Distributed Protocols for Access Arbitration in Tree-Structured Communication Channels

Riccardo Melen and Jonathan S. Turner, *Fellow, IEEE*

***Abstract*—We consider the problem of arbitrating access to a tree structured communication channel with large geographic extent, providing multipoint communication among a set of terminals. In our model, terminals transmit information in bursts consisting of many packets and compete for the right to transmit bursts. In the simplest case, the channel allows only one terminal to transmit at a time; this can be extended to *k* concurrent transmitters. The problem resembles contention resolution in local area networks. It is distinguished by the topology of the channel, the magnitude of the delays involved and the potential for multiple transmitters. In this paper, we identify two general approaches and several specific assess arbitration algorithms and make a preliminary assessment of their promise.**

I. Introduction

THE problem considered in this paper is motivated by recent research on the design of wide-area packet communication networks supporting high speed multipoint communication [6]. Such multipoint networks are constructed from switching systems connected in a mesh topology, and having the ability to replicate packets and forward them over pre-established multipoint virtual circuits, as illustrated in Fig. 1. The switching systems used in such networks may lose packets through buffer overflows or transmission errors which alter the contents of packets. In the simplest case each multipoint channel has a single transmitter and many receivers, and the channel distributes the signal to the multiple receivers. In this paper we consider the case where every terminal in the channel can both transmit and receive on the channel. Each terminal transmits data in the form of packets, which are replicated by the channel and delivered to all the other terminals. In typical applications, such as teleconferencing or LAN interconnection, information is transmitted in bursts comprising many packets and while every endpoint is a potential transmitter, typically only a few transmit at one time. Note that each channel is tree-structured and can be considered in isolation from the remainder of the network. At the same time however, because the transmission links may be shared by several different connections, it is important to ensure that each connection does not use more than its share of the link bandwidth.

Given that the network must provide bandwidth to support

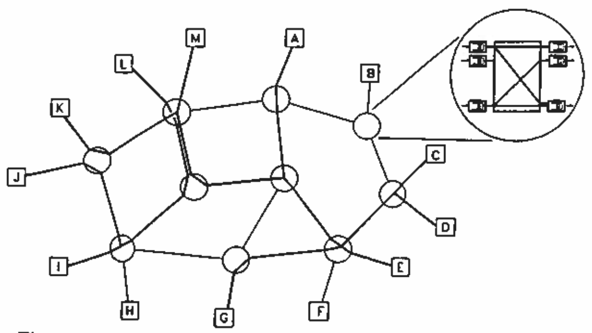

Fig. 1. Packet network supporting multipoint connections.

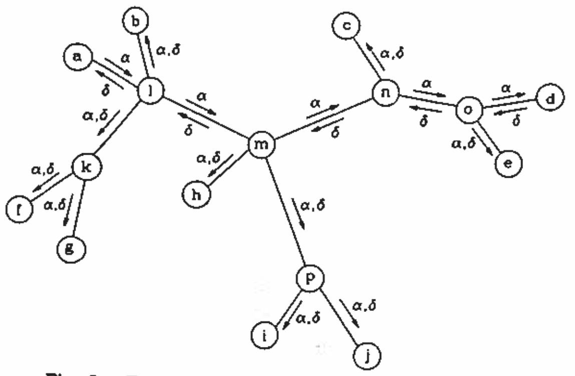

Fig. 2. Example multipoint communication channel.

each channel, there is the question of how much bandwidth to allocate. A worst case allocation provides bandwidth for all n sources in a multipoint channel to transmit simultaneously. While this may be appropriate in some applications it is unacceptably inefficient if only one or two transmitters are active at one time. On the other hand, if the network allocates resources for only a few active transmitters, it must provide mechanisms to ensure that only a few transmitters can be active at one time.

