Algorithmica (1989) 4: 417–436

Algorithmica

Distributed Deadlock Resolution in Store-and-Forward Networks

Jeffrey M. Jaffe[1] and Moshe Sidi[2]

Abstract. We present a simple distributed algorithm that resolves store-and-forward deadlocks in data communication networks. The basic idea of the algorithm is to detect cycles of nodes that may cause store-and-forward deadlocks, and to rotate packets along these cycles. The algorithm uses a fixed amount of storage in each node for its execution, and, under reasonable assumptions upon the routing and packet handling, it ensures that packets that enter the network arrive at their destinations in finite time.

Key Words. Deadlock resolution, Distributed algorithm, Store-and-forward networks.

1. Introduction. Store-and-forward deadlocks prevent packets in the nodes involved in a deadlock to arrive at their destinations. Therefore, the buffer deadlock problem is one of the crucial problems in the design of store-and-forward computer communication networks. There are two traditional approaches to handle deadlocks. One is to design networks and schemes that prevent the occurrence of deadlocks [1]. For instance, packets can be forwarded according to an acyclic buffer graph [2], or the buffer pool can be divided into buffer classes and packets are forwarded through these classes in a manner that prevents deadlocks [3]. The main drawback of this approach is that it introduces restrictions on the possible transmissions of packets, causing inefficient utilization of the buffers and thus limiting the potential throughput of the network.

The other traditional approach is not to prevent deadlocks, but to have a distributed algorithm that detects a deadlock when it is formed and then to extricate the network from the deadlock situation by some distributed resolution algorithm [6], [14]. Many distributed deadlock detection algorithms have been suggested [6]–[15]. In those of [6]–[11], the amount of local storage required at each node to perform the detection algorithm grows (at least) proportionally with the size of the network (number of nodes in the network). This is an unacceptable property in store-and-forward networks, since a buffer deadlock at the network level implies storage shortage and there might not be enough buffers to execute the detection algorithm when a deadlock exists! Hence, in store-and-forward networks the deadlock detection and resolution problems require that

[1] IBM, Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA.
[2] Electrical Engineering Department, Technion, Israel Institute of Technology, Haifa 32000, Israel. Part of this work was done while this author was on sabbatical leave with IBM, Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA.

Received December 31, 1987; revised May 29, 1988. Communicated by Israel Cidon and Inder S. Gopal.

nodes be able to solve them using a *finite* number (independent of the network size) of auxiliary buffers specifically set aside for that purpose at each node [12], [14], [15], [17]. However, the latter property implies rather slow operation of the detection [12]-[15] and resolution [14] algorithms, since a node can store only a finite number of control messages at any time. In addition, unless some global coordination is used [14], a deadlock resolution procedure with a finite number of buffers that operates only when deadlocks are identified, has never been reported.

In this paper we take a new approach to resolving deadlocks, when they are formed. We waive the "challenge" of detecting nodes involved in a deadlock, and the attempt to identify a deadlock, when one exists. Instead, we concentrate on a rather fast operation—distributively detecting cycles of nodes that *might* cause deadlocks. Since deadlocks always contain a cycle (or cycles) of *full* nodes, detecting such a cycle is treated as a *potential deadlock* that triggers a fast resolution phase during which packets are rotated along the detected cycle. This rotation causes some packets to be forwarded toward their destinations. When routing is "reasonable," iterating these actions of detecting a cycle and rotating packets, ensures that if there is a deadlock in the network, then at least one of the nodes that causes the deadlock will become not-full, thus resolving the deadlock. Moreover, with appropriate assumptions upon the way a node serves its neighbors, we show that with our distributed resolution algorithm, deadlocks will be transparent to the users of the network in the sense that it is guaranteed that each packet that enters the network will be delivered to its destination in finite time, though deadlocks might be formed (and implicitly resolved) during the operation of the network. This is done without *explicit detection* that a deadlock exists and without any preallocation of resources for *each* packet that enters the network. The amount of control buffers needed for performing the algorithm is finite and it does not change the routing tables at the nodes, namely no data packets are routed on links other than those specified by the routing algorithm.

The approach of having a "background" algorithm that resolves (or prevents) deadlocks without detecting them have been recently used in [17], where a time-stamp approach has been utilized. We further discuss the algorithm of [17] in Section 5. The rest of the paper is organized as follows. In Section 2 we present the underlying model that we use. In Section 3 we introduce our resolution algorithm and in Section 4 we validate its correctness. Section 5 contains a discussion.

2. The Model. The model that we use here is similar to that of [12]. In the following we describe the model of a communication node of the network and introduce the underlying assumptions on the operation of the network and the way messages are handled. Then we give a formal definition of store-and-forward deadlocks and of deadlock resolution algorithms.

2.1. Model of a Communication Node. We assume that the total storage available at a node of the network is *finite* and that it is divided into three parts (see Figure 1):

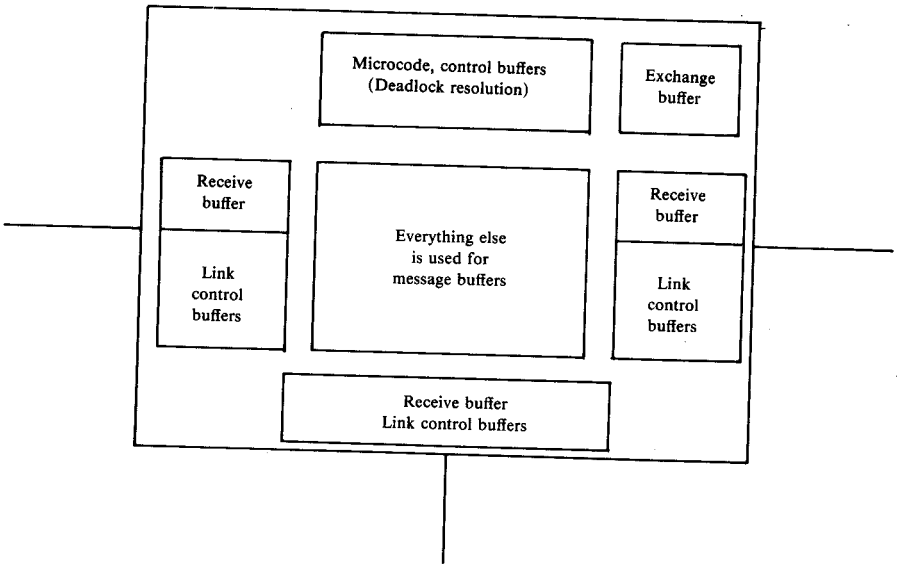

Fig. 1

(i) Overhead storage—needed for overhead that includes the code for the machine, data structures, variables, control blocks, etc.

