Local Distributed Deadlock Detection by Cycle Detection and Clustering

ISRAEL CIDON, MEMBER, IEEE, JEFFREY M. JAFFE, MEMBER, IEEE, AND MOSHE SIDI, MEMBER, IEEE

Abstract—**A distributed algorithm for the detection of deadlocks in store-and-forward communication networks is presented. At first, we focus on a static environment and develop an efficient knot detection algorithm for general graphs. The knot detection algorithm uses at most $O(n^2 + m)$ messages and $O(\log(n))$ bits of memory to detect all deadlocked nodes in the static network. Using the knot detection algorithm as a building block, a deadlock detection algorithm in a dynamic environment is developed. This algorithm has the following properties: It detects all the nodes which cause the deadlock. The algorithm is triggered only when there is a potential for deadlock and only those nodes which are potentially deadlocked perform the algorithm. The algorithm does not affect other processes at the nodes.**

Index Terms—**Clustering, computer networks, cycle detection, deadlock detection, distributed algorithms.**

I. Introduction

DEADLOCKS in computer systems and transaction systems have been extensively studied in the literature [1]. Various schemes to prevent the occurrence of deadlocks have been presented [2], [3]. Distributed deadlock detection algorithms for detecting resource deadlocks in computer systems have been suggested [4]–[6]. Distributed algorithms for detection [7]–[9] and resolution [7], [9] of buffer deadlocks have been described.

In a network without deadlock prevention, having a distributed deadlock detection algorithm is invaluable. Detection of deadlocks could be a key to notifying a "help-desk," that manual intervention is necessary. If one also has a distributed deadlock resolution scheme [7], [9], then deadlock detection is the first step towards starting such a scheme. Finally, in a network in which throwing away messages is an acceptable resolution of a "bad situation," deadlock detection is a catalyst for determining when and where messages should be discarded.

Various properties are desirable in distributed deadlock detection algorithms. When the algorithm is to detect buffer deadlocks, then it should use only a small (preferably fixed) portion of the nodal storage [8], [9]. Algorithms that accumulate a large amount of information, such as the "wait-for graph" [7], would have insufficient storage in practice. Preferably, deadlock detection algorithms should be local, so that if a potential deadlock occurs in a portion of the network, nodes in remote portions do not ever find out that deadlock detection algorithm is in progress [8]. By contrast, the algorithm in [9] is global, as all nodes of the network are involved in the detection algorithm.

In addition, the detection algorithm should be efficient and should limit the amount of control messages transmitted during its execution. The algorithm should be started only when necessary, should find all deadlocked nodes in the network, and should not interfere with the normal operation of the network.

In this paper we present a new distributed deadlock detection algorithm with all the above properties. Our approach is to describe the algorithm as it applies to computer networks. However, with slight modifications this algorithm can be applied to transaction systems too.

The algorithm that we present, uses only $O(\log(n))$ bits of memory per node, is local, is started only upon events that may lead to deadlock, and does not interfere the normal operation of the network. In a static network with n' potentially deadlocked nodes and m' links that are attached to such nodes, the number of control messages transmitted for detecting *all* the deadlocked nodes is $O(n'^2 + m')$ in the worst case, better than the previously known deadlock detection algorithms.

A main feature of our algorithm is that it finds out all deadlocked nodes in a single invocation of the algorithm, as opposed to the "search" type algorithms [5], [6], [8], [11], [12] in which each node starts an independent invocation of the search to find if it is deadlocked. Here we use a new approach to deadlock detection using cycle detection and clustering techniques which are similar to [13], [14].

II. The Model

We adopt here the model in [8], [9], [15]. For completeness we give a concise description.

Manuscript received January 31, 1986; revised June 16, 1986.

I. Cidon was with the Department of Electrical Engineering, Technion—Israel Institute of Technology, Haifa, 32000, Israel. He is now with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

J. M. Jaffe is with the IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598. Most of this work was performed while he was on leave to the IBM Scientific Center, Technion City, Haifa 32000, Israel.

M. Sidi is with IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, on leave from the Department of Electrical Engineering, Technion—Israel Institute of Technology, Haifa 32000, Israel.

IEEE Log Number 8611357.

0098-5589/87/0100-0003$01.00 © 1987 IEEE

A. Network Model

A *network* consists of a set of *communication nodes* N, and a set of links L that interconnect nodes of N. At any point in time, any node may create a new *packet*, which we assume to be of variable but bounded size.

When a node receives a packet (created by itself or sent from another node), it determines a next node, based on information from the packet and the routing tables. The type of routing does not concern us—only the fact that the packet and (possibly dynamic) tables uniquely determine a next adjacent node. Once the next node is determined, the node queues up a packet for that next node.

Regarding links, the following properties are assumed: they are FIFO (do not lose, reorder or duplicate messages); there is no deterministic bound on the amount of time that it takes a message to traverse a link; any message placed on the link arrives at the other side of the link in finite time; links never fail.

B. Model of a Communication Node

We assume that the storage available at a node is finite. The storage is assumed to be divided into three parts. The first is the storage needed for *overhead* that includes the code for the machine, data structures, variables, control blocks, etc. In addition the node may reserve storage (message buffers and variables) for specific emergency measures such as deadlock detection. This must be a *fixed amount of storage* as it must be enough irrespective of network size. Note that messages may carry node identities needed for routing and control, and thus, a message buffer should be of size $O(\log(|N|))$ bits (where $|N|$ is the cardinality of N). However, in practice, since the network size is reasonably bounded, a fixed number of bits is preallocated to identify a node, assuming it is enough to distinctly identify all potential nodes within the network.

Next one considers the maximum number of links that may be assigned to a node and assigns a fixed amount of storage per link. This storage is needed to control the physical link. Since storage in any case must be allocated on a per link basis, we also allow *a fixed amount of storage per link* to be reserved for deadlock detection. Thus the total storage allowable for the entire deadlock detection procedure is a constant number of buffers, plus a constant number of buffers per adjacent link. Once storage has been reserved for overhead and the links, all other storage is left over for message buffers for transit traffic.

Regarding the management of transit message buffers we assume that a link control protocol prevents the sending of a message to an adjacent node, until a free buffer is allocated for this message at the adjacent node. Similarly, if the packet was created at the given node, if there is storage, a buffer is allocated for it, and if not, the packet stays in the same machine, but does not enter the communication subsystem.

C. Deadlocks

A node is *FULL* if all of its transit buffers are occupied with packets destined away from this node, otherwise it is *NOT_FULL*. Let (N, L) be a directed graph where N is the set of nodes in the network and L a set of directed links, where a directed link (i, j) indicates that node i has at least one packet whose next node is j. A *tie* T, in (N, L) is a set of *FULL* nodes with no links directed from T to $N - T$. We say that a *deadlock* exists in the network at time t, if a tie exists in (N, L) at time t.

Another important notion is that of a knot (of *FULL* nodes) [10]. A knot K is a tie of which any subset is not a tie. This implies that K is a set of strongly connected nodes, i.e., there is a directed path which leads from each node of K to all others. Alternatively, node i is a member of a knot if i and all nodes which are reachable from node i are *FULL* and can reach node i. In that case the knot is the set of nodes which are reachable from i. Obviously, any tie contains at least one knot.

A node *suffers* from deadlock if it is in a tie. Such a node can never forward any packet. A node *causes* a deadlock if it is in a knot. In order to resolve all deadlocks one must remove at least one packet from each knot (i.e., from at least one node in such a knot), or alternatively to add to each knot at least one empty buffer place. It would not help if buffers were added to nodes which only suffer from the deadlock.