To make the discussion more concrete, consider the example channel shown in Fig. 2. This channel serves ten terminals interconnected by several intermediate nodes. Two terminals are transmitting bursts labelled α and δ. Note how the burst are propagated by the intermediate nodes. Also note that each link in the channel carries both bursts, but in some cases they propagate in the same direction, while in other cases they propagate in opposite directions. To provide sufficient bandwidth for any pair of terminals to transmit simultaneously, the network must allocate bandwidth for two bursts to be carried in both directions on each internal link. Assuming resources are allocated for just two transmitters, the network must provide mechanisms to ensure that no more than two transmit at a time. Each link has an

Paper approved by the Editor for Network Protocols of the IEEE Communications Society. Manuscript received July 18, 1988; revised April 19, 1989. This work was supported by the National Science Foundation under Grant DCI 8600947, Bell-Northern Research, Italtel SIT, and NEC. This paper was presented in part at the International Communications Conference, June 1988.

R. Melen is with Centro Studi e Laboratori Telecomunicazioni (CSELT), 10148 Torino, Italy.

J. S. Turner is with the Department of Computer Science, Washington University, St. Louis, MO 63130.

IEEE Log Number 9041714.

associated delay. As we are interested in networks with geographically distributed nodes, delays of tens of milliseconds are typical.

The necessity of access arbitration arises primarily from the network's need to prevent terminals on one channel from usurping resources allocated to other channels that may be sharing the same links. However, access arbitration may also be viewed as a service provided by the network for the terminals, since it regulates the flow of data *into* each terminal, in an orderly fashion.

We note that our problem is similar in spirit to media access protocols for local area networks, and indeed current LAN arbitration schemes have given us several useful ideas. (See [4] for an introduction to popular LAN media access protocols.) What distinguishes our problem is the tree-structured channel in which the internal nodes can play an active role in access arbitration, the relatively long delays involved and the possibility of having multiple transmitters. These factors have a strong influence on the performance and implementation of various solutions, as will be seen in subsequent sections. Previous work of some relevance to our problem can be found in [1], [5]

Formally, we denote a channel C by a pair (T, δ) where $T = (N, L)$ is an undirected tree with node set N and link set L; $\delta : L \rightarrow Z^+$ is a function that assigns a positive integer delay to each link. The nodes of T with only one incident link are called the *terminals* and are collectively denoted by N_t; all other nodes are called internal. We define the *distance* between two nodes u and v to be the sum of the link delays on the path joining u and v and denote it by $\delta(u, v)$. Packets transmitted at one end of a link $\{u, v\}$ are delivered to the other end after a delay $\delta(u, v)$. We define the *diameter* of the channel to be the length of the longest simple path in T joining two terminals and denote it by Δ. Packets delivered to a node are replicated and sent out over all of the other links incident to the node. While in an actual system, this involves some (relatively small) stochastic delay, we will neglect it in this paper and assume that the nodes operate instantaneously. Packets that arrive simultaneously at a node are processed sequentially in some arbitrary order. Constant link delays and zero node delays are adopted to simplify the presentation and are not essential to any of the algorithms described here; in general, the only essential properties are sequentiality for links and in some cases for nodes.

Our primary objective is to identify algorithms which can operate in an environment where packets can be lost and in which link speeds are high enough that hardware implementation of access arbitration algorithm is necessary. Two of the five algorithms we consider (2.2 and 3.1) cannot meet this objective, but they are worth consideration for the insights they give to alternative approaches and their possible application to other environments.

We present two fundamental approaches to access arbitration. The first, described in Section II, is based on the idea of transmit permits or *tokens*; that is a terminal must have explicit permission to transmit before starting a burst. We given two algorithms using this approach; one is an essentially passive algorithm that provides the minimum set of facilities to support token-based access arbitration and the other, an active token circulation algorithm that seeks to reduce token latencies by adding intelligence to the internal nodes. The second approach, described in Section III, allows terminals to transmit whenever the number of bursts they can observe from their vantage point is less than the limiting number; the network then performs arbitration internally, possibly aborting some bursts in the process, to prevent too many burst from being active on a link at one time. We present three algorithms using this second approach. We conclude with a brief assessment of the various methods and suggest some possible topics for future investigation.

II. Access Arbitration Using Transmit Tokens

Perhaps the most obvious approach to access arbitration in a tree-structured channel is to supply the connection with some number k of transmit permits or tokens, and require that a terminal possess at least one token before being allowed to transmit a packet. This approach limits the number of simultaneous transmitters to k, while allowing the set of transmitters to vary over time, through the passing of tokens. The network can allocate bandwidth for k simultaneous transmitters, independent of the total number of terminals in the connection.

