Distributed Store-and-Forward Deadlock Detection and Resolution Algorithms

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

Abstract—**Distributed algorithms for the detection and resolution of deadlocks in store-and-forward computer communication networks are presented and validated. The algorithms use a *fixed amount of storage* at each node (that is independent of the size of the network). The detection algorithm is simple but requires network-wide coordination. The resolution algorithm is based on earlier approaches, but uses the network-wide coordination to address certain synchronization problems. When the detection and resolution algorithms are merged, it is guaranteed that packets will arrive at their destinations in finite time.**

I. Introduction

ONE of the crucial problems in the design of store-and-forward computer communication networks is the buffer deadlock problem. A comprehensive survey of potential types of deadlocks appears in [1]. Various schemes can be devised to prevent the occurrence of deadlocks in networks [1]–[4]. However, these schemes are often too costly to implement. Also, redundant buffers allocated to nodes to prevent deadlocks are often not utilized, especially in well-designed networks where deadlock situations are rare. In addition, "overcontrol" of the network degrades performance under normal conditions. Thus, IBM's SNA does not explicitly avoid transport deadlock [5], and Digital's Decnet prevents deadlocks only by occasionally throwing away messages [6]. In a network without deadlock prevention, having a distributed deadlock detection algorithm is invaluable. Once a deadlock is detected, having a distributed algorithm to extricate the network from deadlock is not less important.

Distributed deadlock detection has been extensively studied [7]–[12]. In all these studies, the amount of local storage required at each node to perform a deadlock detection algorithm grows (at least) proportionally with the size of the network. This property is unacceptable in store-and-forward networks since a buffer deadlock at the network level implies storage shortage and there might not be enough storage to execute the deadlock detection algorithm when a deadlock exists! Consequently, the *network* versions of the distributed deadlock detection and resolution problems require that nodes be able to detect and resolve the deadlock using a *finite amount* of auxiliary buffers specifically set aside for that purpose [13].

This paper first presents and validates a simple distributed deadlock detection algorithm that leaves the network in a state that allows resolution of the deadlock (Section IV). Then an accompanying distributed deadlock resolution algorithm is introduced and validated (Section V). Both algorithms use only *finitely many buffers* during their execution and assume the existence of a single Leader in the network. This implies (by the Leader performing a single iteration of the PIF algorithm described in [17]) the existence of a spanning-tree rooted at the Leader and that each node knows its adjacent links that belong to the tree. When routing in the network is "reasonable," the combined deadlock detection and resolution insures the essential property for proper operation of a network, that all packets will arrive at their destinations in finite time, as we prove in Section III. In Section II, we describe the underlying model that is used in this paper.

Paper approved by the Editor for Random Access Systems of the IEEE Communications Society. Manuscript received December 17, 1986. This paper was presented in part at ICC '86, Toronto, Ont., Canada, June 1986.

I. Cido and J. M. Jaffe are with IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

M. Sidi is with the Department of Electrical Engineering, Technion—Israel Institute of Technology, Haifa, 32000 Israel.

IEEE Log Number 8717077.

II. The Model

We adopt the model of [13]. For completeness, we give a concise description of that model.

A. Network Model

A *network* consists of a set of *communication nodes N,* and a set of links L that interconnect nodes of N. Two nodes interconnected by a link are called neighbors. 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 by a neighbor), it determines a next node, based on the packet's header and routing tables. The type of routing does not concern us—only the fact that the header and the (possibly dynamic) routing tables uniquely determine a next neighbor and that:

Assumption 1): The routing is "reasonable," i.e., it ensures that packets do not loop forever in the network, but each packet traverses a finite (not necessarily bounded) number of hops on its way from the source to the destination.

Once the next node is determined, the node queues up a packet for that next node. The way a node handles its queue to an adjacent node is not very important. We only need that:

Assumption 2): If a queue is not permanently blocked, then each packet will eventually be transmitted. Note that a FIFO discipline is one example in which this assumption holds.

The way in which a node decides to send to an adjacent node is also not too important, but we need that:

Assumption 3): A node n never sends to node m if m is unable to receive more packets. Furthermore, if m is not permanently unable to receive packets, then each of its neighbors which has a nonempty queue for m has an opportunity to send to m and will send the first message in the queue to m.

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.

Finally, we assume that the response time of a node to control massages that it receives is finite.

B. Model of a Communication Node

We assume that the total storage available at a node is *finite* and that it is divided into three parts (see Fig. 1). One part 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 for specific emergency measures such as deadlock detection and resolution. This must be a fixed amount of storage as it must be enough irrespective of network size. (When storage management is organized in the machine, one does not know how large the network is or whether it will grow over time.) Thus, in the overhead portion, one may reserve a *fixed number of message buffers* to aid in deadlock detection and resolution.

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 and resolution. Thus, the total storage allowable for the entire deadlock detection and resolution procedures is a constant number of message buffers, plus a constant number of buffers per adjacent link.

Once storage has been reserved for overhead and for the links, all other storage is left over for message buffers for transit traffic.

