Efficient Deadlock-Free Routing

Baruch Awerbuch * Shay Kutten † David Peleg ‡

Abstract

This paper deals with store-and-forward deadlocks in communication networks. The goal is to design deadlock-free routing schemes with small overhead in communication and space. Our main contribution is designing efficient protocols that are superior to existing ones in terms of their performance.

*Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM. Part of the work was done while visiting IBM T.J. Watson Research Center.

†IBM T.J. Watson Research Center, Yorktown Heights, NY 10598.

‡Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot 76100, Israel. E-mail: peleg@wisdom.bitnet. Supported in part by an Allon Fellowship, by a Bantrell Fellowship and by a Walter and Elise Haas Career Development Award. Part of the work was done while visiting IBM T.J. Watson Research Center.

1 Introduction

The store-and-forward deadlock is one of the major concerns in the design of routing protocols for communication networks. Informally, a store-and-forward deadlock occurs when messages are "stuck" at some set of nodes, since all the buffers of these nodes are full, and each of the messages in these buffers needs to be forwarded to some other node in the set. Thus in order to move any of the messages, some buffer must be emptied, and for this some other message must be moved, and so on. In day-to-day life, this is analogous to car "gridlocks" that often occur on a busy intersection. Avoidance or fast resolution of such store-and-forward deadlocks is essential for efficient utilization of available network resources.

This problem has been extensively studied in the literature. Roughly speaking, the proposed solutions can be classified into two main categories. The first involves solutions attempting to avoid the occurrence of deadlocks. This is done, for instance, by dividing the buffer pool into buffer classes and utilizing these classes so as to prevent deadlocks [Gop84, MS80, TU81]. Some of these solutions are based on restricting the family of allowed routes in order to avoid deadlocks (say, by forwarding packets according to an acyclic buffer graph).

The second approach to handling store-and-forward deadlocks, usually referred to as "deadlock detection / resolution", is based on the philosophy that it may be better to "ignore" the problem as long as it doesn't harm us (and thus let the network run into deadlocks from time to time), and rely on efficient mechanisms for detecting deadlocks and resolving them (for instance, by dropping some of the packets involved in the deadlock). Numerous solutions based on this strategy were proposed in the literature; a partial list is [AM86, BT84, CM82, CJS87, Gop84, Gö81, JS89], [MM79, MM84, Obe82].

We feel that those distinctions are somewhat artificial. The ultimate goal is always the same, namely to route the messages to their destinations, without wasting too much of the network resources. So the only relevant measures for comparison of the routing protocols are their complexities. The standard complexity measures for network protocols are communication, time, and space; let us define these measures in our context.

We make the standard assumptions of an asynchronous point-to-point communication network, cf. [GHS83]. The *time delay* of a packet is the total time it takes since the packet is entered into the subnet until it is delivered at the destination. For computing the time delay only we make the (common) normalization assumption that the maximum time it take for a packet to traverse a link is one unit of time. The *inherent communication requirement* of a packet is the distance from its source to its destination, i.e., the number of hops it needs to perform until it arrives its destination. The *actual communication* of a protocol is the actual number of messages sent during its execution. The *communication overhead* (or *amortized communication cost*) of a routing protocol is its worst case ratio of actual to inherent communication, where the maximum is taken over all possible sequences of message arrival histories. Space is measured by the number of buffers used in each node. We omit a more formal definition from the abstract.

In this paper we restrict ourselves to what we may call *non-obtrusive* solutions. These are algorithms obeying the following two requirements. First, they do not interfere with the routing process, and are limited to scheduling decisions. (We use the common assumptions that either the routes are carried in the packet headers, or the next node on the route can always be computed locally.) The second requirement excludes the possibility of dropping packets. This possibility is viewed as a rather serious drawback. Indeed, packets that have been dropped are eventually returned back into the system, possibly creating further future deadlocks, and thus making it difficult to assess the amount of progress ultimately made in the network.