While this strategy seems simple enough, finding a practical implementation for a high speed packet network is not as straightforward as it might appear. The reason is that passing of tokens must be completely reliable; since the underlying network may lose packets on occasion, a protocol is required that allows tokens to be passed reliably, while at the same time preventing terminals from creating new tokens. We consider two algorithms in this section which take two different approaches to the problem. The first provides a simple and practical solution that can be implemented using a set of distributed monitor processes at the access links connecting the terminals to the remainder of the channel. In this algorithm, the network plays a passive role, with the terminals handling most of the work associated with token passing, while the network provides minimal support for reliable transmission and prevents token creation by the terminals. The essential simplicity of the algorithm makes a hardware-based implementation practical, allowing its use in high speed networks. In the second algorithm, the network plays a more active role, distributing tokens to users based on request messages; this approach, while more complex can reduce the latency associated with token passing. On the other hand, this second algorithm requires reliable control communication, which in our environment requires an additional protocol to ensure messages are not lost. This seems to preclude efficient hardware implementation, limiting the algorithm to networks with lower performance needs.

A. A Passive Algorithm

In this section, we describe a passive algorithm, which we refer to as Algorithm 2.1, for support of token-based access arbitration. Terminals transmit two types of packets, *data*, and *token* packets. Every token packet has three fields, one containing the *token id*, another containing the *destination terminal*, that is the identity of the terminal that is to receive the token and a third field called the *generation identifier*, which is described below. The internal nodes of the channel, replicate all received packets and propagate them throughout the channel.

The algorithm is implemented by a collection of *monitor processes*, located at the terminals' access links. The monitor processes observe the flow of packets over the channels and are responsible for preventing a terminal from transmitting a packet unless it is in possession of a token. The monitor processes, also prevent creation of new tokens by making sure that terminals pass only those tokens that are in their possession. The monitors provide indirect support for reliable token transmission; if a terminal passes a token to another and determines that the token packet was lost, it is allowed to retransmit the packet. The tricky

```
do termport?p →relay(p ,termport ,nodeport)
 | nodeport?p →relay(p ,nodeport,termport );
od;

procedure relay(packet p , port from, port to)
        if p. typ=data ∧ from=termport ∧ n >0→
                nodeport!p ;
        | p. typ=data ∧ from=nodeport→
                termport !p ;
        | p. typ=token ∧ from=termport →
                i := p. tid;
                if state_i =present→
                        state_i := passing;  dest_i := p. dest;
                        gen_i := gen_i +1;  p. gen := gen_i ;
                        n := n −1;  nodeport!p ;
                 | state_i =passing→
                        p. dest := dest_i ;  p. gen := gen_i ;
                        nodeport!p ;
                fi;
        | p. typ=token ∧ from=nodeport→
                i := p. tid;
                if p. dest=termid ∧ p. gen≥gen_i →
                        if state_i ≠present→ n := n +1; fi;
                        state_i := present;  gen_i := p.gen ;
                        termport !p ;
                | p. dest≠termid ∧ p. gen≥gen_i →
                        state_i := absent;  gen_i := p.gen ;
                fi;
        fi;
end;
```

Fig. 3. Program for monitor process.

part, is allowing such retransmissions without introducing a mechanism that allows the user to create new tokens.

Each monitor maintains several variables for each token allowed in the connection. For token i, the variable $state_i$ = **present** if token i is present at the terminal (meaning the terminal can use it to transmit packets), $state_i$ = **absent** if token i is not present and $state_i$ = **passing** if the terminal is in the process of passing the token to another terminal. More precisely, $state_i$ = **passing** if the terminal has transmitted a token packet for token i and the monitor has not yet received any positive indication that the token has been received. If $state_i$ = **passing**, the variable $dest_i$ is the identity of the terminal that the token was passed to; additional token packets can be sent to that destination, but not others. The variable gen_i is the generation number of token i; the generation number of a token is incremented whenever the token is passed and used to help prevent replication of tokens. In addition, each monitor has a variable n, which gives the number of tokens present at the node, and a variable *termid* that uniquely identifies the terminal that the monitor is associated with.