The way that the message buffers are used is that they are a common "pool" shared by various components of the node. Each link has a queue of outgoing messages. When a link control learns that an adjacent node is sending a message, it allocates a free message buffer (if one exists) to accept the new transit data. Once in a buffer, the node determines the outgoing link and assigns this buffer to the relevant outgoing queue. (We assume that this entire message processing step is an atomic action.) When the message is transmitted, the buffer is freed. If no free buffer is available, we assume that via pacing or polling mechanisms, the adjacent node knows not to send. 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 it has no free message buffers. Otherwise it is *not full*. If a node i has any messages on the outgoing queue for neighbor l then (i, l) is an *outgoing link*. Alternatively, we call l an outgoing neighbor. Let D be a collection of full nodes. The set D is a *deadlock set* if all outgoing links from nodes in D lead to nodes in D. A node $n \in N$ is *deadlocked* if it is full and is a member of some deadlock set. These definitions imply that:

Lemma 1: If a node i is permanently full after time t, then it becomes deadlocked at some time $t' \geq t$.

Proof: Assume that node i is permanently full after time t. Assumption 3) then implies that $\exists t_1 \geq t$ such that after t_1 all outgoing neighbors of node i are permanently full. Using Assumption 3) for these outgoing neighbors implies that $\exists t_2 \geq t_1$ such that after t_2, all outgoing neighbors of the outgoing neighbors of node i are permanently full. Using the same argument inductively, we conclude that (since the number of nodes in the network is finite) there is a deadlock set in the network that node i belongs to it.

A *deadlock detection algorithm* (DDA) is an algorithm with the following properties: ($D1$) it never returns a result of deadlock to a node that is not deadlocked; ($D2$) if a deadlock set is formed and deadlocked nodes never become not full (by some resolution mechanism), then in finite time the algorithm will (be initiated and) return a result of deadlock to all deadlocked nodes.

A *deadlock resolution algorithm* (DRA) is an algorithm with the following properties: ($R1$) it is initiated upon deadlock detection; ($R2$) if node n becomes deadlocked at time t then in finite time after t, it will become not full; ($R3$) no packets are lost during its execution.

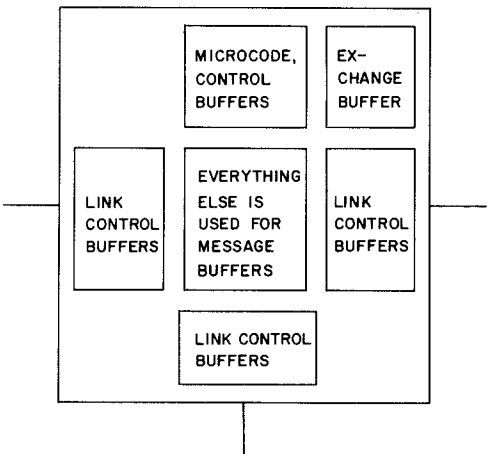


Fig. 1. Storage partition.

In addition, we require that both the detection and the resolution algorithms will use only the (*fixed* amount of) storage reserved for them in the overhead and link control portions of a node, and that they will not interfere with the normal operation of the network. Finally, we require that during the execution of a resolution algorithm no packets will be routed on links other than those specified by the routing algorithm.

The main focus of this paper then is to develop simple deadlock detection and deadlock resolution algorithms. As we show in the next section, when a DDA and a DRA are applied in a network, it is ensured that packets will be delivered to their destinations in finite time. This latter property is essential for proper operation of a communication network.

III. Deadlock Transparency

In computer communication networks, buffer deadlocks may occasionally occur, and if no means are taken to face such situations, they will cause some of the packets to be infinitely delayed and never forwarded to their destinations. Applying deadlock detection and resolution algorithms with the above properties will not inhibit deadlocks from being formed, but these deadlocks will be transparent to the users of the network, in the sense that all packets will be delivered to their destinations in finite time, as we now prove.

Theorem 1: Let the assumptions upon routing and queue and packet handling stated in Section II-A hold. Assume that a network operates with deadlock detection and resolution algorithms with the properties stated in Section II-C. Then, each packet that enters the network will be delivered to its destination in finite time.

Proof: Assume that the theorem does not hold, namely that there is a packet in the network that is never forwarded to its destination. Since the routing is reasonable, i.e., the packet only traverses finitely many hops, and since no packets are lost ($R3$), the packet must be permanently stuck at some node. Consequently, it must be that the outgoing queue is permanently blocked. Thus, the packet is to be sent to a permanently full neighbor. By Lemma 1), this neighbor will become deadlocked and by ($D1$) it will find out that it is deadlocked in finite time. Then by ($R1$) resolution will be initiated and by ($R2$) node i will become not full in finite time, contradicting the fact that it is permanently full.

It is interesting to note that if both detection and resolution algorithms are applied in a network, then it is not guaranteed that each deadlocked node will find out that it is deadlocked,

because the deadlock might be resolved before some nodes become aware that it has been ever formed.

In the following two sections, we introduce distributed deadlock detection and deadlock resolution algorithms with the above properties.

IV. Deadlock Detection Algorithm

The distributed deadlock detection algorithm to be described is based on a simple idea for determining the deadlock set in a network. To facilitate the understanding of the distributed DDA, we first give a centralized DDA that introduces the basic idea.