In this paper, we focus on the trade-off between communication and space complexities of deadlock-free routing algorithms. We are not aware of any previous work explicitly attempting to address this issue. For comparison, let us consider two typical schemes among the ones listed above, namely, those of [JS89, MS80]. These two schemes obey the non-obtrusiveness requirements postulated earlier.

Let us first discuss the communication-efficient schemes. Several of the solutions described earlier require $\Omega(n)$ buffers per node in an n-vertex network (e.g. [Gop84, MS80]). This is usually unreasonable, and will become more unreasonable in the future, as network sizes are expected to outgrow even the most advanced buffer technology. (Currently, networks use typically less than 50 buffers per node.) Moreover, even though memory in general becomes less expensive, buffers at intermediate nodes in fast networks are not a part of the (relatively slow) general purpose

computer with its large memory. Instead, they are a part of a "slim" and fast switch. Putting a lot of the fastest memory on this switch (and in such a way that it remains fast and cheap) is not easy. Indeed, other papers (such as [CJS87, JS89]) pointed out this problem, and proposed solutions using only a small number of buffers per node.

Although both schemes seem very reasonable time-wise, and may behave well in most common situations, we note the surprising fact that it is possible to devise simple examples demonstrating that both schemes require time that is exponential in n in the worst case. It is interesting to note that the number of buffers is not at all the bottleneck leading to exponential time in these examples. Intuitively, the cause is the fact that the next packet to be forwarded is selected on a solely local basis. Any additional number of buffers will not prevents these bad scenarios, nor are we aware of an obvious way to circumvent this problem.

An example of a communication-efficient scheme is the algorithm of [MS80] which has only $O(1)$ communication overhead, i.e., is communication-optimal. However, this algorithm uses $O(n)$ buffers, and thus is memory-inefficient. This scheme can be contrasted with the memory-efficient algorithm ($O(1)$ buffers) of [JS89], which has communication overhead $\Omega(n)$ in the worst case.

This immediately suggests the existence of some tradeoff between communication and space complexities. Is this tradeoff real? Our results clearly indicate that even if it is, its manifestation is not in the polynomial range.

In this paper we present an algorithm whose communication overhead is $O(1)$ but it requires $O(\log n)$ buffers per node. We comment that a modification of this algorithm (omitted from this abstract) results in reducing the space overhead to $O(1)$ buffers per node, at the expense of increasing the communication overhead to $O(\log n)$. (See Table 1.) Thus our results exhibit the same kind of tradeoff between the communication overhead and the number of buffers needed as that of [MS80] vs. [JS89], although with considerably lower complexities.

Author	Commun.	Space
[MS80]	$O(1)$	$O(n)$
[JS89]	$O(n)$	$O(1)$
This paper	$O(1)$	$O(\log n)$
This paper	$O(\log n)$	$O(1)$

Figure 1: Comparison of our algorithms with existing ones.

2 The hierarchical algorithm

2.1 Overview of the Algorithm

First let us discuss flow control. Notice that if packets are sent only to, say, d destinations, who may each have only one incoming edge, then the network can at best deliver d packets per time unit. If the rate at which new packets are introduced into the network is higher than the network delivery rate, than the waiting time will grow to infinity, and so will the requirements for buffers. This is solved in most communication networks by flow control. Specifically, we shall let every node have a window of size 1 on the average. That is, in our algorithm, a packet that is delivered to its destination permits some node (not necessarily its source or destination) to introduce a new packet. The mechanism for guaranteeing this behavior will be discussed later. In fact, this restriction can be relaxed, e.g., every node v may have a window of size one (or of polynomial size) on the average with respect to each possible destination of

v's packets.

Let us now proceed with a description of the entire forwarding process, beginning with a high-level overview.

