Efficient Detection and Resolution of Generalized Distributed Deadlocks

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

***Abstract*—We present an efficient one-phase algorithm that consists of two concurrent sweeps of messages to detect generalized distributed deadlocks. In the outward sweep, the algorithm records a snapshot of a distributed wait-for-graph (WFG). In the inward sweep, the algorithm performs reduction of the recorded distributed WFG to check for a deadlock. The two sweeps can overlap in time at a process. We prove the correctness of the algorithm. The algorithm has a worst-case message complexity of $4e - 2n + 2l$ and a time complexity of $2d$ hops, where e is the number of edges, n is the number of nodes, l is the number of leaf nodes, and d is the diameter of the WFG. This is a notable improvement over the existing algorithms to detect generalized deadlocks.**

***Index Terms*—Distributed snapshot, graph reduction, distributed system, generalized deadlock.**

I. Introduction

DISTRIBUTED systems are prone to deadlocks—system states in which some processes wait indefinitely for each other for their requests to be satisfied. Detecting deadlocks is therefore an important problem in distributed systems. The wait-for relation between processes is modeled by a wait-for graph (WFG), which is a directed graph whose nodes are processes. An edge from node i to node j indicates that process i has requested a resource from process j and that process j has not granted the resource to process i. A system in which a process makes requests for Q resources and remains blocked until it is granted any P out of the Q resources is said to follow the P-out-of-Q request model [1]. The P-out-of-Q request model is also known as the generalized request model, because it is a generalization of the AND and OR request models [7]. The generalized request model also includes the AND–OR request model, in which the condition for a blocked process to get unblocked can be expressed as a disjunction of P-out-of-Q type requests [4]. Generalized deadlocks correspond to deadlocks in the P-out-of-Q request model.

We present an efficient algorithm for detecting generalized distributed deadlocks, and prove its correctness. Detecting generalized deadlocks [8] in a distributed system is a difficult problem [1], [7], [11] 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 processes 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. The algorithm presented is for the P-out-of-Q request model (a single disjunct) to simplify the presentation. In the conclusions, we point out how the algorithm can easily detect deadlocks in the AND–OR request model without increasing complexity.

Although several distributed deadlock detection algorithms have been proposed [1], [2], [4], [7], [10], [11], [12], [13], only the algorithms in [1], [13] are designed to detect generalized distributed deadlocks. The algorithm presented by Bracha and Toueg [1] consists of two phases. In the first phase, the algorithm records a snapshot of a distributed WFG, and, in the second phase, the algorithm simulates the granting of requests 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. Wang *et al.* [13] proposed a distributed two-phase algorithm for detecting generalized deadlocks in the distributed WFG. In the first phase, the algorithm records a snapshot of a distributed WFG. A termination detection technique is used to detect the end of the first phase, after which the second phase is initiated to reduce the static WFG recorded in the first phase to detect a deadlock.

The proposed algorithm consists of a single phase. The single phase comprises a diffusion of messages outward from an initiator process along the edges of the WFG (called the *outward sweep*), and then the echoing of diffusion messages inward to the initiator process (called the *inward sweep*). In the outward sweep, the algorithm records a consistent snapshot of a distributed WFG [3]. In the inward sweep, a *reduction procedure* that simulates unblocking of those processes whose requests can be granted is applied to the recorded distributed WFG to determine whether a deadlock exists. An edge in a WFG is *reduced* if the request it represents can be granted during the reduction process. A node in a WFG is *reduced* if P out of the Q requests on which it is blocked can be granted during the reduction process. After the reduction procedure has been applied to the recorded WFG, all of the processes that are not reduced are deadlocked.

Note that both the outward and the inward sweeps are carried out concurrently in the sense that the inward sweep can begin before the outward sweep is over. The algorithm deals with the complications introduced because the two sweeps can overlap in time at any node in the WFG; that is, the reduction of a process can begin before the state of all

Manuscript received July 1992; revised October 1993. Recommended by S. Lam.

A. D. Kshemkalyani was with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210. He is now with IBM Corp., Research Triangle Park, NC 27709.

M. Singhal is with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210.

IEEE Log Number 9214490.

TABLE I
PERFORMANCE COMPARISON, GIVEN A WFG, n = NUMBER OF ITS NODES, l = NUMBER OF ITS LEAF NODES, e = NUMBER OF ITS EDGES, d = ITS DIAMETER.

Criterion	Bracha- Toueg [1]	Wang et al. [13]	Proposed algorithm
Phases	2	2	1
Delay	$4d$	$3d+1$	$2d$
Messages	$4e$	$6e$	$4e-2n+2l$

WFG edges incident at that process have been recorded. The algorithm has a message complexity of $4e-2n+2l$ and a time complexity of $2d$ hops, where e is the number of edges, n is the number of nodes, l is the number of leaf nodes, and d is the diameter of the WFG. Table I compares the performance of the proposed algorithm and the algorithms in [1], [13]. The proposed algorithm performs better than the algorithms in [1], [13]. It is conjectured that the algorithm has the best time complexity that can be achieved by an algorithm that applies reduction to a distributed WFG to detect generalized distributed deadlocks.

The rest of the paper is organized as follows: In Section II, we discuss preliminaries, which include the system model and the problem definition/statement. In Section III, we present the algorithm. In Section IV, we prove the algorithm's correctness. In Section V, we analyze the performance of the algorithm. In Section VI, we briefly discuss how to extend the algorithm to handle deadlock resolution. Section VII contains concluding remarks.

II. PRELIMINARIES

A. System Model

The system has n nodes, and every two nodes are connected by a logical channel. There is no shared memory in the system. A variable or constant x local to process i is denoted by x_i. The nodes communicate by sending messages, which are delivered in the order sent on each logical channel. An event in a computation can be an internal event, a message send event, or a message receive event. Events are assigned time stamps using Lamport's logical clocks [9]. The clock at process i is t_i, which denotes the local time at process i.

Events in the system can be classified as computation events and control events. A *computation event* is any event that occurs as a result of the execution of an application process. A *control event* is any event that occurs as a result of the execution of the (system) deadlock detection algorithm. A *computation message* is any message that is sent because of the execution of an application process. A *control message* is any message that is sent because of the execution of the deadlock detection algorithm.

The computation messages that occur in the underlying computation include REQUEST, REPLY, and CANCEL messages. A node sends a REQUEST message to q other nodes when it blocks (goes from active to idle state) on a p-out-of-q request. A REPLY message denotes the granting of a request. The node unblocks (goes from idle to active state) when p out of its q requests are granted. When the node unblocks, it sends CANCEL messages to withdraw the remaining $q-p$ requests it had sent.

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

Axiom 1: A node blocks when it sends a p-out-of-q request, and then it 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 of the underlying 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 if its requests are satisfied as a result of the resolution of a deadlock of which it is a part [8]. We do not allow a node to unblock abnormally for simplicity.