A program implementing the monitor process is shown in Fig. 3. The program is written using Dijkstra's guarded command notation [2]. Input and output are denoted using a variant on Hoare's notation for CSP [3]. In particular, *portname?x* reads an item from the named port into the variable x if there is any data available and *portname!x* transmits the value of x on the named port. Each monitor has two bidirectional ports, one for communication with its associated terminal (*termport*) and the other for communication to the associated internal node (*nodeport*).

It is tempting to simplify the algorithm by omitting the token generation numbers. Unfortunately, such a change allows the creation of multiple tokens. Consider, for example, if terminal A sent a token packet to terminal B, which in turn sent a token packet to C. If this latter packet is not seen by the monitor at A (because of an error on one of the links between B and A), then A might send a second token packet to B. The algorithm described above uses the token generation number to filter out this second token packet; without it, the token would be passed on to B and we would be left in a situation where B and C possess copies of the same token. Algorithm 2.1 can never enter a state in which a given token is present at more than one terminal.

In practice, the mechanism implementing the monitor process must simultaneously implement monitor processes associated with other channels that are statistically multiplexed on the same link. When a packet is received on a link, *a logical channel number* is extracted from the packet and used to extract information from an internal table that records information about all the channels using that link. This information includes the state of the monitor process controlling each channel. This information is used to make decisions, then the state is changed if necessary and written back to the table.

In Algorithm 2.1, the network plays the smallest possible role, leaving to the terminals, the real work of ensuring that tokens are reliably exchanged. This approach keeps the network simple and provides a great deal of flexibility. A variety of token distribution strategies can be implemented by the terminals; we note here a few possibilities, without going into detail. One simple method is to have a logical ring associated with each token and allow each token to circulate around its ring; the assignment of terminals to rings can be optimized to satisfy performance requirements that may vary among the different terminals. Another method is for one terminal to play the role of token dispenser, with other terminals explicitly requesting tokens when needed. Alternatively, the task of token dispensing could be distributed, so any terminal with an available token might respond to a token request that was broadcast to all.

B. An Active Token Distribution Algorithm

While Algorithm 2.1 keeps the internal network mechanisms fairly simple, it places a lot of the responsibility for token management on the terminals and may give poor performance as a result of the token latencies involved. In this section we sketch an alternative strategy in which the network plays a more active role, explicitly managing the token distribution so as to reduce the amount of time that terminals spend waiting for tokens.

In the new strategy, a terminal with data to send must first request a token, then wait for the network to provide one; once the token has been assigned the transmission can start; at the end of the burst, the terminal issues a token release. Note that terminals do not simply wait for the arrival of a free token circulating through the connection, but play an active role; this choice, together with the tree-shaped topology of the connection permits faster token circulation.

The algorithm is implemented by two types of processes. The first is a monitor process similar to the one in Algorithm 2.1, which observes the passage of token control messages and allows data packets to be sent only when the terminal is in possession of a token; we omit the details of this process. The second type of process implements the actual token circulation; there is one such process for each internal node in the connection. As we will see, these processes are sufficiently complicated that a hardware implementation is probably impractical; consequently, we assume that the token circulation processes are implemented in software. This may limit the token handling capacity of an actual implementation, but we do not consider that issue in detail here. We also assume that the various token control messages are passed between adjacent nodes using a

```
do parent?p →circ(p ,parent)
  | child(i )?p →circ(p ,child(i ));
od;

procedure circ(packet p , port from)
        if p. typ=request ∧ parent≠null →
                parent!p ;  R := R&from;
         | p. typ=request ∧ parent=null ∧ T > 0→
                T := T−1;  from!tokp;
         | p. typ=request ∧ parent=null ∧ T =0→
                R := R&from;
         | p. typ=token ∧ R ≠null →
                R [1]!p ;  R := R [2..];
         | p. typ=token ∧ R =null ∧ parent≠null →
                parent!p ;
         | p. typ=token ∧ R =null ∧ parent=null →
                T := T+1;
        fi;
end;
```

Fig. 4. Program for token circulation process.

reliable communication protocol to prevent token loss. This also adds additional complexity to the algorithm, which further limits its potential for application in high speed networks.

