Distributed Detection of Generalized Deadlocks

Ajay D. Kshemkalyani
IBM Corporation
P. O. Box 12195
Research Triangle Park
NC 27709

Mukesh Singhal
Department of Computer
and Information Science
The Ohio State University
Columbus, OH 43210

Abstract

Fast and efficient detection of deadlocks remains an important problem in distributed operating systems. In this paper, we present a distributed algorithm to detect generalized deadlocks in distributed systems. The algorithm performs reduction of a distributed wait-for-graph (WFG) to determine 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 $2d$ hops, where e is the number of edges and d is the diameter of the WFG. The algorithm is shown to perform better in both time and message complexity than the best known distributed algorithms to detect distributed generalized deadlocks. We conjecture that the algorithm is optimal in the number of messages and time delay, among distributed algorithms to detect generalized deadlocks.

1 Introduction

In computer systems, a deadlock occurs when a set of processes wait indefinitely on each other for their requests to be satisfied. A deadlock hampers the progress of the processes and lowers the resource availability; therefore, all deadlocks must be promptly detected and eliminated [10, 14]. Distributed systems are prone to deadlocks and detecting deadlocks is an important problem in the design of distributed systems. For the purpose of deadlocks, interaction between processes is modeled by a directed graph, called a wait-for graph (WFG) [10, 14]. Nodes in a WFG are processes and an edge from node i to node j indicates that process i has requested a resource from process j and process j has not granted the resource to process i. A deadlock is characterized by topological properties of the WFG that depend upon the underlying process 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 [6].

In the generalized request model, also called the AND-OR request model, the condition for a blocked process to get unblocked is expressed as a predicate involving requested resources and AND and OR operators [8]. For example, predicate $i \wedge (j \vee k)$ denotes that the process is waiting for a resource from i and for a resource from either j or k. In the P-out-of-Q request model [2], a process makes requests for Q resources and remains blocked until it is granted any P out of the Q resources. The P-out-of-Q and the AND-OR models are equivalent because a predicate in the AND-OR model can be expressed as a disjunction of P-out-of-Q type requests and vice-versa [8]. Thus the P-out-of-Q request model is also known as the generalized request model. A generalized deadlock corresponds to a deadlock in the AND-OR (or P-out-of-Q) request model. AND and OR models are special cases of the generalized deadlock model.

Although the problem of deadlock detection has been relatively well explored in the single-request, the AND request, and the OR request models, e.g., [4, 6, 10, 11, 14], there remains much to be done towards the detection of generalized deadlocks in distributed systems. The generalized request model is used frequently in many domains (such as resource allocation in distributed operating systems [2], communicating processes [9], and quorum consensus algorithms for distributed databases [7]) and efficient detection of generalized deadlocks remains an important problem.

Previous Work on Generalized Deadlock Detection

Detecting generalized deadlocks in a distributed 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 processes in the WFG to unblock. A cycle in the WFG is a necessary but not sufficient condition, whereas a knot in

0-8186-7813-5/97 $10.00

the WFG is a sufficient but not necessary condition for a generalized deadlock.

Design of correct and efficient algorithms to detect generalized distributed deadlocks is an extremely difficult problem and only four distributed algorithms (e.g., [2, 3, 13, 15]) exist to detect generalized distributed deadlocks. We do not consider centralized algorithms such as [1, 5] in which a snapshot of a WFG is collected by some process and then examined by that process for a generalized deadlock. The algorithms in [2, 13, 15] 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 [10]. The algorithm in [3] is based on the principle of detection of weak termination of a distributed computation.

The algorithm by Bracha and Toueg [2] 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. [15], the first phase records a snapshot of a distributed WFG. The end of the first phase is detected using a termination detection technique, after which the second phase is initiated to reduce the recorded WFG to detect a deadlock. In the one-phase algorithm of Kshemkalyani and Singhal [13], the recording of a snapshot of the distributed, dynamically changing WFG and reduction of the recorded WFG is done concurrently. The algorithm has to deal with the complications introduced because the reduction of a process can begin before the state of all WFG edges incident at that process have been recorded. Brzezinski et al. [3] define a generalized deadlock in terms of weak termination of a distributed computation and develop an algorithm that detects generalized distributed deadlocks by detecting weak termination of a distributed computation. Nodes are logically arranged as a ring and a token circulates on the ring to monitor the state of the nodes. The token keeps circulating until the monitored states of the nodes is the same in two consecutive rounds.

Paper Objectives

We present a decentralized algorithm for detecting generalized distributed deadlocks and outline its correctness proof. The algorithm performs reduction of a distributed wait-for-graph (WFG) to detect a deadlock. During the distributed reduction, 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.