(ii) Storage for emergency measures (such as deadlock 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 we do not know how large the network is, or whether it will grow over time.) Thus, in the overhead portion, we may reserve a *fixed number of shared control buffers* to aid in deadlock resolution. In addition, we allow *a fixed amount of storage per link* to be reserved in the overhead portion for the resolution of deadlocks. Thus the total storage allowable for the entire deadlock resolution procedure is a constant number of control buffers per adjacent link.

(iii) Message buffers for transit traffic.

The way that message buffers are used is as follows: for each incoming link, a single *receive* buffer is reserved and whenever a packet arrives via that link, it enters the receive buffer (if it is not occupied). The rest of the message buffers 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 receives that message in the corresponding receive buffer (if it is empty). Once in a receive buffer, the node looks whether it is the final destination of that message. If so, that message is consumed and the receive buffer it occupied is freed. Otherwise, the node determines the outgoing link for that message and assigns the message to the relevant outgoing queue. If at least one of the shared buffers is free, then the message is moved from the receive buffer to a free shared buffer and the receive buffer is freed. (We assume that this entire message processing step is an atomic action and, in practice, pointers are used to indicate packet movements within a node.) When the message is

completely transmitted, the buffer it occupies is freed. If the receive buffer is not 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 (from the shared buffers) is allocated for it, and if not, the packet stays in the same machine, but does not enter the communication subsystem.

2.2. Network Model. A *network* consists of a set of *communication nodes* N and a set of *communication 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. A packet consists of a fixed-size *header portion* and a variable-size *data portion.* Each message buffer can contain one packet. (In principle, a user of a communication network might generate messages of unbounded size. Generally, before such a message reaches the "network layer," it is segmented into bounded-sized packets. Our study of deadlocks relates to deadlocks involving these packets, not to higher-level reassembly deadlock that may occur.) The header of a packet includes information which theoretically requires an unbounded number of bits to specify. For example, the header includes the network address of the destination, even though one needs $\log n$ bits in a network of n nodes. In addition, the header might include various sequence number fields (e.g., each node may assign successive sequence numbers to messages it initiates). For any network, a sequence number field may theoretically grow unboundedly. Nevertheless, in practice, if we use a relatively small number of bits (e.g., 64) for each field, then the network will work as long as there are only a few trillion nodes, and the network must operate for only a few trillion centuries.

For this reason, real networks utilize fixed-size headers [4], [5]. Also, to model real networks accurately, we allow theoretically unbounded sequence number fields and node name fields to be part of a bounded-size packet in our model. (Formally we can argue that we are looking for solutions that work for networks of up to 2^{64} nodes, working for up to 2^{64} units of time, and using packets bounded by a few hundred bytes.)

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 of a packet is determined, the node queues up the packet for that next node. We say that a queue at node i with packets to an adjacent node j is *permanently blocked* if node i can never forward to j any message of the queue. The way a node handles its queue to an adjacent node is not very

important. We only need that none of the messages in a queue will be neglected, i.e.,

ASSUMPTION 2. If a queue is not permanently blocked, then each packet of the queue will eventually be transmitted.

Note that a FIFO discipline is one example in which this assumption holds.

If several of the receive buffers at a node are occupied and all shared buffers are full, then when one of the shared buffers is freed, the node moves a message from one of the receive buffers to the freed shared buffers. The way in which a nodes decides which receive buffer to free is not too important, but we need that none of the receive buffers is neglected, i.e.,

ASSUMPTION 3. If the shared buffers are not permanently full, then each of the receive buffers will be eventually freed.

Note that one example in which this assumption holds is when a node frees its receive buffers in a round-robin fashion.

Essentially, the three assumptions guarantee that packets arrive at their destinations in finite time, under normal conditions (i.e., no store-and-forward deadlocks). If any of these assumptions does not hold, packets might never arrive at their destinations even if deadlocks are never formed.

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.

2.3. Store-and-Forward Deadlocks. A node is said to be *blocked* if all its shared buffers are occupied with packets destined to neighbors whose corresponding receive buffers are occupied; otherwise it is said to be *not-blocked.* Alternatively, we say that a node is in a blocked or a not-blocked state.

If at some time t there is in the network a set T of blocked nodes with the property that all packets in their shared buffers are to be forwarded to nodes in T and also the corresponding receive buffers (the receive buffers that correspond to nodes in T with packets destined to these receive buffers) are occupied with packets to be forwarded to nodes in T, then we say that at time t there is a *deadlock* in the network. The reason is that if no special measures are taken, then the nodes of T will not be able to ever forward any of the packets in their shared buffers. The set T is called a *tie.* A tie T of which any subset of T is not a tie is called a *knot.* Obviously, any tie contains at least one knot. Note that the nodes of a knot are causing the deadlock.

Consider one packet at a blocked node i, say the oldest one among the shared buffers, and denote by $NEXT_i$ the neighbor to which that oldest packet should be forwarded. A set of blocked nodes $i_1, i_2, \ldots, i_l$ with $NEXT_{i_j} = i_j + 1, 1 \leq j \leq l-1$, and $NEXT_{i_l} = i_1$ is called a *cycle.* Obviously, each knot contains at least one cycle. However, the existence of a cycle does not imply the existence of a knot.