The token circulation processes view the tree induced by the channel as a directed tree. One of the internal nodes is designated the *root* of the channel; all other nodes u in the tree have a unique *parent*, which is the neighboring node that lies on the path from u to the root. The root can be any node, but the best performance is obtained when it is at the center of the tree. Whenever there are no pending requests for tokens, unused tokens propagate up the tree to the root. Token requests also propagate up the tree, but each node maintains a list of token requests from its subtrees and if a token is received from either the parent or a subtree, while a request is pending, that token is used to satisfy the request.

A program implementing a simple version of the token circulation process appears in Fig. 4. The process can receive messages from the *parent*, or from any of several children, denoted *child*(i). The process at the root has its *parent* variable set to **null**. The variable R is a list of children with pending requests; R [1] is the first item on the list and $R[2 \cdots]$ denotes the sublist with the first item removed. The assignment R := *R&from* adds the value of *from* to the end of the list. The variable T, records the number of tokens available at the root. The variable, *tokp* is just a packet with the type field set to **token**.

There are a few aspects of the algorithm that can be improved upon. Suppose a node u has a single pending token request from a child $c(i)$ and has requested a token from its parent. If u receives a token from one of its children, that token will be used to satisfy the pending request. When the token requested from the parent arrives later, it will be returned, assuming no other requests have arrived in the meantime. The time spent by that second token traveling to u and back is essentially wasted; it's possible that overall performance could be improved, if in this situation u sent a *cancellation* packet to its parent. A node receiving such a packet from one of its children would respond by deleting any pending request for that child and sending the cancellation on to its parent. If the node no longer had a pending request for that child (because it had already sent a token in response to the earlier request), it would simply ignore the cancellation.

Note also, that as written, the algorithm permits starvation; that is, it is possible for a node with a pending request to never get served since the token may stay in another subtree. We can avoid starvation by constraining the token circulation somewhat. In particular, whenever a token is received from *child*(i), the token is used to satisfy a request from *child*(j) where j is the smallest integer *greater than* i for which there is a pending request. If there is no such request, the token is sent to the parent. With this change, the waiting time of a pending request is bounded if the time that a terminal holds a token is bounded. We refer to the algorithm incorporating these two refinements as Algorithm 2.2.

III. Contention-Based Access Arbitration

The algorithms of the previous section required that a terminal acquire an explicit transmit permit or token before starting a burst. In this section, we consider access arbitration algorithms in which terminals contend for access to the channel by simply transmitting their bursts at will and allowing the channel to select the bursts to be delivered. Again our objective is algorithms which admit hardware implementation and are robust in the face of occasional packet loss, although we also consider algorithms that fall short of this objective.

We are interested in access arbitration algorithms that can be implemented by a collection of *arbiters*; each link having an arbiter at each of its two ends. Preferably, these should be simple enough to be implemented within a hardware packet processor that handles many channels multiplexed on the common link. An arbiter is a sequential process that monitors the flow of traffic at its position in the channel and either allows packets to pass or discards them. Arbiters may also exchange control packets, but they may not delay user packets. While practical arbiters require some time to operate, we neglect that here and assume that they operate instantaneously.

For the purposes of exposition, we assume that the terminals transmit packets in the form of *bursts* comprising a *start packet*, zero or more *data packets* and an *end packet*. We assume initially that the start and end packets are never lost, but we relax this assumption where possible to obtain robust algorithms. Each packet has a *source* field that identifies the terminal from which it originated. We say that a burst is *contending* if the originating terminal has transmitted the start packet, the start packet has not yet been received by all the other terminals and no arbiter has yet discarded the start packet. We say that a burst is *active* if its start packet has been received by every other terminal and its end packet has not yet been transmitted. The set of active bursts at time t is denoted by $\alpha(t)$ and the set of contending bursts at time t is denoted by $\gamma(t)$. We say that a burst is *active at a node* u, if the start packet of the burst has been transmitted from u and the last packet of the burst to be transmitted from u has not yet left u. We denote the set of active bursts at u at time t by $\beta_u(t)$. We say that a burst is *received* by a terminal u if both the start and end packets of the burst are received.