Each node is internally organized as a hierarchy of δ levels, for $\delta = \log n + 1$. In particular, each node has 3δ buffers, partitioned into δ levels, with level i owning two regular buffers A_i and B_i and a special *elevator* buffer E_i, for $1 \leq i \leq \delta$.

The packet is packed in a token T (in level 1), whose task is to carry the packet to its destination and terminate. A token in level i proceeds using only buffers of level i. Let us first give the simplified idea of the action taken when the token cannot proceed (because of the lack of an empty buffer in the next node). The way the idea is implemented (to be described later) comprises the main the technical difficulty of this paper.

Intuitively, a token of level i "represents" 2^i basic tokens (of level 1). Hence the higher the level, the less congested the buffers are. Thus when two tokens of level i find themselves "stuck" at two buffers A_i, B_i of the same vertex, say w, they perform a conceptual "merge" operation in which one of them is promoted to proceed in buffers of level $i+1$. In order to preserve the invariant, the other token remains locked in the rendezvous vertex w, (in fact, both buffers A_i and B_i remain locked, as part of the flow control mechanism), until it receives an appropriate "permit" to proceed. Such a permit can be thought of as informing the locked token that its partner has arrived its destination and exited the network. The permit thus allows the locked token to become active again and proceed in level $i+1$ as well. However, the two buffers A_i and B_i of vertex w remain locked, until another permit arrives (intuitively signifying the arrival of the second token to its destination) and releases them.

This simplistic description creates an algorithm that is hard to implement. More specifically, the task of freeing the locked tokens is tricky. It would seem that a permit message (or process) sent to free them, must itself use buffers. An attempt to solve this using the same "go up one level" strategy may result in increasing the communication complexity. Intuitively, the reason is that if a "freeing message" locks another "freeing message" and goes up a level, it is later required to send yet another freeing message that will free the locked freeing message, and so forth. It turns out that such a strategy implies a communication stretch of $\log n$.

In order to overcome this problem we relax the releasing paradigm described above. The idea is to view a permit as a "general purpose currency", or *check*, rather than a dedicated process. I.e., we do not require a level i permit generated by some token T to free the particular token T' with whom T was merged. Rather, the permit can be used to free any other token that happens to be locked at level i on the permit's way.

This approach enables us to treat permits "unanimously", and hence makes it possible to manipulate, keep track of and transfer permits "bunched" together, simply keeping account of the number of permits at each level. This in turn enables us to drastically limit the amount of information carried by a $CHECK$. For example, there is no need for a $CHECK$ to carry the name of the token it is going to release, or the node in which it resides, or a route to that node. On the other hand, it should be clear that this relaxation has to be carefully devised, in order to prevent starvation of locked tokens.

Technically, the $CHECK$ approach implies the following modifications. Instead of releasing separate tokens, we have counters (called `Debt`) on every edge e, specifying, intuitively, "how much freeing work remains to be done in the direction of e". Similarly we

have counters ($CREDIT$'s) specifying "how much freeing work" we are now permitted to do. Suppose now that a $CHECK$ wishes to enter a node, while another $CHECK$ has not left the node yet. The new $CHECK$ does not require another buffer. Instead, it enters (using the single temporary communication buffer in the node) and changes the value of a $CREDIT$ counter. The node knows how many $CHECK$ processes reside in it by the values of the $CREDIT$ counters. These processes always enter a node while they are in the same state. Thus no buffers are needed to store the information carried by these processes. (The process itself, of course, does not require storage, as long as its state information is stored implicitly, as described above.)

The algorithm uses the values of the `Debt` counters in order to route the $CHECK$ processes. In Section 3 we analyze the process and prove that all tokens are indeed found and released.

2.2 Detailed description

The flow control mechanism has to tell v when it may introduce another packet into the network. This will be done by setting a flag called `Get_Msg`. We assume that the internal node process responsible for introducing new packets may be invoked only when `Get_Msg`=*Ready*. The first action taken after allowing a new packet into the network is to set `Get_Msg` to *Waiting*. The resetting of `Get_Msg` to *Ready* will be done by a $CHECK$ process.