A node i has the following local variables to keep track of its state:

1) t_i: **integer** $\leftarrow$ 0; /*current local time.*/
2) t_block_i: **integer** $\leftarrow$ 0; /*the time at which node i was last blocked.*
3) in_i: **set of integer** $\leftarrow \emptyset$; /*set of nodes whose requests are outstanding at node i.*/
4) out_i: **set of integer** $\leftarrow \emptyset$; /*set of nodes for which node i is waiting since t_block_i.*/
5) p_i: **integer** $\leftarrow$ 0; /*the number of replies required for unblocking.*/

The operations of the underlying computation that pertain to acquisition and release of resources follow. REQUEST and REPLY messages contain the requester's clock value at the time it blocked, so that a REPLY can be matched with its corresponding REQUEST. (The updating of the local clock and the assignment of time stamps to messages is not shown.) The semantics of "$a \longrightarrow b$" is *if a then b else skip.*

```
send REQUEST's
/*Executed by process $i$ when it blocks for a $p_i$-out-of-$q_i$ request. Parameters $p_i$ and $q_i$ depend on the application and the $q_i$ processes are identified by the application process and/or the operating system.*/
$p_i \leftarrow$ application-dependent value;
For each node $j$ of $q_i$ nodes on which $i$ blocks, do
    $out_i \leftarrow out_i \bigcup \{j\}$;
    send REQUEST($i$) to $j$;
$t\_block_i \leftarrow t_i$.

receive REQUEST($k$)
/*Executed by process $i$ when it receives a request made by process $k$.*/
$in_i \leftarrow in_i \bigcup \{k\}$.

send REPLY
/*Executed by process $i$ when it replies to a request by process $k$.*/
$in_i \leftarrow in_i - \{k\}$;
send REPLY($i$) to $k$.

receive REPLY($j$)
/*Executed by process $i$ when it receives a reply from process
```

```
$j$ to its corresponding request. A reply to an outdated request
can be identified by the timestamp of the outdated request on
the reply and is ignored.*/
$out_i \leftarrow out_i - \{j\}$;
$p_i \leftarrow p_i - 1$;
$p_i = 0 \longrightarrow$
    $t\_block_i \leftarrow 0$;
    for every $j$ in $out_i$, send CANCEL($i$) to $j$;
    $out_i \leftarrow \emptyset$.
receive CANCEL($k$)
/*Executed by process $i$ when it receives a cancel from process
$k$.*/
$in_i \leftarrow in_i - \{k\}$.
```

The only computation events that are relevant to deadlock detection are the events of send REQUEST, receive REQUEST, send REPLY, receive REPLY, send CANCEL, and receive CANCEL. The local state variables that they alter are t_block_i, in_i, out_i, and p_i. To detect deadlocks, we need to observe only the above-listed variables at processes in the WFG consistently. Thus, in a snapshot of the WFG, we are concerned only with the local state of a process (active/blocked, and how many requests from the ones that are outstanding need to be granted so that unblocking can occur), and with the sending and receiving of REQUEST, REPLY, and CANCEL messages.

Definition 1: A consistent snapshot [3] is a collection of local states of processes (defined by t_block_i, in_i, out_i, and p_i for process i) such that if the receipt of a REQUEST, REPLY, or CANCEL message is recorded in a local state, the sending of the message is also recorded in the local state at the process which sent the message.

Thus, every receive event need not be recorded, but if a receive event is recorded, the corresponding send event must also be recorded.

B. Problem Statement

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

Definition 2: A generalized deadlock is a graph (D, K) where D is a nonempty set of nodes that are blocked on p-out-of-q requests in a WFG, and K is the set of WFG edges between nodes in D, such that $\forall i \in D$, at least $q_i - p_i + 1$ outgoing WFG edges are incident on other nodes in D.

From Axiom 1 and 2, it follows that none of the nodes in D gets unblocked. All nodes in D thus remain blocked forever. After all of the grantable requests of nodes in D have been granted, each WFG consisting of a minimal subset of D that satisfies the condition for a generalized deadlock forms a knot topologically. All of the nodes in the WFG that do not belong to any D have at least p_i edges to nodes that do not belong to any D. All of these nodes are not deadlocked, because their requests can be satisfied.

C. Correctness Conditions

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

1) If a deadlock exists, it is detected by the algorithm within a finite time after the deadlock has formed.
2) If a deadlock is declared, the deadlock exists in the system.

At the time that a node blocks, it initiates a deadlock detection algorithm. Note that only the processes that are reachable from a process in the WFG can be involved in deadlock with that process. Therefore, to detect whether a process is deadlocked, we need to examine only that part of the WFG that is reachable from that process.

In the outward sweep of the algorithm, a snapshot of only those processes that can be reached from the initiator in the WFG is taken. Also, at a node in the WFG, only those wait-for edges that can be traversed starting from the initiator are recorded. Thus, a snapshot of complete WFG of the entire system is not taken. Only that part of the WFG that is obtained by projecting the WFG over the reachability set of the initiator is of concern.[1]

A correctness proof of the algorithm must include the following two points to show that the above two correctness criteria are satisfied. First, it must be shown that the entire WFG reachable from the initiator is recorded by the algorithm consistently. Second, it must be shown that the reduction of the recorded WFG snapshot is performed correctly.

III. A Distributed Deadlock Detection Algorithm

When a node i blocks on a P-out-of-Q request, it initiates the deadlock detection algorithm. The algorithm records that part of the WFG that is reachable from i (hereinafter referred to as i's WFG) in a distributed snapshot [3]. Such a distributed snapshot includes only those wait-for edges and nodes that form i's WFG. Since multiple nodes may block concurrently, they may each initiate the deadlock detection algorithm concurrently. Each invocation of the deadlock detection algorithm is treated independently and is identified by the initiator's identity and the local time at which the initiator blocked. Every node maintains a local snapshot for the latest deadlock detection algorithm initiated by every other node. We will focus on a single instance of the deadlock detection algorithm.

Next, we discuss the basic idea behind the algorithm with the help of an example (shown in Figs. 1 and 2). Fig. 1 shows initiation of deadlock detection by node A and Fig. 2 shows the state after node D is reduced. The notation x/y beside a node in the figures indicates that the node is blocked and needs replies to x out of the y outstanding requests so that it can unblock.

A. Basic Idea

Distributed WFG is recorded using FLOOD messages during the outward sweep and is examined for deadlocks using ECHO messages during the inward sweep. The initiator records its local state and sends FLOOD messages along its outgoing wait-for edges at the time it blocks.

At the time a node i receives the first FLOOD message along an existing incoming wait-for edge, it records its local

[1] The *reachable set* of a node is the set of nodes that can be reached from it in the WFG.