The proposed algorithm performs better than the algorithms in [2, 3, 13, 15]; it has a message complexity of $2e$ messages and the worst-case time complexity of $2d$ hops, where e is the number of edges and d is the diameter of the WFG. It is conjectured in [12] 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: (i) no process has complete knowledge of the topology of the WFG or the system, and (ii) the deadlock detection is to be carried out in a distributed manner. If the initiator of the deadlock detection algorithm is deadlocked, at the time the algorithm terminates, it has all the necessary information to adequately resolve the deadlock, unlike the algorithms in [2, 3, 13, 15].

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 sketch the algorithm's correctness proof. In Section 6, we analyze the performance of the algorithm, and compare it with that of previous algorithms. Section 7 concludes.

2 System Model

The system consists of P processes (called nodes) which are connected by communication channels. There is no shared memory in the system and nodes communicate solely by sending messages over the channels. The messages are reliably delivered with finite but unpredictable delays. The system is assumed to be fault-free.

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 simplify 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, or are granted but not used in the evaluation of f_i.

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 during the normal course 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 its

requests are satisfied due to the resolution of a deadlock of which it is a part [11]. 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 timestamping mechanism to consider the dynamically changing WFG along the latest observable state [11, 13]. We do not allow a node to unblock abnormally for simplicity.

The interaction between processes 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, $|N| \ll |P|$.

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.

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 (N, E) where (i) each $i \in D(\neq \emptyset)$ is blocked by function $f_i(OUT_i)$ such that $f_i(OUT_i|_{(\forall j \in D, j=false) \wedge (\forall j \in OUT_i - D, j=true)}) = false$, and (ii) 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 gets 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 $OUT_i - D$, i.e., $f_i(OUT_i|_{(\forall j \in D, j=false) \wedge (\forall j \in OUT_i - D, j=true)}) = true$. All these nodes 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, it initiates a deadlock detection algorithm. Note that only the nodes that are reachable from a node in the WFG can be involved in deadlock with that node. Thus, the complete WFG is not examined to detect if a node is deadlocked; only that part of the WFG which is reachable from that node needs to be examined.

3 Basic Idea

No node has 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 its successor nodes. When a node receives the first FLOOD message, it propagates it to all its successor nodes, and so on. This is the flood phase. 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.

The 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 non-sink 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 non-sink node in the graph receives a second or subsequent FLOOD message, it responds with an ECHO message provided it has been reduced by then. However, a dilemma arises if a non-sink node in the graph has not been reduced when it receives the 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 are generated and moved up in the graph. Such a node can not 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.

Lazy Evaluation

If a non-sink node in the graph has not been reduced when it receives a second or subsequent FLOOD message, it immediately responds to such a FLOOD message with a PIP message. A PIP message conveys the indeterminate state of the node. 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 until the time 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, (i) the reduced status of nodes that previously sent a PIP message is propagated upwards in the DST towards the initiator node. Also, (ii) when a (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 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.

A node j keeps a set of residual functions, denoted by Z_j, that contains tuples of the form $<k, f_k>$, 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 its successor nodes), the message contains the residual function set of the sender node. An ECHO or a PIP message sent to a non-parent node carries null as the value of the residual function set. When a node j receives an ECHO or a PIP message with a non-null value of the residual function set, it adds the received residual function set to Z_j. This retarded collection of residual function 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 upwards 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 responses to all FLOOD messages it sent, it evaluates every residual function in the set Z_j as follows: select a tuple $<k, f_k>$ 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 $<k, f_k>$ from Z_j. This is done repeatedly until no more entries in Z_j can be reduced.

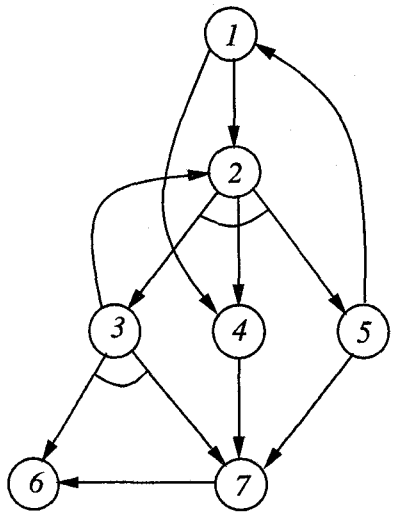

Figure 1. An example of a WFG.

Thus, a node j uses information in R_j about its successor nodes that sent PIP but later 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.