2.4. Resolution Algorithms. We define a deadlock resolution algorithm as an algorithm with the following properties:

(RA1) If there is a knot in the network at time t, then in finite time after t the knot is resolved, i.e., at least one of its nodes becomes not-blocked.
(RA2) No packets are lost during its execution.

Before proceeding with a presentation of a deadlock resolution algorithm, we state and prove the following important theorem:

THEOREM 1. *Let Assumptions* 1–3 *upon routing and queue and packet handling stated in Section* 2.2 *hold. Assume that a network operates with a deadlock resolution algorithm with the properties stated above. 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. Because of the assumption upon reasonable routing (Assumption 1) and since packets are not lost (property (RA2)), the packet must be permanently stuck at some node after some time t_0. By Assumption 2, it must be that the outgoing queue that it belongs to is permanently blocked after some time $t_1 \geq t_0$. This implies that the packet is to be sent to a node whose corresponding receive buffer and all its shared buffers are permanently occupied (otherwise, by Assumption 3, the outgoing queue to which the packet belongs would not have been permanently blocked) after some time $t_2 \geq t_1$. This situation implies that the packet at the receive buffer and the shared buffers at this neighbor are all to be forwarded to neighbors whose corresponding receive buffers and their shared buffers are also permanently occupied (again because of Assumption 3). Continuing this reasoning we conclude that (since the number of nodes in the network is finite) there is a knot in the network that none of its nodes ever becomes not-blocked, contradicting the assumption that the network operates with a deadlock resolution algorithm with property (RA1). □

Note that the property that each packet eventually arrives at its destination is the ultimate goal of any resolution algorithm. Some other desirable properties of a deadlock resolution algorithm have been summarized in [17]. The algorithm should use only the (*fixed* amount of) storage reserved for its execution in the overhead and link control portions of a node. It should operate in an arbitrary network topology. It should allow for efficient buffer utilization, by allowing to assign any free buffer to a packet requiring it (besides the limited number of buffers that are reserved for control purposes). Finally, the deadlock resolution algorithm should not change the routing tables at the nodes, namely no data packet will be routed on links other than those specified by the routing algorithm, during the execution of the resolution algorithm. We further discuss this property in Section 5.

The main focus of this paper then is to develop a simple distributed deadlock resolution algorithm with the above properties.

3. Description of the Distributed Resolution Algorithm

3.1. Overview. The distributed resolution algorithm that we present in this section guarantees that knots that are formed during the operation of the network are resolved, provided that Assumptions 1–3 stated in Section 2.1 hold. As proved in the previous section, it also ensures that all packets that enter the network arrive at their destinations in finite time.

As alluded before, the algorithm does not attempt to identify a deadlock, undoubtedly, when one exists. Instead, it concentrates on a relatively fast operation—detecting *cycles of blocked nodes* that *might* cause deadlocks. Since deadlocks are always caused by a cycle (or cycles) of blocked nodes, the algorithm treats any detected cycle (of blocked nodes) as a *potential deadlock*, that triggers a rotation phase during which packets are forwarded one hop ahead toward their destination along the detected cycle, as is subsequently described.

Basically, the algorithm is iterative and each iteration is composed, in principle, of two phases. During the first phase an attempt is made to identify a cycle of blocked nodes, using a distributed cycle-detection algorithm. A node that is involved in the first phase (and only blocked nodes are, since only they can cause a deadlock) may either find out that it is not a part of a deadlock (thus completing the iteration without performing the second phase), or a cycle might be detected by one of the nodes of the cycle. That node appoints itself to be the *leader* of the cycle. A detected cycle hints upon a possible deadlock. The leader informs all other nodes of the cycle that they belong to the cycle and then it triggers the second phase of the iteration—the rotation phase. During this phase each node along the cycle forwards a packet one hop ahead, in its turn, using a dedicated buffer called an *exchange buffer* (each node has a single exchange buffer, see Figure 1). In Section 3.2 we describe the two phases in detail.

Conceptually, iterating the two phases described above guarantees that deadlocks would be resolved, because cycles of blocked nodes would be detected and their packets would be forwarded, thus ensuring that they eventually arrive at their destinations.

The main advantage of the approach of this algorithm is that a distributed algorithm for detection of cycles is relatively simple and fast, while detecting a deadlock and identifying the nodes of the deadlock needs rather sophisticated and time- and communication-consuming algorithms [12]–[15]. Evidently, a cycle of blocked nodes that are not part of any deadlock might also be detected during the first phase of an iteration and thus the rotation phase would be activated though it is not obligatory. However, during the rotation phase packets are forwarded in the same manner as they would be forwarded, sooner or later, if the cycle had not been detected. Therefore, this operation is not harmful, and it serves the network, even if a deadlock does not exist.

The algorithm that we describe is to operate in a dynamic environment, where nodes transit between blocked and not-blocked states, due to packet transmissions and receptions. As we shall see, whenever a node transits from a not-blocked state to a blocked state, or when a node ends the rotation phase in a blocked state, it initiates a new iteration of the algorithm. It is necessary to distinguish between

different iterations of the algorithm, otherwise it might happen that cycles of blocked nodes in a knot will not be detected, because control messages of older iterations will arrive to nodes performing newer iterations. To distinguish between different iterations, each control message of some iteration is stamped with an *iteration number.* At the beginning this iteration number is zero at all nodes and then, each time a node starts to perform a new iteration of the algorithm, it increases its iteration number by one and stamps each control message that it sends during this iteration, with the corresponding iteration number.

The details of how iteration numbers are handled in different nodes are described in Section 3.2.4.