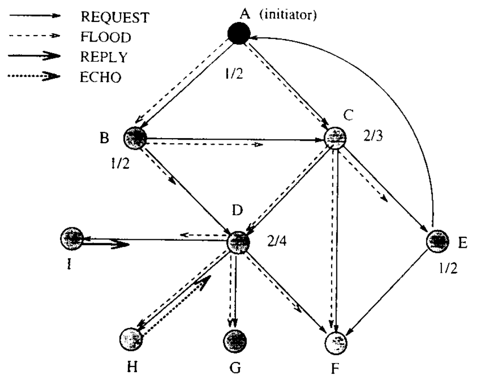


Fig. 1. An example-run of the algorithm.

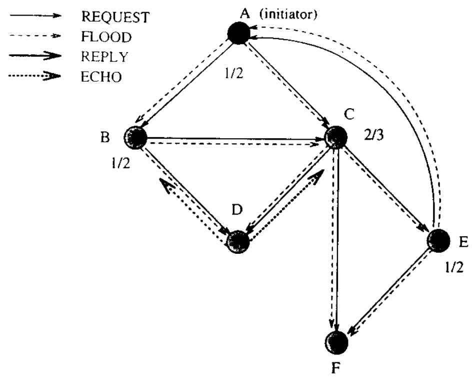


Fig. 2. An example-run of the algorithm (continued from Fig. 1).

state (out_i, p_i, t_block_i, and this particular incoming wait-for edge). If the node happens to be blocked at this time, it sends FLOOD's along its outgoing wait-for edges to ensure that all nodes in the reachability set of the initiator participate in recording the WFG in the outward sweep. For example, in Fig. 1, when node C receives FLOOD from node A, it sends FLOOD's to nodes D, E, and F. If the node happens to be active at this time, (i.e., it does not have any outgoing wait-for edges), then it initiates reduction of the incoming wait-for edge by returning an ECHO message on it. For example, in Fig. 1, node H returns an ECHO to node D in response to a FLOOD from it. Note that such an active node can initiate reduction (by sending back an ECHO in response to a FLOOD along an incoming wait-for edge) even before the states of all other incoming wait-for edges have been recorded in the WFG snapshot at that node. For example, node F in Fig. 1 starts reduction after receiving a FLOOD from C even before it has received FLOOD's from D and E.

At the time a node receives a FLOOD, it need not have an incoming wait-for edge from the node that sent the FLOOD, because it may have already sent back a REPLY to the node. In this case, the node returns an ECHO in response to the FLOOD. For example, in Fig. 1, when node I receives a FLOOD from node D, it returns an ECHO to node D.

ECHO messages perform reduction of the nodes and edges in the WFG by simulating the granting of requests in the inward sweep. A node gets reduced at the time it has received p ECHO's. When a node is reduced, it sends ECHO's along all the incoming wait-for edges incident on it in the WFG snapshot to continue the progress of the inward sweep. These ECHO's in turn may reduce other nodes. The initiator node detects a deadlock if it is not reduced when the deadlock detection algorithm terminates. The nodes in the WFG snapshot that have not been reduced are deadlocked. The order in which reduction of the nodes and edges of the WFG is performed does not alter the final result.

In general, WFG reduction can begin at a nonleaf node before recording of the WFG has been completed at that node. This happens when ECHO's arrive and begin reduction at a nonleaf node before FLOOD's have arrived along all incoming wait-for edges and recorded the complete local WFG at that node. For example, node D in Fig. 1 starts reduction (by sending an ECHO to node C) after it receives ECHO's from H and G, even before FLOOD from B has arrived at D. When a FLOOD on an incoming wait-for edge arrives at a node that is already reduced, the node simply returns an ECHO along that wait-for edge. For example, in Fig. 2, when a FLOOD from node B arrives at node D, node D returns an ECHO to B. Thus, the two activities of recording the WFG snapshot and reducing the nodes and edges in the WFG snapshot are done concurrently in a single phase, and no serialization is imposed between the two activities, as is done in [13]. Since reduction is applied to an incompletely recorded WFG at nodes, the local snapshot at each node has to be judiciously manipulated to give the effect that reduction is initiated after WFG recording has been completed. The reduction operations at any node can be permuted with the WFG recording operations that follow them at that node, without affecting the result.

B. Termination Detection

A termination detection technique based on weights [6], [13] detects the termination of the algorithm by using SHORT messages (in addition to FLOOD's and ECHO's). A weight of 1 at the initiator node, when the algorithm is initiated, is distributed among all FLOOD messages sent out by the initiator. When the first FLOOD is received at a nonleaf node along an existing WFG edge, the weight of the received FLOOD is distributed among the FLOOD's sent along outgoing wait-for edges at that node. Weights in all subsequent FLOOD's arriving along existing WFG edges at a nonleaf node that is not yet reduced are returned to the initiator through SHORT messages. For example, in Fig. 1, node C receives a FLOOD from node A followed by a FLOOD from node B. When node C receives a FLOOD from B, it sends a SHORT to the initiator node A.

When a FLOOD is received at a leaf node, its weight is returned in the ECHO message sent by the leaf node to the sender of the FLOOD. Note that an ECHO is like a reply in the simulated unblocking of processes. When an ECHO arriving at a node does not reduce the node, its weight is sent directly to the initiator through a SHORT message. For example, in Fig. 1, when node D receives an ECHO from node H, it sends a SHORT to the initiator node A. When an

ECHO that arrives at a node reduces that node, the weight of the ECHO is distributed among the ECHO's that are sent by that node along the incoming edges in its WFG snapshot. For example, in Fig. 2, at the time node C gets reduced (after receiving ECHO's from nodes D and F), it sends ECHO's to nodes A and B. When an ECHO arrives at a reduced node, its weight is sent directly to the initiator through a SHORT message. For example, in Fig. 2, when an ECHO from node E arrives at node C after node C has been reduced (by receiving ECHO's from nodes D and F), node C sends a SHORT to initiator node A. The algorithm maintains an invariant that the sum of the weights in FLOOD, ECHO, and SHORT messages plus the weight at the initiator (received in SHORT and ECHO messages) is always 1. The algorithm terminates when the weight at the initiator becomes 1, signifying that all WFG recording and reduction activity has completed.

C. The Algorithm

FLOOD, ECHO, and SHORT control messages use weights introduced in [6] for termination detection. The weight in a message is a real number from 0 to 1. An event at which a FLOOD, ECHO, or SHORT message is sent or received is a control event.

A node i stores the local snapshot to detect deadlocks in a data structure LS_i, which is an array of records. Record $LS_i[init]$ stores a snapshot at node i corresponding to deadlock detection initiation by the initiator node $init$. (See Snapshot Data Structure at the bottom of this page.)

In addition, the initiator has a variable in which it collects returned weights.

```
$wt_{init}$: real $\leftarrow$ 1.0;      /*weight to detect termination of
                                      deadlock detection algorithm.*/
```

The other variables at a node have been defined in Section II.A. The procedures in the algorithm (next page) define the deadlock detection algorithm and are executed atomically. For the sake of notational convenience, the initiator's identity is dropped when referring to LS_i; that is, $LS_i[init]$ is denoted by LS_i.

