An Efficient Distributed Algorithm for Detection of Knots and Cycles in a Distributed Graph

D. Manivannan, *Member*, *IEEE*, and Mukesh Singhal, *Fellow*, *IEEE*

Abstract—Knot detection in a distributed graph is an important problem and finds applications in deadlock detection in several areas such as store-and-forward networks, distributed simulation, and distributed database systems. This paper presents an efficient distributed algorithm to detect if a node is part of a knot in a distributed graph. The algorithm requires $2e$ messages and a delay of $2(d + 1)$ message hops to detect if a node in a distributed graph is in a knot (here, e is the number of edges in the reachable part of the distributed graph and d is its diameter). A significant advantage of this algorithm is that it not only detects if a node is involved in a knot, but also finds exactly which nodes are involved in the knot. Moreover, if the node is not involved in a knot, but is only involved in a cycle, then it finds the nodes that are in a cycle with that node. We illustrate the working of the algorithm with examples. The paper ends with a discussion on how the information about the nodes involved in the knot can be used for deadlock resolution and also on the performance of the algorithm.

Index Terms—Distributed graph, distributed systems, knot detection, deadlock detection, distributed algorithms, distributed simulation.

1 Introduction

A knot in a directed graph is a subgraph such that every node in the subgraph can be reached from *every* other node in the subgraph and no node outside the subgraph is reachable (a node B in a directed graph is reachable from another node A if there is a directed path from A to B) from any node in the subgraph. Thus, a node in a directed graph belongs to a knot if and only if the node can be reached from all the nodes that are reachable from it. The problem of knot detection in a distributed graph arises in several domains. For example, in store-and-forward networks, packets at a switching node are blocked if there is no empty buffer at the next switch, and a deadlock occurs if there is a knot in the buffer graph of the switching nodes [7], [11]; in distributed simulation where a process waits for messages from other processes, a knot in the process graph implies a deadlock [1], [16], [10], [18]; in distributed database systems, a transaction waits for response (e.g., a quorum) from remote hosts and a deadlock occurs if the transaction-wait-for graph contains a knot [8]. Chang [3] shows that knot is a useful concept in deadlock detection. Thus, knot detection in a distributed graph is an important problem and efficient knot detection algorithms are required. For example, in Fig. 1, the subgraph consisting of the vertices {A, B, C, D, E} is a knot because every node in this subgraph is reachable from every other node in the subgraph and no node outside this subgraph is reachable from any node in this subgraph.

1.1 The OR Request Model

In the OR request model, a process can simultaneously request multiple resources and it remains blocked until it is granted *any one* of the requested resources [23]. Thus, in the OR model, a process is *deadlocked* if *none* of the resources it is requesting will ever be granted. The existence of a knot in the wait-for-graph is necessary and sufficient for deadlock in the OR Request model.

1.2 The AND Request Model

In the AND Request Model, a process can simultaneously request multiple resources and it remains blocked until it is granted *all* of the requested resources [23]. Thus, in the AND model, a process is *deadlocked* if at least one of the resources it is requesting will never be granted. In the AND request model, the existence of cycle in the wait-for-graph is necessary and sufficient for deadlock.

Thus, in general, deadlocks occur in distributed systems due to the existence of cycles or knots in the communication graph. Hence, efficient distributed algorithms for detecting knot and cycles in a distributed graph are required. A good survey about deadlock detection can be found in [10], [13]. Several algorithms for deadlock detection under various models have been proposed in the literature [2], [14], [15], [20], [21], [22], [25]. Several distributed knot detection algorithms [1], [4], [7], [19] have been proposed in the literature. To our knowledge, none of the distributed knot detection algorithms proposed in the literature actually finds the nodes involved in the knot. Our algorithm not only detects the existence of a knot, but also finds the nodes involved in the knot. Moreover, if a node is not involved in

• *The authors are with the Computer Science Department, University of Kentucky, 301 Rose Street, Hardymon Building, Room 231, Lexington, KY 40506. E-mail: {mani, singhal}@cs.uky.edu.*

Manuscript received 28 May 2002; revised 13 Jan. 2003; accepted 15 May 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 116638.

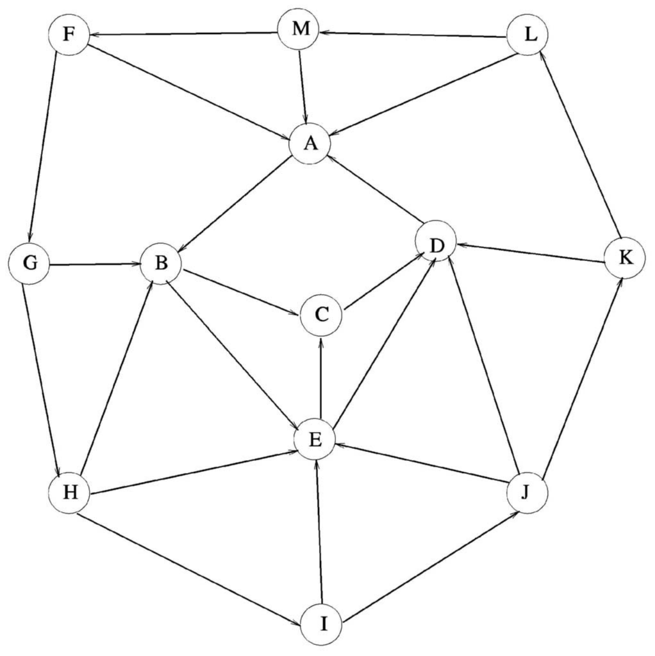

Fig. 1. The subgraph formed by the vertices {A, B, C, D, E} is a knot.

a knot, then our algorithm finds the set of all nodes that are involved in a cycle with that node. Information about the nodes involved in the knot or cycle can be very useful for efficient resolution of deadlocks.

1.3 Related Work

Various distributed knot detection algorithms have been proposed to detect knots in distributed graphs of n vertices and e edges. These algorithms can be categorized into three basic classes [4]:

1. In the algorithms in the first class, the complete distributed graph is collected at each node by extensive message passing and knots are detected by a centralized algorithm at each node. The message complexity of one such algorithm is O(ne) [7]. This approach is simple, but very inefficient, since experimental results show that the number of nodes involved in a deadlock is very small, in general [9].
2. In the algorithms in the second class, each node independently searches to discover whether it is a member of a knot. A distinct search is used at each node [19]. The Misra-Chandy algorithm [19], for instance, is based on termination detection of diffusing computation and requires at most $4e$ messages. Dijkstra [5] proposed a distributed algorithm for detecting whether a network of processes is in a knot. His algorithm is based on his previous work with Scholten [6] on termination detection of diffusing computations. Boukerche and Tropper [1] proposed a knot detection algorithm which requires $2e$ messages; however, like other algorithms, their algorithm also does not find the nodes involved in the knot.
3. The algorithms in the third class [4] exploit the property that a knot is a *strongly connected subgraph*[1] with no edge directed outside the subgraph. A group of nodes that forms a strongly connected subgraph is called a *cluster*. A cycle of clusters can be merged together to form a bigger cluster. This process of merging a cycle of clusters is continued. If a cluster with no edge directed outside the cluster is found, then a knot is found. Cidon's algorithm [4] follows this approach and has message complexity $O(e + n\ log(n))$.

1.4 Objectives

This paper presents an efficient distributed knot detection algorithm in the second class described above. In the previous algorithms of the second class, the initiator generally sends messages along all incoming and outgoing edges of the initiator. These messages propagate along the edges of the graph and are echoed back to the initiating node. Based on the information received in the echoes, the initiating node determines whether it is in a knot. All of the algorithms in the second class, except the one proposed by Boukerche and Tropper [1], have the following drawbacks:

1. The node initiating the knot detection and all nodes that receive the knot detection message must know their incoming as well as outgoing edges. In real-world applications, however, a node may not know its incoming edges. For example, in store-and-forward networks, a node only knows the nodes to which it is waiting to send a packet; it may not know about the nodes that are trying to send it a packet.
2. In these algorithms, knot detection messages travel along the backward edges of the graph even to nodes that are not reachable from the initiator. However, such unreachable nodes are irrelevant to determine whether the initiator is in a knot, thus increasing the message complexity.
3. When these algorithms terminate, they only detect if the initiating node is in a knot [19]. The initiator discovers no information about what other nodes are involved in the knot. However, such knowledge about the nodes involved in the knot can be essential for resolving the deadlock appropriately and efficiently.