Each new packet is inserted to buffer E_1 of the origin vertex. The packet is wrapped in a frame T called a *token*. This token is individually carried by a process $TOKEN(T)$. Token T has the format

$$T = (\texttt{Packet}, \rho, \texttt{Level}, \texttt{Status}, \texttt{Temp}),$$

where ρ is the route description, `Level` is an indicator of the level of the packet, `Status` is a status indicator, which may assume one of four possible values, to be described later on, and `Temp` is a temporary variable. We define $\texttt{Dest}(\rho)$ as the destination of the packet, and $Next(\rho, v)$ executed in a vertex v (which occurs in ρ) to be the vertex appearing just after v in ρ.

Buffers may be in one of the following modes: *Empty*, *Active*, *Stuck* and *Locked*. Tokens may be in one of the following modes: *Active*, *Stuck* and *Locked*. When a buffer contains no token, its status is either *Empty* (meaning, available for accepting an arriving token) or *Locked* (meaning, empty but not ready to accept a new token). When a buffer contains a token, its status is identical to that of the token.

In addition to the buffers, each node v maintains also a set $\texttt{InQueue}(i)$ on every level i, and a *debt* counter $\texttt{Debt}_v(u, i)$ on every level i for every neighbor u.

Let us now proceed with a more detailed description of the process of forwarding a token. The basic mode of operation for a token is the *Active* mode. A level-i token enters this level by moving from some level $i-1$ buffer into the "elevator" buffer E_i. While in *Active* mode, a token travels in the buffers A_i, B_i.

A token T travels as follows. Suppose that T is currently in buffer A_i of a vertex v and it needs to proceed to a neighboring vertex $w = Next(\rho, v)$. Then T tests whether w is ready to accept it. Two conditions need to be satisfied for that to happen. First, w has to have an *Empty* buffer A_i or B_i at level i. Secondly, w manages a round-robin ordering on its incoming edges, and it has to be the turn of the incoming edge from v. (For clarity the code uses a simple implementation of the above. The communication can be improved by a constant factor.)

If both tests are answered positively, then

T traverses the edge and enters the appropriate buffer at w. The buffer A_i in v emptied by T now switches to state *Empty*. This buffer may now be used in order to accommodate another token waiting to enter v on level i. The accepting buffer at w switches its mode to *Active*. Further, the debt counter $\texttt{Debt}_w(v,i)$ at w is increased by 1.

Now suppose that there is no *Empty* buffer in w (or it is not the turn of the edge from v to enter w at that level). Then T acts as follows. It first adds itself to the queue $\texttt{InQueue}(i)$ at w. This is the queue of neighbors with tokens that are currently awaiting entrance to w on level i. T then checks to see whether there is another token waiting in another buffer at level i of v. If not, then T has to change its status to *Stuck* and wait until it is released by some other process.

The releasing task is carried out "automatically". I.e., there is a background "demon" process at every vertex, that is activated whenever some buffer becomes *Empty*, examines the appropriate InQueue queue and initiates the release of the *Stuck* token that is at the top of the queue waiting for this buffer (if there is such a token). The code for this trivial release process is omitted from the abstract.

In case there is a second token T' waiting in another level-i buffer of v, then these two tokens perform a "merge" operation as follows. The two buffers switch into *Locked* status, with one of the tokens, say T', remaining in its buffer and switching into $\texttt{Status}(T') =$ *Locked* as well. The other token, T, increases its level to $\texttt{Level}(T) \leftarrow i+1$. Consequently, it moves into the elevator buffer on that level, E_{i+1}, and proceeds in level $i+1$ buffers. For the purpose of unified description and proof, we introduce an "imaginary" internal edge at node v, connecting its i'th and $i+1$'st levels, and associated with a variable of internal debt $\texttt{Int\_Debt}_v(i)$. This variable $\texttt{Int\_Debt}_v(i)$ is now incremented by one.