IV. Correctness Proof

Different instances of the algorithm can be distinguished from each other by using time stamps, and various instances of the algorithm do not interfere with each other. Therefore, correctness proof is given for a single instance of the algorithm. We prove that the algorithm satisfies the correctness conditions given in Section II.B by showing the following results:

1) The execution of the algorithm terminates.
2) The entire WFG reachable from the initiator is recorded in a consistent distributed snapshot in the outward sweep.
3) In the inward sweep, ECHO's correctly reduce the recorded snapshot of the WFG.

The algorithm is initiated within a timeout period when a node blocks. By theorem 2, only all nodes that are not reduced on the termination of the algorithm are deadlocked. Thus, the correctness conditions of Section II.B are satisfied.

A. Algorithm Termination

Let $w(TYPE)$ represent the weight of a message of type $TYPE$. The following *weight invariant* is satisfied by the latest algorithm instance initiated by the initiator $init$:

Invariant 1:

$$\sum_{\forall FLOOD} w(FLOOD) + \sum_{\forall ECHO} w(ECHO) + \sum_{\forall SHORT} w(SHORT) + wt_{init} = 1$$

A message (and its weight) is discarded only if the message belongs to an outdated instance of the algorithm.

Definition 3: The algorithm is said to be "terminated" when $wt_{init} = 1$.

Observation 1: A weight in a FLOOD, ECHO, or SHORT message is always in transit until it reaches the initiator node $init$ and is added to wt_{init}.

Observation 2: Table II illustrates how a node i reacts when it receives a FLOOD from node k.

Corollary 1: At most one FLOOD is sent on a WFG edge.

Proof: Follows from observation 2. Note that only the initiator initiates the FLOOD phase, and other nodes send FLOOD's on outgoing edges only at the time that they receive the first FLOOD (and Case F1–A in Table II holds). Any other FLOOD received is sent as a SHORT or ECHO, because the local snapshot recording is not initiated by such a FLOOD. Moreover, the receipt of an ECHO never causes the sending of a FLOOD. □

Observation 3: A node j is recorded in $LS_i.in$ only at the time that a FLOOD is received by node i from that node and $j \in in_i$ (Case F1 or F4). From corollary 1, this happens at most once.

Observation 4: After $LS_i.p$ has been recorded, it is monotone nonincreasing.

Observation 5: At the time node i receives an ECHO from node j, if $LS_i.p$ is decremented to 0 (Case E2–A) (which can happen at most once by observation 4), then an ECHO is sent to all nodes in $LS_i.in$; otherwise (Cases E1, E2–B), the weight in the ECHO is sent to $init$ in a SHORT.

Observation 6: Node i sends an ECHO to node j only at the times (i) Case (F1–B), (F4-A), (F2), or (F3) hold at the time a FLOOD is received from j, or (ii) Case (E2–A) holds and $j \in LS_i.in$, at the time some ECHO is received.

Snapshot Data Structure

```
$LS_i$ : array [1..N] of record
    $out$: set of integer $\leftarrow \emptyset$;    /*nodes for which node $i$ is waiting in the snapshot*/
    $in$: set of integer $\leftarrow \emptyset$;     /*nodes waiting for node $i$ in the snapshot*/
    $t$: integer $\leftarrow$ 0;             /*time when $init$ initiated snapshot.*/
    $p$: integer $\leftarrow$ 0.             /*value of $p$ as seen in snapshot.*/
```

Algorithm

```
initiate a SNAPSHOT
/*Executed by process $i$ to detect whether it is deadlocked. */
$init \leftarrow i$;
$wt_{init} \leftarrow 0$;                                    /*initial weight.*/
$LS_i.t \leftarrow t_i$;
$LS_i.out \leftarrow out_i$;
$LS_i.in \leftarrow \emptyset$;
$LS_i.p \leftarrow p_i$;
send $FLOOD(i, i, t_i, 1/|out_i|)$ to each $j$ in $out_i$.

receive FLOOD($k$, $init$, $t\_init$, $w$)
/*Executed by process $i$ on receiving a FLOOD message from $k$.*/
[
/*Valid FLOOD for a new snapshot. */                        /*Case F1*/
$LS_i.t < t\_init \bigwedge k \in in_i \longrightarrow$
    $LS_i.out \leftarrow out_i$;
    $LS_i.in \leftarrow \{k\}$;
    $LS_i.t \leftarrow t\_init$;
    $LS_i.p \leftarrow p_i$;
    $p_i > 0 \longrightarrow$                                /* $i$ is blocked. Case F1-A*/
        send FLOOD($i, init, t\_init, w/|out_i|$) to each $j \in out_i$;
    $p_i = 0 \longrightarrow$                                /* $i$ is unblocked. Case F1-B*/
        send $ECHO(i, init, t\_init, w)$ to $k$;
□
/* FLOOD for a new snapshot is received
along an edge that has ceased to exist. A reply
has already been sent back to the sender of the FLOOD.*/     /* Case F2*/
$LS_i.t < t\_init \bigwedge k \notin in_i \longrightarrow$
    send $ECHO(i, init, t\_init, w)$ to $k$.
□
/* FLOOD for a current snapshot is received along
an edge that has ceased to exist. A reply has already been sent back to the
sender of the FLOOD.*/
                                                             /* Case F3 */
$LS_i.t = t\_init \bigwedge k \notin in_i \longrightarrow$
    send $ECHO(i, init, t\_init, w)$ to $k$.
□
/*Valid FLOOD for current snapshot.*/                        /*Case F4*/
$LS_i.t = t\_init \bigwedge k \in in_i \longrightarrow$
    $LS_i.in \leftarrow LS_i.in \bigcup \{k\}$;
    $LS_i.p = 0 \longrightarrow$                             /* $i$ is unblocked in snapshot. Case F4-A */
        send $ECHO(i, init, t\_init, w)$ to $k$;
    $LS_i.p > 0 \longrightarrow$                             /* $i$ is blocked in snapshot. Case F4-B*/
        send $SHORT(init, t\_init, w)$ to $init$.
□
/*Outdated FLOOD. */
                                                             /* Case F5 */
$LS_i.t > t\_init \longrightarrow$ discard the flood message.
]

receive ECHO($j$, $init$, $t\_init$, $w$)
/*Executed by process $i$ on receiving an ECHO from $j$ for a current snapshot
for which $LS_i.t = t\_init$. ECHO for an outdated snapshot ($LS_i.t > t\_init$)
is discarded. */
$LS_i.t = t\_init \longrightarrow$
    $LS_i.out \leftarrow LS_i.out - \{j\}$;
    $LS_i.p = 0 \longrightarrow$                             /* $i$ is unblocked in snapshot. Case E1 */
```