A. Centralized Algorithm for Deadlock Detection

Let N, E, and D be variables that at the end of the algorithm represent the set of all nodes, the set of not-deadlocked nodes, and the set of deadlocked nodes, respectively. The idea of the algorithm (listed in Fig. 2) is that at its beginning, all full nodes are in D and all nodes that are not full are in E. Then each node in D with an outgoing link to a node in E, is deleted from D and added to E, repeatedly. Thus, at the end of the algorithm D contains all full nodes that do not have any path through outgoing links to a not-full node and therefore they are deadlocked.

Initialize $E = \phi$; $D = N$;
Step 1 For all $i \in N$, if i is not full then do: $E \leftarrow E \cup i$; $D \leftarrow D - i$;
Step 2 $FLAG \leftarrow 0$; For all $i \in D$, if $j \in E$ and j is an outgoing neighbor of i do:
$E \leftarrow E \cup i$; $D \leftarrow D - i$; $FLAG \leftarrow 1$;
Step 3 If $FLAG = 1$ then go to Step 2;
End {D contains all deadlocked nodes; E contains all nondeadlocked nodes}

Fig. 2. Centralized algorithm.

B. Distributed Algorithm for Deadlock Detection

The following is an algorithm that exploits the same idea of the centralized algorithm, but now in a distributed manner. We assume that there is a static spanning tree that covers all nodes of the network and is rooted at a Leader. Each node i knows its adjacent links that belong to the spanning tree and its neighbors on this spanning-tree are called *tree neighbors*. Different distributed algorithms can be used for determining the Leader [15] or for constructing the spanning tree [16], [17] using a finite amount of storage per adjacent link.

At any time, each node can be in one of two modes. When it is not participating in the algorithm it is *ASLEEP* and when participating it is *AWAKE*. Each node i holds and updates the Boolean variable $ST(i)$ during the execution of the algorithm. $ST(i)$ indicates whether node i is potentially deadlocked or not; 0—not deadlocked (belongs to E), 1—deadlocked (belongs to D). At the end of a cycle, this variable indicates whether a node is deadlocked.

The distributed algorithm operates in cycles. Each cycle is composed of three phases and is started by the Leader after being triggered by some request (*REQ*) from a node that became full when it was *ASLEEP* or when it was *AWAKE*. In the former case, the *REQ* message is forwarded immediately through the tree to the Leader, while in the latter case, it is held until the whole cycle of the algorithm terminates, and only then is forwarded.

The first phase of the algorithm is called the *START* phase and it is started by the Leader sending *START* messages to all its tree neighbors. The *START* messages propagate as follows. Each node i that receives a *START* message from the tree neighbor that leads to the Leader (call it p_i), sends *START* messages to all its tree neighbors, except p_i. When it receives *START* messages from *all* its tree neighbors it sends a *START* message to p_i. When a node i receives a *START* message for the first time in a cycle (i.e., when it is *ASLEEP*), it *AWAKE*'s and sets its $ST(i)$ properly, i.e., if it is full, $ST(i) \leftarrow 1$ and if it is not full $ST(i) \leftarrow 0$.

During the whole cycle, a node i that becomes not full while being *AWAKE*, sets $ST(i) \leftarrow 0$. If node i becomes full during a cycle it does not change $ST(i)$, but keeps a *REQ* message to indicate that it should trigger a new cycle when the current one ends.

Whenever node i sets $ST(i)$ to 0, it starts a *freeing* mechanism by sending *FREE* messages to *all* its neighbors. A node j that receives a *FREE* message from a node i and is *AWAKE* acts as follows: if $ST(j) = 0$ or if it does not have any data message intended to i, it sends an immediate *ACK* to node i. Otherwise (i.e., $ST(j) = 1$ and j has a data message intended to i), it sets $ST(j) \leftarrow 0$ and starts a freeing mechanism. Only after receiving *ACK* messages from *all* its neighbors, will node j send an *ACK* message to node i. Thus, when a node i receives *ACK* messages from all its neighbors, it is ensured that all nodes that are not deadlocked because i is not, have been informed of that. We say that at this point node i completes its freeing mechanism. A node j that becomes not full by sending a data message to node i acts as if it received a *FREE* message from i when $ST(j) = 1$ with a message intended for i. Finally, a node j that receives a *FREE* message from a node i and is still *ASLEEP*, waits until it *AWAKE*'s, and then acts as described above.

When the Leader receives *START* messages from all its tree neighbors, it knows that each node in the network started the algorithm and the *START* phase has been completed. It then begins the second phase, called the *TERMINATE* phase, by sending *TERMINATE* messages to all its tree neighbors. *TERMINATE* messages propagate in the same way as *START* messages, except that when a node receives *TERMINATE* messages from all its tree neighbors, it sends a *TERMINATE* message to p_i, only if it has $ST(i) = 1$ or if it has $ST(i) = 0$ and it completed its self-initialized freeing mechanism. Otherwise, it will send a *TERMINATE* message to p_i upon completion of its freeing mechanism.