III. Outline of the Paper

In this paper a distributed algorithm for detecting all nodes which cause a deadlock is presented. Such an algorithm is most suitable if the deadlock will be later resolved by releasing packets from nodes (see [7], [9]), and may serve as a trigger to the subsequent resolution process.

To simplify the description of the algorithm we progress in stages. In Section IV, we describe a version of the alogrithm for a "static environment." In this case any node which participates in the algorithm freezes its buffer state, i.e., never considers any packet movement to or from its buffer which takes place after it started the algorithm. This implies that the directed graph which represents the network does not change after the initiation of the algorithm at all participating nodes. The algorithm for the static environment can be interpreted from two points of view.

1) A knot detection algorithm for a general graph which consists of two types of nodes (*FULL*, *NOT_FULL*), similar to [10].

2) As a single iteration of the dynamic deadlock detection algorithm to be described later. From this point of view the algorithm for the static environment is a building block of the dynamic deadlock detection process.

In Section V, we describe how the static algorithm can be adapted to the general situation where nodes may move from *FULL* to *NOT_FULL* states and vice versa. The main idea is to iteratively use the static algorithm by reinitiating it when there is a potential for a new deadlock. Using similar ideas a tie detection algorithm is developed and described in [15].

IV. Deadlock Detection in the Static Environment

A. Outline

In this section we describe an algorithm which operates in a static environment, i.e., nodal state transitions are not considered. Appendix A contains a complete specification of the algorithm.

At the end of the algorithm each node is in one of two modes: 1) *FREE*—in this case the node does not belong to a knot. (Obviously, any *NOT_FULL* node is always *FREE* and immediately sets its mode to *FREE*). 2) *DEADLOCK*—in this case the node belongs to a knot.

The algorithm exploits the property that nodes of a knot are strongly connected and therefore a knot contains a *cycle* of *FULL* nodes. A group of *FULL* nodes which were found to be strongly connected is called a *cluster*. This algorithm is based on looking for cycles of clusters and merging them into bigger clusters. If a cluster with no link directed outside the cluster is detected, then, a knot is found. At the beginning, each cluster consists of a single *FULL* node. At the end of the algorithm each cluster contains all nodes of a knot, if one exists.

The algorithm consists of two basic steps which are repeated to the end. First, within each cluster of *FULL* nodes a single outgoing (outgoing from the cluster) link is selected. In the second step it is checked whether this link is directed to a *FREE* node. If affirmative, then all nodes of the cluster set their mode to *FREE*. These nodes will not be involved in any further action and their cluster is considered as erased. On the other hand, if none of the outgoing selected links is directed to a *FREE* node, then, a cycle of clusters is detected. All clusters of this cycle are merged into a single cluster. These two steps are repeated until all nodes are either *FREE*, or clusters without any outgoing links are found. In the latter case all nodes of such clusters set their modes to *DEADLOCK*.

In order to accelerate the starting of the algorithm and the way nodes become *FREE* we add a sub-phase to the algorithm in which each *FULL* node firstly checks if any of its next nodes are *FREE*. In this phase all *FULL* nodes before choosing a selected outgoing link send an *ASK* message over all outgoing links. A *NOT_FULL* node receiving such an *ASK* message responds by sending a *FREE* message to the sender. When a *FULL* node which never started the algorithm receives such an *ASK*, it initiates the algorithm. If a node receives a *FREE* message over one of its outgoing links, it changes its mode to *FREE* and broadcasts *FREE* messages over all links.

Using the above, two competing algorithms are executed at the same time: the flooding of *ASK* and *FREE* messages, and the cluster formation process. The first one does not affect nodes that are in knots since the *FREE* messages can only be sent by nodes that have a directed path to a *NOT_FULL* node. Consequently, these two phases are independent and are described separately.

B. Detailed Description of the Static Algorithm

The important parts of the algorithm is the detection of cycles of clusters (a modification of [11], [12]) and the technique to combine a cycle of clusters into a single cluster. The cycle detection is facilitated by having nodes/clusters pass maximal ID's through the links of the cycle and have the maximum ID node receive back its own ID. The cycle combination is facilitated by having this maximum ID node establish a tree structure whereby all nodes of the cycle report to it. We start by describing clusters that contain a single node, and then generalize it to any cluster.

The cluster formation phase is started at a *FULL* node i, by choosing an arbitrary outgoing link l, designating it as a selected outgoing link (SO_i), and sending a *TEST* message over it. If l is directed to a *FREE* node, since an *ASK* message is always sent prior to the *TEST*, a *FREE* message will be received over l and i will become *FREE* as described in the first phase. Any non-*FREE* node upon receiving a *TEST* message over l, adds l to its list of selected incoming links (SI). In addition, it sends over l a value called MX, where MX is the maximal node identity "known" at this node. MX is calculated to be the maximum of its own and all other identities received in the past by this node (over its SO link). Each time a new identity is received it is compared with the current maximal identity. If it is larger, it is recorded and sent over all SI links. Since maximal identities are sent over SI links, the existence of a cycle implies that the node with the maximal identity in the cycle will receive its own identity back over its SO link. When this happens this node detects a cycle and appoints itself as cluster leader.

Next we describe how this leader causes cluster combination. The leader builds a tree rooted at itself. This tree is used by the leader to coordinate the cluster activities, i.e., to make it "act" as a single node. Through this tree messages will be exchanged between the leader and the rest of the members of the cluster.

First, the leader sends a message (*CHANGE_LEADER*) stamped with its own identity over its SO link. A node receiving this message over an SI link records its leader identity, deletes this link from SI and designates it as a leader link (LL). Messages to the leader are forwarded over this link. The node forwards the *CHANGE_LEADER* message over its SO link, and designates this link as a branch link (member of LB). Messages from the leader are forwarded over the branch links. Finally, the *CHANGE_LEADER* message arrives through the cycle back to the leader. The link on which the leader receives this message is not a part of the leader tree. The leader deletes this link from SI and in addition sends a *DELETE* message over this link. Receiving this *DELETE* the receiver deletes this link from LB. In addition, a completion message (*CHANGE_TERM*) is sent from this leaf node back to the leader through the leader tree. Any node receiving this acknowledgment from all LB links (here LB consists of a single link), forwards it over LL. In Fig. 1

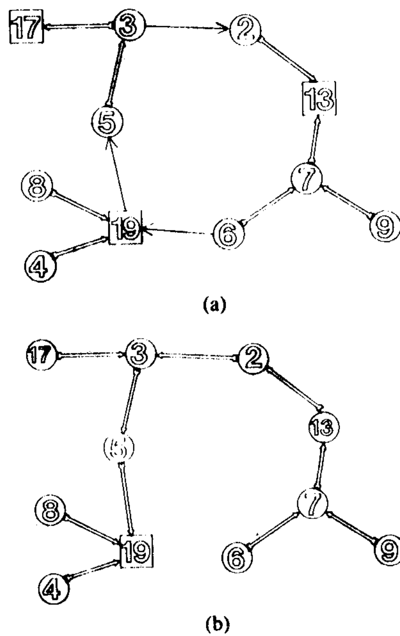

Fig. 1. Typical snapshot.

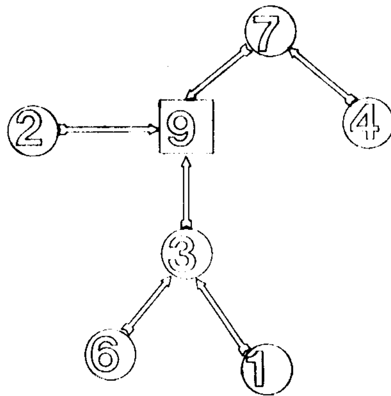
Fig. 2. Cluster's structure.

we demonstrate a typical snapshot of the algorithm. In Fig. 1(a) the *FULL* nodes with their selected outgoing links are depicted. In Fig. 1(b) the resultant cluster along with its leader tree is depicted.