Algorithm (***continued***)

```
        send $SHORT(init, t\_init, w)$ to $init$.
    $LS_i.p > 0 \longrightarrow$                                  /* $i$ is blocked in snapshot. Case E2 */
      $LS_i.p \leftarrow LS_i.p - 1$;
      $LS_i.p = 0 \longrightarrow$                        /* $i$ becomes unblocked in snapshot. Case E2-A */
        $init = i \longrightarrow$ declare not deadlocked; exit.
        send $ECHO(i, init, t\_init, w/|LS_i.in|)$ to all $k \in LS_i.in$;
      $LS_i.p \neq 0 \longrightarrow$                       /* $i$ remains blocked in snapshot. Case E2-B */
        send $SHORT(init, t\_init, w)$ to $init$.

receive SHORT$(init, t\_init, w)$
/*Executed by process $i$ (which is always $init$) on receiving a
SHORT for which $t\_init = t\_block_i$. SHORT for an outdated
snapshot ($t\_init < t\_block_i$) is discarded.*/
$t\_init = t\_block_i \bigwedge LS_i.p = 0 \longrightarrow$
        discard the message.
$t\_init = t\_block_i \bigwedge LS_i.p > 0 \longrightarrow$
    $wt_{init} \leftarrow wt_{init} + w$;
    $wt_{init} = 1 \longrightarrow$ declare deadlock and abort.
```

TABLE II
ACTIONS AT NODE i **at the time** IT RECEIVES A FLOOD FROM NODE k.

$k \in in_i$?	Has snapshot recording already begun at node i ?	Action at node i at the time it receives a FLOOD from node k.	
Yes	No	**if** $p_i > 0$ **then** record $LS_i.out$, record $LS_i.p$, add k to $LS_i.in$, and send FLOOD's to nodes in $LS_i.out$.	**(Case F1-A)**
		if $p_i = 0$ **then** record $LS_i.p$ as 0, add k to $LS_i.in$, record $LS_i.out$ as { }, and return an ECHO to k.	**(Case F1-B)**
	Yes	**if** $LS_i.p > 0$ **then** add k to $LS_i.in$ and send a SHORT to $init$.	**(Case F4-B)**
		if $LS_i.p = 0$ **then** add k to $LS_i.in$ and return an ECHO to k.	**(Case F4-A)**
No	does not matter	return an ECHO to k.	**(Cases F2,F3)**

Corollary 2: At most one ECHO is sent on an edge recorded in the WFG.

Proof: At the time node i receives a FLOOD from node j (which can happen at most once according to corollary 1), one of the following cases arises:

1) If Case (F2) or (F3) holds at node i, an ECHO is returned to j. An ECHO was not sent to j before, because $j \notin LS_i.in$ before. Node j's identifier is not inserted in $LS_i.in$ now or later (observation 3), and another ECHO is never sent to j, because $j \notin LS_i.in$ henceforth.
2) If Case (F1–B) or (F4–A) holds at node i, an ECHO is returned to j and j's identifier is inserted in $LS_i.in$ at this time. An ECHO was not sent before, because $j \notin LS_i.in$ before (observation 6). By observation 4 and due to the fact that $LS_i.p$ is already 0, it follows that Case E2–A will not be executed later. Therefore, from observation 6, it follows that an ECHO will not be sent to j again.
3) If Case (F1–A) or (F4–B) holds at node i, an ECHO is not sent to j and node j's identifier is added to $LS_i.in$. From observation 5, an ECHO is sent to each node in $LS_i.in$ only at the time $LS_i.p$ is decremented to zero on receipt of an ECHO. The prior receipt of an ECHO could not have caused an ECHO to be sent to j because the nonincreasing $LS_i.p$ is > 0 at this time. At a later time, $LS_i.p$ may decrement to zero and cause an ECHO to be sent. This can happen at most once by observation 4.

Hence, at most one ECHO is sent on an edge recorded in the WFG. □

A message hop denotes one logical transfer of a message from the sender node to the receiver node. We use the number of hops that occur serially as a measure of time.

Lemma 1: The algorithm terminates in a finite number of message hops, bounded by $2e$, where e is the number of edges in the WFG.

Proof: The algorithm terminates when wt_{init} becomes one at the initiator process, at which time there are no FLOOD's, ECHO's, or SHORT's in transit for that deadlock detection initiation. At most one FLOOD is sent on a WFG edge (corollary 1), and at most one ECHO is sent on a WFG edge (corollary 2). Weights in control messages are always in transit (observation 1), and a weight in a SHORT message, which takes a single hop, is never transferred to a FLOOD or ECHO message. It follows that the algorithm terminates in at most $2e$ message hops. □

It should be noted that $2e$ message hops is a very loose upper bound because many control messages may be in transit concurrently.

B. Recording Reachability Set in the Snapshot

Theorem 1: The algorithm records a consistent snapshot of a distributed WFG of the initiator's reachability set.

Proof: Let $init$ be the initiator node. It needs to be shown that **[Completeness]** all of the nodes reachable from node i in the WFG record their local state, and **[Consistency]** if the receipt of a message is recorded in a local state, the send of that message is also recorded in the local state of the node that sent the message (definition 1).

Completeness follows from the following:

1) The initiator records its state $LS_{init}.out$ and $LS_{init}.p$ and sends FLOOD's along its outward WFG edges. No other node initiates FLOOD's.
2) A node i records $LS_i.out$ and $LS_i.p$ *only* at the time at which it receives the first FLOOD from some j and $j \in in_i$ (Case F1). Because of ordered message delivery on the channels, the WFG edge from j to i exists at node i iff $j \in in_i$. At this time, if i is blocked, i.e., node i has outgoing edges in the WFG, it propagates FLOOD's along its outward edges (Case F1-A). Therefore, only all nodes in the initiator's reachability set record their local states.
3) Whenever node i receives a FLOOD from node j, j is not recorded in $LS_i.in$ if $j \notin in_i$ to reflect the fact that the WFG edge from j to i does not exist (Cases F2 and F3). Because of the ordered message delivery on channels, the WFG edge from j to i exists at node i iff $j \in in_i$. To undo the recording of i in $LS_j.out$, i sends an ECHO to j. Thus, if an edge does not belong to the initiator's WFG, it is not recorded in its snapshot.

Note that if node i is active at the time it receives a FLOOD, it does not propagate FLOOD's. If node i blocks sometime later, then $init$'s WFG is extended. However, the distributed snapshot that is recorded for $init$ shows i as active and does not consist of the portion of the WFG extended because of the blocking of node i. If a deadlock results because node i blocks, the deadlock will be detected by the instance of the algorithm initiated by node i when it blocks.

Consistency follows from the following:

1) Node j is recorded in $LS_i.in$ only at *all* times that a FLOOD is received from j and $j \in in_i$ (Cases F1 and F4). This reflects the fact that the WFG edge from j to i indeed exists. (Because of the ordered delivery of messages, the WFG edge from j to i exists at node i iff $j \in in_i$.) By observation 2, j must have recorded i in $LS_j.out$ when it sent a FLOOD to i (Case F1-A). Therefore, the local states recorded collectively form a consistent snapshot of the initiator's WFG. (That is, the causes of all of the effects are recorded.) □

C. Reduction of the Recorded WFG Snapshot

Lemma 2: If a node i sends an ECHO to node j, then node i must have received a FLOOD from node j.