3.2. Detailed Description. In the following we describe in more details the two phases of an iteration of the algorithm—the cycle-detection phase and the rotation phase. Then we indicate how different iterations of the algorithm are handled. A pseudocode of the algorithm is given in the Appendix.

3.2.1. The Cycle-Detection Phase. A node that changes its state from a not-blocked to a blocked state starts an iteration of the algorithm by initiating a cycle-detection phase.

A node that performs a cycle-detection phase, can either end that phase by finding out that it belongs to a cycle of blocked nodes, in which case it enters the rotation phase, or it can end the phase in a FREE mode (i.e., not-deadlocked). A node is FREE if it is not-blocked, or if it is blocked but it knows that one of its packets is to be forwarded to a FREE neighbor.

The cycle-detection phase that we describe is similar to that of [18] and [19]. Whenever a node i initiates this phase, it chooses one of its neighbors (say the neighbor to which the oldest packet in its shared buffers should be forwarded), designates it as a next neighbor ($NEXT_i$), and sends a TEST message to it.

A FREE node receiving a TEST message sends back a FREE message. A node that receives a FREE message becomes FREE. Also a node that becomes not-blocked, changes its mode to FREE (except for one specific situation that is described in Section 3.2.2). A node that becomes FREE sends FREE messages to all nodes that it received TEST messages from and discontinues its participation in the current detection phase and thus completes the current iteration of the algorithm.

Any non-FREE node receiving a TEST message from neighbor l, adds l to its list of neighbors for which a TEST message has been received (this list is denoted by SI) and it sends to 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 during the current iteration by this node (from its $NEXT$ neighbor). Each time a new identity is received, it is compared with the currently known maximal identity. If it is larger, it is recorded and sent to all SI neighbors (MAX messages). When a node receives its own identity from its $NEXT$ neighbor, it concludes that a cycle of blocked nodes exists, and that it has the highest identity among the nodes of the cycle. Consequently, it appoints itself to the *leader* of the cycle, and triggers the initiation of a rotation phase.

3.2.2. The Rotation Phase. During the rotation phase packets are forwarded along the detected cycle. Recall that the nodes of the cycle are blocked, so, if they belong to a knot, there is no way to move packets through these nodes. To overcome this we assume (as in [6]) that each node contains a *single* buffer called the *exchange buffer* that is used only during the rotation phase.

The rotation phase is started by the leader l of the detected cycle. It forwards the corresponding packet to the exchange buffer of $NEXT_l$, reserving the buffer that becomes free to the packet that it will receive from its predecessor on the cycle at the end of the rotation phase. A node i of the cycle that receives a packet, stores it in its exchange buffer and then exchanges (again, this is done by changing pointers) that packet with the packet intended to $NEXT_i$ and then forwards the packet in its exchange buffer to $NEXT_i$ and ends the rotation phase, thus completing the current iteration of the algorithm. The leader ends the rotation phase when it receives a packet from its predecessor on the cycle and stores that packet in the free buffer it reserved since the beginning of the rotation phase.

Note that when the rotation phase starts, all the exchange buffers are empty, and when it ends, all the exchange buffers are empty again. The one important thing achieved during the rotation phase is that, in each node of the detected cycle, one packet has been forwarded one hop ahead toward its destination.

Recall that a detected cycle of blocked nodes is not necessarily a part of any knot. Therefore, it is not guaranteed that the cycle will exist until the rotation phase ends, because some nodes of the cycle may become not-blocked after the leader has started the rotation phase. This may lead to deadlocks in the exchange buffer level and we would not be able to guarantee the termination of the rotation phase.

To overcome this problem the leader has to guarantee the *rotation path.* To that end, when a node appoints itself to a leader, it first sends a CYCLE control message forward on its *NEXT* link. A node that receives a CYCLE message but is FREE sends back a CANCEL message, otherwise, it designates itself as part of the cycle and sends forward a CYCLE message on its *NEXT* link. A node that designates itself as part of the cycle will not change its mode to FREE (even if it receives a FREE message or becomes not-blocked during the current iteration—this is the specific situation mentioned in Section 3.2.1) until it participates in the rotation phase, i.e., receives and forwards a transit packet or until it receives a CANCEL message. If the node receives a CANCEL message, on its *NEXT* link, it sends back a CANCEL message to its predecessor on the cycle (from which it received a CYCLE message), it designates that it is no longer a part of a cycle, enters a FREE mode, and behaves as indicated earlier for a node that becomes FREE.

From the above description it is obvious that once a CYCLE message arrives at the leader, the path for the rotation phase is guaranteed, and that phase, as described earlier, is initiated by the leader. The only necessary change is that it might happen that a node in the cycle, say i, will not have its packet to $NEXT_i$ (the node to which i sent TEST and CYCLE messages) when its turn to forward a packet along the cycle arrives (because node i might have forwarded that packet

before, if it was not involved in a deadlock). In that case node i just forwards to $NEXT_i$ a dummy packet to indicate that it completed the rotation phase.

Once a node completes its participation in a rotation phase, it returns to normal operation, or it may start a new iteration by starting a new cycle-detection phase if it is still blocked.

3.2.3. Properties of the Cycle-Detection and Rotation Phases. If no packets (other than those involved in the rotation phase) are moving while the cycle-detection and rotation phases are performed, i.e., the network is static (no node changes its mode from not-blocked to blocked and vice versa), then it follows (see below) that a cycle is detected (if one exists), and one packet of each node in the detected cycle is moved one hop forward.

A simple but important observation is that if there is a knot in the network, it is static (no node in the knot becomes not-blocked, at least until the rotation phase is completed) and it contains a cycle. Consequently, when the two phases are performed by the nodes of the knot we have:

LEMMA 1. *Assume that there exists a knot in the network and the nodes of the knot are performing the cycle-detection phase (all nodes participate in the same iteration). Then:*

(1) *A cycle will be detected and the node with the highest identity in this cycle will appoint itself as the leader of the cycle (in finite time).*
(2) *One packet of each node in the detected cycle is moved one hop forward (in finite time).*

PROOF. We first realize that the only way that packets can be moved within a knot is by completion of the rotation phase by some nodes and this can happen only if (1) and (2) hold. So in the sequel we assume that no packets are moved within the knot (at least until the initiation of the rotations).

