_i is $\emptyset$. Due to eager dissemination of reduced node information, R_i can contain any node in OUT_i^{+} that sent a PIP before local reduction or that was remotely reduced at some other node in OUT_i^{+}. Hence, in the worst case, R_i is the set of $|OUT_i^{+}|$ node identifiers. To analyze the size of Z_i, we use the P-out-of-Q representation of generalized requests; as reviewed in Section 1, the P-out-of-Q request model and the AND-OR request model have equivalent expressive power and the presented algorithm can be applied directly to both request models. A P-out-of-Q formula when translated to the AND-OR model can become exponentially large in Q. An AND-OR formula can always be translated into a P-out-of-Q formula, as shown in [19]. In the P-out-of-Q model, the unblocking function X_i at a node i requires the representation of $|OUT_i|$ node identifiers. In the best case, the residual function of i at local termination can be *true* and Z_i can be $\emptyset$. In the worst case, the local residual function can be X_i for node i and Z_i contains X_j for every j in OUT_i^{+}, thus requiring the representation of

$$\sum_{j \in OUT_i^{+} \cup \{i\}} |OUT_j|$$

node identifiers. The size of a PIP or ECHO is the sum of the estimates of the sizes of R_i and Z_i. In the best case, this is the representation of $\emptyset$; in the worst case, it is the representation of

$$|OUT_i^{+}| + \sum_{j \in OUT_i^{+} \cup \{i\}} |OUT_j|$$

node identifiers, with the added constraint that a node identifier cannot appear in both R_i and Z_i. Although no empirical data on the size of WFGs or deadlocks is available for the P-out-of-Q request model, for the single-request and AND request models, it is argued in [3], [13] that most deadlock cycles are of length 2 and WFGs are also relatively small. We expect that for the P-out-of-Q request model, the WFGs will not be significantly larger than those for the AND request model. Hence, even in the worst case, messages are expected to be small.

For the computational complexity at a node, we again consider the equivalent P-out-of-Q representation of unblocking functions. We need to determine the computational complexity of procedure *eval*, which is executed once by each node i when it receives its last ECHO or PIP. For each pass of the outer *repeat* loop, at least one new node is inserted in *tempR*. In the worst case, there will be t_i executions of the *repeat* loop, where t_i is the number of nodes in the subtree of the WFG rooted at i. Within each

repeat loop, at most t_i Zs are instantiated by the new members of $tempR$, and evaluated; this can be done in parallel in $O(1)$ steps, and serially in $O(t_i)$ steps. Hence, in the worst case, the processing at node i without any parallelization is $O(t_i^2)$. In the best case, it is $O(1)$ when $R_i = \emptyset$ or $Z_i = \emptyset$.

Table 1 compares the performance of the proposed algorithm and the algorithms in [6], [7], [21], [28] in terms of the number of phases, time complexity, and message complexity. The proposed algorithm performs better than the algorithms in [6], [7], [21], [28]; it has a message complexity of

$$(2 * \textit{the number of edges in the WFG})$$

and the worst-case time complexity of

$$(2 * \textit{the diameter of the WFG}) + 2$$

hops. Also, the proposed algorithm has information locally available at the initiator to determine how to resolve a detected deadlock; other algorithms incur extra time and message overhead to achieve this. In addition, even if the initiator is not deadlocked, Z_i at the initiator contains information on the residual function of each node in the WFG that could not be even remotely reduced.

We conjecture that the proposed algorithm is optimal in the number of messages and in time delay if detection of generalized deadlocks is to be carried out under the following framework:

- No node has complete knowledge of the topology of the WFG or the system.
- The deadlock detection is to be carried out in a distributed manner.

This framework is similar to that in [6], [21], [28]. The algorithm does not introduce the latent message overhead to acknowledge computation messages (in [7], every computation message sent has to be individually acknowledged; this greatly increases the message complexity) and does not have latent delays (as in [7], where the algorithm blocks until all previously sent computation messages are acknowledged).