Proof: Node i sends an ECHO to node j only in one of the following two situations:

1) On the receipt of a FLOOD from j if Case F1-B, F2, F3, or F4-A in the algorithm holds; or
2) On the receipt of an ECHO if $LS_i.p$ decrements to 0 and $j \in LS_i.in$ at this time. By observation 3, j is added to $LS_i.in$ only if a FLOOD has been received from j as per Case F1 or F4. (Note that at the instant such a FLOOD was received from j, only Case F1-A or F4-B could have been executed, because $LS_i.p$ was greater than 0.)

In both of the above situations, i must have received a FLOOD from j. Hence, the lemma. □

Observation 7: If node i receives an ECHO from node j, the local snapshot recording at node i has already begun, node i has already sent a FLOOD to node j, and $j \in LS_i.out$ (*follows from lemma 2 and observation 2*).

Before proving that the reduction is performed correctly, we define a function h on the nodes in the WFG as follows:

$$h(i) \longleftarrow \begin{cases} 0 & \text{if } i \text{ is a leaf node in the WFG} \\ \infty & \text{if } i \text{ is deadlocked} \\ 1 + \left(p_i{}^{th} \text{ smallest of } \{h(j) \mid j \in out_i\}\right) & \text{otherwise.} \end{cases}$$

A node i blocked on a p_i-out-of-q_i request needs to receive p_i to unblock. $h(i)$ indicates the length of the $p_i{}^{th}$ shortest path traversed by all of the REPLY's to reach i. If node i were to get unblocked by receiving REPLY's, at least one of them would have to travserse a path of length $h(i)$. However, this would not preclude node i from getting unblocked by receiving a REPLY that had traversed a path of length greater than $h(i)$, because more than p_i number of nodes may send REPLY's.

A node that is not deadlocked has a finite value of h, because it has at least p_i edges to other nodes that are not deadlocked and there is a sequence of REPLY's by which the node can get unblocked. If a node is deadlocked, it is assigned a value of infinity for h because there is no sequence of REPLY's by which the node can get unblocked. The length of the shortest path traversed by a series of REPLY's to unblock a deadlocked node is ∞.

The above definition of h needs three modifications before it is applicable to a dynamically recorded WFG.

1) $LS_i.p$ and $LS_i.out$ should be used instead of p_i and out_i in the definition.
2) Because of the dynamic recording of the WFG, a node j may receive a FLOOD from node i after node j has sent a REPLY to node i. When node j receives the FLOOD from node i in this situation, $i \notin in(j)$ (Cases F2, F3) and j returns an ECHO to i. The edge from i to j does not exist, but got recorded at i because of finite message delays. Such an edge is termed a *phantom* edge. In such situations, node i is defined to perceive $h(j)$ as 0, because the edge from i to j is a phantom edge and

node j is perceived active by node i. Note that node j may not be a part of the WFG; but if it is, it must be through some other node k, and k may see a nonzero value of $h(j)$.

3) A leaf node in the WFG may send FLOOD's along phantom edges. $h(i)$ is defined to be 0 only if it first records its state as active and does not send any FLOOD's, i.e., executes Case F1-B.

The modified definition of h based on the dynamic recording appears at the bottom of the next page.

We now show that the algorithm performs reduction correctly. A node in the recorded WFG gets reduced iff it receives a sufficient number of ECHO's. A node in the WFG performs reduction iff it gets reduced and then sends an ECHO only in response to every FLOOD it receives.

Lemma 3: A node i for which $h(i) < \infty$ performs reduction.

Proof: We prove that reduction of the recorded WFG (as per the statement of theorem 1) is performed correctly by nodes whose h is finite by using induction on $h(i)$.

$\underline{\text{Base Case } h(i) = 0:}$ It will be shown that a node i for which $h(i) = 0$ performs reduction correctly. When such a node i recieves the first FLOOD, which is from node j and $j \in in_i$ at the time, node i executes Case F1-B by initiating its local snapshot recording and recording itself as "active" ($LS_i.p = 0$). Node i is reduced because it has received a sufficient number of ECHO's, which is 0 in this case. Such a node returns an ECHO for every FLOOD that it receives (Cases F1-B, F4-A, F2, and F3). Recall that Cases F2 and F3 correspond to a phantom edge. Cases F1-A, F4-B, E1, and E2 do not occur at this node. From lemma 2, it follows that if an ECHO is sent, it is only in response to a FLOOD. Therefore, a node for which $h(i) = 0$ sends an ECHO only in response to every FLOOD it receives, and thus performs reduction correctly.

$\underline{h(i) = x > 0:}$ Assume that a node i with $h(i) = x$ performs reduction correctly.

$\underline{h(i) = x + 1:}$ It needs to be shown that a node i with $h(i) = x + 1$ performs reduction correctly. At the time node i receives the first FLOOD, which is from node j and $j \in in_i$, node i executes Case F1-A and records $LS_i.p > 0$ because it is blocked. Each node k of at least p_i number of nodes in $LS_i.out$ 1) has $h(k) \leq x$, by definition, and performs reduction correctly by the induction assumption, or 2) returns an ECHO along a phantom edge by executing Case F2 or F3. Since an ECHO is sent at most once on an incoming WFG edge (corollary 2), node i will receive exactly one ECHO from each of these nodes in $LS_i.out$. Node i gets reduced when it receives p_i ECHO's and the monotone nonincreasing function $LS_i.p$ becomes 0. It needs to be shown that node i sends an ECHO only to each node from which it receives a FLOOD.

Node i returns an ECHO to node j when it receives a FLOOD from node j and $j \notin in_i$ (Cases F2 and F3), regardless of whether node i is reduced. It remains to be shown that node i sends an ECHO only to each node from which it receives a FLOOD and executes Cases F1 and F4. If a FLOOD is received before the node gets reduced, Case F4-B or F1-A holds and no ECHO is returned; instead, the identity of the sender of the FLOOD is added to $LS_i.in$, so that when node i gets reduced later (Case E2-A), it can send an ECHO to the sender of the FLOOD. At the time i receives the $p_i{}^{th}$ ECHO, it executes Case E2-A, gets reduced, and sends an ECHO to each node in $LS_i.in$. If a FLOOD is received after the node gets reduced, Case F4-A holds and an ECHO is returned. Therefore, when node i gets reduced, it sends ECHO's to all of the nodes from which it has received FLOOD's and returns ECHO's to all nodes from which it subsequently receives FLOOD's. Hence, a node whose $h = x + 1$ performs reduction correctly; i.e., it sends ECHO's in response to all FLOOD's, and only in response to all FLOOD's. Hence, the lemma. □

Lemma 4: 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. All of these nodes are recorded in the snapshot of the WFG (theorem 1). For any node $i \in D$, i has at least $q_i - p_i + 1$ WFG edges to nodes in D, which belong to the set K. None of the edges in K are phantom edges. At the time that node i receives the first FLOOD, which is from node j and $j \in in_i$, node i records $LS_i.p = p_i (> 0)$ and propagates FLOOD on its q_i outward edges without returning an ECHO (Case F1-A).