An Example

We now illustrate the basic idea behind the algorithm with the help of an example. Figure 1 shows a distributed WFG that spans seven nodes numbered 1 through 7. All nodes except node 6 are blocked and the unblocking functions of these nodes are as follows: 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=true, f_7=6.

Suppose node 1 initiates deadlock detection and sends out FLOOD messages to nodes 2 and 4. Figure 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.

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

[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.

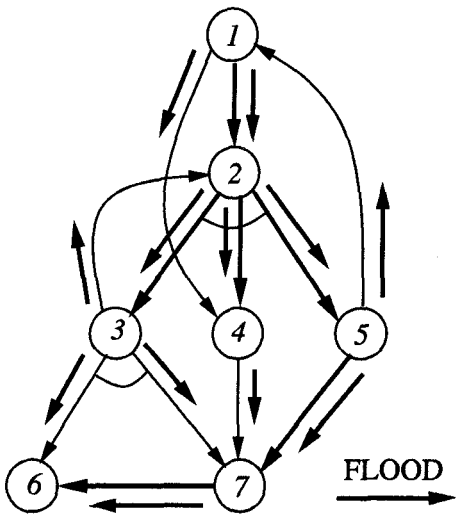

Figure 2. Diffusion of FLOOD messages.

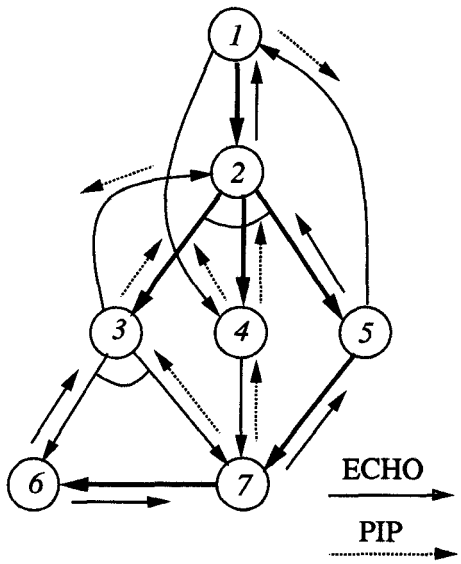

Figure 3. Flow of ECHO/PIP messages.

Node 7 responds to these FLOOD messages by PIP(7, 1, $\emptyset$, $\emptyset$) messages because the state of node 7 is indeterminate at this instant. On the receipt of the ECHO 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 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 FLOOD from node 1 before it receives PIP from node 7. Consequently, it responds to 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, $\emptyset$, {<4, 7>}) to its parent in the DST (node 2).

Node 2 sends a PIP(2, 1, $\emptyset$, $\emptyset$) message to 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, $\emptyset$, {<3, 2 $\vee$ 7>}) message to node 2.

On receipt of PIP(4, 1, $\emptyset$, {<4, 7>}) from node 4 and PIP(3, 1, $\emptyset$, {<3, 2 $\vee$ 7>}) message from node 3, Z_2 at node 2 becomes {<3, 2 $\vee$ 7>, <4, 7>}. On receipt of ECHO(5, 1, {7}, $\emptyset$) message from node 5, node 2 sets R_2 to {7}. It adds its residual function <2, 3$\wedge$4> 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".

4 Distributed Deadlock Detection Algorithm

The pseudo-code for the algorithm uses the symbol $\leftarrow$ for the assignment operator, and the CSP-like symbol $\Box$ for the selection operator. The semantics of "$a \longrightarrow b$" is "if a then b else $skip$." A node i has variables OUT_i, IN_i, and f_i which describe the WFG locally. The deadlock detection algorithm uses the following variables.

$parent_i$: **integer** $\leftarrow$ 0; /*node id of parent of node i.*/
out_i: **set of integer** $\leftarrow OUT_i$;
/*nodes for which 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 func. */
str: FNRES ← ⊥;                        /* 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 substituting *true* for j in X_i. Step E1. */
$X_i \leftarrow X_i(OUT_i \mid_{j=true})$;
$X_i = true \longrightarrow$
$init = i \longrightarrow$ NO deadlock; exit;
$pip\_sent_i = true \longrightarrow R_i \leftarrow R_i \bigcup \{i\}$;
/* 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. */
$out_i \leftarrow out_i - \{j\}$;
/* update local variables out_i, R_i, Z_i. Step EP1. */
$R_i \leftarrow R_i \bigcup R$;
$Z_i \leftarrow Z_i \bigcup Z$;
$out_i = \emptyset \longrightarrow$ /* all expected replies received. Step EP2. */
$X_i = false \longrightarrow$
/* i is not yet reduced. Add to Z_i. Step EP2.1. */
$str.id \leftarrow i$;
$str.uc \leftarrow X_i$;
$Z_i \leftarrow Z_i \bigcup \{str\}$;
eval;
/* use R_i to evaluate unreduced nodes in Z_i. Step EP2.2. */
$(X_i = true \bigwedge pip\_sent_i) \longrightarrow$
/* examine X_i using R_i which was updated in *eval*.*/
$R_i \leftarrow R_i \bigcup \{i\}$;
/* if i sent PIP, update R_i to indicate so. Step EP2.3.*/
$X_i = true \longrightarrow$
/* i is locally reduced using updated R_i. Step EP2.4. */
$init = i \longrightarrow$ NO deadlock; exit;
send $ECHO(i, init, R_i, Z_i)$ to $parent_i$;
$X_i = false \longrightarrow$
/* i is not locally reduced using updated R_i. Step EP2.5. */
$init = i \longrightarrow$ 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** $\leftarrow R_i$; /* working variable for R_i. */