The informal argument to support our conjecture of optimality is as follows: The only way to identify the WFG when the topology of neither the WFG nor the system is known is to use the diffusion of messages along the WFG edges (folklore). This takes $d+1$ time units and e messages. Due to the following two reasons, it is necessary that a node respond to every FLOOD message it receives:

1. If a node is in the indeterminate state when it receives a second or subsequent FLOOD message, it must immediately respond to it (e.g., by a PIP) to avoid deadlocking of the algorithm itself. Consider an example given in Fig. 5. The thick edges define the DST. If node 4 does not immediately respond to the FLOOD messages from nodes 3 and 5, and node 5 does not immediately respond to the FLOOD message from node 4, the algorithm is deadlocked.

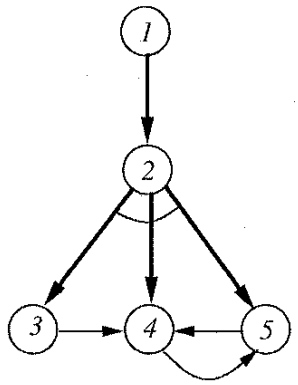

Fig. 5. An example detection.

2. If evaluation of the WFG topology for deadlock is to be conducted in a distributed manner, each node involved in the diffusion process must receive a response to the diffusion messages from each of its direct successor nodes so that it can decide about its own reducibility. In turn, it must send a response containing information about its reducibility status to all nodes from which it received a diffusion (FLOOD) message.

Thus, at least e return messages are required. Since return messages sequentially traverse the WFG, they take $d+1$ time units to reach the initiator node. Thus, an additional $d+1$ time units and at least e messages are required after the diffusion of messages is over. Hence, we conjecture that the time and the message complexity of $2d+2$ and $2e$, respectively, are optimal.

7 Discussion and Conclusions

Handling Dynamic WFGs. The algorithm was presented for a static WFG. In practice, nodes are making requests and requests are being satisfied; therefore, a WFG is dynamic. Consider an edge from i to j. By the time a FLOOD sent by i reaches j, j has already replied to i. This edge is a *phantom* edge. Treatment of phantom edges that arise due to the dynamic nature of the WFG is described in [21]. When a node j receives a FLOOD from a node i, j checks if this is a phantom edge, i.e., is $i \notin IN_j$? If so, then j immediately returns an ECHO to i. For a phantom edge from node i to node j, node i is defined to perceive $h(j)$ as zero (0). Node j may still be a part of the WFG, but $parent_j \neq i$ and $parent_j$ may perceive a nonzero value of $h(j)$. The message and the time complexities of the modified algorithm remain unchanged; however, phantom edges are appropriately handled in the reachable WFG during the diffusion of FLOOD messages.

Handling Concurrent Initiations. Due to the symmetric nature of the algorithm, multiple nodes may initiate the deadlock detection concurrently and a particular node may initiate it multiple times. Sequence numbers and initiator-ids distinguish between different instances of the algorithm. An optimization on the number of messages can be performed by maintaining a timestamp-based

priority order on all invocations of the algorithm and suppressing lower priority algorithms.

In summary, we presented an efficient algorithm for detecting deadlocks in replicated databases. Replicated databases offer increased fault-tolerance and better responsiveness but require quorum algorithms to serialize concurrent read and write operations from different transactions for concurrency control [1], [4], [5], [12]. Replicated databases that use quorum-consensus algorithms are prone to deadlocks because transactions which are waiting for a quorum to be satisfied may be involved in an indefinite wait. Requests in replicated databases are P-out-of-Q or AND-OR type requests and the resulting deadlocks are generalized deadlocks. The presented algorithm to detect generalized deadlocks is based on the principle of diffusion computation and performs reduction of a distributed WFG to detect a deadlock. Deadlock detection is performed by echoing the diffusion computation messages at terminating nodes and reducing the graph when an appropriate condition at a node in the echo phase is found. If sufficient information to decide the reducibility of a node is not available at that node, the algorithm optimizes the performance by attempting the reduction later in a lazy manner.