Next let us consider the time when a token T has reached its destination. It first delivers the packet Packet to the vertex processor, and then creates a $CHECK$ process on level $\texttt{Level}(T)$, and terminates. The task of the check process is essentially to release the tokens that T has locked on its way. (It may, however, be diverted and end up releasing some other *Locked* token it finds on its way back at the same level.)

A $CHECK$ process C on level $i > 1$ at vertex v proceeds as follows. It considers every edge (v,u) such that $\texttt{Debt}_v(u,i) > 0$, and also the internal edge we defined, if $\texttt{Int\_Debt}_v(i)$ is larger than zero. Process C chooses the next such edge by a round robin policy among the edges of v, i.e., the edge served now goes to the end of the line to await its next turn. Such a queue is kept for every level separately (that is, the next edges to be served on two different levels may be different).

If the chosen edge is not the internal one (but some edge (v,u)), then the $CHECK$ process C reduces the $\texttt{Debt}_v(u,i)$ counter of the edge by 1 and traverses this edge to u. Now consider the case that the internal edge is chosen. This means that a debt of level i was generated locally at this vertex. This could have happened in one of two ways. If $\texttt{Level} = 1$, then the internal edge is selected only if $\texttt{Get\_Msg} =$ *Waiting*. This represents a packet that has been introduced into the network at this node. Otherwise, the debt was generated through a merge operation between two token processes in level $i-1$.

Let us first consider the second case. In this case, the buffers in level $i-1$ are necessarily *Locked*. There are now two sub-cases to consider. If there is a token T' in one of these buffers, then the process C releases this token (i.e., sets $\texttt{Status}(T') =$ *Active*), supplies it with $\texttt{Level} = i$, and terminates. Note that the process leaves both buffers *Locked*,

although both are now empty. Otherwise, both level $i-1$ tokens merged at this vertex have already been delivered. In this case the process C releases both buffers at level $i-1$ (i.e., sets $\texttt{Status}(A_{i-1}) = \texttt{Status}(B_{i-1}) = \textit{Empty}$), spawns two new $CHECK$ processes C_1 and C_2, both on level $i-1$, and terminates.

Finally, in the case that $\texttt{Level} = 1$ and the internal edge was the one selected, the $CHECK$ sets `Get_Msg` to *Ready*, allowing a new packet to be entered into the network, and terminates.

The processes $TOKEN(T)$ and $CHECK(C)$ are described Figures 2 and 3. The code for these algorithms is described using processes (or "tokens"), that can migrate over an edge, carrying their variables along with them [KKM85]. Such a process is executed at a processor until it either migrates, or executes a Wait instruction. This description has an efficient translation to any standard description format.

3 Analysis

Let us define some basic concepts regarding the status of the system. It is convenient to view each vertex v of the graph as composed of $\delta+1$ distinct *layer vertices* $v^{[0]}$, $v^{[0]}$, $\ldots, v^{[\delta]}$, one for each level. Vertex $v^{[i]}$, for $i > 0$, consists of the buffers A_i, B_i and E_i and the queue $\texttt{InQueue}(i)$. (Vertex $v^{[0]}$ represents the external entity in v that provides v with the packets it must send.) The edges adjacent to v are duplicated for each layer $v^{[i]}$. We think of the edges as directed (i.e., each edge connecting the vertices u and v is represented by directed edges $(u^{[i]}, v^{[i]})$ and $(v^{[i]}, u^{[i]})$ for $1 \leq i \leq \delta$. There is also a directed debt edge leading from $v^{[i]}$ to $v^{[i-1]}$ for every i. Moreover, in order to represent the `Debt` configuration at any given moment, we consider the directed multigraph obtained by taking each edge $(u^{[i]}, v^{[i]})$ with multiplicity $\texttt{Debt}_u(v,i)$. I.e., at any given time there are $\texttt{Debt}_u(v,i)$ parallel edges going from the i'th layer of u to the i'th layer of v. (Similarly, the number of internal edges from $u^layeri+1$ to $u^{[i]}$ is the value of $\texttt{Int\_Debt}(i)$.) We term these edges *debt edges*. Also, if $\texttt{Get\_Msg} = \textit{Waiting}$ then there is an internal debt edge from $u^{[1]}$ to $u^{[0]}$. (This represents the event in which a packet was introduced into the network in vertex u). We refer to the resulting directed graph as the *layered graph* $\tilde{G}$.