The only difference in the resultant tree after detecting a cycle of nodes and after detecting a cycle of clusters is that in the first each node has only one branch link and in the second each node may have many branch links. In both cases the leader tree connects all nodes of the cluster. In each node the identities of the links that belong to the tree are recorded. The node distinguishes between the leader link (*LL*) which leads to the leader (the root node of the tree) and the branch links (*LB*) which lead to leaf nodes of the tree. In Fig. 2 an example of such a structure is depicted.

Next, we describe how clusters are coordinated to detect a cycle of clusters. After a new cluster is formed, a single link, outgoing of the cluster is chosen. The search for this link is done using a distributed depth first search through the leader tree, starting from the leader. A node presently active in the search process looks in its outgoing links list for an untested link (i.e., a *TEST* message was never sent over it). If all links are tested, the node pass the search deeper into the tree by sending a search message (*SEARCH*) over one of its branch links from which a search termination message (*SEARCH_TERM*) has not been received. This passes the activity to another node.

If an untested link is found, a *TEST* message is sent over it (the link becomes the cluster outgoing link). If the identity of the present leader is received over this outgoing link along with a flag indicating it is the leader of the sender, then this link connects two nodes of the same cluster. A *DELETE* message is sent over it in order to inform the node at its end to delete it from *SI*. This *DELETE* message is acknowledged by a second *DELETE* message. Receiving this acknowledgment the node designates the link as tested, and proceeds with the search.

A node which receives a *SEARCH_TERM* message over all its branch links, or, does not have any branch links, nor any untested links, sends a *SEARCH_TERM* message back over its *LL* to "back-up" the search. If a leader receives such messages over all its branch links it determines that its cluster forms a knot. The leader informs all nodes of its cluster of this situation by broadcasting a *SPAN* message with this information. Receiving such a message over their leader links, nodes forward the *SPAN* message over their branch links, set their modes to *DEADLOCK*, and send a *FREE* message over all their *SI*'s. Note that, all links of the cluster were tested and incoming links that were founded to be directed from nodes of the same cluster were deleted from *SI* list (either after receiving a *DELETE* message during the search or when the link was designated as a leader link). This implies that all the remaining *SI* links are directed from nodes which are not members of this cluster and thus are not members of a knot. These *FREE* messages inform the receiving clusters that they are not the cause of the deadlock.

Other situations which may arise when an untested link is chosen to be the cluster outgoing link, are the following:

If a new identity higher than the known one is received over this link, this knowledge is delivered to all nodes of the cluster so each may broadcast this identity over all cluster *SI* links. This is accomplished by sending a *DELIVER* message including this identity to the leader through the leader link. When the leader receives such an identity higher than the known one, it broadcasts it to all nodes of the cluster using a *SPAN* message. They in turn broadcast it over all their *SI* links.

If the identity received over the cluster outgoing link is the cluster leader's identity along with a flag which indicates it is not the sender's leader, then a cycle of clusters has been detected. At this point all clusters of that cycle are combined into a single cluster whose leader has the identity just received. The process is to combine the leader trees of the clusters in the cycle into a single tree, rooted at the new leader. The node which is responsible for this action is the node that received this identity, this node is

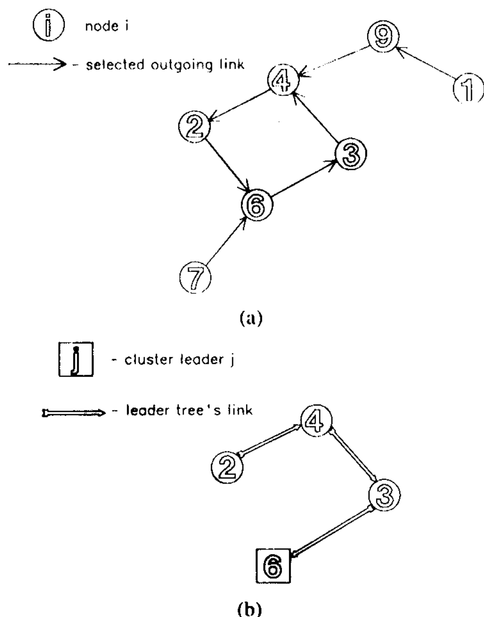

Fig. 3. Cluster formation process.

the node which is currently active in the search process (adjacent to the cluster selected outgoing link). The new part of the leader tree will be rooted at this node. The node starts the construction by sending a *CHANGE_LEADER* messge stamped with the leader identity over the selected outgoing link. A node which receives such a *CHANGE_LEADER* message over a link, changes its leader identity, designates this link as the new leader link, designates all other old leader tree links and the cluster selected outgoing link (if any) as branch links, and forwards the *CHANGE_LEADER* message over all the branch links. This changing leader process is terminated by a *CHANGE_TERM* messages flowing back through the new *LL*'s to the node that initiated the process. After this node has received the *CHANGE_TERM* message from the cluster outgoing link it designates this link both as a branch link and as tested, and proceeds the search for a new cluster outgoing link (the search can now be forwarded to the new branch links as well). In Fig. 3 we demonstrate this cluster combining process. In Fig. 3(a) the clusters are depicted along with their leader trees and selected outgoing links. In Fig. 3(b) the resulted combined cluster is depicted.

Any time a node receives a *FREE* message over one of its outgoing links, it leaves the cluster formation process and acts as previously described. Since clusters are strongly connected subgraphs, and *FREE* messages are forwarded over every incoming link, the termination is guaranteed at all cluster nodes. The algorithm ends when every node is in one of the two modes *FREE* or *DEADLOCK*.

In Appendix A a formal description of the algorithm for the static environment is presented along with a calculation of the communication and memory costs. The communication cost is $O(n'^2 + m')$ messages in the worst case, where n' is the number of *FULL* nodes and m' the number of links attached to those nodes. The memory cost is $O(n \log_2 n + m)$ bits in total and $O(\log_2 n + k)$ bits for a node, where n is the total number of nodes, m the total number of links, and k is the number of links attached to this node.

V. Deadlock Detection in the Dynamic Environment

In this section we describe the extension of the knot detection algorithm to the dynamic environment, i.e. when nodes transit between *FULL* and *NOT_FULL* states due to packet transmissions and receptions. In Appendix B we formally describe the additional lines needed to implement this dynamic knot detection algorithm. In Appendix C a complete proof of correctness of the dynamic algorithm is given.

The extended algorithm iteratively uses the static algorithm where in each new iteration messages are stamped with a different iteration number. A new iteration is started whenever there are state transition that may affect the correct operation of the current iteration. In the following, we first explain that a transition from *FULL* to *NOT_FULL* does not require starting of a new iteration. Then we show how the static algorithm is restarted following a transition from *NOT_FULL* to *FULL*.

A transition from *FULL* to *NOT_FULL* occurs when a packet is sent from a *FULL* node to a *NOT_FULL* neighbor. Since a packet can be transmitted only to a *NOT_FULL* node, the sender node must have had a path in the graph to a *NOT_FULL* node at the start of the current iteration. Moreover, since a deadlock is permanent, this node always had a path to a *NOT_FULL* node. Thus, its correct final mode for the current iteration (and all previous ones) is *FREE*. Therefore the transition from *FULL* to *NOT_FULL* can be considered as if a *FREE* message of the current iteration were received by this node. Since the sender did not suffer from a deadlock in the first place, no harm can be caused by such an interpretation. The *FULL* to *NOT_FULL* transition tells the node sooner (before a *FREE* message does) that it is not deadlocked.