The algorithm proposed by Boukerche and Tropper [1] does not have the first two drawbacks; however, their algorithm only detects if a node is involved in a knot and does not gather any information about what nodes are involved in the knot. Our proposed algorithm does not have any of the drawbacks mentioned above. Knot detection messages sent by the initiator reach only the nodes reachable from the initiator; no knot detection message is propagated backward along the edges of the graph. Unlike previous algorithms of the second class, our algorithm collects the set of all the nodes involved in the knot, which can help very much in resolving the deadlock represented by the knot. Our algorithm requires $2e$ messages and $2(d + 1)$ message hops delay to detect a knot, where e is the

1. A subgraph of a directed graph is called a strongly connected subgraph if, for any two vertices x and y in the subgraph, there is a directed path from x to y in the subgraph.

number of edges in the *reachable part* of the graph and d is the diameter of that part of the graph.

The rest of the paper is organized as follows: In the next section, we present the system model and definitions. In Section 3, we present our algorithm for knot detection. In Section 4, we prove its correctness, illustrate the algorithm with examples, compare our algorithm with existing work, and describe how knot detection can help in deadlock resolution. In Section 5, we analyze the performance of our algorithm. Section 6 concludes the paper.

2 System Model and Definitions

Computation is organized in spatially separated processes $\{P_1, P_2, \ldots, P_n\}$, running on geographically distributed nodes in a distributed system which are connected by a set of bidirectional communication channels. The processes do not share a common memory and they communicate with each other solely by passing messages over the communication channels. Messages are delivered reliably with finite but arbitrary time delay. The organization can be described as a graph in which vertices represent the processes and edges represent communication channels. In this paper, for simplicity of exposition, we assume that there is only one process running on each node and, hence, we use the terms nodes, sites, and processes interchangeably. Each node has a unique id which is an integer that lies between 1 and n. We assume that if a node i sends a message to node j, that message gets appended to the end of j's unbounded length input buffer. This assumption is for ease of exposition.

The problem is to detect if a node i is in a knot in the problem graph, which is a directed subgraph of the communication graph. For instance, the problem graph could be a wait-for-graph of a resource allocator. We assume that the problem graph does not change during the execution of the algorithm. For simplicity, we assume that only one process initiates the knot detection. If multiple processes initiate knot detection, the initiations can be distinguished by initiator id and a sequence number associated with the initiation.

3 Our Knot Detection Algorithm

In this section, we present the basic idea behind our algorithm. Then, we present the algorithm formally and provide a detailed explanation of the algorithm.

3.1 Basic Idea

We assume that node i, called the initiator, initiates the knot detection algorithm. A node j is said to be reachable from node i if there is a directed path in the graph from i to j. First, observe that a node i is in a knot if and only if every node that is reachable from i lies on a cycle containing i. Thus, to determine if the initiator is involved in a knot, it only needs to check if every node that is reachable from the initiator node is on a cycle that passes through the initiator. Since no node has complete knowledge of the topology of the graph, the initiator diffuses $detect\_knot$ message to all nodes in the *reachable part* of the graph; the reachable part of the graph is the set of all those nodes to which there is a directed path from the initiator. In other words, to initiate knot detection, the initiating node sends a $detect\_knot$ message to all its immediate successors; a node j is an immediate successor of a node i in the graph if and only if there is a directed edge (i, j) from node i to node j. Each node receiving the first $detect\_knot$ message propagates it to all its immediate successors. Thus, $detect\_knot$ messages propagate to all the nodes that are reachable from the initiator. The edges along which the first $detect\_knot$ message is received by each node form a directed spanning tree (DST) rooted at the initiator.

If the initiator i receives the $detect\_knot$ message from a node j, i sends a $cycle\_ack$ reply to j. This $cycle\_ack$ message informs node j that node j is on a cycle with the initiator. It is easy to see that all the nodes in the DST that lie on a path from the initiator i to node j are also on a cycle with the initiator. In general, if a node k lies on a directed path that joins i and a node that is known to be on a cycle with i, then node k also lies on a cycle with i. The problem is to find an efficient technique to determine if a node in the DST lies on a path that passes from i to a node that is known to be on a cycle with i. This problem is solved as follows:

After a node knows that it is on a cycle with i, it sends $cycle\_ack$ acknowledgment to all the nodes from which it receives a $detect\_knot$ message thereafter. This message informs the recipients that they are also on a cycle with i. However, after receiving the first $detect\_knot$ message, a node k may receive a $detect\_knot$ message from some other node j before knowing if it is on a cycle with i; in this case, node k sends $seen\_ack$ acknowledgment to node j as soon as it receives $detect\_knot$. This message basically informs node j that k has already seen a $detect\_knot$ message. When j receives the $seen\_ack$ acknowledgment from k, it includes the ordered pair (j, k) in a local variable, $Seen_j$. Later, if k is found to be on a cycle with i, it can be concluded from this ordered pair that j is also on a cycle with i. After node j has received acknowledgments from all its immediate successors, it sends a $parent\_ack$ acknowledgment to its parent. The $parent\_ack$ lists the nodes that are known to j to be on a cycle with i, as well as the set of all ordered pairs of the form (j, l) that j has stored in the local variable $Seen_j$ as a result of $seen\_ack$ and $parent\_ack$ acknowledgments from its immediate successors in the wait-for-graph. This information about the nodes that are known to be on a cycle with the initiator, as well as the set of ordered pairs (j, l) collected, is propagated back in the DST until it reaches i.

The initiator i keeps the set of all nodes that are known to be on a cycle with i in a variable called $Incycle_i$. It also keeps in the variable $Seen_i$, the set of all the ordered pairs (j, k) collected in the $parent\_ack$, and $seen\_ack$ replies received from its immediate successors. For each $j \in Incycle_i$, if $(k, j) \in Seen_i$, then node i concludes that k is on a cycle with i and removes (k, j) from $Seen_i$ and includes k in the set $Incycle_i$. If $Seen_i$ becomes empty, then i concludes that it is in a knot. If i concludes that it is in a knot, then the variable $Cycle\_nodes$ contains exactly the ids of the nodes involved in the knot (see the algorithm in Section 3.2). If i concludes that it is not in a knot, then the variable $Cycle\_nodes$ contains the set of nodes that are in cycle with i.

3.2 The Algorithm

We now give the pseudocode for the knot detection algorithm: As before, we call the initiator i; it has the following variables:

- OUT_i: A set of node ids, initialized to the set of immediate successors of i in the graph.
- $Seen_i$: A set of ordered pairs of node ids. An ordered pair $(k, j) \in Seen_i$ implies that there is a path from i to k and an edge from k to j in the graph; it also means that both k and j have seen the *detect_knot* message and, when j received the *detect_knot* message from k, j did not know yet if it is on a cycle with the initiator. Initially, $Seen_i$ is empty.
- $Incycle_i$: A set of node ids, initially empty. Node $k \in Incycle_i$ implies k is in a cycle with i.
- $Cycle\_nodes_i$: A set of node ids, initially empty. When the algorithm terminates, if i is in a knot, this variable contains the ids of all the nodes involved in the knot.
- $done_i$: This is a Boolean variable, initially set to false. It is set to true after it receives acknowledgments from all its immediate successors.

A node k other than the initiator has the variables OUT_k, $Seen_k$, $Incycle_k$, and $done_k$, whose functions are similar to the corresponding variables at node i. In addition, node k also has the following variables:

- has_seen_k: This is a Boolean variable which is set to true when node k receives the first *detect_knot* message. Initially, it is set to false.
- $parent_k$: Node k sets this variable to the id of the node from which it received the first *detect_knot* message. Initially, it is set to 0, meaning it has no parent node. After all nodes that are reachable from i receive the *detect_knot* message, the value of the *parent* variables at each node helps form a distributed spanning tree of the reachable nodes from i.