We proved the correctness of the algorithm. The algorithm detects all deadlocks in a finite time and if it reports a deadlock, the deadlock exists in the system. The algorithm performs considerably better than the existing algorithms to detect generalized deadlocks in distributed systems. It has a message complexity of $2e$ messages and the worst-case time complexity of $2d+2$ hops. We conjectured that the algorithm is optimal in time and message complexity to detect generalized deadlocks if no node has complete knowledge of the topology of the WFG or of the system and the deadlock detection is to be carried out in a distributed manner.

The presented algorithm is applicable to detecting deadlocks in other domains such as resource allocation in distributed operating systems, store-and-forward communication networks, and communicating processes, where generalized deadlocks occur, as well as to traditional domains where single request, AND request, and OR request deadlocks occur.

Acknowledgments

A preliminary version of this algorithm appears in "Distributed Detection of Generalized Deadlocks," by Ajay D. Kshemkalyani and Mukesh Singhal, presented at the 17th IEEE International Conference on Distributed Computing Systems (pages 545-553) that was held in May 1997.

References

[1] D. Barbara, H. Garcia-Molina, and A. Spauster, "Increasing Availability Under Mutual Exclusion Constraints with Dynamic Vote Reassignment," *ACM Trans. Computer Systems*, vol. 7, no. 4, pp. 394-426, Nov. 1989.

[2] C. Beeri and R. Obermarck, "A Resource Class Independent Deadlock Detection Algorithm," Technical Report No. RJ-3077, IBM Research Laboratory, San Jose, Calif., 1981.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] B. Bhargava and A. Helal, "Efficient Availability Mechanisms in Distributed Database Systems," *Proc. Int'l Conf. Information and Knowledge Management*, pp. 645-654, Nov. 1993.

[5] B. Bhargava and A. Helal, "Performance Evaluation of Quorum Consensus Replication Method," *Proc. Int'l Conf. Computer Performance and Dependability Symp.*, pp. 165-172, Apr. 1995.

[6] G. Bracha and S. Toueg, "Distributed Deadlock Detection," *Distributed Computing*, vol. 2, pp. 127-138, 1987.

[7] J. Brzezinski, J.M. Helary, M. Raynal, and M. Singhal, "Deadlock Models and Generalized Algorithm for Distributed Deadlock Detection," *J. Parallel and Distributed Computing*, vol. 31, no. 2, pp. 112-125, Dec. 1995.

[8] K.M. Chandy, J. Misra, and L.M. Haas, "Distributed Deadlock Detection," *ACM Trans. Computer Systems*, vol. 1, no. 2, pp. 144-156, 1983.

[9] E.W. Dijkstra and C.S. Scholten, "Termination Detection for Diffusing Computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1-4, Aug. 1980.

[10] D.K. Gifford, "Weighted Voting for Replicated Data," *Proc. Seventh ACM Symp. Operating Systems Principles*, pp. 150-162, Dec. 1979.

[11] V.G. Gligor and S.H. Shattuck, "On Deadlock Detection in Distributed Systems," *IEEE Trans. Software Eng.*, vol. 6, no. 5, pp. 435-440, 1980.

[12] K. Goldman and N. Lynch, "Quorum Consensus in Nested Transaction Systems," *ACM Trans. Database Systems*, vol. 19, no. 4, pp. 537-585, Dec. 1994.

[13] J.N. Gray, P. Homan, H.F. Korth, and R.L. Obermarck, "A Strawman Analysis of the Probability of Waiting and Deadlock in a Distributed System," Technical Report No. 3066, IBM Research Laboratory, San Jose, Calif., 1981.

[14] T. Herman and K.M. Chandy, "A Distributed Procedure to Detect AND/OR Deadlocks," Technical Report No. TR-LCS-8301, Univ. of Texas, Austin, Feb. 1983.

[15] G.S. Ho and C.V. Ramamoorthy, "Protocols for Deadlock Detection in Distributed Database Systems," *IEEE Trans. Software Eng.*, vol. 8, no. 6, pp. 554-557, Nov. 1982.