Next, we examine the transition from *NOT_FULL* to *FULL*. This transition may create a new deadlock which no previous iteration can detect. Thus, it is essential to start a new iteration of the algorithm following this transition. Our goal is to handle iteration numbers in such a way that the static algorithm will be initiated at each node of a knot after it becomes *FULL*. Thus, in the last iteration started by a node of that knot, a deadlock will be detected.

To allow for different iterations we assume that a nondecreasing iteration number is attached to each node. Initially the iteration number is zero. Lower iteration numbers are considered older. A node increments its iteration number by one each time the node becomes *FULL*. Following this increment, the node initiates a new iteration of the algorithm whose messages are stamped with the new iteration number. The main difficulty addressed in this section is the coordination between different iteration numbers at different nodes and still using a fixed amount of memory.

To allow the orderly completion of previous attempts of deadlock detection, a priority is given to old iterations of the algorithm over newer ones. A node n_1 attempting to initiate a new iteration at a neighbor n_2, which is performing an older iteration, will have to wait until the completion of all older iterations at n_2. Until n_2 is finished, n_2 records the latest received message for each link until it can respond to it. After n_2 completes all previous iterations it responds to messages of new iterations (if any). Its response depends on its current state and mode.

Generally, each node participates in all iterations, i.e., increments its iteration number one by one, and initiates the algorithm for each new iteration number. This insures that all nodes of a knot reach the highest iteration number started in the knot and complete this iteration. Otherwise, they may skip this iteration due to attempts of higher iteration numbers received from outside. Two exceptions may be considered.

1) A *NOT_FULL* node always completes its current iteration of the algorithm in *FREE* mode. In this case all iterations between the node's current iteration and the highest received from outside are considered as being completed and the node immediately responds to the highest iteration number heard by it.

2) A *DEADLOCK* node always remains in *DEADLOCK* mode. In this case a node considers itself as *DEADLOCK* for all iterations higher than its own. In this case the node responds to higher iteration *TEST* messages with a *FREE* message stamped with the received iteration number, without actually changing its current iteration number. Note, that since this is knot detection (and not tie detection) as viewed by other nodes in the network, a *DEADLOCK* node is considered as a permanently *NOT_FULL* node. (This is changed for tie detection [15].) These two exceptions may also explain why a node should record only the highest attempt received from a neighbor. For all iterations below the highest received it is guaranteed that the neighbor completed them as *FREE* (a node which completes the algorithm in the *DEADLOCK* mode never increments its iteration number nor does it initiate any new iteration). Since the neighbor already became *FREE* in these iterations, it no longer participates in these iterations and does not wait for any response.

To allow the completion of old iterations, some nodes have to respond to messages stamped with iteration numbers lower than their current one. Since the node completed these older iterations as *FREE*, it answers those messages as a *FREE* node with the same (older) iteration number, would do.

We now describe the details. The algorithm is operated at each node just as described in the static stage. The only difference is that the current iteration number is attached to each message of the algorithm. Accepting a message stamped with its current iteration number, the node operates as in the previous section. A *FULL* node which becomes *NOT_FULL* acts as a *FREE* message was received, i.e., sets its mode to *FREE* and broadcasts *FREE* messages. A *NOT_FULL* node which becomes *FULL* increments its iteration number by one and initiates the algorithm using a *TEST* message as previously described. A node which receives a *TEST* message stamped with an iteration number which is lower than its own, sends a *FREE* message stamped with the received iteration number back to the sender.

The case when a node receives messages stamped with a higher iteration number, is divided into several subcases. Since a *TEST* message is always the first to be sent, we assume such a message has been received.

1) The node is in *NOT_FULL* state. In this case the node increments its current iteration number to the received one and sends a *FREE* message back to the sender. In this case there is no point in incrementing the iteration number one by one, since a *NOT_FULL* node completes all iterations in *FREE* mode.

2) The node has completed the current iteration in a *DEADLOCK* mode. The node never increments its iteration number, and considers itself as deadlocked for all iterations higher or equal to its own. In this case the node answers this *TEST* message with a *FREE* message, stamped with the received iteration number.

3) The node is *FULL* but has completed its current iteration in *FREE* mode. In this case the node increments its iteration number by one and initiates the algorithm for this iteration number. If the message just received has the same iteration number (the node previous iteration number plus one) it is considered for the current iteration. If it is higher, then it is recorded and kept until current iteration completion.

4) The node is *FULL* but has not yet completed its current iteration of the algorithm. In this case the node records the highest iteration number received over each link. After the completion of the current iteration, the node takes care of the recorded messages, and acts as if they were just received (according to rules 1-3).

The actions described in cases 3 and 4 guarantee that any message with higher iteration number, received by a node at a lower iteration, will not be processed until the node reaches the higher iteration. As the increment in the iteration number occurs only at nodes which already completed previous iterations, only the highest iteration received over each link should be considered.

VI. Discussion and Conclusions

A new approach for deadlock detection in computer network has been presented. The algorithm was incrementally developed for two environments of increasing complexity. We first derived an algorithm for a static environment in which the state of the transit buffers at nodes is frozen. Given a network with n nodes of which n' are *FULL* and m links of which m' are attached to the *FULL* nodes, the worst case communication cost of this algorithm is $O(n'^2 + m')$ messages, each of $O(\log(n))$ bits. The amount of memory needed in a single node is $O(\log_2(n) + k)$ where k is the link degree of the node and the total memory needed is $O(n \log(n) + m)$ bits.

Next we showed how to detect deadlocks in a full dy-

namic environment where packet transmissions and receptions constantly change the states of the nodal transit buffers, and the resulting directed graph. This is done by invoking a new iteration of the static algorithm each time a new potential for deadlock occurs, and by giving higher priority to "older" iterations. Practically, the adaptation to the dynamic environment uses only a small (fixed) amount of additional memory for each link.

In the static environment one may compare the differences in the performance between our algorithm and the previously published algorithms. However one should be careful in comparing these algorithms since there are differences in the definition of deadlocks, nodes, links, etc. However since the problems seem similar we will compare our performance results to the best known results. In [11] a *N-out-of-M* deadlock detection algorithm for a distributed processing system is presented. This is a "search" type algorithm in which for each process an independent invocation of the algorithm is performed. The communication cost of this algorithm for each invocation is: $O(m')$ messages of size $O(\log(n))$ bits. Since no mechanism for restricting the number of independent invocations is proposed and in order to detect the deadlock at least one deadlocked node must invoke the algorithm (and there is no way to tell which nodes are deadlocked at this point), it is evident that all nodes which are potentially in deadlock must initiate the algorithm. Since up to n' independent invocations of the same algorithm are needed (even in the completely static version), the total communication cost is $O(n'm')$ messages. The total amount of memory needed for a single invocation is $O(m)$ bits. Therefore for n independent invocations $O(nm)$ bits are needed.

On the other hand, the algorithm of [11] is considerably faster and it detects more general types of deadlocks. Similar communication and memory costs in the static environment are common to most search type algorithms known in the literature. One of them [8] needs less memory.

In the dynamic environment the comparison is more difficult as all algorithms are involved in multiple invocations. We only note that our adaptation uses only a total of $O(m)$ extra bits of memory (if we consider the iteration number to be a fixed size). No storing of the complete state is needed, as in [11].

Appendix A
Formal Description of the Knot Detection Algorithm

Variables at Node i

S_i — Buffer state of node i (*FULL* or *NOT_FULL*).

M_i — Mode of node i. (*IDLE, OPERATE, FREE* or *DEADLOCK*, initial values: if S_i = *NOT_FULL* then: M_i = *FREE* else: M_i = *IDLE*).

O_i — Set of outgoing links at node i.

SI_i — Set of selected incoming links over which a *TEST* message has been received (initial value $SI_i = \Phi$).