The algorithm at the initiating node i (i.e., Process P_i), and the algorithm at any other node k (i.e., Process P_k) are given below in CSP (communicating sequential processes) [12] like syntax. The algorithm at the initiating node i is given as a process with a *repetitive command* having five alternative *guarded commands*. The algorithm at any other node k is given as a process with a *repetitive command* having five alternative *guarded commands*. A guarded command is of the form:

$$Guard \longrightarrow Command.$$

We call a guarded command an *Action*. An *Action* is enabled if its *Guard* becomes true. Each enabled *Action*, when executed, is executed atomically. If several *Actions* are enabled at an instant, then only one of these enabled *Actions* is picked randomly and executed at a time. We assume that each enabled action is given a fair chance for execution. We first present the algorithm and then describe the algorithm, in detail later (see Fig. 2).

3.3 An Explanation of the Algorithm

When node i wants to find out if it is in a knot, it sends a $detect\_knot(i, i)$ message to all its immediate successors (i.e., the nodes in OUT_i). The first parameter of the *detect_knot* message is the id of the initiator and the second parameter is the id of the node propagating the message. When node k receives the $detect\_knot(i, j)$ message from node j, it takes one of the following actions (*Action 1* of P_k):

1. If it is the first *detect_knot* message received, it sets its *parent* to j and propagates the *detect_knot* message to all the nodes in OUT_k.
2. If it has already seen a *detect_knot* message and $Incycle_k \neq \emptyset$, then it sends $cycle\_ack(k)$ to j. When node j receives this acknowledgment, it includes the id k in the set $Incycle_j$. ($Incycle_k \neq \emptyset$ implies that k is on a cycle with i which implies that j is also on a cycle with i).
3. If it has already seen a *detect_knot* message and $Incycle_k = \emptyset$, then it sends $seen\_ack(k)$ acknowledgment to site j. After receiving this message, j includes the ordered pair (j, k) in its local variable $Seen_j$ so that if, later, k is found to be on a cycle with the initiator, the initiator can conclude that j is also on a cycle with the initiator.

When the initiator receives a *detect_knot* message back from a node k, it sends a $cycle\_ack(i)$ message to node k so that k knows that it is on a cycle with i (*Action 4* of P_i). So, when node k receives $cycle\_ack(i)$ for a *detect_knot* message, it includes id i in the set $Incycle_k$ so that k can conclude that it is in cycle with the initiator (*Action 4* of P_k). When node k receives $seen\_ack(j)$ for a *detect_knot* message from node j, it includes the ordered pair (k, j) in the variable $Seen_k$ (*Action 3* of P_k). $(k, j) \in Seen_k$ implies j is an immediate successor of k and k is reachable from i. Later, when this information reaches node i, if node i finds that j is on a cycle with i, then it can conclude that node k is also on a cycle with i. When k receives a $parent\_ack(Seen_j, Incycle_j)$ message from one of its children in the DST, it updates its variables $Seen_k$ and $Incycle_k$ (*Action 2* of P_k).

After a node k has received acknowledgment from all its immediate successors (i.e., when $OUT_k = \emptyset$), it includes its id in the set $Incycle_k$ if it finds at least one of its immediate successors is on a cycle with i (i.e., $Incycle_k \neq \emptyset$) and sends a $parent\_ack(Seen_k, Incycle_k)$ message to its parent (*Action 5* of P_k).

After receiving an acknowledgment from all its immediate successors (i.e., when $OUT_i = \emptyset$), *Action 5* is executed by the initiating node i. Note that, after node i has received an acknowledgment from all its immediate successors, for each node j that is reachable from i, either $j \in Incycle_i$ or $(k, j) \in Seen_i$ for some node k, reachable from i (see Observation 4 below). If $j \in Incycle_i$ and $(k, j) \in Seen_i$, then k is also on a cycle with i. Therefore, after the execution of *Action 5*, the variable $Cycle\_nodes_i$ contains the ids of all the nodes reachable from i that are on a cycle with i. If $Seen_i$ becomes empty, which means all the nodes reachable from i are on a cycle with i (see Theorem 1 below), then node i declares *"knot detected."*

4 Properties of the Algorithm

In this section, we prove the correctness of the algorithm and illustrate the algorithm with some sample executions.

The algorithm

```
Process $P_i$: /* the knot detection algorithm at the initiator node $i$ */
$OUT_i$: set of node ids(initially the set of all immediate successors of $i$);
$Seen_i$: set of ordered pairs of node ids(initially empty);
$Incycle_i$: set of node ids(initially empty);
$Cycle\_nodes_i$: set of node ids(initially empty); /* nodes that are in cycle with $i$ */
$done_i$: boolean(initially $false$); /* set to $true$ when the algorithm terminates */

send $detect\_knot(i, i)$ to all nodes in $OUT_i$;

*[$\neg done_i \wedge$                               $\longrightarrow$ $OUT_i := OUT_i - \{k\}$;                /*Action 1*/
  receive $parent\_ack(Seen_k, Incycle_k)$                $Seen_i := Seen_i \cup Seen_k$;
  from $P_k$                                              if $Incycle_k = \emptyset$ then
                                                              $Seen_i := Seen_i \cup \{(i,k)\}$;
                                                          else
                                                              $Incycle_i := Incycle_i \cup Incycle_k$;

  ⟦
  $\neg done_i \wedge$ receive $seen\_ack(k)$ from $P_k$    $\longrightarrow$ $OUT_i := OUT_i - \{k\}$;    /* Action 2*/
                                                          $Seen_i := Seen_i \cup \{(i,k)\}$;

  ⟦
  $\neg done_i \wedge$ receive $cycle\_ack(k)$ from $P_k$   $\longrightarrow$ $OUT_i := OUT_i - \{k\}$;    /*Action 3*/
                                                          $Incycle_i := Incycle_i \cup \{k\}$;

  ⟦
  $\neg done_i \wedge$ receive $detect\_knot(i,k)$ from   $\longrightarrow$ send $cycle\_ack(i)$ to $P_k$;   /*Action 4*/
  $P_k$
  ⟦
  $\neg done_i \wedge OUT_i = \emptyset$                  $\longrightarrow$ for each $j \in Incycle_i$ do     /*Action 5*/
                                                              $Incycle_i := Incycle_i - \{j\}$;
                                                              $Cycle\_nodes_i := Cycle\_nodes_i \cup \{j\}$;
                                                              for each $k$ $(1 \le k \le n)$ do
                                                                  if $(k,j) \in Seen_i$ then
                                                                      $Incycle_i := Incycle_i \cup \{k\}$;
                                                                      $Seen_i := Seen_i - \{(k,j)\}$;
                                                          if $Seen_i = \emptyset$ then
                                                              Declare 'knot detected';
                                                          else
                                                              Declare 'no knot detected';
                                                          $done_i := true$
]
```

Fig. 2. The algorithm.

We also illustrate with examples how the information about the nodes in the knot can be used for deadlock resolution. First, we make the following observations:

Observation 1. When an initiator propagates a $detect\_knot$ message, it is eventually received by all nodes of the communication graph that can be reached from node i.

Observation 2. After a $detect\_knot$ message has been received by all nodes that are reachable from i, if we follow the $parent$ variables at each node, we get the directed spanning tree (DST) of the nodes reachable from i.

Observation 3. Every node in the DST diffuses $detect\_knot$ messages along all its outgoing edges in the communication graph.

Definition. *$pred(j)$ denotes the predecessor node of node j in the DST. The distance of a node j, denoted by $dst(j)$ in the DST is defined as:*

$$dst(i) = 0$$
$$dst(j) = dst(pred(j)) + 1 \text{ if } j \neq i.$$

Let $dst_{max} = max\ \{dst(j)\}$. Clearly, $dst_{max} \le$ d, the diameter of the communication graph.

Observation 4. When *Action 5* of process P_i is enabled, for every node j that is reachable from i, either or both of the following holds: 1) $j \in Incycle_i$, 2) $(k, j) \in Seen_i$ for some k that is reachable from i.