A *trace tree* is a directed tree in the layered graph $\tilde{G}$ with the following properties. A leaf is a vertex who sent a packet and its `Get_Msg` is still *Waiting*. The edges of the tree point downwards towards the leaves. The out-degree of each vertex in the tree is one, and its it-degree is 1 or 2. A leaf or a vertex with in-degree 2 is called a *merge* vertex. The vertices of the tree are of the following kinds. The leaves are all of layer 0. A non-merge vertex is of the same level as its parent (i.e., they are $v^{[i]}$ and $u^{[i]}$ for some i). For a merge vertex of level i, its parent is of level $i+1$. For each non-leaf merge vertex two of its buffers are in status *Locked* and one of them may contain a token in status *Locked*. The root of the tree is a node $v^{[j]}$ containing a *debtor* of the appropriate level. By this we mean either a token in status *Active* or *Stuck*, or a $CHECK$ process, of level j.

We say that a certain trace tree *covers* the debt of a token T in status *Locked* currently residing in a buffer of the layer vertex $v^{[i]}$ if the vertex $v^{[i]}$ occurs in the tree as a merge vertex. Similarly, a certain debt tree *charges* the debtor D (which, again, may be a token in status *Active* or *Stuck* or a check) currently residing in a layer vertex $v^{[j]}$ if this vertex occurs as the root of the tree.

A *trace cover* is a collection of trace trees in the graph $\tilde{G}$ with the following properties.

```
Process TOKEN(ρ, Packet),                                   /* deliver Packet on path ρ */
Level ← 1,
Temp ← self
Wait until Get_Msg = Ready
Get_Msg ← Waiting                                           /* initially Ready */
E_1 ← (ρ, Packet, Active)
While Temp ≠ Dest(ρ) do :
   if Status ≠ Active then wait until Status = Active
   Traverse the edge from Temp to z = Next(ρ, Temp).
   If ∃X, X ∈ {A_Level, B_Level}, Status(X) = Empty, InQueue(Level) = Empty then do :
      X ← T
      Debt_z(Temp, Level) ← Debt_z(Temp, Level) + 1
      Traverse back to Temp.
      Buffer ← (∅, ∅, Empty)                                /* the packet has been forwarded */
      Traverse to z = Next(ρ, Temp)                         /* ready to continue moving forward */
   Else do :                                                /* no available buffer */
      Put {(Temp, Buffer)} in InQueue(Level)
      Traverse back to Temp.
      If ∃Y, Y ∈ {A_Level, B_Level, E_Level}, Y ≠ Buffer, Status(Y) = Stuck then do :
         E_Level+1 ← Buffer
         Status(Buffer) ← Locked
         Status(Y) ← Locked
         Level ← Level + 1
         Set Int_Debt_self(Level) ← Int_Debt_self(Level) + 1
      Else Status(Buffer) ← Stuck.
End_while                                                   /* reached destination */
Deliver Packet to the local processor.
Create a check process C = ⟨Level⟩.
Set Status(Buffer(T)) ← Empty.
```