Fig. 5 illustrates contention-based access arbitration. It shows a simple sequence involving burst from a, d, and b. In the figure, arbiters are shown as circles with a cross and the numbers labeling the links give the link delays. The number of transmitters that are allowed to transmit simultaneously is one. The notation $S_a(t)$ indicates a start packet from a passing a given point in the channel at time t. Similarly, $D_a(t)$ indicates a data packet from a passing a given point and $E_a(t)$, an end packet from a. The sequence starts with a burst sent from a, which starts to propagate through the channel. Shortly after, a burst starts from d and the bursts from a and d cross on the

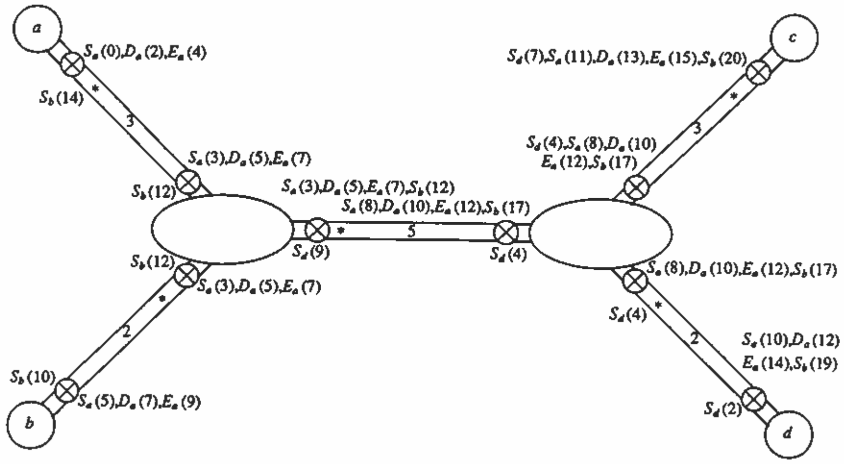

Fig. 5. Example of contention-type access arbitration.

link joining the two internal nodes. Note that the burst from a wins the contention and continues to propagate, while the burst from d is not allowed to propagate beyond the central link. After the burst from a completes, another burst is started from b and propagates through the channel without contention.

We can now list the defining properties for an access arbitration algorithm allowing up to k concurrent bursts.

P_1 If a packet is transmitted by a terminal u at time t, it is delivered to terminal $v \neq u$ by time $t + \delta(u, v)$ or not at all.

P_2 If a terminal v does not receive a packet p because an arbiter discarded it, it will not receive any packets that are part of the burst containing p and are transmitted after p.

P_3 At all times t, $|\alpha(t)| \leq k$ and $|\beta_u(t)| \leq k$ for all nodes u.

P_4 If no terminal transmits any packet after time t, then $|\alpha(t + \Delta)| = \min\{k, |\alpha(t)| + |\gamma(t)|\}$.

P_5 If the start packet of a burst transmitted by u is delivered to every terminal in $N_t - \{u\}$, then the burst is received by every terminal.

We also expect access arbitration algorithms to be fair in the sense that they not favor some terminals at the expense of others.

Properties P_1–P_5 have some useful consequences if all bursts have a duration of at least 2Δ. In this case, if a terminal x starts a burst and during the period $[t, t + 2\Delta]$, there is no time when k burst are arriving, then the transmitted bursts is received by all other terminals. To see this, suppose that some terminal fails to receive the burst and let t_1 be a time at which some arbiter u discards the start packet of the burst (this must occur at some time by P_5). By P_1, $t \leq t_1 \leq t + \Delta$ and by P_4 we must have $\beta_u(t) = k$. Since u is propagating k burst at time t, x must be receiving k bursts at time $t + \delta(u, x)$. Also, if there is some time in the interval $[t, t + 2\Delta]$ interval when k bursts are arriving, the outgoing burst is not received completely by any other terminal. This follows since the arbiter on the access link to x must have reached a decision not to propagate the burst from x and so (by P_2) will not propagate the remaining packets in the burst. Thus, either the burst is completely received by everyone, or it is received by no one; moreover, the transmitting terminal can determine which is the case, allowing the possibility of retransmission at a later time, if appropriate.