Consider a cycle C in the knot. During the iteration a blocked node sends the highest identity it knows, each time it is updated, to all SI neighbors. Let m be the highest identity among the nodes of C. Node m sends its identity to its SI neighbors and, therefore, its predecessor along the cycle, say i ($NEXT_i = m$), will receive that identity. Since m is the highest identity among the nodes of the cycle, i will update (in finite time) the highest identity it knows, to m, and it will send it to all its SI neighbors. Repeating this argument it is obvious that, in finite time, m will receive its own identity on $NEXT_m$, and therefore will appoint itself to the leader of the cycle. Moreover, no other node of the cycle will appoint itself to a leader. This is true since any other node of the cycle has an identity lower than m, and, therefore, that identity will not be forwarded back by node m. This proves (1).

Once m appoints itself to be a leader of C, it starts a procedure to guarantee the rotation path. This procedure will fail only if the CYCLE message sent by the leader, and forwarded by nodes of C, arrives at a FREE node. However, this is impossible since no node in C can complete the iteration in FREE mode (because we assumed that no packet is moved within the knot). Consequently,

the leader will succeed in guaranteeing the rotation path. Then it follows that the nodes of C will complete the rotation phase, advancing one packet (from their shared buffers) one hop ahead, thus proving (2). □

The above properties guarantee that some packets (at nodes of the detected cycle) are making some progress. Intuitively, iterating this procedure will resolve the deadlock. The main difficulty, though, is that the network is not static, and this is addressed in the following section.

3.2.4. Iteration Numbers. The cycle-detection and the rotation phases of each iteration are performed in a dynamic environment where nodes transit between blocked and not-blocked states (due to packets transmissions and receptions). As discussed earlier, in order to ensure the proper operation of the algorithm in such an environment, there is a need to distinguish between different iterations. This is achieved by giving different numbers to different iterations of the algorithm and stamping the control messages (TEST, FREE, etc.) of each iteration with the corresponding 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 at each node.

Each node has a variable (SEQ_i for node i) that indicates the number of the iteration that the node is involved in. This variable is initially set to 0 at each node. Every control message that a node sends is stamped with the current iteration number of the node. Whenever a need arises to start a new iteration at some node, it is started with a higher iteration number (namely, the node, say i, increments SEQ_i by 1). In principle, each iteration is completed to its end, namely, either a blocked node that started to participate in the iteration becomes aware that it is FREE, or it becomes aware that it is a part of a cycle and performs the rotation phase.

We first note that a transition of a node from a blocked state to a not-blocked state, does not require that the node start a new iteration, because that node cannot be part of any knot and therefore cannot cause a deadlock.

The transition of a node from a not-blocked state to a blocked state, can cause a knot and form a deadlock that did not exist earlier, and therefore it is essential that a new iteration of the algorithm be started by the node, to allow the detection of a cycle of blocked nodes and the rotation of packets. Consequently, whenever a node transits to a blocked state, it increments its *iteration number* by one and starts a new iteration of the algorithm with control messages stamped with this new number.

To allow the orderly completion of previous attempts to detect a cycle (previous iterations), priority is given to old iterations of the algorithm (low iteration numbers) over newer ones. A node i attempting to initiate a new iteration (a new cycle-detection phase), by sending a TEST message, to a neighbor j which is performing an older iteration, will have to wait until the completion of all older iterations at j. Until j is finished, j records the *highest* received message (the one with the highest iteration number) for each link, until it can respond to it. After j completes all previous iterations, it responds to messages (if any) of new

iterations. Its response depends on its current state and mode, as is described in the sequel.

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

Generally, each node that ever becomes blocked uses all iteration numbers, i.e., increments its iteration number one by one, and initiates a cycle-detection phase for each new iteration number. This ensures that if a knot exists, all nodes of the knot will reach the highest iteration number started within the knot and a cycle-detection phase and a rotation phase will be completed during that iteration. Otherwise, nodes of a knot may skip that iteration due to attempts of higher iteration numbers received from outside the knot. Nevertheless, a not-blocked node is always FREE (except for the situation described in Section 3.2.2). Therefore, all iterations between the (not-blocked) node's current iteration and the highest number received from its neighbors, are considered as being completed. Consequently, a not-blocked node can immediately respond to the highest iteration number heard by it.

To summarize, the resolution algorithm is operated at each node just as described in Sections 3.2.1 and 3.2.2. The only difference is that the current iteration number of a node is attached to each control message of the algorithm that the node sends. Accepting a message stamped with its current iteration number, the node operates as described in Sections 3.2.1 and 3.2.2. A blocked node which becomes not-blocked acts as a FREE message was received, i.e., sets its mode to FREE and broadcasts FREE messages to its neighbors from which it received TEST messages (except if it received a CYCLE message, in which case it continues to participate in the rotation phase until either it receives a CANCEL message or it forwards a packet). A not-blocked node which becomes blocked increments its iteration number by one and initiates the algorithm by sending 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.

When a node receives messages stamped with a higher iteration number, it does the following. (Recall that a TEST message is always the first to be sent, so we assume such a message has been received.) If the node is in a not-blocked state, then it 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-blocked node cannot be part of a knot. If the node is blocked but has completed its current iteration in FREE mode, then it increments its iteration number by one and initiates a cycle-detection phase for this iteration number. If the message just received has the same iteration number (the node's previous iteration number plus one) it is considered for the current iteration. If it is higher, then it is recorded and kept until the current iteration is completed. If the node is blocked but has not yet completed its current iteration of the algorithm, then it 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. The above implies that any message with a higher iteration number, received by a node at a lower iteration, will not be processed until the node reaches the higher iteration.