This is because of the following: Before P_i executes *Action 5*, from Observation 2, every node that is reachable from i is in the DST. Also, every node $j \neq i$ in the DST sends

```
Process $P_k(k \neq i)$: /* algorithm at any node $k$ other than the initiator $i$ */
$has\_seen_k$: boolean(initially $false$);
$OUT_k$: set of node ids(initially the set of all immediate successors of $k$);
$Seen_k$: set of ordered pairs of node ids(initially $empty$);
$Incycle_k$: set of node ids(initially $empty$);/* nodes that are in cycle with $i$ */
$parent_k$: a node id (initially 0);
$done_k$: boolean(initially $false$); /* set to $true$ when $P_k$ sends $parent\_ack$ to its parent */

*[ receive $detect\_knot(i, j)$ from $P_j$   $\longrightarrow$ if $\neg has\_seen_k$ then                    /*Action 1*/
                                                      $has\_seen_k := true$;
                                                      $parent_k := j$;
                                                      send $detect\_knot(i, k)$ to all
                                                      nodes in $OUT_k$;
                                                  else
                                                      if $Incycle_k \neq \emptyset$ then
                                                          send $cycle\_ack(k)$ to $P_j$;
                                                      else
                                                          send $seen\_ack(k)$ to $P_j$;
 ‖
 $\neg done_k \wedge$                     receive   $\longrightarrow$ $OUT_k := OUT_k - \{j\}$;             /*Action 2*/
 $parent\_ack(Seen_j, Incycle_j)$            from      $Seen_k := Seen_k \cup Seen_j$;
 $P_j$                                                if $Incycle_j = \emptyset$ then
                                                          $Seen_k := Seen_k \cup \{(k, j)\}$;
                                                      else
                                                          $Incycle_k := Incycle_k \cup Incycle_j$;
 ‖
 $\neg done_k \wedge$ receive $seen\_ack(j)$ from $P_j$   $\longrightarrow$ $OUT_k := OUT_k - \{j\}$;   /*Action 3*/
                                                  $Seen_k := Seen_k \cup \{(k, j)\}$;
 ‖
 $\neg done_k \wedge$ receive $cycle\_ack(j)$ from $P_j$   $\longrightarrow$ $OUT_k := OUT_k - \{j\}$;  /*Action 4*/
                                                  $Incycle_k := Incycle_k \cup \{j\}$;
 ‖
 $\neg done_k \wedge OUT_k = \emptyset \wedge has\_seen_k$   $\longrightarrow$ if $Incycle_k \neq \emptyset$ then   /*Action 5*/
                                                      $Incycle_k = Incycle_k \cup \{k\}$;
                                                  send $parent\_ack(Seen_k, Incycle_k)$ to $parent_k$;
                                                  $done_k := true$;
 ]
```

Fig. 2. The algorithm (continued).

$parent\_ack(Seen_j, Incycle_j)$ to its parent (*Action 5* of P_j). Before sending this acknowledgment, node j includes its id j in the set $Incycle_j$ if $Incycle_j \neq \emptyset$ (*Action 5* of P_j); if $Incycle_j = \emptyset$, then the node k receiving the $parent\_ack$ $(Seen_j, Incycle_j)$ message includes the ordered pair (k, j) in its variable $Seen_k$ (*Action 2* of P_k, $k \neq i$, or *Action 1* of P_i). Thus, for each node j that is reachable from i, either j was included in the variable $Incycle_j$ or an ordered pair (k, j) was included in the variable $Seen_k$ for some node k that is reachable from i. The values of the variables $Incycle_k$ and $Seen_k$ are passed up in the DST with the $parent\_ack$ message. Each node receiving these values adds them to their corresponding local variables and passes those values to its parent, and so on (*Action 2*, *5* of P_k). Thus, for each node j that is reachable from i, either j is included in $Incycle_i$ or an ordered pair (k, j) is included in $Seen_i$ before P_i executes *Action 5*.

4.1 Correctness Proof

Lemma 1. *If node i is in a knot, then P_i declares "knot detected."*

Proof. Note that P_i declares "knot detected" only after the execution of *Action 5*. Thus, it is sufficient to prove that, after the execution of *Action 5*, $Seen_i = \emptyset$; this is because, after executing *Action 5*, node i declares "*knot detected*" only if $Seen_i = \emptyset$. Suppose $(l, m) \in Seen_i$ just before the execution of *Action 5*, where $1 \leq l, m \leq n$.

Claim. (l, m) will be removed from $Seen_i$ during the execution of *Action 5*. □

Proof of Claim. $(l, m) \in Seen_i$ implies that node m has sent either $seen\_ack(m)$ or $parent\_ack(Seen_m, Incycle_m)$ with $Incycle_m = \emptyset$ to node l (*Action* 2 and 3 of P_l, $l \neq i$ or *Action* 1 and 2 of P_i, $l = i$). However, since node i is inside a knot, there is a path $\{m_1(= m), m_2, \cdots, m_k(= i)\}$ from node m to node i in the communication graph.

Clearly, when i received the $detect\_knot(i, m_{k-1})$ message from node m_{k-1}, it must have sent $cycle\_ack(i)$ acknowledgment to m_{k-1} (*Action 4* of P_i). After receiving this acknowledgment, node m_{k-1} must have included i as well as m_{k-1} in the set $Incycle_{m_{k-1}}$ before sending a $parent\_ack$ acknowledgment to its parent (*Actions 4* and *5* of $P_{m_{k-1}}$). This means node $m_{k-1} \in Incycle_i$ before *Action 5* was executed at node i. Let j be the smallest integer $(1 \leq j < k-1)$ such that $m_j \notin Incycle_i$ before the execution of *Action 5* at node i (if there exists no such integer, then $m_j \in Incycle_i \; \forall \; j$; this implies $m_1 (= m) \in Incycle_i$, which implies that (l, m) would have been removed from $Seen_i$ during the execution of *Action 5* of P_i and this proves our claim in this case). This means, for each integer p $(1 \leq p \leq j)$, node m_p has either received $seen\_ack_{m_{p+1}}$ or $parent\_ack(Seen_{m_{p+1}}, Incycle_{m_{p+1}})$ with $Incycle_{m_{p+1}} = \emptyset$ from m_{p+1} (*Actions* 2 and 3 of P_{m_p} and *Action 1* of P_i). This means $(m_p, m_{p+1}) \in Seen_i$ for all p such that $(1 \leq p \leq j)$ before the execution of *Action 5*. Since $m_{j+1} \in Incycle_i$, the ordered pairs (m_j, m_{j+1}), (m_{j-1}, m_j), $\ldots$ $(m_1 (= m), m_2)$ and then (l, m) would have been removed from $Seen_i$ in succession during the execution of *Action 5*. This proves our claim.

Since (l, m) is an arbitrary element in $Seen_i$, we have proven that all the elements of $Seen_i$ would have been removed during the execution of *Action 5*. So, $Seen_i = \emptyset$ after the execution of *Action 5* and, hence, P_i declares "*knot detected*" if i is indeed in a knot. □

Lemma 2. *If P_i declares "*knot detected,*" then node i is in a knot.*

Proof. P_i declares "*knot detected*" only if, after the execution of *Action 5*, $Seen_i = \emptyset$. So, it is sufficient to prove that $Seen_i = \emptyset$ after the execution of *Action 5* implies that node i is in a knot. From Observation 4, for each node j that is reachable from i, either j was included in $Incycle_i$ or an ordered pair (k, j) was included in $Seen_i$ for some node k that is reachable from i. Note that, during the execution of *Action 5*, every node in the set $Incycle_i$ is on a cycle with i and an ordered pair $(k, j) \in Seen_i$ is removed from $Seen_i$ during the execution of *Action 5* only if $j \in Incycle_i$.