repeat
For every $r \in tempR$ **do par**
for every $z \in Z_i$ **do par** instantiate each occurrence of r in $z.uc$ by $true$;
rap od;
$tempR \leftarrow tempR - \{r\}$;
rap od;
for every $z \in Z_i$ **do par**
$z.uc = true \longrightarrow$
$tempR \leftarrow tempR \bigcup \{z.id\}$;
$z.id \neq i \longrightarrow R_i \leftarrow R_i \bigcup \{z.id\}$;
/* if $z.id = i$, then X_i will also be true. */
$Z_i \leftarrow Z_i - \{z\}$;
rap od;
until $tempR = \emptyset$.

5 Correctness Proof

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.

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

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 pro-*

cessed 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* $X_i = true$ *after* $out_i = \emptyset$.

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 *by the time* $out_j = \emptyset$ *(these have all been reduced either locally or remotely) 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 $true$. However 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.

We sketch a correctness proof by stating some lemmas and observations. Detailed proofs are given in [12].

Lemma 1 states that node i may be reduced at most at one node in $\{i\} \bigcup parent_i^+$.

Lemma 1 $reduce^i \implies reduce_i^i \oplus (reduce^i_{(j_1 \oplus j_2 \oplus \ldots \oplus j_w) \neq i}, j_1, j_2, \ldots, j_w \in parent_i^+)$

Lemma 2 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 2 $reduce^i_{k \neq i} \implies \forall j \in (parent_k^+ \bigcup \{k\})$, *when* $out_j = \emptyset$, $i \in R_j$

Lemma 3 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 before i locally terminates.

Lemma 3 *(Node* j *sent a PIP and then* $reduce_j^j) \implies \forall i \in (parent_j^+ \bigcup \{j\})$, *at the time* $out_i = \emptyset$, $j \in R_i$

Lemma 4 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 $l \in OUT_j^+ \bigcup \{i\}$.

Lemma 4 $i \in R_j \implies reduce_l^i$, $l \in OUT_j^+ \bigcup \{i\}$ *and* $reduce_l^i$ *happened before* i *was placed in* R_j

Observation 1 *If node* i *belongs to the* R *parameter in some ECHO or PIP received by* j, *then* $reduce_k^i$ *where* $k \in OUT_j^+$.

Lemma 5 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, [$(z \in Z_j$, where $z.id = i) \oplus (i \in R_j)$].

Lemma 5 *(*i *sends a PIP* $\bigwedge reduce_i^i) \bigvee \neg reduce_i^i$ *when* $out_i = \emptyset \iff$
$\forall j \in (parent_i^+ \bigcup \{i\})$, *when* $out_j = \emptyset$, *[*$(z \in Z_j$, *where* $z.id = i) \oplus (i \in R_j)$*]*

Observation 2 *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.*

Observation 3 *If node* i *receives an ECHO or a PIP from node* j, *node* i *has already sent a FLOOD to node* j *and* $j \in out_i$.

Observation 4 *Node* i *does not send any ECHOs unless* $reduce_i^i$.

The above observations and lemmas are used to prove the following result [12].

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

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. The algorithm considerably facilitates efficient and fast resolution of a detected deadlock, whereas the other algorithms [2, 13, 15] require an additional round of messages to collect the information that is needed to resolve the deadlock.

6 Performance

For a WFG with e edges and a diameter d, the following results are shown in [12].

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

Note 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 2 message hops [12].