Finally, a node that completes the rotation phase sends FREE messages on its links in which it recorded TEST messages with its current iteration number. If it is blocked, it increments its iteration number by one and initiates a new iteration (starts a new cycle-detection phase) with the new number.

Note that the increment in the iteration number occurs only at nodes which have already completed previous iterations. Assume that a node receives, on some incoming link, a control message with some iteration number higher than its own iteration number (this message is recorded). If the node later receives a control message, on the same incoming link, with an iteration number higher than the one recorded, it is obvious that the sending node has already completed the previous iteration. Consequently, a node should consider only the highest iteration received over each incoming link.

Before proceeding with the formal validation of the correctness of the algorithm, let us describe the basic idea in using iteration numbers. Consider a knot in the network. If all nodes of the knot have the same iteration number, then no node in it can increase its iteration number, at least until a rotation phase is completed. Consequently, all the control messages sent within the knot contain the same iteration number, and therefore Lemma 1 holds (for this knot). If the nodes of the knot do not have the same iteration number, then eventually they will increase their numbers to the highest iteration number in the knot when it is formed. This is proved in the following section.

4. Validation of the Algorithm. The resolution algorithm presented in Section 3 ensures that store-and-forward deadlocks that are formed during the operation of the network are resolved, thus (by Theorem 1) ensuring that each packet that enters the network is forwarded to its destination in finite time. This is proved in this section.

THEOREM 2. *Assume that there is a knot in the network at time t and the algorithm presented in Section 3 is in effect. Then the nodes of at least one of the cycles in that knot will forward one packet (from their shared buffers) one hop ahead toward its destination in finite time.*

PROOF. Assume the contrary, i.e., that there is a knot in the network at time t and none of the nodes of any of its cycles ever forward a packet (from the shared buffers). This implies that no packets in the shared buffers of any node in the knot is ever forwarded and none of the cycles of the knot are ever changed.

Let i_1 be the last node to become blocked among the nodes of the knot. Then i_1 starts the algorithm (either immediately, if it is not involved in a rotation phase, or when it completes the rotation phase it is involved in) by sending TEST

message to $NEXT_{i_1} = i_2$. By induction, taking into account that the number of the nodes in the network is finite, it follows that the nodes of at least one cycle in the knot start the algorithm in finite time. Call this cycle C.

Let S be the highest iteration number among the nodes of C when they start the algorithm. We consider two cases:

(i) Not all nodes of C have the same iteration number.
(ii) All nodes of C have the same iteration number S.

Consider first case (i). Divdie the nodes of C into two groups. One contains the nodes with the lowest iteration number $r_{\min}$ and the other contains all other nodes of C. There is at least one node in the first group that sends a TEST message (with iteration number $r_{\min}$) and in finite time receives a FREE message with that number (from a node in the second group). The above implies that all nodes in the first group becomes FREE in finite time in the iteration $r_{\min}$ (either by receiving a FREE message from a node in the second group or from a node in the first group).

At least one TEST message with an iteration number $r' > r_{\min}$ has been sent by a node of the second group and received by a node in the first group, i.e., a node that is blocked and becomes FREE in finite time for iteration number $r_{\min}$. Consequently, that node increases its iteration number by 1 (to $r_{\min}+1$) and starts an iteration with the new number. Thus, we showed that at least one node leaves the first group and joins the second group.

Repeating the above argument we deduce that all nodes of the cycle with iteration number $r_{\min}$, will increase their iteration number by 1 in finite time. Again, by dividing the nodes of the cycle into two groups and repeating the arguments above, we deduce that all nodes of C have, in finite time, the iteration number S (it is easy to see that no node in C can ever have an iteration number higher than S if no rotation is ever performed), and we are back to case (ii).

Thus we only have to consider the case where all nodes of C have the same iteration number S, but then Lemma 1 holds. Consequently, the nodes of C will complete the rotation phase, advancing one packet (from their shared buffers) one hop ahead, contradicting our initial assumption. □

THEOREM 3. *When Assumptions* 1-3 (*stated in Section* 2.2) *hold, the algorithm presented in Section* 3 *is a deadlock resolution algorithm.*

PROOF. That no packets are lost during the execution of the algorithm is obvious, hence (RA2) holds. To show that (RA1) holds, assume the contrary, i.e., that there is a knot that none of its nodes ever becomes not-blocked.

Divide the nodes of the network into two groups: B contains all nodes that are permanently blocked; A contains all other nodes. After a finite time, A is the set of nodes that are not blocked infinitely often (meaning that, for each node i in A and for any time t, there exists some finite time $\tau_i \geq t$ such that node i is not blocked at τ_i). By our assumption, B is not empty. We now consider two cases:

(i) A is empty. In this case, all nodes of the network are permanently blocked, hence there is always a knot in the network. By Theorem 2, some packets

will progress toward their destination, as long as there is a knot in the network. Since the routing is reasonable (Assumption 1, Section 2.2), at least one packet will arrive at its destination in finite time. Hence, at least one node will free at least one of its shared buffers, contradicting the fact that all nodes of the network are in B.

(ii) A is not empty. In this case, each of the receive buffers at nodes in A are freed infinitely often (Assumption 3, Section 2.2). Therefore, after a finite time, a node in B cannot be blocked and have any packets to nodes in A. It follows then that B always contains a knot with no packets to A. By Theorem 2, packets are continually rotated within the knot, but none of the packets ever arrives at its destination and they are always destined to nodes in B (Assumptions 2 and 3, Section 2.2), contradicting our assumption upon reasonable routing (Assumption 1, Section 2.2). □

5. Discussion. A simple distributed algorithm that ensures that packets that enter the network arrive at their destination in finite time, has been presented. This property is achieved without any preallocation of resources for each packet that enters the network, which imply that it will hold even in a network in which the nodes have a small number of shared buffers.