Finally, when the Leader receives *TERMINATE* messages from all its tree neighbors, it knows that each node in the network completed the *TERMINATE* phase. It then (if its freeing mechanism is in progress, it first waits for its completion) begins a final phase (the *INFORM* phase) by sending *INFORM* messages to all its tree neighbors. *INFORM* messages propagate in the same way as *START* messages. When a node receives an *INFORM* message and it has $ST(i) = 1$ it concludes that it is deadlocked (indicating that by $DS(i) \leftarrow$ *Deadlocked*). A node that sends an *INFORM* message to p_i goes *ASLEEP*. When the Leader receives *INFORM* messages from all its tree neighbors, it knows that each deadlocked node in the network knows that it is deadlocked and that the current cycle of the detection algorithm has been completed.

In addition, the *INFORM* phase messages contain a variable $X(i)$ that allows the Leader to find out if there is any deadlocked node in the network. This feature will become important when we design deadlock resolution algorithms. Finally, during the propagation of the *INFORM* messages, if any node has an *REQ* message (indicating that the node became full during the execution of the current cycle), it is forwarded to the Leader, thus triggering it to start a new cycle.

The formal specification of the algorithms performed by the Leader and the nodes in the network is given in Fig. 3. From the above description of the DDA, it is clear that only a fixed amount of local storage is used during its execution. In the following subsection, we prove that it operates correctly, i.e., that it has properties (*D*1) and (*D*2), too. Note that the assumptions upon routing and queue handling stated in Section II-A are not needed for the detection algorithm to be correct. These assumptions are important when a resolution algorithm is designed, as we show in Section V.

Finally, it is easy to see that the number of control messages

Detection Algorithm for the Leader (L)

<1> For *NOT FULL*
  <1.1> If became *NOT FULL* by sending a data message to neighbor l, then perform <6.2.1>-<6.2.3>; Else $ST(L) \leftarrow 0$, $FR_L \leftarrow nil$, Send *FREE* messages to all neighbors
<2> For *FULL* or *REQ*
  <2.1> If $Q(L) = ASLEEP$ then *START*; Else hold *REQ*
<3> For *START*
  <3.1> If $Q(L) = ASLEEP$ then do:
    <3.1.1> $Q(L) \leftarrow AWAKE$
    <3.1.2> Send *START* messages to all tree-neighbors
    <3.1.3> $DS(L) \leftarrow$ *Not-Deadlocked*
    <3.1.4> If full then $ST(L) \leftarrow 1$; Else *NOT FULL*
  <3.2> If $Q(L) = AWAKE$ then do:
    <3.2.1> If received *START* messages from all tree-neighbors then send *TERMINATE* messages to all tree-neighbors
<4> For *TERMINATE*
  <4.1> If received *TERMINATE* messages from all tree-neighbors then send *INFORM* messages to all tree-neighbors (if freeing mechanism is in progress wait until it is completed)
<5> For *INFORM*($X(l)$)
  <5.1> If $X(l) = 1$ then $X(L) \leftarrow 1$
  <5.2> If received *INFORM* messages from all tree-neighbors then do:
    <5.2.1> If $ST(L) = 1$ then L is in a *DEADLOCK*; $DS(L) \leftarrow$ *Deadlocked*; $X(L) \leftarrow 1$
    <5.2.2> $Q(L) \leftarrow ASLEEP$
    <5.2.3> If have *REQ* then *START* and delete *REQ*
<6> For *FREE* from neighbor l
  <6.1> If $ST(L) = 0$ then send *ACK* to l
  <6.2> If $ST(L) = 1$ and L has a packet to l then do:
    <6.2.1> $ST(L) \leftarrow 0$
    <6.2.2> $FR_L \leftarrow l$
    <6.2.3> Send *FREE* messages to all neighbors except FR_L
<7> For *ACK*
  <7.1> If received *ACK* message from all neighbors then send *ACK* messages to FR_L

Detection Algorithm for node i

<1> For *NOT FULL*
  <1.1> If became *NOT FULL* by sending a data message to neighbor l, then perform <6.2.1>-<6.2.3>; Else $ST(i) \leftarrow 0$, $FR_i \leftarrow nil$, Send *FREE* messages to all neighbors
<2> For *FULL* or *REQ*
  <2.1> If $Q(i) = ASLEEP$ then send *REQ* to p_i; Else hold *REQ*
<3> For *START*
  <3.1> If $Q(i) = ASLEEP$ then do:
    <3.1.1> $Q(i) \leftarrow AWAKE$
    <3.1.2> Send *START* messages to all tree-neighbors except p_i
    <3.1.3> $DS(i) \leftarrow$ *Not-Deadlocked*
    <3.1.4> If full then $ST(i) \leftarrow 1$; Else *NOT FULL*
  <3.2> If $Q(L) = AWAKE$ then do:
    <3.2.1> If received *START* messages from all tree-neighbors then send *START* message to p_i
<4> For *TERMINATE*
  <4.1> If received *TERMINATE* message from p_i then send *TERMINATE* messages to all tree-neighbors except p_i
  <4.2> If received *TERMINATE* messages from all tree-neighbors then send *TERMINATE* message to p_i (if freeing mechanism is in progress wait until it is completed)
<5> For *INFORM*($X(l)$)
  <5.1> If $X(l) = 1$ then $X(L) \leftarrow 1$
  <5.2> If received *INFORM* message from p_i then send *INFORM* messages to all tree-neighbors except p_i
  <5.3> If received *INFORM* messages from all tree-neighbors then do:
    <5.3.1> If $ST(i) = 1$ then i is in a *DEADLOCK*; $DS(i) \leftarrow$ *Deadlocked*; $X(i) \leftarrow 1$
    <5.3.2> Send *INFORM*($X(i)$) to p_i
    <5.3.3> $Q(i) \leftarrow ASLEEP$
    <5.3.4> If have *REQ* then send it to p_i and delete it
<6> For *FREE* from neighbor l (upon *AWAKE*)
  <6.1> If $ST(i) = 0$ then send *ACK* to l
  <6.2> If $ST(i) = 1$ and i has a packet to l then do:
    <6.2.1> $ST(i) \leftarrow 0$
    <6.2.2> $FR_i \leftarrow l$
    <6.2.3> Send *FREE* messages to all neighbors except FR_i
<7> For *ACK*
  <7.1> If received *ACK* message from all neighbors then send *ACK* to FR_i