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

Criterion	Bracha-Toueg [2]	Wang et al. [15]	Kshemkalyani-Singhal [13]	Proposed algorithm
Phases	2	2	1	1
Delay	$4d$	$3d+1$	$2d$	$2d$
Messages	$4e$	$6e$	$4e-2n+2l$ ($\leq 4e$)	$2e$

Table 1. Comparison of worst-case performance complexities. Given a WFG (N, E), $n = |N|$, l = number of sink nodes, $e = |E|$, d = diameter.

Note that the message size in the algorithm is variable, and some messages may be larger than those in earlier algorithms [2, 15, 13]. However, message headers are usually large, so a slightly larger message body should not be a drawback.

Table 1 compares the performance of the proposed algorithm with the existing distributed algorithms to detect generalized deadlocks [2, 13, 15]. The proposed algorithm performs better than the algorithms in [2, 13, 15]; it has a message complexity of $2e$ messages and the worst-case time complexity of $2d$ hops. Also, in the proposed algorithm the initiator has information locally available to resolve the detected deadlock; the other algorithms incur extra time and message overhead to achieve this. It is conjectured in [12] that the proposed algorithm is optimal in the number of messages and time delay if detection of generalized deadlocks is to be carried out under the following framework: (i) no process has complete knowledge of the topology of the WFG or the system, and (ii) the deadlock detection is to be carried out in a distributed manner.

7 Conclusions and Discussion

We presented a distributed algorithm for detecting generalized deadlocks in a distributed system. The algorithm 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 sketched the correctness proof 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 distributed algorithms to detect generalized deadlocks in distributed systems. It has a message complexity of $2e$ messages and the worst-case time complexity of $2d$ hops. We conjecture that the algorithm is optimal in the number of messages and time delay, among distributed algorithms to detect generalized deadlocks [12].

In practice, a WFG is dynamic, i.e., processes make requests and requests get satisfied on a continual basis. Consider an edge from i to j. By the time a FLOOD sent by i reaches j, j had already replied to i. This edge is a *phantom* edge. Phantom edges that arise due to the dynamic nature of the WFG are dealt with as in [13].

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 invocations [2].

References

[1] C. Beeri, R. Obermarck, A Resource Class Independent Deadlock Detection Algorithm, *IBM TR RJ-3077*, March 1981.

[2] G. Bracha, S. Toeug, Distributed Deadlock Detection, *Distributed Computing*, 2:127-138, 1987.

[3] J. Brzezinski, J. Helary, M. Raynal, M. Singhal, Deadlock Models and Generalized Algorithm for Distributed Deadlock Detection, *Jour. Par. Distrib. Comput.*, Dec. 1995.

[4] K. M. Chandy, J. Misra, A Distributed Graph Algorithm: Knot Detection, *ACM Transactions on Programming Languages and Systems*, 678-686, Oct. 1982.

[5] S. Chen, Y. Deng, P. Attie, W. Sun, Optimal Deadlock Detection in Distributed Systems based on Locally Constructed Wait-For Graphs, *Proc. IEEE ICDCS*, 613-619, May 1996.

[6] I. Cidon, An Efficient Distributed Knot Detection Algorithm, *IEEE Trans. on Software Engg.*, 15(5), 644-649, May 1989.

[7] D. K. Gifford, Weighted Voting for Replicated Data, *Proc. 7th ACM Symp. on Op. Syst. Principles*, 150-162, Dec. 1979.

[8] T. Herman, K. M. Chandy, A Distributed Procedure to Detect AND/OR Deadlocks, *TR-LCS-8301*, University of Texas, Austin, Feb. 1983.

[9] C.A.R. Hoare, Communicating Sequential Processes, *Communications of the ACM*, 666-677, 21(8), Aug. 1978.

[10] R. C. Holt, Some Deadlock Properties of Computer Systems, *ACM Computing Surveys*, 4(3), 1972.

[11] A. D. Kshemkalyani, M. Singhal, Characterization and Correctness of Distributed Deadlock Detection and Resolution, *Jour. Par. Distrib. Computing*, 44-59, 22(1), July 1994.

[12] A. D. Kshemkalyani, M. Singhal, Optimal Detection of Generalized Distributed Deadlocks, *IBM Tech. Rep. 29.2039*, July 1995.

[13] A. D. Kshemkalyani, M. Singhal, Efficient Detection and Resolution of Generalized Distributed Deadlocks, *IEEE Trans. on Software Engg.*, 43-55, 20(1), Jan. 1994.

[14] M. Singhal, Deadlock Detection in Distributed Systems, *IEEE Computer*, 37-48, Nov. 1989.

[15] J. Wang, S. Huang, N. Chen, A Distributed Algorithm for Detecting Generalized Deadlocks, Tech. Rep., National Tsing-Hua Univ., 1990.