The earliest that node i in D can send an ECHO to another node in D along an edge in K is when it receives p_i ECHO's, $LS_i.p$ becomes 0, and i gets reduced (Case E2-A) because of observation 6 and the following three facts:

1) Node i never executes Cases F1-B;
2) Node i will not send an ECHO by Case F4-A until after it gets reduced;
3) Node i does not send an ECHO to another node in D along an edge in K by Cases F2 and F3, because the edges in K exist; i.e., the edges in K are not phantom edges.

From observation 7 and corollary 2, a node may receive at most one ECHO only on an outgoing WFG edge. A node i in D may receive at most $p_i - 1$ ECHO's from 1) nodes outside D (whose h is finite), possibly along phantom edges, and 2) from nodes in D (whose h is ∞) along phantom edges that do not belong to K. In both 1) and 2), the ECHO's are received along edges that do not belong to K. A node i in D does

$$h(i) \longleftarrow \begin{cases} 0 & \text{if } i \text{ executes Case F1-B} \\ \infty & \text{if } i \text{ is deadlocked} \\ 1 + \left(LS_i.p^{th} \text{ smallest of } \{h(j) \mid j \in LS_i.out\}\right) & \text{otherwise /* Note that } h(j) \text{ is read as 0 if } (i, j) \\ & \text{is a phantom edge */} \end{cases}$$

not get reduced and does not send an ECHO to another node in D along an edge in K at least until it receives an ECHO from another node in D along an edge in K. Consequently, no node in D receives an ECHO from another node in D along an edge in K. All such nodes are not reduced when the algorithm terminates. □

Theorem 2: Reduction of the recorded snapshot is performed correctly.

Proof: Follows from lemmas 3 and 4. The order of reduction of nodes is unpredictable because of unpredictable message delays. From Holt's result [5], however, the nodes can be reduced in any order without changing the final outcome. □

V. Performance

The proposed algorithm has a better message and time complexity than existing algorithms for detecting generalized distributed deadlocks. Consider a system of n nodes with e edges in the WFG and a diameter of d for the WFG. In comparing message complexity, we 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, and that is not an issue here.

The Herman–Chandy algorithm for the specialized AND–OR request model [4] has a message complexity of n messages per edge of the WFG and a time complexity of $2d$ hops. The Bracha–Toueg algorithm [1] has a message complexity of four messages per edge of the WFG and a time complexity of $4d$ hops. The algorithm in [13] has a message complexity of six messages per edge of the WFG and a time complexity of $3d + 1$ hops.

The message complexity of the proposed algorithm has four components.

1) There are at most e FLOOD's, by corollary 1.
2) There are at most e ECHO's, by corollary 2.
   a) The number of FLOOD's converted to SHORT's is at most "$e - n + l - (number\ of\ edges\ to\ leaf\ nodes)$," where l is the number of leaf nodes in the WFG. This is so because $(n - l)$ FLOOD's cannot be converted to SHORT's as each internal node uses one FLOOD to expand the WFG recording (Case F1-A), and because $(number\ of\ edges\ to\ leaf\ nodes)$ FLOOD's cannot be converted to SHORT's, since a leaf node always sends back an ECHO when a FLOOD arrives (Case F1-B).
   b) The number of ECHO's converted to SHORT's is at most "$e - n + l$," because if a nonleaf node in the WFG receives an ECHO along each outgoing edge in the WFG, one of the ECHO's will reduce the node and will not be converted to a SHORT.

Thus, the upper bound on the message complexity of the algorithm is $4e - 2n + 2l -$ (number of edges to leaf nodes) $< 4e$ messages. (This can be approximately stated as $(4e - 2n + 2l) < 4e$ messages.)

Recall that the logical transfer of each message takes one hop. The algorithm can be viewed to execute in steps. In step 0, the initiator sends FLOOD's. In step $i > 0$, a node receives any messages sent to it in step $i - 1$, performs local action, and sends messages.

A FLOOD that causes a node to execute Case F1 when the FLOOD is received causes that node to be included in the recorded WFG. A spanning tree can be defined on the recorded WFG as follows. The initiator is the root of the spanning tree. A node i is the parent of node j iff the FLOOD received from node i causes node j to execute Case F1. The height of the spanning tree is at most d. From observation 1, corollary 1, and the fact that a FLOOD is generated only at the time a node executes Case F1-A, it follows that the last step in which a FLOOD is sent denotes one less than the length of the longest spanning tree path from the initiator to some node. This is at most $d - 1$. Thus, all FLOOD's are delivered in at most d steps, and the FLOOD's delivered in step d are converted to SHORT's (Case F4-B) or ECHO's (Case F4-A, F3, F2, or F1-B).

An ECHO is initiated only in Cases F1-B, F4-A, F2, and F3 with the transfer of the weight from a FLOOD. The last step in which ECHO's may be initiated is d. ECHO's may travel only backward along the edges traversed by FLOOD's (lemma 2). ECHO's can travel at most $d - 1$ hops before being converted to SHORT's, or at most d hops before reaching the initiator. Thus, the worst case time complexity of the algorithm is $2d$.

It is conjectured that the algorithm has the optimal time complexity for detecting generalized deadlocks by using the technique of WFG recording and reduction, as explained next. The time complexity of $2d$ hops represents the round-trip travel time from the initiator to the farthest node in the WFG. This is the lower bound on the time because a given edge has to be recorded in the WFG before it can be reduced. To record the farthest edge in the WFG requires d hops. From the time the farthest edge is recorded in the WFG snapshot, d message hops are required in the worst case to complete reduction of the rest of the WFG.

Table I compares the performance of the proposed algorithm and the algorithms by Bracha and Toueg, and Wang *et al.*

Each blocked node may initiate its own detection algorithm. This gives rise to m instances of the algorithm in a deadlock of m nodes. An optimization on the number of messages can be performed by maintaining a time stamp–based priority order on all invocations of the detection algorithm and by suppressing lower-priority invocations [1].

VI. Handling Deadlock Resolution

The presented algorithm works for a system in which axiom 2 is valid; i.e., a blocked node can get unblocked only if its requests are satisfied without any intervention of the underlying computation. Axiom 2 ensures that if a deadlock exists in the system, it will be stable. Hence, if a deadlock is detected, it still exists in the system.

In another system model, a node can get unblocked "abnormally" if it spontaneously withdraws its requests, or if its requests are satisfied because of the resolution of a deadlock of which it is a part. An example of this model is one that allows a waiting process to time out. In this model, deadlocks are not stable, and the latest possible state should be observed when detecting deadlock. The algorithm presented in this paper can be modified to achieve this as follows: If a node's state is recorded as blocked in a local snapshot, then, if it unblocks abnormally, it initiates reduction of the (may be partially) recorded snapshot. This is necessary because the node does not know whether it will get reduced and will be able to send out ECHO's during the normal course of the algorithm to implicitly propagate the fact that it is now unblocked. However, the node has become unblocked and would like to ensure that the initiator becomes aware of this and does not detect deadlock falsely.