Fig. 3. Detection algorithm.

exchanged during one cycle is $O(|L|)$ (where $|L|$ is the cardinality of the set L), since in the worst case two messages (*FREE* and *ACK*) will be transmitted on each link, in addition to six messages on each of the links of the spanning tree. Assuming that the maximal propagation delay for a message is one unit of time, the time required to complete one cycle is $O(|N|)$.

C. Validation of the DDA

The DDA presented above operates in cycles. Each cycle is composed of three phases, each of which is started by the Leader and ends when the respective control messages are received by the Leader from all its tree neighbors. From the way the control messages of each phase propagate, it is obvious that the *START* and the *INFORM* phases are completed in finite time. We now show that the *TERMINATE* phase also ends in finite time.

Lemma 2: The *TERMINATE* phase ends in finite time.

Proof: Assume the contrary. Then there exists a node, say i_1, that started a freeing mechanism and it never receives an *ACK* message from all its neighbors. Hence, one of its neighbors, say i_2, started a freeing mechanism upon receiving a *FREE* message from i_1 (otherwise, it sends an immediate *ACK*) and never received *ACK* messages from all its neighbors. Applying the same argument, it follows that there must exist nodes $i_1, i_2, i_3, \cdots$, such that i_{j+1} started a freeing mechanism upon receiving a *FREE* message from i_j and each of them never received *ACK* messages from all its neighbors. However, since the number of nodes in the system is finite, and since the response time to a control message is finite, there is some j for which $i_j = i_l$ for some $1 \leq l \leq j - 1$, hence contradiction.

Now that we showed that each of the phases of a cycle completes in a finite time, it immediately follows that a cycle completes in finite time. Therefore, an *REQ* message generated at some node arrives to the Leader in finite time.

We now prove that our DDA has property ($D1$).

Theorem 2: A node that is not deadlocked has $ST(i) = 0$ when it receives an *INFORM* message.

Proof: Assume that at least one node in the network, say node i, is not deadlocked (otherwise, the theorem trivially holds), and that the theorem does not hold, i.e., $ST(i) = 1$ when node i receives an *INFORM* message. Obviously, node i was full since the beginning (when it received a *START* message) of the cycle (otherwise, it would have set $ST(i) = 0$). Let *ND* be the set of all nodes such as i, i.e., full, not deadlocked, and have $ST(i) = 1$ when completing the *INFORM* phase. If all nodes in *ND* have packets only to nodes within *ND* or to deadlocked nodes, then they are deadlocked, contradicting our assumption about *ND*. So at least one node k in *ND* has a packet to a node l that is not deadlocked and is not in *ND*. Note that $ST(l) = 0$ since if l is not full $ST(l) = 0$ automatically and if l is full then $ST(l) = 0$ by the assumption $l \notin ND$. When l set $ST(l)$ to 0, it started a freeing mechanism by sending *FREE* messages to all its neighbors. Hence, k should have set $ST(k)$ to 0 (otherwise, l would not have received an *ACK* from k and the *TERMINATE* phase would not have been completed yet), contradicting the fact that k is in *ND*.

The next theorem shows that our DDA has property ($D2$). We assume that no resolution mechanisms are applied to deadlocked nodes.

Theorem 3: If a node becomes deadlocked, then in finite time it will find it out.

Proof: A node i becomes deadlocked only if it remains full forever. At the instant i becomes deadlocked, some node becomes full and generates an *REQ* message. This *REQ* message arrives at the Leader in finite time, causing it to initiate a cycle. As a cycle ends in finite time, we only need to show that $ST(i) = 1$ when node i sends an *INFORM* message to p_i. Assume that there are nodes that are deadlocked, started the cycle with $ST(i) = 1$, but ended it with $ST(i) = 0$ (thus not detecting that they are deadlocked). Let n_0 be the first node that changed $ST(n_0) = 1$ to $ST(n_0) = 0$ among the deadlocked nodes. To do that it must have received a *FREE* message from

a neighbor to which it has a packet to send, that by definition is also deadlocked, contradicting the fact that n_0 was the first to change *ST*.

The importance of the final lemma that we prove will become apparent when the deadlock resolution algorithm is presented.

Lemma 3: If some node finds out that it is deadlocked, then the Leader will know that in finite time.

Proof: Straightforward, as when a node becomes aware that it is deadlocked, it sets $X(i) \leftarrow 1$ and that information is propagated to the Leader during the *INFORM* phase.