So, after the execution of *Action 5*, $Seen_i = \emptyset$ implies that all the nodes reachable from i are on a cycle with node i. Hence, node i is in a knot. □

Theorem 1. *When Process P_i terminates, P_i declares "*knot detected*" if and only if node i is in a knot.*

Proof. Follows from Lemmas 1 and 2. □

Theorem 2. *When Process P_i terminates after executing Action 5:*

1. *If P_i declares "*knot detected,*" its variable $Cycle\_nodes_i$ contains the set of all the nodes that are involved in the knot containing i.*
2. *If P_i declares "*no knot detected,*" then its variable $Cycle\_nodes_i$ contains the set of all those nodes that lie on a cycle passing through i.*

Proof. During the execution of *Action 5* by Process P_i, a node j is added to the variable $Cycle\_nodes_i$ if and only if it is reachable from i and also lies in a cycle containing i (see proof of Lemma 1). If $Seen_i = \emptyset$, then each node that is reachable from i lies on a cycle containing i. This means $Cycle\_nodes_i$ contains precisely those nodes that are involved in the knot. Thus, if P_i declares "*knot detected,*" its variable $Cycle\_nodes_i$ contains the set of all the nodes that are involved in the knot containing i.

A node that is reachable from i is added to $Cycle\_nodes_i$ if and only if it lies on a cycle containing i (see proof of Lemma 1). Thus, if P_i declares "*no knot detected,*" then its variable $Cycle\_nodes_i$ contains the set of all those nodes that lie on a cycle containing i. □

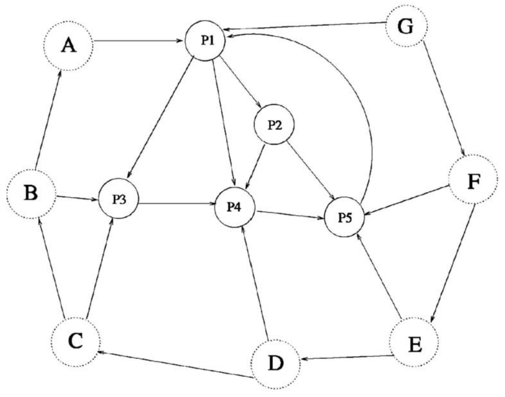

Fig. 3. A communication graph.

Theorem 3. *The algorithm is deadlock-free.*

Proof. We prove that the algorithm is deadlock-free by proving that circular wait does not occur. After forwarding a *detect_knot* message, a node waits for reply. If the node that received the *detect_knot* message has already seen the *detect_knot* message, then it sends a $seen\_ack()$ or *cycle_ack* immediately; otherwise, it forwards the message along all its outgoing edges and waits until it receives replies from all its immediate successors. If there is a circular wait, that means there is a process which has already seen the *detect_knot* message, but it is not sending a reply to a newly received *detect_knot* message; this is impossible because a node that has seen the *detect_knot* message replies to a newly received *detect_knot* message immediately. So, circular wait does not occur and, hence, the algorithm is deadlock-free. □

4.2 Example Executions of the Algorithm

In this section, we illustrate the execution of the algorithm with an example. Consider the communication graph of a distributed system shown in Fig. 3. In this graph, P1, P2, P3, P4, and P5 are nodes. Each of A, B, C, D, E, F, and G is a subgraph, which may contain knots within them; we assume that there is no incoming edge to any of the nodes in these subgraphs from any of the nodes in the set {P1, P2, P3, P4, P5}. Nodes P1, P2, P3, P4, and P5 are clearly in a knot. When P1 wants to find out if it is involved in a knot, it sends a knot detection message to all its immediate successors in the graph which in turn propagate the knot detection message to all their immediate successors and so on. So, the knot detection message propagates to only nodes involved in the knot if P1 is in a knot. We want to empahzize that the knot detection message does not propagate to any of the nodes in the subgraphs A ... G,

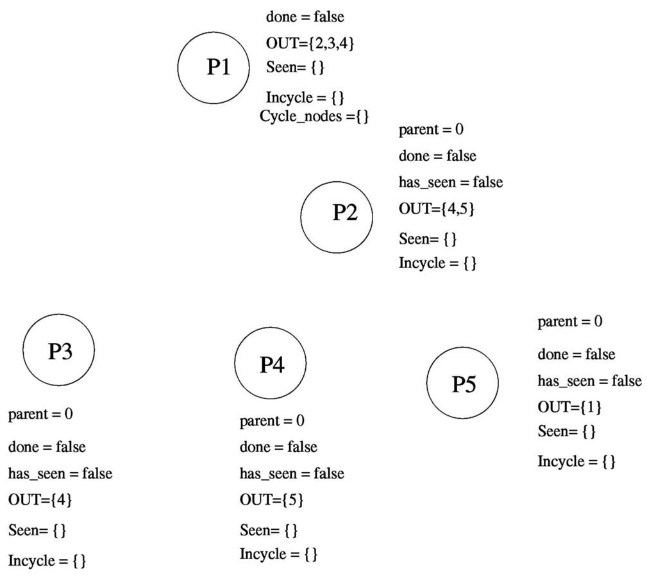

Fig. 4. Initial values of the variables at various processes at various nodes.

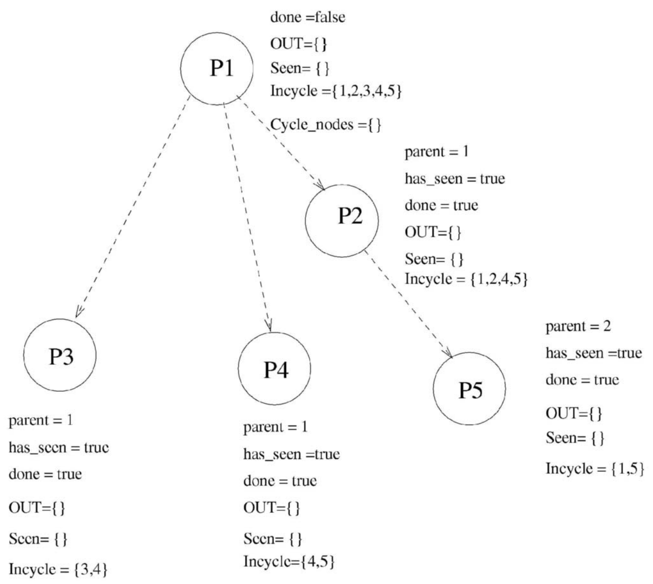

Fig. 5. State of the variables at all locations just before the execution of *Action 5* at P_1.

thus reducing the propagation of knot detection message to the entire wait-for-graph, unlike Misra and Chandy's algorithm [19].

Suppose node P1 initiates the knot detection by sending a $detect\_knot(1,1)$ message. The initial values of the variables of processes P1, P2, P3, P4, and P5 are shown in Fig. 4. We left out the suffixes for the variables in the figure as it is clear from the figure which variable belongs to which process. We do not show the rest of the communication graph because our algorithm does not send messages to nodes that are not reachable from the initiator. The following is one scenerio of propagation of the knot detection messages and acknowledgments.