Figure 2: Process $TOKEN$

```
Process CHECK(Level):
/* process has form ⟨Level⟩ */
While Int_Debt_self(Level) > 0
Or there exists an edge (self, u) such that Debt_self(u, Level) > 0 do :
   If the next edge in the round robin order of Level is not the internal edge then
      Debt_self(u, Level) ← Debt_self(u, Level) − 1
      Traverse to u.
   Else                                                    /* internal edge is next */
      If ∃X ∈ {A_Level−1, B_Level−1, E_Level−1}, Packet(X) ≠ ∅, Status(X) = Locked then do :
         E_Level ← X                                      /* move to elevator buffer */
         X ← (∅, ∅, Locked)                               /* empty and keep locked */
         Status(E_Level) ← Active                         /* Release token */
      Else                          /* two Locked empty buffers at level Level − 1 */
         If Level > 1 then do :
            Let X, Y be the buffers in Level-1 in Status Locked
            Status(X), Status(Y) ← Empty.
            Create two check processes C1 = ⟨Level − 1⟩ and C2 = ⟨Level − 1⟩.
            Set Int_Debt_self(Level) ← Int_Debt_self(Level) − 1
         Else                                             /* Level = 1*/
            If Get_Msg = Waiting then do
               Get_Msg ← Ready
               Terminate
End_while
```

Figure 3: Process $CHECK$.

1. Every token T with status *Locked* has exactly one trace tree covering its debt.

2. Every debtor D has exactly one trace tree charging it.

3. Every debt edge is included in at most one trace tree. (Recall that there may be parallel debt edges between the same vertices.)

Intuitively, a trace tree in which a buffer is locked keeps trace of the token that locked the buffer. However, since only the *number* of tokens and checks passing at each level is recorded (in the `Debt` variables), a check generated by a token on one tree may be used to release another tree. Still, we can prove the following invariant. (All proofs are omitted from this extended abstract.)

Lemma 3.1 At any time during the execution of the algorithm, there exists a trace cover for the graph $\tilde{G}$. ∎

The trace trees already enable us to prove that the algorithm is well-defined. This is stated in Lemmas 3.2 and 3.4.

Lemma 3.2 For every token or check of level i there are 2^i vertices with `Get_Msg` = *Waiting*. ∎

Corollary 3.3 At no time does the algorithm require a buffer of `Level` larger than δ.

Lemma 3.4 . At any time, if two tokens wish to perform a merge operation at level i, then the elevator buffer E_{i+1} is *Empty*. ∎

A trace tree traces the route from a *Locked* buffer to either a token or a check. However, the token may be *Stuck*. We now extend the directed multigraph defined above to help us trace the way from these *Stuck* tokens to a non-*Stuck* token or to a check, or to a *Stuck* token that will eventually be released. (Note that this is only an auxiliary definition; the algorithm does not "know" which token will eventually be released.) This will enable us to show later that every *Stuck* token and every *Locked* buffer are eventually released. For that purpose let us add to the layered graph a directed *Stuck edge* from u to v at level i if there is a token *Stuck* at v in level i trying to get into u.

To prove the above fairness property, it suffices to show that every directed edge in the layered graph, leading to (a vertex in the layered graph with) a *Locked* or *Stuck* buffer is eventually deleted. Note that an edge that from some time on is never deleted, undergoes only a finite number of deletions and insertions throughout the execution. Consider all the edges that are inserted and deleted only a finite number of times. (The last event of such an edge may be a deletion). Let us call such edges *eventually stable edges*. Out of these edges, let the *permanent* edges be those which from some time on are never deleted.

Let T be the time after the last event (either deletion or insertion) happening to any eventually stable edge. Let a *return* route at time T be a directed path in the layered graph from a non-*Stuck* debtor to (a vertex in the layered graph with) a *Locked* or *Stuck* buffer, such that a permanent *Stuck* edge can be used only if the previous edge is one from some $u^{[i]}$ to $u^{[i-1]}$. (Again, we shall not need to know which edge is permanent.)