V. Deadlock Resolution Algorithm (DRA)

Once a deadlock has been detected, one should have some mechanism to resolve that deadlock. In [7] Gambosi *et al.* have proposed a deadlock resolution algorithm. Their basic idea is that deadlocked nodes will release packets through dedicated reserved buffers called *exchange buffers*. Each node reserves exactly one buffer for this purpose. Unfortunately, the algorithm of [7] may lead to deadlock among the exchange buffers. We use similar ideas to those in [7] to devise a deadlock resolution algorithm, but by exploiting the possibility for network-wide coordination (through the Leader) we avoid the difficulties of the algorithm in [7].

A. The Removal Process

The building block of the distributed deadlock resolution algorithm that we present is a *removal iteration* that enables one packet from a deadlocked node to leave the network in finite time.

The removal iteration has two phases, both of which are started by the Leader. The first phase is devoted for determining the deadlocked node that will release a packet and during the second phase, that packet is actually released. The first phase is triggered by the end of a previous removal iteration or when the Leader finds out that a deadlock exists in the network.

The first phase is started by the Leader sending *SEARCH* control messages to its tree neighbors. These messages propagate in the network as the *START* messages of the DDA. *SEARCH* messages contain the identity of the sender, as well as some other information *MN* (can be *nil*) that is used to determine the node that will release a packet which is the node with the lowest identity among the nodes of the current deadlock set. Upon receiving a *SEARCH* message, a node i that is not deadlocked sets its local variable $MN(i)$ to that received from neighbor l if $MN(i) > MN(l)$. A deadlocked node i compares its identity to $MN(l)$, and if it is smaller than $MN(l)$, it changes $MN(i)$ to i. Eventually, *SEARCH* messages arrive at the Leader from all its tree neighbors and it knows the node with the lowest identity among the nodes of the current deadlock set and that node will release a packet.

During the second phase, the Leader broadcasts the identity of the chosen node with *RELEASE* messages (through the spanning tree). When a node receives its identity, it starts to release a packet at that iteration. To that end we assume, as in [7], that all nodes have an extra buffer, called an *exchange buffer* that is dedicated for removal purposes and is otherwise unoccupied. The chosen node then picks one of its packets, tags it, and sends it to the intended neighbor (according to the routing tables) and leaves the deadlock set ($DS(i) \leftarrow$ *Not Deadlocked*). That neighbor stores the tagged packet in its exchange buffer and then forwards it again according to its routing table. Thus, that packet is forwarded along a route of exchange buffers. As there is a single tagged packet within the network at any time, there is no way that the packet will be blocked at some node and that it would not be forwarded. Assuming that routing decisions are "reasonable" (packets do not loop permanently within the network), the tagged packet will eventually arrive at its destination and therefore leave the network. Then the node at which the tagged packet has been consumed notifies the Leader (through the tree) of that event by sending an *END_RELEASE* message. When this message arrives at the Leader, the second phase of the resolution iteration ends. Note that at the end of the removal iteration, the exchange buffers at all nodes are empty and one packet has been deleted from the chosen deadlocked node. Then other similar removal iterations are performed; each time the current node with the lowest identity among the nodes that are still deadlocked is chosen to be the node that will start to release a packet. This continues until each node that belonged to the deadlock set upon initiation of the DRA has released a packet and thus became not full (at least temporarily) and therefore, not deadlocked. During the execution of the DRA, the Leader does not start any detection cycle. Only when the DRA ends ($MN(L) = nil$ when *SEARCH* messages are received from all tree neighbors), then if the Leader is required to start a detection cycle (by an *REQ* message), it does. In any case, the goal of resolving the deadlock is achieved.

Several remarks are in place here. The first is that, at each node along the route, the tagged packet traverses, it can be *replaced* by any other packet at that node. The replaced packet would become the new tagged packet and forwarded along the exchange buffers and the previous tagged packet would remain at the node. (Replacements are assumed to be atomic actions.) Such replacements would not invalidate the correctness of the removal iteration. The reason is that some packet is forwarded during the second phase and if routing is "reasonable," some packet will leave the network (be consumed at some node). Note that if replacements take place, it is unlikely that the packet from the chosen node (where the removal starts) will leave the network. Yet, at the end of the removal iteration, the network would be in the same situation as it would have been without replacements, namely, all exchange buffers are empty and there is one less packet at the chosen node.

Allowing replacements has several important implications. The first is that it enables some optimization during the second phase. To speed up the execution of the second phase, a node with the tagged packet may try to replace it with a packet whose destination is "closer." In particular, if the node has a packet destined to one of its neighbors, it may replace it with the tagged packet and thus the second phase will terminate more quickly. Replacements are also important because they enable to use removal iterations in networks that preserve the first-in-first-out rule along sessions paths as SNA [16] and TYMNET [17] do. When a tagged packet is at some node and it should be sent to some neighbor, then it is replaced by a packet at the head of the queue of some outgoing link while the previous tagged packet is put at the end of the corresponding queue and thus the FIFO property is preserved if at each outgoing queue packets are served in a FIFO order.