$T_i(l)$ — Indicates if link l has been tested (initial value $T_i(l) = 0$).

SO_i — Presently tested outgoing link (initial value $SO_i = \Phi$).

L_i — The identity of node i's leader (initial value $L_i = i$).

LL_i — Direction to present leader (a part of the leader tree which leads toward the leader). (Initial value $LL_i = \Phi$).

MX_i — Maximal identity heard (initial value $MX_i = i$).

LB_i — Set of branch links (the part of the leader tree which leads to the branches). (Initial value $LB_i = \Phi$).

$CT_i(l)$ — Indicates if a *CHANGE_TERM* message has been accepted on link l (initial value $CT_i(l) = 0$).

$ST_i(l)$ — Indicates if a *SEARCH_TERM* message has been accepted on link l (initial value $ST_i(l) = 0$).

CF_i — A flag indicates if node i has initiated a changing leader process (initial value $CF_i = 0$).

DF_i — A flag indicates if node i has sent a *DELETE* message over a link which connects two nodes of the same cluster (initial value $DF_i = 0$).

Messages Sent and Received by Node i

START	Message accepted from the outside world which signals the start of the algorithm.
ASK	Message sent on outgoing links prior to the *TEST*.
TEST	Message sent on selected outgoing link.
FREE	Message sent in response to an *ASK* message by a *FREE* node.
MAX(MX_i, f)	Message sent with maximal identity MX_i and $f = 1$ if $MX_i = L_i$ (f = leader indicator).
SEARCH	Search for an untested outgoing link.
SEARCH_TERM	Search has terminated.
DELETE	Delete link from lists.
CHANGE_LEADER(j)	Change leader to j.
CHANGE_TERM	Change leader process has terminated.
DELIVER	Message sent through LL_i to inform leader of a new identity received from cluster outgoing link.
SPAN	Message sent from leader through LB_i to inform nodes with new information (can contain information about 1) new identity to be broad-

casted 2) final mode of cluster (*FREE* or *DEADLOCK*)).

Note: A received message is usually indexed by link identity from which it was accepted, i.e., *TEST*(l) – *TEST* received over link l.

Algorithm for Node i

a1. **For** *START* **then:**
- *a1.1* **If** S_i = *FULL* and M_i = *IDLE* **then:** send *ASK* message to all O_i; $M_i \leftarrow$ *OPERATE*; for some $k \in O_i$ send *TEST* to k; $T_i(k) \leftarrow l$; $SO_i \leftarrow k$;

a2. **For** *ASK*(l) **then:**
- *a2.1* as <a1.1>;
- *a2.2* **If** S_i = *NOT_FULL* **then:** send *FREE* to l;
- *a2.3* **If** M_i = *DEADLOCK* **then:** send *FREE* to l;

a3. **For** *FREE*(l) **then:**
- *a3.1* **If** M_i = *OPERATE* and $l \in O_i$ **then:** $M_i \leftarrow$ *FREE*; send *FREE* to all links; reset all variables except M_i;

a4. **For** *TEST*(l) **then:**
- *a4.1* **If** M_i = *OPERATE* **then:** $SI_i \leftarrow SI_i \cup \{l\}$;
  - *a4.1.1* **If** $MX_i = L_i$ **then:** *send MAX*(MX_i, 1) to l;
  - *a4.1.2* **Else:** send *MAX*(MX_i, 0) to l;

a5. **For** *MAX*(l, MX_i, f) **then:**
- *a5.1* **If** $MX_l > MX_i$ and $L_i \neq i$ **then:** send *DELIVER*(MX_i, L_i) to LL_i;
- *a5.2* **If** $MX_l > MX_i$ and $L_i = i$ **then:** $MX_i \leftarrow MX_l$; send *SPAN*(MX_i, L_i) to LB_i; send *MAX*(MX_i, 0) to SI_i
- *a5.3* **If** $MX_l = MX_i = L_i$ and $f = 1$ and $l = SO_i$ **then:** send *DELETE* to l, $DF_i \leftarrow 1$;
- *a5.4* **If** $MX_l = MX_i = L_i$ and $f = 0$ and $l = SO_i$ **then:** $LB_i \leftarrow LB_i \cup SO_i$; $CF_i \leftarrow 1$; send *CHANGE_LEADER*(L_i) to l;

a6. **For** *CHANGE_LEADER*(l, L_l) **then:**
- *a6.1* **If** $L_l = L_i$ **then:** send *DELETE* to l; delete l from SI_i;
- *a6.2* **Else:** $L_i \leftarrow L_l$; $MX_i \leftarrow L_l$; $LB_i \leftarrow LB_i \cup (LL_i - l) \cup SO_i$; $SO_i \leftarrow \Phi$; $LL_i \leftarrow l$; for all j: $CT_i(j) \leftarrow 0$; send *CHANGE_LEADER*(L_i) to LB_i;
  - *a6.2.1* **If** $LB_i = \Phi$ **then:** send *CHANGE_TERM* to LL_i;

a7. **For** *CHANGE_TERM*(l) **then:** $CT_i(l) \leftarrow 1$;
- *a7.1* **If** for all $k \in LB_i CT_i(k) = 1$ **then:**
  - *a7.1.1* **If** $CF_i = 0$ **then:** $SO_i \leftarrow \Phi$; send *CHANGE_TERM* to LL_i;
  - *a7.1.2* **Else:** $SO_i \leftarrow \Phi$; $CF_i \leftarrow 0$; **goto** <a11> act as *SEARCH* was received;

a8. **For** *DELETE*(l) **then:**
- *a8.1* **If** $l = SO_i$ **then:**
  - *a8.1.1* **If** $DF_i = 1$ **then:** $SO_i \leftarrow \Phi$; $DF_i \leftarrow 0$; **goto** <a11> act as *SEARCH* was received;
  - *a8.1.2* **Else:** $SO_i \leftarrow \Phi$; delete l from LB_i; **goto** <a7.1>;
- *a8.2* **Else:** delete l from SI_i; send *DELETE* to l.

a9. **For** *SPAN*(MX_l, L_l) from LL_i **then:**
- *a9.1* **If** $L_l = L_i$ and MX_l **then:** send *SPAN*(MX_l, L_i) to LB_i,
  - *a9.1.1* **If** $MX_l < \infty$ **then:** $MX_i \leftarrow MX_l$; send MAX_i(MX_i, 0) to SI_i;
  - *a9.1.2* **Else:** $m_i \leftarrow$ *DEADLOCK*; send *FREE* to SI_i;

a10. **For** *DELIVER*(MX_l, L_l) from some $k \in LB_i$ **then:**
- *a10.1* **If** $MX_l > MX_i$ and $L_l = L_i \neq i$ **then:** send *DELIVER*(MX_l, L_i) to LL_i;
- *a10.2* **If** $MX_l > MX_i$ and $L_l = L_i = i$ **then:** $MX_i \leftarrow MX_l$; send *MAX*(MX_i, 0) to SI_i; send *SPAN*(MX_i, L_i) to LB_i;

a11. **For** *SEARCH* from LL_i **then:**
- *a11.1* **If** for some $k \in O_i$ $T_i(k) = 0$ **then:** $SO_i \leftarrow k$; send *TEST* to k; $T_i(k) \leftarrow 1$;
- *a11.2* **Else: If** for some $k \in LB_i ST_i(k) = 0$ **then:** send *SEARCH* to k;
  - *a11.2.1* **Else: If** $L_i \neq i$ **then:** send *SEARCH_TERM* to LL_i;
    - *a11.2.1.1* **Else:** $M_i \leftarrow$ *DEADLOCK*; send *FREE* to SI_i; send *SPAN*(∞, L_i) to LB_i; reset all variables except M_i;

a12. **For** *SEARCH_TERM* from $k \in LB_i$ **then:** $ST_i(k) \leftarrow 1$; same as <a1>;

Communication Cost

To compute the communication cost, measured in number of messages sent in the network, we will consider separately each message type. Let n' be the number of *FULL* nodes in the system, and m' the number of bidirectional links which are attached to those nodes.