[16] C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, vol. 21, no. 8, pp. 666-677, Aug. 1978.

[17] M. Hofri, "On Timeout for Global Deadlock Detection in Decentralized Database Systems," *Information Processing Letters*, vol. 51, no. 6, pp. 295-302, 1994.

[18] R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, vol. 4, no. 3, pp. 179-195, 1972.

[19] E. Knapp, "Deadlock Detection in Distributed Databases," *ACM Computing Surveys*, vol. 19, no. 4, pp. 303-328, Dec. 1987.

[20] A.D. Kshemkalyani and M. Singhal, "Characterization and Correctness of Distributed Deadlock Detection and Resolution," *J. Parallel and Distributed Computing*, vol. 22, no. 1, pp. 44-59, July 1994.

[21] A.D. Kshemkalyani and M. Singhal, "Efficient Detection and Resolution of Generalized Distributed Deadlocks," *IEEE Trans. Software Eng.*, vol. 20, no. 1, pp. 43-55, Jan. 1994.

[22] A.D. Kshemkalyani and M. Singhal, "Invariant-Based Verification of a Distributed Deadlock Detection Algorithm," *IEEE Trans. Software Eng.*, vol. 17, no. 8, pp. 789-799, Aug. 1991.

[23] A.D. Kshemkalyani and M. Singhal, "Correct Two-Phase and One-Phase Deadlock Detection Algorithms for Distributed Systems," *Proc. Second IEEE Symp. Parallel and Distributed Processing*, pp. 126-129, Dec. 1990.

[24] D.A. Menasce and R.R. Muntz, "Locking and Deadlock Detection in Distributed Databases," *IEEE Trans. Software Eng.*, vol. 5, no. 3, pp. 195-202, 1979.

[25] R. Mukkamala, "Storage Efficient and Secure Replicated Distributed Databases," *IEEE Trans. Knowledge and Data Eng.*, vol. 6, no. 2, pp. 337-341, June 1994.

[26] M. Singhal, "Deadlock Detection in Distributed Systems," *Computer*, vol. 22, no. 11, pp. 37-48, Nov. 1989.

[27] G. Vossen and S.S. Wang, "Correct and Efficient Deadlock Detection and Resolution in Distributed Database Systems," *Proc. Fifth IEEE Int'l Conf. Data Eng.*, pp. 287-294, 1989.

[28] J. Wang, S. Huang, and N. Chen, "A Distributed Algorithm for Detecting Generalized Deadlocks," technical report, Dept. of Computer Science, National Tsing-Hua Univ., 1990.

[29] C. Yeung and S. Hung, "A New Deadlock Detection Algorithm for Distributed Real-Time Database Systems," *Proc. 14th IEEE Int'l Symp. Reliable Distributed Systems*, pp. 146-153, Sept. 1995.

Ajay D. Kshemkalyani received a BTech degree in computer science and engineering from the Indian Institute of Technology, Mumbai, in 1987; and a PhD degree in computer and information science from Ohio State University in 1991. He has been an assistant professor in the Electrical Engineering and Computer Science Department at the University of Illinois at Chicago since 1998. From 1991 to 1997, he worked at IBM Research Triangle Park in computer networks and distributed systems. His current research interests are in distributed computing, computer networking, and operating systems. He is a member of the ACM and a senior member of the IEEE.

Mukesh Singhal received a bachelor of engineering degree in electronics and communication engineering (with high distinction) from the University of Roorkee, Roorkee, India, in 1980; and a PhD degree in computer science from the University of Maryland, College Park, in May 1986. He is presently an associate professor in the Computer and Information Science Department at Ohio State University, Columbus. His current research interests include operating systems, database systems, distributed systems, mobile computing, high-speed networks, computer security, and performance modeling. He has published over 100 refereed articles in these areas. He has coauthored two books entitled *Readings in Distributed Computing Systems* (IEEE Computer Society Press, 1993) and *Advanced Concepts in Operating Systems* (McGraw-Hill, 1994). He is currently the program director of the Operating Systems and Compilers Program at the National Science Foundation.