The basic idea of the algorithm is to detect cycles of nodes that may cause a deadlock, and to rotate packets along these cycles. The cycle-detection algorithm presented here is similar to the algorithms in [18] and [19]. When a cycle contains m nodes, all of them having the same iteration number, the algorithm requires $O(m^2)$ messages to detect the cycle and elect a leader in the cycle, in the worst case (the average number of messages required is $O(m \log m)$). Then $O(m)$ control messages are required in the rotation phase. Note that the number of cycle-detection and rotation phases needed to remove a deadlock, depends on the explicit structure of the deadlock and also on the routing of packets within the network. Other algorithms for cycle detection that are known to be more efficient in the worst case (requiring only $O(m \log m)$ control messages), such as the one presented in [20], can also be used, without any changes in other parts (the rotation phase and the iteration numbering) of the algorithm.

An important feature of our deadlock resolution algorithm is that it does not change the routing tables in the nodes. This is important in networks with routing restrictions implied by security and network control management. The resolution algorithm of [17], for instance, does not have this property and therefore it cannot be used in a network that requires fixed routing. Furthermore, the algorithm in [17] imposes an additional cost which is due to the necessity of introducing extra (enforced) hops for some packets.

Independently of our work, the idea of detecting cycles and moving packets along the detected cycles in order to resolve deadlocks, has been used in [16]. The underlying model used in [16] is different from our model. For instance, in [16] the existence of a cycle implies a deadlock, while in our model it does not. In addition, the deadlock detection and resolution algorithm of [16] does not take into account that data packets can still move while the algorithm is in

operation. It is this dynamic environment in which some of the packets are moving that the algorithm has to cope with, and that gives rise to the main difficulties in designing such an algorithm and in proving its correctness.

Finally, we note that the algorithm in [21], for detecting general resource-allocation deadlocks, uses ideas similar to ours (electing leaders in cycles, iteration numbers). Our resolution technique can be combined with that algorithm.

Appendix. Resolution Algorithm for Node i.

Variables at node i

STATE_i	State of node i (Blocked, Not-Blocked).
MODE_i	Mode of node i (FREE, OPERATE, CYCLE); initially FREE.
SEQ_i	Iteration number of node i $(0, 1, 2, \ldots)$; initially 0.
LEADER_i	Indicate if node i is a leader of a cycle (No, Yes); initially No.
NEXT_i	Neighbor to which TEST message has been sent; initially nil.
RETURN_i	Neighbor from which CYCLE message has been received; initially nil.
SI_i	Set of nodes from which TEST messages have been received; initially nil.

Messages sent and received by node i

TEST	Message sent to NEXT_i.
MAX	Message sent to SI_i neighbors (contain the maximal identity known at i).
FREE	Message sent in response to TEST message.
CYCLE	Message sent to establish a cycle.
CANCEL	Message sent to destroy a partially established cycle.
BLOCKED	Local message that indicates that node i became blocked.
NOT-BLOCKED	Local message that indicates that node i became not-blocked.
START	Local message that initiates the algorithm.

REMARK. In the following, whenever a control message is received at i and none of the conditions that are specified hold, then the control message is discarded.

```
For TEST(SEQ_j) from neighbor j
    If STATE_i = Not-Blocked then
        Send FREE(SEQ_j) to j
        If SEQ_j > SEQ_i then SEQ_i ← SEQ_j
    If STATE_i = Blocked and MODE_i = FREE then
        If SEQ_j ≤ SEQ_i then
            Send FREE(SEQ_j) to j
        If SEQ_j > SEQ_i + 1 then record the message and START_nil
```

```
        If SEQ_j = SEQ_i + 1 then START_j
    If STATE_i = Blocked and MODE_i = OPERATE then
        If SEQ_j < SEQ_i then
            Send FREE(SEQ_j) to j
        If SEQ_j = SEQ_i then
            Send MAX(MX_i, SEQ_i) to j
            SI_i ← SI_i ∪ j
        If SEQ_j > SEQ_i then record the message.
    If STATE_i = Blocked and MODE_i = CYCLE then
        If SEQ_j < SEQ_i then
            Send FREE(SEQ_j) to j
        If SEQ_j ≥ SEQ_i then record the message.

For MAX(MX_j, SEQ_j) from neighbor j
    If MODE = OPERATE and SEQ_j = SEQ_i and j = NEXT_i then
        If MX_j = MX_i = i then
            MODE_i ← CYCLE
            LEADER_i ← Yes
            Send CYCLE(SEQ_i) to NEXT_i
        If MX_j > MX_i then
            MX_i ← MX_j
            Send MAX(MX_i, SEQ_i) ∀k ∈ SI_i

For FREE(SEQ_j) from neighbor j
    If STATE_i = Not-Blocked and SEQ_j > SEQ_i then SEQ_i ← SEQ_j
    If STATE_i = Blocked and SEQ_j ≥ SEQ_i then
        MODE_i ← FREE
        Send FREE(SEQ_i) ∀k ∈ SI_i
        SI_i ← nil
        NEXT_i ← nil
        Treat each recorded TEST message as if it has just been received

For CYCLE(SEQ_j) from neighbor j
    If LEADER_i = No then
        If STATE_i = Not-Blocked or (STATE_i = Blocked and MODE_i = FREE)
          then
            Send CANCEL(SEQ_j) to j
        If STATE_i = Blocked and MODE_i = OPERATE then
            If SEQ_j < SEQ_i then send CANCEL(SEQ_j) to j
            If SEQ_j = SEQ_i then
                MODE_i ← CYCLE
                RETURN_i ← j
                Send CYCLE(SEQ_i) to NEXT_i
    If LEADER_i = Yes then
        Send the corresponding data (or dummy) message to the exchange
          buffer of NEXT_i; keep the freed buffer.
```

```
For CANCEL(SEQ_j) from neighbor j
    If LEADER_i = No then
        Send CANCEL(SEQ_j) to RETURN_i
        MODE_i ← FREE
        Send FREE(SEQ_i) ∀k ∈ SI_i (k ≠ RETURN_i)
        SI_i ← nil
        NEXT_i ← nil
        RETURN_i ← nil
        Treat each recorded TEST message as if it has just been received
    If LEADER_i = Yes then
        LEADER_i ← No
        MODE_i ← FREE
        Send FREE(SEQ_i) ∀k ∈ SI_i
        SI_i ← nil
        NEXT_i ← nil
        Treat each recorded TEST message as if it has just been received

For BLOCKED
    If MODE_i = FREE then START_nil

For NOT-BLOCKED
    STATE_i ← Not-Blocked
    If MODE_i = OPERATE then
        MODE_i ← FREE
        Send FREE(SEQ_i) ∀k ∈ SI_i
        SI_i ← nil
        NEXT_i ← nil
        Treat each recorded TEST message as if it has just been received

For START_j
    SEQ_i ← SEQ_i + 1
    MODE_i ← OPERATE
    SI_i ← j
    STATE_i ← Blocked
    Choose NEXT_i
    Send TEST(SEQ_i)
    If j ≠ nil then
        MX_i ← i
        Send MAX(MX_i, SEQ_i) to j

For a data (or dummy) message received in the exchange buffer
    If LEADER_i = No then
        Send the corresponding data (or dummy) message to NEXT_i
        Move the message from the exchange buffer to the freed buffer; if it is
          a dummy message, discard it.
```