1) A *TEST*, ASK, and *FREE* message can be sent over any outgoing unidirectional link, and thus at most $6m'$ such messages are sent.

2) *SPAN* and *DELIVER* messages are sent only over the leader trees, each contains different identities, and thus at most n'^2 messages are sent.

3) CHANGE_LEADER and CHANGE_TERM messages are sent only over trees or cycles, once for each leader. Since no more then n' leaders can be found, then, their total number is bounded by $4n'^2$.

4) *MAX* messages are received over selected outgoing links. However for a node which is not in a knot, such a link is only selected once. For nodes in knots such messages can be received only over outgoing links which are part of a cycle (except when the leader identity is received) over links which connect nodes of the same cluster) and thus no more then $2n'^2 + m'$ of such messages can be sent. We can conclude that communication cost at

worst case is of $O(m' + n'^2)$ messages, each of no more than $O(\log (n'))$ bits.

Memory Cost

In order to compute the memory cost of the algorithm, we assume that one bit variable is allocated for describing the membership of each adjacent link to each set of links (O_i, SI_i, L_i, LL_i, SO_i, LB_i). This implies that except MX_i which contains a node identity, all variables are of one bit length, and a fix number of such variables is allocated for each link of the node.

We can conclude that the total memory cost of this algorithm is $O(n \log_2 (n) + m)$ bits for the total network, and $O(\log_2 (n) + k)$ for each node where k is the link degree of that particular node.

Appendix B
Formal Description of Adaptation to the Dynamic Environment

Variables at Node i

Same as in the static version with the addition of

- CN_i Iteration number of node i (initial value $CN_i = 0$).
- $MCN_i(l)$ In this variable the node records the highest CN which was accepted over link l while in *OPERATE* mode, it also indicates the reception of an *ASK* message (initial value $MCN_i(l) = 0$).
- $MT_i(l)$ Indicates if a *TEST* message stamped with iteration number $MCN_i(l)$ was accepted over link l (initial value $MT_i(l) = 0$).

Messages Sent and Received by Node i

The same as in the static version with the addition of iteration number.

Additional Actions Taken by Node i

b1. **For** $S_i \leftarrow$ *FULL* **then:** $CN_i \leftarrow CN_i + 1$; $M_i \leftarrow$ *IDLE*; initiate all variables; initiate the algorithm for the new iteration (as if a *START* is received);

b2. **For** $ASK(l, CN_l)$ **then:**
- *b2.1* **If** $CN_i > CN_l$ or S_i = *NOT_FULL* **then:** $CN_i \leftarrow max\{CN_i, CN_l\}$; send $FREE(CN_l)$ to l;
- *b2.2* **If** $CN_i = CN_l$ and S_i = *FULL* **then:** same as in static algorithm;
- *b2.3* **If** $CN_i < CN_l$ and S_i = *FULL* **then:**
  - *b2.3.1* **If** M_i = *FREE* **then:** $CN_i \leftarrow CN_i + 1$; $M_i \leftarrow$ *IDLE*; initiate all variables; initiate the algorithm for the new iteration (as if a *START* is received); consider this message as just received;
  - *b.2.3.2* **if** M_i = *OPERATE* **then:** $MCN_i(l) \leftarrow CN_l$, $MT_i(l) \leftarrow 0$;
  - *b2.3.3* **If** M_i = *DEADLOCK* **then:** send $FREE(CN_l)$ to l;

b3. **For** $TEST(l, CN_l)$ **then:**
- *b3.1* **If** $CN_l = CN_i$ **then:** same as in the static algorithm when a *TEST* received;
- *b3.2* **If** CN_i and S_i = *FULL* and $M_i \neq$ *DEADLOCK* **then:** $MT_i(l) \leftarrow 1$

b4. **For** $FREE(l, CN_l)$ **then:**
- *b4.1* **If** $CN_l = CN_i$ **then:** same as in the static algorithm when a *FREE* message is received;

b5. **For** $M_i \leftarrow$ *DEADLOCK* or $M_i \leftarrow$ *FREE* **then:** after finishing original algorithm lines: act as recorded messages (if any) were just received.

Appendix C
Proof of Correctness

Theorem 1: All nodes of a knot eventually set their mode to *DEADLOCK*. If at time t a node is not deadlocked, then, it has never been in *DEADLOCK* mode, before t.

In order to prove Theorem 1, which is the main theorem of this section, we present some introduction lemmas. The following two lemmas prove some properties of the static algorithm.

Lemma 1: Consider the algorithm of Appendix A. If a knot exists at time t and a *START* is given at least to one of the nodes in the knot, then all nodes of the knot will complete the algorithm in the *DEADLOCK* mode without leaving the *OPERATE* mode prior to completion. Furthermore, they will complete the algorithm after they join the cluster whose leader is the node with the maximal identity in the knot.

Proof of Lemma 1: First, we prove that none of the knot's nodes can complete the static algorithm in *FREE* mode, before at least one of the nodes has completed the static algorithm in *DEADLOCK* mode. To see that, assume in contradiction that i is the first node of the knot which completes the algorithm in *FREE* mode. In order to complete the algorithm as *FREE*, a *FULL* node must receive a *FREE* message over an outgoing link. As no link is outgoing from the knot this implies that node i receives the *FREE* message from another member of the knot. Let us denote as j this *FREE* message originator. Clearly, in order to send a *FREE* message, node j must be either in the *FREE* mode or in *DEADLOCK* mode prior to the time of i's reception. The former case contradicts our assumption that node i is the first to become *FREE.* The latter case implies that j has completed the algorithm in the *DEADLOCK* mode. Since there is an outgoing link from j to i they are members of the same knot which proves that a member of a knot cannot become *FREE* before some other member of the same knot becomes *DEADLOCK.*

Now we prove that all nodes complete the algorithm in *DEADLOCK* mode. To see that, let m be the node with the maximal identity in the knot. As identities are always sent over incoming links it is clear that m is the maximal identity which might be known at every node of the knot.

First, we prove that in finite time after t all nodes initiate the algorithm. This is clearly true since a knot is a

strongly connected graph and *ASK* messages are sent over all outgoing links of any node which initiates the algorithm. Recall that at least one of the knot's nodes receives a *START*.

Secondly, we prove that each outgoing link of the knot's nodes is tested before any node become *DEADLOCK*. Clearly no cluster leader may become *DEADLOCK* before it receives *SEARCH_TERM* messages over all its branch links and before it tests all its outgoing links (<a11.2.1.>). By repeating this argument for all nodes of the leader tree starting from the leader, it is clear that if some outgoing link is not tested, at least one *SEARCH_TERM* message is not delivered. This implies that if an untested link exists in the cluster then no node becomes *DEADLOCK* in that cluster.