1. P1 sends $detect\_knot(1,1)$ to P2, P3, and P4.
2. P2 receives $detect\_knot(1,1)$ from P1 and sends $detect\_knot(1,2)$ to P5 and P4.
3. P5 receives $detect\_knot(1,2)$ from P2 and then sends $detect\_knot(1,5)$ to P1.
4. P1 receives $detect\_knot(1,5)$ from P5 and sends $cycle\_ack(1)$ to P5.
5. P5 receives $cycle\_ack(1)$ from P1 and adds 1 to its local variable $Incycle$.
6. P4 receives $detect\_knot(1,1)$ from P1 and sends $detect\_knot(1,4)$ to P5.
7. P5 receives $detect\_knot(1,4)$ from P4 and sends $cycle\_ack(5)$ to P4.
8. P4 receives $cycle\_ack(5)$ from P5 and adds 5 to its local variable $Incycle$.
9. Having received replies from all its immediate successors, P5 adds 5 to its local variable $Incycle$ and sends $parent\_ack(\{\},\{1,5\})$ to P2.
10. P2 receives $parent\_ack(\{\},\{1,5\})$ from P5 and adds $\{1,5\}$ to its local variable $Incycle$.
11. P4 receives $detect\_knot(1,2)$ from P2 and sends $cycle\_ack(4)$ to P2.
12. P2 receives $cycle\_ack(4)$ from P4, adds 4 as well as 2 to its local variable $Incycle$, and sends $parent\_ack(\{\},\{1,2,4,5\})$ to P1.
13. P3 receives $detect\_knot(1,1)$ from P1 and sends $detect\_knot(1,3)$ to P4.
14. P4 adds 4 to its local variable $Incycle$ and sends $parent\_ack(\{\},\{4,5\})$ to P1.
15. P4 receives $detect\_knot(1,3)$ from P3 and sends $cycle\_ack(4)$ to P3.
16. P3 receives $cycle\_ack(4)$ from P4, adds 4 as well as 3 to its local variable $Incycle$, and sends $parent\_ack(\{\},\{3,4\})$ to P1.
17. P1 receives $parent\_ack(\{\},\{3,4\})$ from P3 and adds $\{3,4\}$ to its local variable $Incycle$.
18. P1 receives $parent\_ack(\{\},\{4,5\})$ from P4 and adds $\{4,5\}$ to its local variable $Incycle$.
19. P1 receives $parent\_ack(\{\},\{1,2,4,5\})$ from P2 and adds $\{1,2,4,5\}$ to its local variable $Incycle$.
20. P1 computes the value of $Cycle\_nodes$ after receiving the $parent_ack$s from P2, P3, and P4 (see *Action 5* of process P_i in the algorithm).

The values of the local variables of various processes after the exchange of the messages in Steps 1 through 19 described above are shown in Fig. 5. The edges in Fig. 5 are the edges of the DST. This is also reflected by the values of the $parent$ variables of each process. Now, P_1 computes the value of $Cycle\_nodes$ in Step 20 after receiving $parent_ack$s from all its children. After that, P_1 decides if it is in a knot by examining its variable $Seen$. If $Seen = \emptyset$, as it is in this case, it declares "knot detected." The way in which messages have propagated in the network kept the computation of the initiator to a minimum.

However, messages could have propagated along the edges of the wait-for-graph in such a way that the initiator comes to know about the set of nodes that lie on only one cycle. In this case, the initiator will have to do more processing (*Action 5*) to determine if all nodes that are

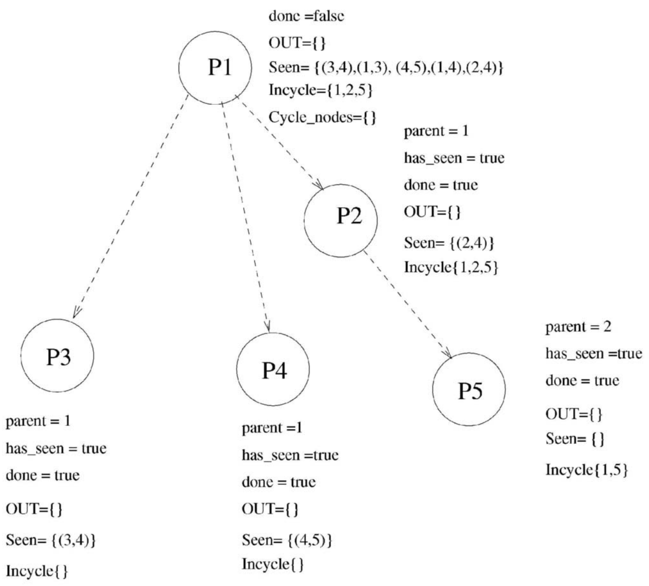

Fig. 6. State of the variables at all locations in another execution just before the execution of *Action 5* at P1.

reachable from it lie on a cycle passing through the initiator. Such a scenario is illustrated below (see Fig. 6):

1. P1 sends $detect\_knot(1,1)$ to P2, P3, and P4.
2. P2 receives $detect\_knot(1,1)$ from P1 and sends $detect\_knot(1,2)$ to P5 and P4.
3. P5 receives $detect\_knot(1,2)$ from P2 and sends $detect\_knot(1,5)$ to P1.
4. P4 receives $detect\_knot(1,1)$ from P1 and sends $detect\_knot(1,4)$ to P5.
5. P5 receives $detect\_knot(1,4)$ from P4 and sends $seen\_ack(5)$ to P4.
6. P4 receives $seen\_ack(5)$ from P5 and adds $(4,5)$ to the variable $Seen$.
7. P3 receives $detect\_knot(1,1)$ from P1 and sends $detect\_knot(1,3)$ to P4.
8. P4 receives $detect\_knot(1,3)$ from P3 and sends $seen\_ack(4)$ to P3.
9. P4 receives $detect\_knot(1,2)$ from P2 and sends $seen\_ack(4)$ to P2.
10. P2 receives $seen\_ack(4)$ from P4 and adds $(2,4)$ to its variable $Seen$.
11. P3 receives $seen\_ack(4)$ from P4 and adds $(3,4)$ to the variable $Seen$.
12. P3 sends $parent\_ack(\{(3,4)\},\{\})$ to P1.
13. P1 receives $parent\_ack(\{(3,4)\},\{\})$ from P3 and adds $(3,4)$ and $(1,3)$ to its variable $Seen$.
14. P4 sends $parent\_ack(\{(4,5)\},\{\})$ to P1.
15. P1 receives $parent\_ack(\{(4,5)\}\{\})$ from P4 and adds $(4,5)$ and $(1,4)$ to its variable $Seen$.
16. P1 receives $detect\_knot(1,5)$ from P5 and sends $cycle\_ack(1)$ to P5.
17. P5 receives $cycle\_ack(1)$ from P1 and adds 1 to its variable $Incycle$.
18. P5 adds 5 to its variable $Incycle$ and sends $parent\_ack(\{\},\{1,5\})$ to P2.
19. P2 receives $parent\_ack(\{\},\{1,5\})$ from P5 and adds 1 and 5 to its variable $Incycle$.
20. P2 adds 2 to its $Incycle$ variable and sends $parent\_ack(\{(2,4)\},\{1,2,5\})$ to P1.
21. P1 receives $parent\_ack(\{(2,4)\},\{1,2,5\})$ from P2, adds 1, 2, and 5 to its variable $Incycle$ and $(2,4)$ to its variable $Seen$. The contents of the local variables at this point are shown in Fig. 6.

After Step 21 above, *Action 5* of P_i is enabled at P1 and, for each $j \in Incycle$, P1 deletes j from $Incycle$, adds j to $Cycle\_nodes_i$, and then, for each k such that $(k,j) \in Seen$, it removes (k,j) from $Seen$ and adds k to $Incycle$. Eventually, $Seen = \emptyset$ and $Cycle\_nodes = \{1,2,3,4,5\}$; P1 reports detection of a knot. At this point, the variable $Cycle\_nodes$ at P1 contains precisely the nodes involved in the knot.

4.3 Comparison with Related Work

Misra and Chandy's [19] knot detection algorithm is one of the earliest ones. In their algorithm, a node initiating knot detection sends a *Suc* message along all outgoing edges and a *Pred* message along all incoming edges. Each of the *Suc* messages received by a node is sent along all outgoing edges of the receiving node; similarly, each of the *Pred* messages is forwarded by the receiving node along all incoming edges. A node receiving these messages also sends an *ack* back to the sender to report the results of the search. Thus, the number of messages exchanged in this algorithm is $4e'$, where e' is the number of edges in the subgraph formed by the nodes that are reachable from the initiator as well as the nodes from which the initiator can be reached. Thus, messages propagate to nodes from which the initiator can be reached, which may not be part of the knot. In our knot detection algorithm, knot detection messages propagate to only nodes that are reachable from the initiator and each node receiving the knot detection message sends an *ack* back to the node from which it received the knot detection message. Thus, our algorithm requires only $2e$ messages, where e is the number of edges in the subgraph formed by the nodes that are reachable from the initiator. In general, e will be very small when compared to e'. Experimental measurements of Gray et al. [9] have shown that 90 percent of all deadlocks involve only two nodes [24]. Thus, our algorithm will have a very low message complexity when compared to Misra and Chandy's algorithm. Moreover, unlike Misra and Chandy's algorithm, our algorithm not only detects if a node is involved in a knot, but also collects the set of nodes involved in the knot, which can be very useful for deadlock resolution.