Note that it is not essential that the released packet will arrive at its destination. Whenever the packet arrives at a not-full node, it can leave the exchange buffer and be put in a message buffer, and that not-full node informs the Leader that the packet has been released. Such a modification will speed up the execution of the second phase of the removal iteration without invalidating its correctness.

Finally, we have to specify how the whole resolution algorithm is triggered, namely, how the Leader finds out that there is a deadlock. The simplest way of doing that is that whenever a node discovers that it is deadlocked (by some deadlock detection algorithm) it generates a *DEADLOCK* message that is forwarded through the tree to the Leader (as is done with the *REQ* messages of our detection algorithm). When a *DEADLOCK* message arrives at the Leader, it becomes aware that the resolution algorithm has to be initiated. When the specific detection algorithm described in Section IV is used to detect deadlocks, then there is no need

Resolution Algorithm for the Leader (L)

Replace <5.2.3> of Fig. 3 by:
<5.2.3> If $X(L) = 1$ then do:
<5.2.3.1> If $DS(L) = Deadlocked$ then $MN(L) \leftarrow L$; Else $MN(L) \leftarrow nil$
<5.2.3.2> send $SEARCH(L,MN(L))$ to all tree-neighbors
Else if have REQ then $Q(L) \leftarrow SLEEP$ and $START$; Delete REQ
<8> For $SEARCH(l,MN(l))$
<8.1> If $MN(l) = nil$ then discard the message
Else if $MN(L) = nil$ then $MN(L) \leftarrow MN(l)$
Else if $MN(l) < MN(L)$ then $MN(L) \leftarrow MN(l)$
<8.2> If Received $SEARCH$ messages from all tree-neighbors then do:
<8.2.1> If $MN(L) = nil$ then if have REQ then $Q(L) \leftarrow SLEEP$ and $START$; Delete REQ
<8.2.2> If $MN(L) = L$ then $REMOVE$
Else send $RELEASE(MN(L))$ to all tree-neighbors
<9> For $RELEASE(l,MN)$
Discard the message
<10> For $END\_RELEASE$
<10.1> $MN(L) = nil$
<10.2> Send $SEARCH(L,MN(L))$ to all tree-neighbors
<11> For $REMOVE$
<11.1> Pick and tag one message; Put it in exchange buffer; Forward it according to routing table; Delete it from exchange buffer
<12> For $TAGGED$ message
If L is the destination of the message then $END\_RELEASE$; Else put it in exchange buffer; Forward it according to routing table; Delete it from exchange buffer

Resolution Algorithm for node i

<8> For $SEARCH(l,MN)$
<8.1> If received $SEARCH$ message from p_l then if $DS(i) = Deadlocked$ then $MN(i) \leftarrow i$; Else $MN(i) \leftarrow = nil$
<8.2> If $MN(l) = nil$ then discard the message
Else if $MN(i) = nil$ then $MN(i) \leftarrow MN(l)$
Else if $MN(l) \leq MN(i)$ then $MN(i) \leftarrow MN(l)$
<8.4> If $l = p_l$ then send $SEARCH(i, MN(i))$ to all neighbors except p_l
<8.5> If received $SEARCH$ messages from all tree-neighbors then send $SEARCH(i,MN(i))$ to p_l
<9> For $RELEASE(l,MN)$
<9.1> If $i \neq MN$ then do:
<9.1.1> If $l = p_l$ then send $RELEASE(i, MN)$ to all tree-neighbors except p_l; Else discard the message
<9.2> If $i = MN$ then $REMOVE$; $DS(i) \leftarrow Not\text{-}Deadlocked$
<10> For $END\_RELEASE$
Send it to p_l
<11> For $REMOVE$
<11.1> Pick and tag one message; Put it in exchange buffer; Forward it according to routing table; Delete it from exchange buffer
<12> For $TAGGED$ message
If i is the destination of the message then send $END\_RELEASE$ to p_l; Else put it in exchange buffer; Forward it according to routing table; Delete it from exchange buffer

Fig. 4. Resolution algorithm.

for special messages as the *DEADLOCK* messages, since the *INFORM* messages can carry [in the variable $X(i)$] the information that a deadlock exists (see Lemma 6). In the formal specification of the algorithms performed by the Leader and the nodes in the network given in Fig. 4, the latter method is used.

Note that if only one packet is released from each of the deadlocked set, it is very likely that a DDA would have to start, or even worse, that a deadlock would recur soon. To overcome this we may very well perform the second phase of the removal algorithm several times and thus releasing more than only one packet from the chosen node. To take it to the extreme, the chosen node might be emptied. The question of exactly how many packets should be released from each chosen node, is up to the designer of the network.

B. Validation of the DRA

Theorem 4: In DRA, if node n becomes deadlocked at time t then it becomes not full in finite time after t.

Proof: Assume that node i is full when it becomes deadlocked (otherwise, the theorem trivially holds). This implies that a DDA would be activated and upon its termination node n (and possibly other nodes as well) found out that it is deadlocked. From Lemma 6, it follows that the DRA is activated and removal iterations are started (or if a different DDA is used it follows from the *DEADLOCK* message discussion in Section V-A). In each removal iteration, at least one node that is deadlocked and full, becomes not full and thus leaves the deadlock set. Each removal iteration ends in finite time (given the assumption of "reasonable" routing) and therefore n's turn to release a packet will come (if it has not become not-full before, in which case the proof is completed) in finite time.