If $LEADER_i$ = Yes then
    Move the message from the exchange buffer to the freed buffer; if it is a dummy message, discard it.
    $LEADER_i \leftarrow$ No
$MODE_i \leftarrow$ FREE
Send FREE(SEQ_i) $\forall k \in SI_i$
$SI_i \leftarrow nil$
$NEXT_i \leftarrow nil$
If blocked then BLOCKED
Treat each recorded TEST message as if it has just been received

References

[1] K. D. Günther, Prevention of Deadlocks in Packet-Switched Data Transport Systems, *IEEE Transactions on Communications*, Special issue on Congestion Control in Computer Networks, Vol. 29, No. 6, 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 Transactions on Communications*, Vol. 28, No. 3, pp. 345–352, March 1980.

[3] S. Toueg and J. D. Ullman, Deadlock-Free Packet Switching Networks, *SIAM Journal on Computing*, Vol. 10, pp. 594–611, August 1981.

[4] J. D. Atkins, Path Control—The Network Layer of System Network Architecture, in *Computer Network Architectures and Protocols* (P. E. Green, Jr., ed.), pp. 297–326, Plenum, New York, 1982.

[5] S. Wecker, DNA—The Digital Network Architecture, in *Computer Network Architectures and Protocols* (P. E. Green, Jr., ed.), pp. 249–296, Plenum, New York, 1982.

[6] 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.), pp. 219–229, Elsevier (North-Holland), Amsterdam, 1984.

[7] D. A. Menascoe and R. Muntz, Locking and Deadlock Detection in Distributed Databases, *IEEE Transactions on Software Engineering*, Vol. 5, No. 3, pp. 195–202, May 1979.

[8] R. Obermarck, Distributed Deadlock Detection Algorithm, *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp. 187–208, June 1982.

[9] K. M. Chandy and J. Misra, A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems, in *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Aug. 1982), pp. 157–164, ACM, New York, 1983.

[10] K. M. Chandy, J. Misra, and L. M. Haas, Distributed Deadlock Detection, *ACM Transactions on Computer Systems*, Vol. 1, No. 2, pp. 144–156, May 1983.

[11] G. Bracha and S. Toueg, A Distributed Algorithm for Generalized Deadlock Detection, *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, pp. 285–301, 1984.

[12] I. Cidon, J. Jaffe, and M. Sidi, Local Distributed Deadlock Detection with Finite Buffers, IBM Technical Report 88.154, April 1985.

[13] I. Cidon, J. Jaffe, and M. Sidi, Local Distributed Deadlock Detection by Cycle Detection and Clustering, *IEEE Transactions on Software Engineering*, Vol. 13, pp. 3–14, January 1987.

[14] I. Cidon, J. Jaffe, and M. Sidi, Distributed Store-and-Forward Deadlock Detection and Resolution Algorithms, *IEEE Transactions on Communications*, Vol. 35, No. 11, pp. 1139–1145, November 1987.

[15] N. Natarajan, A Distributed Scheme for Detecting Communication Deadlocks, *IEEE Transactions on Software Engineering*, Vol. 12, pp. 531–537, April 1986.

[16] C. Chan and T. Yum, An Algorithm for Detecting and Resolving Store-and-Forward Deadlocks in Packet-Switched Networks, *Proceedings of the IEEE International Conference on Communications*, Toronto, pp. 114–118, June 1986.

[17] J. Blazewick, J. Brzezinski, and G. Gambosi, Time-Stamp Approach to Store-and-Forward Deadlock Prevention, *IEEE Transactions on Communications*, Vol. 35, No. 5, pp. 490–495, May 1983.

[18] P. A. Humblet, A Distributed Algorithm for Minimum Weight Directed Spanning Trees, *IEEE Transactions on Communications*, Vol. 31, No. 6, pp. 756–762, June 1983.

[19] E. Chang and R. Roberts, An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes, *Communications of the ACM*, pp. 281–283, May 1979.

[20] G. L. Peterson, An $O(n \log n)$ Unidirectional Algorithm for the Circular Extrema Problem, *ACM Transactions on Programming Languages and Systems*, pp. 758–762, October 1982.

[21] D. P. Mitchell and M. Merrit, A Distributed Algorithm for Deadlock Detection and Resolution, *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, pp. 282–284, 1984.