Gambosi et al. [7] present a distributed algorithm for detection and removal of deadlocks in store-and-forward networks. Their algorithm is based on the fact that the existence of knot in the Site Blocking Graph (SBG)[2] is a necessary and sufficient for the existence of deadlock in store-and-forward networks. Their algorithm only presents a method for constructing the SBG, but does not give any efficient method for detecting knot in the SBG.

Cidon's knot detection algorithm exploits the property that knots are strongly connected graphs [4]. A group of nodes that are found to be strongly connected is called a

2. SBG is defined as the directed graph whose vertices are the set of sites N and edges E, are defined by $E = \{(i,j) : (i,j) \in N'XN$ *and* $\exists$ *at least one packet at site* i *whose next destination is site* $j\}$, where N' is the set of blocked sites.

cluster. This algorithm looks for cycles of clusters and merges them to form new clusters. If a cluster with no outgoing link is detected, then a knot is detected. The message complexity of their algorithm is $O(n\ log(n)\ +\ e)$; however, the number of messages required for knot detection in our algorithm is $2e$, where e is the number of edges in the subgraph consisting of the nodes reachable from the initiator. Boukerche and Tropper [1] proposed an algorithm for detecting knots and cycles. Their algorithm diffuses knot detection messages along all outgoing edges of the initiator and replies are propagated back. Based on the replies received, the initiator determines if it is involved in a knot or is involved in cycles only. However, their algorithm does not determine which nodes are involved in the knot or cycles. How the information about the nodes involved in the knot gathered by our algorithm can be used in deadlock resolution is discussed next.

4.4 Application to Deadlock Resolution

We describe briefly how the proposed knot detection algorithm can be used for deadlock resolution. First, we describe how deadlock can be resolved in store and forward networks. In store and forward networks, deadlock occurs if there is a knot in the buffer graph [7], [11]. In other words, all the buffers at all the nodes in the knot are full. We assume that each node has a sufficient number of buffers reserved for receiving control messages, which are used for storing the messages related to the knot detection algorithm while knot detection is in progress. To detect and resolve deadlock in the case of store and forward networks, the following actions can be taken:

- The initiator finds if it is in a knot by running the knot detection algorithm. If it finds itself to be in a knot, it takes the following steps:
  - **Step 1:** It discards one or more of the packets in its buffers so that it can receive packets from nodes that are trying to send packets to it.
  - **Step 2:** Allows one of the nodes in the knot that is waiting to send packets to transmit packets.
  - **Step 3:** Sends the ids of all the nodes in the knot to every other node in the knot.
  - **Step 4:** Upon receiving the ids of all the nodes in the knot, a node that has been able to send packets as a result of the deadlock resolution process in Step 2 will be able to receive packets from other nodes in the knot that are waiting to send packets to it and this process can continue at all nodes until the deadlock is resolved.
- Nodes involved in the knot receive packets only from nodes that are involved in the knot until the deadlock is resolved. A node in the knot concludes that deadlock has been resolved as soon as it finds that there is no node in the knot that tries to send packets to it. We can achieve this by using a timer, i.e., a process waits for a fixed time period before it receives packets from a node that is not in the knot. (In some store-and-forward networks, a node sends a ready to send (RTS) packet to the receiver before sending data packets and waits until it receives a ready to receive (RTR) packet from the receiver before sending the data packets to it. In such networks, at any time, a node knows about all nodes that are trying to send packets to it, so, a node can easily find out when there is no node that is trying to send packets to it). At this point, it starts receiving packets from nodes outside the knot.

We presented one way of resolving deadlock in store and forward networks above. We do not claim this is the most efficient way to resolve deadlock. Efficient deadlock resolution takes into consideration several factors such as which node needs to discard packets, priority of the packets, etc. Our focus in this paper is not to design an efficient deadlock resolution algorithm, but only to design an efficient knot detection algorithm. To resolve a deadlock, it is important to know the set of nodes that are involved in the deadlock.

In the OR model, the existence of a knot in the communication graph is necessary and sufficient to cause deadlock. In this model, the following actions can be taken to detect and resolve deadlock.

- The initiator finds if it is in a knot by running the knot detection algorithm. If it finds itself in a knot,
  - it rolls back and releases the resources it is holding,
  - grants the resources released to one or more requesting processes in the knot, and
  - sends the ids of the nodes involved in the knot to all the nodes in the knot (this is possible because the initiator knows exactly what nodes are involved in the knot).
- Nodes involved in the knot allocate resources only to nodes that are involved in the knot until the deadlock is resolved. A node in the knot concludes that deadlock has been resolved as soon as it finds that there is no node in the knot which is requesting a resource from it. At this point, it starts honoring requests for resources from nodes outside the knot. Note that, after a node in the knot becomes active, it can finish and grant the resources it was holding to one or more nodes in the knot that were requesting from it. Thus, at least one other node in the knot can become active after a node finishes and this can continue. Thus, all the nodes in the knot will eventually become active and the deadlock will be resolved.

5 Performance of the Algorithm

When we look at *Action 5* of process P_i in the algorithm, it appears as though the initiator has to do a lot of processing after receiving replies from all its immediate successors. However, as we saw in Section 4.2, it is clear that, in the best case, the initiator does not have to do much processing after it receives replies from all its immediate successors in order to determine if it is in a knot; otherwise, the initiator will have to do more processing than the other nodes to determine if the initiator is in a knot as well as to determine the nodes that are part of the knot. To distribute the work

```
$\neg done_k \wedge OUT_k = \emptyset \wedge has\_seen_k$   $\longrightarrow$ if $Incycle_k \neq \emptyset$ then          /*Action 5*/
        $Incycle_k = Incycle_k \cup \{k\}$;
    for each $j \in Incycle_k$ do
        $Incycle_k := Incycle_k - \{j\}$;
        $Cycle\_nodes_k := Cycle\_nodes_k \cup \{j\}$;
        for each $m$ do
            if $(m, j) \in Seen_k$ then
                $Incycle_k := Incycle_k \cup \{m\}$;
                $Seen_k := Seen_k - \{(m, j)\}$;
    $Incycle_k := Cycle\_nodes_k$;
    send $parent\_ack(Seen_k, Incycle_k)$ to $parent_k$;
    $done_k := true$;
```

Fig. 7. Enhancement to the algorithm.

more evenly among all nodes, we can change *Action 5* of process P_k as follows: Like the initiating process P_i, every other process P_k has a variable $Cycle\_nodes_k$ initialized to the empty set and is used to determine the nodes on cycle with the initiator based on the current information as illustrated above (see Fig. 7):

If we modify *Action 5* of process P_k as in Fig. 7, then, before sending $parent\_ack$, each process computes all the nodes involved in a cycle with the initiator based on the current information. When a node k executes this modified *Action 5*, it is possible that the set $Seen_k$ becomes empty. If $Seen_k$ is empty in the $parent_ack$s sent by all processes, then the initiator does not have to do any processing before declaring that it is involved in a knot. This reduces the amount of work done by the initiator.

Next, we discuss the performance of the algorithm with respect to the following two metrics: 1) message complexity: the number of messages exchanged in an execution of the algorithm and 2) time delay: the time interval from an invocation of the algorithm till its termination.

5.1 Message Complexity

When the algorithm declares "knot detected," each edge in the subgraph generated by the nodes reachable from i is traversed twice—once by a $detect\_knot$ message and once by an ack message (a $parent\_ack$, a $seen\_ack$, or $cycle\_ack$). Therefore, message complexity of the algorithm in this case is $2e$, where e is the number of edges in the subgraph formed by the vertices reachable from i. Experimental measurements of Gray et al. [9] have shown that 90 percent of all deadlocks involve only two nodes [24]. So, even though the messages in our algorithm are longer than the algorithm of Boukerche and Tropper [1], it will not cause much overhead because it propagates only to the nodes involved in the knot and the knot, if it exists, will be very small, in general; moreover, our algorithm has the advantage of finding all nodes involved in the knot which can help in resolving the deadlock.