VI. Summary

Simple distributed algorithms for the detection and resolution of deadlocks in store-and-forward computer communication networks have been presented and validated. The algorithms find and resolve buffer deadlocks using only a fixed number of buffers per node plus a fixed number of buffers per adjacent link. When the detection and resolution algorithms are merged, it is guaranteed that packets will arrive at their destinations in finite time.

Distributed deadlock detection algorithms that does not assume any *a priori* structure among the nodes of the network and still use a finite amount of storage at each node for their execution have been presented in [13], [14], and [20]. The absence of such structure tremendously complicates the algorithms and makes it difficult to devise a resolution algorithm. Thus, the present detection algorithm is considerably simpler than those of [13], [14], and [20]. Finally, we note that the control messages exchanged by nodes in the detection algorithms in [13], [14], and [20] contain sequence numbers that may theoretically grow unboundedly, while in the simpler algorithms presented here, no such numbers are employed.

References

[1] K. D. Günther, "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, pp. 345–352, Mar. 1980.

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

[4] S. Toueg and J. D. Ullman, "Deadlock-free packet switching networks," *SIAM J. Comput.*, vol. 10, pp. 594–611, Aug. 1981.

[5] J. D. Atkins, "Path control—the network layer of system network architecture," in *Computer Network Architectures and Protocols*, P. E. Green, Jr., Ed. New York: Plenum, 1982, pp. 297–326.

[6] S. Wecker, "DNA—the digital network architecture," in *Computer Network Architectures and Protocols*, P. E. Green, Jr., Ed. New York: Plenum, 1982, pp. 249–296.

[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. New York: Elsevier, 1984, pp. 219–229.

[8] D. A. Menascoe and R. Muntz, "Locking and deadlock detection in distributed databases," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 195–202, May 1979.

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

[10] K. M. Chandy and J. Misra, "A distributed algorithm for detecting resource deadlocks in distributed systems," in *Proc. ACM SIGACT-SIGOPS Symp. Prin. Distr. Comput.* (Ottawa, Canada, Aug. 1982), ACM, New York, 1983, pp. 157–164.

[11] 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.

[12] G. Bracha and S. Toueg, "A distributed algorithm for generalized deadlock detection," *PODC*, pp. 285–301, 1984.

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

[14] ——, "Local distributed deadlock detection by cycle detection and

clustering," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 3–14, Jan. 1987.
[15] R. G. Gallager, "Choosing a leader in a network," Internal Memorandum, M.I.T., 1982.
[16] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum weight spanning trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 66–77, Jan. 1983.
[17] A. Segall, "Distributed network protocols," *IEEE Trans. Inform. Theory*, vol. IT-29, Jan. 1983.
[18] J. D. Atkins, "Path control—the network layer of system network architecture," in *Computer Network Architectures and Protocols*, P. E. Green, Jr., Ed. New York: Plenum, 1982, pp. 297–326.
[19] L. Tymes, "Routing and flow control in TYMNET," *IEEE Trans. Commun.*, vol. COM-29, pp. 392–398, Apr. 1981.
[20] N. Natarajan, "A distributed scheme for detecting communication deadlocks," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 531–537, Apr. 1986.

★

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

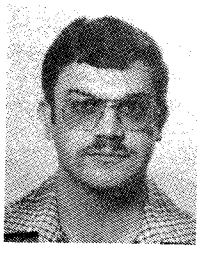
From 1980 to 1984, he was a Teaching Assistant and a Teaching Instructor at the Technion. From 1984 to 1985, he was a Faculty member with the Faculty of Electrical Engineering at the Technion. Since 1985, he has been with IBM Thomas J. Watson Research Center, Yorktown Heights, NY. His current research interests are in distributed algorithms and communication networks.

Jeffrey M. Jaffe (M'80–SM'86) received the B.S. degree in mathematics, and the M.S. and Ph.D. degrees in computer science from the Massachusetts Institute of Technology, Cambridge, MA, in 1976, 1977, and 1979, respectively. He is currently the Department Manager of the Department of Communications Systems. In 1984–1985, he spent a sabbatical year at the IBM Scientific Center, Haifa, Israel.

He has been a Research Staff Member with the Department of Computer Sciences at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, since 1979. 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. In 1982 and again in 1986, Dr. Jaffe received IBM Outstanding Innovation Awards for his work in dynamic routing. He has also received two divisional awards and two patent awards within IBM.

Dr. Jaffe's current professional activities include being a 1986–1987 IEEE Communications Society Distinguished Lecturer, an Editor for the IEEE TRANSACTIONS ON COMMUNICATIONS, and the Technical Program Chairman for the 1988 Computer Networking Symposium. Dr. Jaffe is a member of ACM and Phi Beta Kappa.

★



Moshe Sidi (S'77–M'82) 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 the Department of Electrical Engineering at the Technion. During the academic year 1983–1984, he was a Post-Doctoral Associate at the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology, Cambridge, MA. During the academic year 1986–1987, he was on a sabbatical leave with IBM Thomas J. Watson Research Center, Yorktown Heights, NY. His current research interests are in queueing systems and in the area of computer communication networks.