The readers are referred to [8] for a detailed treatment of this modified algorithm and its correctness proof. The upper bound on message complexity of this modified algorithm is between $(4e - 2n + 2l)$ and $(5e - 2n + 2l)$, and the upper bound on the time complexity is $2d$. Thus, it has a better time complexity than and a comparable message complexity to the best existing algorithms [1], [13], which do not even consider deadlock resolution (i.e., treat deadlock as a stable property).

VII. Conclusion

We have presented a distributed algorithm for detecting generalized deadlocks. Existing algorithms that detect generalized deadlocks use two distinct phases: one to record the snapshot of the WFG and the other to reduce the WFG to detect a deadlock. The proposed algorithm is a single-phased algorithm; it records a snapshot of the WFG of the initiator and concurrently reduces the WFG to check whether the initiator is deadlocked. Since reduction may be begun on an incompletely recorded WFG, the local snapshot at each node has to be carefully manipulated to give the effect that reduction of the WFG snapshot begins after the WFG snapshot has been fully recorded.

The correctness proof of the algorithm was provided in two steps. First, it was shown that the algorithm records a consistent distributed snapshot of the initiator's reachability set in the outward sweep. Then it was shown that reduction of the recorded snapshot, which begins in the inward sweep before the outward sweep is over, is performed correctly. Hence, when the algorithm terminates, a node in the WFG is not reduced iff it is deadlocked.

Since the algorithm uses a single phase to detect generalized distributed deadlocks, the message complexity and the time complexity of the algorithm are better than those of existing algorithms that use two phases. It is conjectured that the algorithm is optimal in time complexity if generalized deadlocks are to be detected by using distributed reduction of the WFG.

Finally, it can be observed that the presented algorithm, as well as the algorithms in [1], [13], can work in a request model in which a node requires replies on an arbitrary combination of its outgoing WFG edges in order to get unblocked. This arbitrary combination can be expressed as a predicate on elements in out_i and can be recorded in the snapshot as $LS_i.pred$. When a node receives an ECHO, it evaluates $LS_i.pred$. The node gets reduced (i.e., executes Case E2-A) when $LS_i.pred$, which is monotonic, changes from $false$ to $true$, instead of when $LS_i.p$ becomes 0.

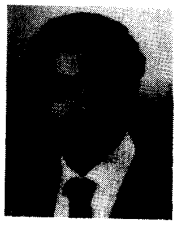
For the correctness proof of the above change to the algorithm, or for an alternate realization of the change described above, one can view the predicate in disjunctive normal form (DNF), where each disjunct is x is of type P-out-of-Q [4]. A disjunct x at node i requires $p_{i,x}$ REPLY's out of $q_{i,x}$ REQUEST's, so that node i will get unblocked. Every node in a deadlock (D, K) has at least $q_{i,x} - p_{i,x} + 1$ outgoing WFG edges to other nodes in D for each disjunct x. In the definition of function $h(i)$, the minimum overall disjuncts x of $\{$ $p_{i,x}^{th}$ smallest value of $\{h(j) | j \in out_{i,x}\}\}$ is used. In the algorithm, there will be a distinct $LS_i.p_x$, $LS_i.out_x$, and $LS_i.q_x$, for each disjunct x at node i. A node will execute Case E2-A (get reduced) at the earliest time that $LS_i.p_x$ becomes 0 for some x. The presented algorithm is a special case of this scheme with a single x and was presented in this form for simplicity. With the extensions to permit multiple disjuncts as described above, the presented algorithm works for the arbitrary request model.

Acknowledgment

The authors are greatly indebted to three anonymous referees for their painstakingly detailed comments on an earlier version of the paper.

References

[1] G. Bracha and S. Toeug, "Distributed deadlock detection," *Distributed Comput.*, vol. 2, pp. 127–138, 1987.
[2] A. L. Choudhary *et al.*, "A modified priority-based probe algorithm for distributed deadlock detection and resolution," *IEEE Trans. Software Eng.*, vol. 16, pp. 10–17, Jan. 1989.
[3] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, pp. 63–75, 1985.
[4] T. Herman and K. M. Chandy, "A distributed procedure to detect AND/OR deadlocks," Dept. Comput. Sci. Tech. Rep. TR-LCS-8301, University of Texas, Austin, TX, Feb. 1983.
[5] R. C. Holt, "Some deadlock properties of computer systems," *ACM Comput. Surveys*, vol. 4, 1972.
[6] S. Huang, "Detecting termination of distributed computations by external agents," *Proc. 9th Int. Conf. Distributed Comput. Syst.*, pp. 79–84, 1989.
[7] E. Knapp, "Deadlock detection in distributed databases," *ACM Computing Surveys*, vol. 19, pp. 303–328, Dec. 1987.
[8] A. D. Kshemkalyani, "Characterization and correctness of distributed deadlock detection and resolution," Ph.D. dissertation, Ohio State University, Aug. 1991.
[9] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communic. ACM*, vol. 21, pp. 558–565, July 1978.
[10] M. Roesler and W. A. Burkhard, "Resolution of deadlocks in object-oriented distributed systems," *IEEE Trans. Comput.*, vol. 38, pp. 1212–1224, 1989.
[11] M. Singhal, "Deadlock detection in distributed systems," *Comput.*, pp. 37–48, Nov. 1989.
[12] M. K. Sinha and N. Natarajan, "A priority-based distributed deadlock detection algorithm," *IEEE Trans. Software Eng.*, vol. 12, pp. 67–80, Jan. 1985.
[13] J. Wang, S. Huang, and N. Chen, "A distributed algorithm for detecting generalized deadlocks," Tech. Rep., Dept. of Comput. Sci., National Tsing-Hua University, 1990.



A. D. Kshemkalyani received the B. Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, India, in 1987, and the M.S. and Ph.D. degrees in computer and information science from the Ohio State University, Columbus, OH, in 1988 and 1991, respectively.

He is currently working in the Networking Systems Architecture Division at IBM Corp. in Research Triangle Park, NC. His current research interests include distributed systems, operating systems, computer architecture, and databases.

M. Singhal received the B. Eng. degree in electronics and communication engineering with high distinction from the University of Roorkee, Roorkee, India, in 1980, and the Ph.D. degree in computer science from the University of Maryland, College Park, MD, in May 1986.

He is currently an Associate Professor of computer and information science at the Ohio State University, Columbus, OH. His current research interests include distributed systems, operating systems, databases, and performance modeling. He has co-authored a book titled, *Advanced Concepts in Operating Systems: Distributed, Multiprocessor, and Database Operating Systems* (New York: McGraw-Hill, 1994).