Let us divide at some time $\tau \geq t$ all the directed links of the knot into two groups: those which are tested and all the rest. Now consider all the clusters which exist within the knot at time τ. If there exists a cluster without any currently selected outgoing link, then, in finite time after τ, at least one untested link will be selected as the cluster selected outgoing link, which implies that at least one link leaves the second group and joins the first. If *all* clusters have already selected outgoing links at τ and the knot consists of more then one cluster, then there exists at least one cycle of clusters. Consider one of these cycles and let h be the leader with maximal identity in this cycle. No higher identity can be accepted at this cycle (identities are accepted only over outgoing tested links). Since at each cluster, the maximal identity accepted over the selected outgoing link is 1) delivered to the leader, 2) spanned to all the cluster's nodes, and 3) broadcasted over all selected incoming links, then in a finite time after τ the cluster whose leader is h accepts its own identity over its outgoing tested link and initiates a cluster formation process using *CHANGE_LEADER* messages. *CHANGE_LEADER* messages are forwarded over all branch links and selected outgoing links. This insures that the *CHANGE_LEADER* messages are accepted at all nodes of the cycle except those of the h cluster (with the exception of a single node which replies with a *DELETE* message, see <a6.1>). The termination of this process is guaranteed by *CHANGE_TERM* messages flowing back over the new combined leader tree, and the search for a new untested outgoing link proceeds (also at nodes that just joined the cluster). In a finite time after t at least one untested outgoing link is found and tested. This implies that at least one link leaves the second group and joins the first. If the knot consists of only one cluster, then again any untested outgoing link will be found and tested by the search process. Repeating this argument results that each untested link is eventually tested before any node become deadlock.

This also proves that no node of the knot may reach the *FREE* mode. No *FREE* message from outside may cause this since no outgoing link in is directed outside to the knot. No *FREE* message is sent within the knot. A *DEADLOCK* node sends *FREE* messages only over its selected incoming links (see <a11.2.1.1>). As all directed links which connects two members of the same knot are deleted from the list of selected incoming links following the test (see <a5.3> and <a.8.1.1>). All these links are already deleted before any node may become *DEADLOCK*.

Third, we prove that in finite time after t all nodes of the knot will accept the identity m. To see that let us divide at some time $\tau \geq t$ the nodes of the network into two groups, those which already accept m, and those which do not. Since at least one outgoing link is directed from the second group to the first and this link is eventually tested, following the second argument, in finite time after τ, at least one node from the second group receives this identity over the tested link and thus leaves the second group and joins the first.

Fourth, we prove that in finite time after t all nodes of the knot join the cluster whose leader is m. To see that assume a time τ after which, following the third argument, all nodes have already accepted the identity m. And let us consider all clusters whose leader is not m. If such clusters exist, then at least one outgoing link is directed from the m cluster to the others and vice versa. As all nodes accepted the identity m, all outgoing links are eventually tested, and no identity above m can be accepted at the knot, then, in finite time after τ the identity m is accepted by one of the m cluster's nodes from outside, which resulted in a cluster formation process using *CHANGE_LEADER* messages. This implies that in finite time after τ at least one node joins the m cluster. Repeating this argument results that all nodes eventually join the m cluster.

Using the above it is clear that the search process in the m cluster will be terminated using *SEARCH_TERM* messages, resulting in a deadlock detection at m which broadcasts this information to all knot's nodes using *SPAN* messages. As no node of the knot may accept a *FREE* message over an outgoing link as described in the second argument, this completes the proof of Lemma 1.

Lemma 2: Consider the static algorithm of Appendix A. If a node j is not a member of a knot then in a finite time after j initialized the algorithm, it becomes *FREE*.

Proof of Lemma 2: If node j is not in a knot then there is a path of outgoing links directed from j to a *NOT_FULL* node or to a knot. Consider the first case, the initialization of the algorithm results in a flooding of *ASK* messages over all outgoing links, and such an *ASK* is responded by a *FREE* message when received by a *NOT_FULL* or a *FREE* node, then if j does not become *DEADLOCK* it becomes *FREE*. In the second case if j has a path to some knot, then following Lemma 1, this knot is detected. Since *FREE* messages are sent over all links incoming to the knot, the above holds for this case as well. Therefore it is sufficient to prove that j does not become *DEADLOCK*.

Assume in contradiction that i is the first node which becomes *DEADLOCK* but is not a member of a knot. Following the proof of Lemma 1, it is clear that i must be a cluster leader. Clearly, before reaching the *DEADLOCK* node i is *FULL*, receives a *SEARCH_TERM* message

over all its branch link and each of its outgoing link is either a branch link or deleted. Now consider all nodes adjacent to the branch links of i. The same argument must hold for these nodes. Using this induction argument for all branch links, we can conclude that all nodes which are reachable from i are *FULL*, belongs to i's cluster, and all their outgoing links are directed to nodes of the same cluster. This implies that these nodes form a knot, which contradicts the assumption. Thus j does not become *DEADLOCK* and thus becomes *FREE*.

Lemma 3: Consider the algorithm of Appendix B. Let t be a time in which a node becomes *FULL* resulting in a knot K. Let R be the highest cycle number known in K at t. The following can be said about the nodes of the knot. All nodes of K with cycle number less than R enter the *FREE* mode in their current cycle. All nodes of K eventually increment their cycle number to R. The R cycle results in a *DEADLOCK* mode at all nodes of K.

Proof of Lemma 3: We prove the lemma by an induction over the cycle numbers. Assume the lemma is correct for all cycle numbers $r < R$. The lemma is true for $R = 0$ following Lemmas 1 and 2 as the operation of the algorithm is static in this case.

As a deadlock exists permanently, the existence of K at time t implies its existence at all times $\tau \geq t$. It also implies that no node of K was a member of a knot prior to t.

We first show that in finite time after t all nodes of the knot will have cycle number which is at least R. Let R_x be the lowest cycle numbers known at the knot at time t. Let us further assume that $R_x < R$.

First, we show that all nodes with cycle number R_x complete this cycle in *FREE* mode. To see that let the graph (N, L) consists of all nodes and directed links of the network with their states at the time they started cycle R_x. Nodes that skipped this cycle number are considered as *NOT_FULL*. The execution of the R_x cycle in the network is identical to an execution of the static algorithm in (N, L). This is true since all *FULL* nodes have the same directed links, nodes with higher cycle numbers are considered as *NOT_FULL* and each packet transmission to a *NOT_FULL* node over an outgoing link is interpreted as *FREE* message reception. Asssume in contradiction that some node k of K in (N, L) becomes *DEADLOCK*. Then by Lemmas 1 and 2, a knot K_x exists in (N, L). Consider the nodes of K_x after the completion of the R_x cycle. K_x forms a cluster of *FULL* nodes where each outgoing link is either a branch link or deleted (following the proof of Lemma 2). This implies that K_x is a knot in the network too. Since k is a member of both K_x and K and each knot is strongly connected, $K_x = K$. The induction hypothesis implies that K is detected in the R_x cycle. As no node changes its cycle number after it becomes *DEADLOCK* this contradicts the assumption that $R_x < R$ and implies that all nodes of K complete the R_x cycle as *FREE*.

Secondly, we show that all nodes with cycle number $R_x(\tau)$ eventually increment their cycle number. Let us consider the time at which all nodes having cycle numbers $R_x(\tau)$ have already reach the *FREE* mode. Divide the nodes of the knot into two groups, those with cycle number $R_x(\tau)$ and the rest. If the first group is empty, then we are done. Else, as a knot is a fully connected subgraph, there exists at least one outgoing link directed from the second group to the first one. Since *ASK* messages are sent for each cycle over each outgoing link, then, at least one such message has been sent stamped with cycle number above $R_x(\tau)$ over that particular link. Receiving such an *ASK* message stamped with higher cycle number at a node which has already completed its current cycle in the *FREE* mode while in the *FULL* state, results in a cycle number increment. This implies that at least one node leaves the second group and joins the first. Repeating this argument results in all nodes with cycle number $R_x(\tau)$ incrementing their cycle number by one.

Using the above argument it is clear that all cycles lower than R result in *FREE* mode. Moreover, by repeating this argument for the remaining set of nodes with the lowest cycle number, and by using the fact that *FULL* nodes increment cycle numbers always one by one, we can conclude that all nodes increment their cycle number to R.