Lemma 3.5 At any time, there is a return route (on the layered graph) to every packet from a debtor who is not *Stuck*, or from a *Stuck* edge which is not permanent. ∎

Lemma 3.5 is used in the proof of the following desired result:

Lemma 3.6 Every edge leading to a vertex (in the layered graph) with a *Locked* or *Stuck* buffer is eventually deleted. ∎

To conclude the correctness part, recall that a vertex that introduced a packet into the network is prevented from introducing

another until its Get_Msg equals *Ready*.

Lemma 3.7 Eventually, the value of every Get_Msg variable becomes *Ready*. ∎

Let us now analyze the performance of the algorithm. It is easy to see that the memory requirement is logarithmic in n. As for the communication overhead, we can show

Lemma 3.8 The communication overhead of the hierarchical algorithm is $O(1)$. ∎

The algorithm as described so far uses only the FIFO discipline locally in order to decide which packet to advance next. Thus the time in the worst case is the same as in [JS89, MS80], namely, exponential. However, with the introduction of additional (more global) scheduling decisions we can prove the following:

Theorem 3.9 The algorithm can be extended so that with the same communication overhead and number of buffers, every packet is delivered within time $O(n^2 \log n)$ of the time it entered the network. ∎

Acknowledgments

We are grateful to Israel Cidon and Moshe Sidi for working with us in the early stages of this research and for many stimulating and helpful discussions.

References

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 358–370. IEEE, October 1987.

[AM86] B. Awerbuch and S. Micali. Dynamic deadlock resolution protocols. In *Proc. 27th IEEE Symp. on Foundations of Computer Science*. IEEE, November 1986.

[BT84] G. Bracha and S. Toueg. A distributed algorithm for generalized deadlock detection. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, pages 285–301. ACM, August 1984.

[CJS87] I. Cidon, J. Jaffe, and M. Sidi. Distributed store-and forward deadlock detection and resolution algorithms. *IEEE Trans. on Commun.*, COM-35:1139–1145, May 1987.

[CM82] K.M. Chandi and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proc. 1st ACM Symp. on Principles of Distributed Computing*, pages 157–164. ACM, August 1982.

[Gün81] K.D. Günther. Prevention of deadlocks in packet-switched data transport systems. *IEEE Trans. on Commun.*, COM-29:512–524, May 1981.

[GHS83] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum weight spanning trees. *ACM Trans. on Programming Lang. and Syst.*, 5:66–77, 1983.

[Gop84] I.S. Gopal. Prevention of store-and-forward deadlock in computer networks. Research Report RC-10677, IBM Yorktown, August 1984.

[JS89] J.M. Jaffe and M. Sidi. Distributed deadlock resolution in store-and forward networks. *Algorithmica*, 4, 1989.

[KKM85] E. Korach, S. Kutten, and S. Moran. A modular technique for the design of efficient distributed leader finding algorithms. In *Proc. 4th ACM Symp. on Principles of Distributed Computing*. ACM, August 1985.

[Mar82] James Martin, *SNA: IBM's Networking Solution*, Prantice Hall, Englewood Cliffs, NJ, 1982

[MM79] D.A. Menascoe and R. Muntz. Locking and deadlock detection in distributed databases. *IEEE Trans. on Software Eng.*, SE-5:195–202, 1979.

[MM84] D.P. Mitchell and M. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proc. 3rd ACM Symp. on Principles of Distributed Computing*, pages 282–284. ACM, August 1984.

[MS80] P.M. Merlin and P.J. Schweitzer. Deadlock avoidance in store-and-forward networks i: Store and forward deadlock. *IEEE Trans. on Commun.*, COM-28:345–352, March 1980.

[Obe82] R. Obermarck. Distributed deadlock detection algorithm. *ACM Trans. on Database Syst.*, 7:187–208, 1982.

[TU81] S. Toueg and J.D. Ullman. Deadlock-free packet switching networks. *SIAM J. on Comput.*, 10:594–611, 1981.