Unlike Misra and Chandy's [19] algorithm, messages travel only among the nodes reachable from the initiator. The message complexity of our algorithm in the worst case is $2e$, where e is the number of edges in the subgraph whose vertices are reachable from the initiator. The message complexity of Misra and Chandy's algorithm in the worst case is $4e'$, where e' is the number of edges in the subgraph whose vertices are either reachable from the initiator or have a path leading to the initiator. Boukerche and Tropper's [1] algorithm requires $2e$ messages; however, it does not find the nodes involved in the knot. Our algorithm, like Boukerche and Tropper's [1] algorithm, also requires $2e$ messages; however, our algorithm not only detects the existence of a knot, but also collects all the nodes involved in the knot, which we have demonstrated to be useful in deadlock resolution.

5.2 Time Delay

A $detect\_knot$ message must travel from the initiator to the farthest leaf node in the DST (dst_{max} message hops), the leaf node must propagate the message to its immediate successors and collect all acks (2 message hops), and then send $parent\_ack$ message to its parent which must travel from the leaf to the initiator with modifications on the way by other nodes. Thus, the time delay in this case is $2(dst_{max} + 1)$ message hops.

6 Concluding Remarks

Knot detection in a communication graph is an important problem and finds applications in several domains of distributed systems design, such as store-and-forward networks, distributed simulation, and distributed databases. In this paper, we presented a distributed algorithm to detect if a given node in a communication graph is in a knot. The algorithm requires $2e$ messages and $2(dst_{max} + 1)$ message hops delay to detect if a node is involved in a knot (e is the number of edges in the subgraph generated by the nodes reachable from the initiator and dst_{max} is the maximum of the distances from the initiator to the nodes that are reachable from the initiator). The algorithm helps not only in detecting if a node is involved in a knot, but also finds exactly which nodes are involved in the knot. We also illustrated with examples how the information about the nodes involved in the knot can be used for deadlock resolution. Another advantage of our algorithm is that our algorithm finds all nodes that are in a cycle with the initiator if the initiator is not in a knot. Nodes involved in a cycle help in the determination of deadlocked nodes in the AND request model.

ACKNOWLEDGMENTS

The authors thank the anonymous referees and the associate editor for their detailed comments and valuable suggestions on an earlier version of the paper. The authors also thank Raphael Finkel for his comments on a preliminary version of this paper. A preliminary version of this paper appears in the *IEEE Proceedings of the 31st International Conference on Parallel Processing* [17]. This research was supported in part by the US National Science Foundation, CAREER Award # CCR-9983584.

REFERENCES

[1] A. Boukerche and C. Tropper, "A Distributed Graph Algorithm for the Detection of Local Cycles and Knots," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 8, pp. 748-757, Aug. 1998.

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

[3] E. Chang, "Decentralized Deadlock Detection in Distributed Systems," technical report, Univ. of Victoria, Victoria, B.C., Canada, 1980.

[4] I. Cidon, "An Efficient Distributed Knot Detection Algorithm," *IEEE Trans. Software Eng.,* vol. 15, no. 5, pp. 644-649, May 1989.

[5] E.W. Dijkstra, "In Reaction to Ernest Chang's Deadlock Detection," EWD702, Plataanstraat 5, 5671 AL Nuenen, The Netherlands, 1979.

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

[7] G. Gambosi, D.P. Bovet, and D.A. Menasce, "A Detection and Removal of Deadlocks in Store and Forward Communication Networks," *Performance of Computer-Comm. Systems,* H. Rudin and W. Bux, eds. pp. 219-229, North-Holland: Elsevier Science, 1984.

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

[9] J.N. Gray, P. Homan, H.F. Korth, and R.L. Obermarck, "A Straw Man Analysis of the Probability of Waiting and Deadlock in Database Systems," Technical Report RJ 3066, IBM Research Laboratory, San Jose, Calif., 1981.

[10] B. Groselj and C. Tropper, "The Distributed Simulation of Clustered Processes," *Distributed Computing,* vol. 4, pp. 111-121, 1991.

[11] K. Gunther, "Prevention of Deadlocks in Packet-Switched Data Transport Systems," *IEEE Trans. Comm. (special issue on congestion control in computer networks),* pp. 512-524, June 1981.

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

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

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

[15] A.D. Kshemkalyani and M. Singhal, "Efficient Detection and Resolution of Generalized Distributed Deadlocks," *IEEE Trans. Software Eng.,* vol. 20, no. 1, pp. 43-54, Jan. 1994.

[16] L. Liu and C. Troppor, "Local Deadlock Detection in Distributed Simulation," *Proc. Distributed Simulation Conf.,* pp. 64-69, 1990.

[17] D. Manivannan and M. Singhal, "A Distributed Algorithm for Knot Detection in a Distributed Graph," *Proc. 31st IEEE Int'l Conf. Parallel Processing,* pp. 485-492, Aug. 2002.

[18] J. Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys,* vol. 18, no. 1, pp. 39-65, Mar. 1986.

[19] J. Misra and K. Chandy, "A Distributed Graph Algorithm: Knot Detection," *ACM Trans. Programming Languages and Systems,* pp. 678-686, 1982.

[20] N. Natarajan, "A Distributed Scheme for Detecting Communication Deadlocks," *IEEE Trans. Software Eng.,* vol. 12, no. 4, pp. 531-537, Apr. 1986.

[21] M. Singhal, "Deadlock Detection in Distributed Systems," *Computers,* pp. 37-48, Nov. 1989.

[22] M. Singhal, "A Class of Deadlock-Free Maekawa-Type Mutual Exclusion Algorithms for Distributed Systems," *Distributed Computing,* vol. 4, no. 3, pp. 131-138, Feb. 1991.

[23] M. Singhal and N.G. Shivaratri, *Advanced Concepts in Operating Systems.* McGraw-Hill, 1994.

[24] P.K. Sinha, *Distributed Operating Systems Concepts and Design.* IEEE Press, 1997.

[25] I. Terekhov and T. Camp, "Time Efficient Deadlock Resolution Algorithms," *Information Processing Letters,* vol. 69, pp. 149-154, 1999.

D. Manivannan received the BSc degree in mathematics with special distinction from the University of Madras, Madras (Chennai), India. He received the MS degree in mathematics, the MS degree in computer science, and the PhD degree in computer science, from The Ohio State University, Columbus, in 1992, 1993, and 1997, respectively. He is an assistant professor in the Department of Computer Science at the University of Kentucky, Lexington, Kentucky. His areas of research interests include distributed systems, mobile computing systems, and wireless networks. He has served as a program committee member for several international conferences and has served as a reviewer for several international journals and conferences. He is a recipient of the CAREER Award from the US National Science Foundation. He is a member of the ACM, IEEE, and IEEE Computer Society.

Mukesh Singhal received the bachelor of engineering degree in electronics and communication engineering with high distinction from the Indian Institute of Technology, Roorkee, India, in 1980, and the PhD degree in computer science from the University of Maryland, College Park, in May 1986. He is a full professor and Gartener Group Endowed Chair in Network Engineering in the Department of Computer Science at the University of Kentucky, Lexington. From 1986 to 2001, he was a faculty member in the Computer and Information Science Department at the Ohio State University. His current research interests include distributed systems, mobile computing, computer networks, computer security, and performance evaluation. He has published more than 160 refereed articles in these areas. He has coauthored three books titled *Data and Computer Communications: Networking and Internetworking*, CRC Press, 2001, *Advanced Concepts in Operating Systems*, McGraw-Hill, New York, 1994, and *Readings in Distributed Computing Systems*, IEEE Press, 1993. He is a fellow of the IEEE and a member of the IEEE Computer Society. He is currently serving on the editorial board of the *IEEE Transactions on Knowledge and Data Engineering* and *IEEE Transactions on Computers*. From 1998 to 2001, he served as the program director of the Operating Systems and Compilers Program at the US National Science Foundation.