Third, we show that the R cycle results in a *DEADLOCK* mode at all nodes of K. The proof follows immediately as a result of Lemmas 1 and 3. As the knot permanently exists (no node can change its state), no link is outgoing outside the knot, and messages with cycle numbers higher than R are not responded until the node reaches its final mode, then the operation of cycle R is equivalent to the operation of the static algorithm resulting in *DEADLOCK* mode at all nodes.

Lemma 4: If the algorithm results in a *DEADLOCK* mode at any node then there is a deadlock.

Proof of Lemma 4: Let us denote by i the first node which reaches the *DEADLOCK* mode, and by R the corresponding cycle number. As all nodes except the clusters leaders reach the *DEADLOCK* mode only by receiving a *SPAN* message which contains this information, i must be a cluster leader. As i concluded that a deadlock exists it must have received *SEARCH_TERM* messages over all branch links. As all nodes send *SEARCH_TERM* messages only after they test that all their outgoing links are directed to members of the same cluster, and only after they received *SEARCH_TERM* messages over all their branch links, it is clear that node i can conclude a deadlock situation only after a *SEARCH_TERM* message has been sent by all the nodes of its cluster. A *SEARCH_TERM* message can be sent by a node only after all its outgoing links were tested and were found to be directed to *FULL* nodes of the same cluster with the same cycle number R. If at the time when i concludes a deadlock situation all nodes which sent the *SEARCH_TERM* message remain *FULL* and with the same outgoing links, then, the cluster is indeed a knot of *FULL* nodes and a deadlock does exist. On the other hand if some node became *NOT_FULL* or has some other outgoing link which did not exist at the time of *SEARCH_TERM* sensing, then it is clear that such a node must have sent at least one packet

over one of the outgoing links existed at that time. As no *NOT_FULL* node can send a *SEARCH_TERM* message, this implies that no node can become *NOT_FULL* between the time it initiates the current cycle until the time it sends the *SEARCH_TERM* message. Now let us assume in contradiction that j is the first node of the cluster which sends a packet after it initiates cycle R. Following the above this packet transmission occurs only after a *SEARCH_TERM* message has been sent. Now let us denote by k, j's packet destination. As packets can be sent only to *NOT_FULL* nodes, then k must have been *NOT_FULL* before j does. Now consider the two possible events: 1) node k becomes *NOT_FULL* after it initiates the R cycle. This clearly contradicts the assumption that j is the first node which does. 2) When j sent the message to k, node k has not yet initiated the R cycle. This contradicts the assumption that node j sent a *SEARCH_TERM* message, since k is outgoing from j, and so j must *TEST* k before deciding on *SEARCH_TERM*. This implies that if all nodes send a *SEARCH_TERM* message a deadlock does exists and the cluster leader conclusion is indeed correct.

Theorem 1 is proved using Lemmas 3 and 4.

REFERENCES

[1] K. D. Gunther, "Prevention of deadlocks in packet-switched data transport systems," *IEEE Trans. Commun.* (*Special Issue on Congestion Control in Computer Networks*), vol. COM-29, pp. 512–524, June 1981.

[2] P. M. Merlin and P. J. Schweitzer, "Deadlock avoidance in store-and-forward networks—I: Store and forward deadlock," *IEEE Trans. Commun.*, vol. COM-28, Mar. 1980.

[3] G. A. Grover and J. M. Jaffe, "Standoff resolution in computer communication networks," IBM Res. Rep. RC11009; also *IEEE Trans. Commun.*, submitted for publication.

[4] R. Obermarck, "Distributed deadlock detection algorithm," *ACM Trans. Database Syst.*, vol. 7, no. 2, pp. 187–208, June 1982.

[5] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144–156, May 1983.

[6] K. M. Chandy and J. Misra, "A distributed algorithm for detecting resource deadlocks in distributed systems," in *Proc. ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, Ottawa, Ont., Canada, Aug. 1982. New York: ACM, 1983, pp. 157–164.

[7] G. Gambosi, D. P. Bovet, and D. A. Menascoe, "A detection and removal of deadlocks in store and forward communication networks," in *Performance of Computer-Communication Systems*, H. Rudin and W. Bux, eds. Amsterdam: The Netherlands: Elsevier-North-Holland, 1984, pp. 219–229.

[8] I. Cidon, J. M. Jaffe, and M. Sidi, "Local distributed deadlock detection with finite buffers," IBM Israel Scientific Center Tech. Rep. 88.154, Apr. 1985.

[9] —, "Global distributed deadlock detection and resolution with finite buffers," IBM Israel Scientific Center Tech. Rep. 88.161, July 1985.

[10] J. Misra and K. M. Chandy, "A distributed graph algorithm: Knot detection," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 4, pp. 678–686, Oct. 1982.

[11] G. Bracha and S. Toueg, "A distributed algorithm for generalized deadlock detection," in *Proc. Symp. Principles of Distributed Comput.*, Oct. 1984, pp. 285–301.

[12] D. P. Mitchell and M. J. Merritt, "A distributed algorithm for deadlock detection and resolution," in *Proc. Symp. Principles of Distributed Comput.*, Oct. 1984, pp. 282–284.

[13] P. A. Humblet, "A distributed algorithm for minimum weight directed spanning trees," *IEEE Trans. Commun.*, vol. COM-31, pp. 756–762, June 1983.

[14] E. Gafni and Y. Afek, "Election and traversal in unidirectional networks," Dep. Comput. Sci., Univ. California, Los Angeles.

[15] I. Cidon, J. M. Jaffe, and M. Sidi, "Local distributed deadlock detection by cycle detection and clustering," IBM Israel Scientific Center Tech. Rep. 88.164, July 1985.

Israel Cidon (M'85) received the B.Sc. (summa cum laude) and D.Sc. degrees from the Technion—Israel Institute of Technology, Haifa, Israel, in 1980 and 1984, respectively, both in electrical engineering.

From 1977 to 1980 he was a consulting Research and Development Engineer involved in the design of microprocessor-based equipment. From 1980 to 1984 he was a Teaching Assistant and a Teaching Instructor at the Technion. From 1984 to 1985 he was a faculty member with the Faculty of Electrical Engineering at the Technion. Since 1985 he has been with IBM Thomas J. Watson Research Center, Yorktown Heights, NY. His current research interests are in distributed algorithms and voice/data communication networks.

Jeffrey M. Jaffe (M'80) received the B.S. degree in mathematics, the M.S. degree in computer science, and the Ph.D. degree in computer science from the Massachusetts Institute of Technology, Cambridge, in 1976, 1977, and 1979, respectively.

He has been a Research Staff Member in the Department of Computer Science at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, since 1979. He is currently the Department Manager of the Department of Communications Systems. During 1984–1985 he spent a sabbatical year at the IBM Scientific Center in Haifa, Israel. His work for the past several years has been in the area of network architecture and protocols, in particular in the area of distributed routing algorithms.

Dr. Jaffe is a member of the Association for Computing Machinery and Phi Beta Kappa.

Moshe Sidi (S'77–M'82) was born in Israel in 1953. He received the B.Sc., M.Sc., and D.Sc. degrees from the Technion—Israel Institute of Technology, Haifa, Israel, in 1975, 1979, and 1982, respectively, all in electrical engineering.

From 1975 to 1981 he was a Teaching Assistant and a Teaching Instructor at the Technion in communication and data networks courses. In 1982 he joined the Faculty of Electrical Engineering Department, Technion, where he is presently a Senior Lecturer. His current research interests are in the area of computer communication networks.

Dr. Sidi is a Bat-Sheva de Rothschild fellow.