A. A Distributed Access Arbitration Algorithm

The first access arbitration algorithm we present allows just a single active transmitter. Extension to multiple transmitters, while possible, is complicated. The key idea underlying the algorithm is that contention between two competing bursts can be resolved at that point in the channel where the two burst meet. This requires the cooperation of the pair of arbiters at opposite ends of the link where the bursts meet, or of the arbiters at the node where they meet. The arbiters that are not at the meeting point can respond in a passive way; they simply allow a later burst to preempt an earlier one that is not yet finished, since the later burst must be the one chosen by the arbiters that were at the meeting point. This idea is illustrated in Fig. 5. The bursts from a and d contend on the central link with d being blocked at that point. When the burst from a arrives at the link incident to c, the arbiters can allow it to preempt the burst from d, because it can then be inferred that a contention has taken place, with the burst from a winning. Most often, the start packets of bursts cross on some link and the arbiters at opposite ends of the link must resolve the contention. The winner resulting from a contention alternates between the two link directions. This requires a simple hand-shake protocol between the arbiters, so that they both properly recognize a contention event and respond consistently. It is also possible for contending start packets to arrive simultaneously at a node. To resolve the contention at this point, we add two additional constraints on the operation of the node. First, we require that start packets sent to a node from an arbiter be sent to all arbiters at the node, including the arbiter that first sent it. This serves as an acknowledgment packet for that arbiter. We also require that the order in which start packets from a node to an arbiter are processed, be the same for all arbiters at the node. In a practical system, this implies that the node arbitrarily serialize start packets that arrive at about the same time and deliver them in the same order to all arbiters.

We now describe the arbiters used by the internal nodes. These can be described as finite state machines with three major states, **stable**, **in_burst** and **out_burst**; **in_burst** is a transitory state, which the arbiter enters upon receiving a start packet from the link. The start packet is sent to the node and when the node returns the packet as an acknowledgment, the arbiter goes to the stable state. Similarly the arbiter enters **out_burst** upon receiving a start packet from the node. The packet is sent to the link

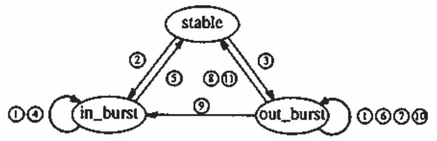

Fig. 6. Transition diagram for internal arbiter.

```
do linkport?p →relay(p ,linkport,nodeport)
 | nodeport?p →relay(p ,nodeport,linkport);
od;

procedure relay(packet p , port from, port to)
(1)     if p. type∈ {data,end} ∧ p. source=current_trans→
               to!p ;
(2)      | state=stable ∧ p. typ=start ∧ from=linkport→
               current_trans := p. source;  nodeport!p ;
               linkport!ackp;  state := in_burst;
(3)      | state=stable ∧ p. typ=start ∧ from=nodeport→
               current_trans := p. source;  linkport!p ;
               pending := 1;  state := out_burst;
(4)      | state=in_burst ∧ p. typ=start ∧ from=linkport→
               current_trans := p.source ;  nodeport!p ;
               linkport!ackp;
(5)      | state=in_burst ∧ p. typ=start ∧ p. source=current_trans ∧ from=nodeport→
               state := stable;
(6)      | state=out_burst ∧ p. typ=start ∧ from=nodeport→
               current_trans := p. source;  linkport!p ;  pending := pending+1;
(7)      | state=out_burst ∧ p. typ=start ∧ from=linkport ∧ pending > 1→
               myturn := ¬ myturn ;  pending := pending−1;
(8)      | state=out_burst ∧ p. typ=start ∧ from=linkport ∧ pending=1 ∧ myturn →
               myturn := false;  state := stable;
(9)      | state=out_burst ∧ p. typ=start ∧ from=linkport ∧ pending=1 ∧ ¬ myturn →
               current_trans := p. source;  nodeport!p ;
               myturn := true;  state := in_burst;
(10)     | state=out_burst ∧ p. typ=ack ∧ from=linkport ∧ pending > 1→
               pending := pending−1;
(11)     | state=out_burst ∧ p. typ=ack ∧ from=linkport ∧ pending=1→
               state := stable;
        fi;
end;
```

Fig. 7. Program for internal arbiter of Algorithm 3.1.

and in the simplest case, when an acknowledgment is received from the far end of the link, the arbiter enters the stable state.

The arbiters contain several supplementary variables. The variable *current_trans* identifies the terminal whose burst is currently active at the arbiter. The variable *my_turn* is used to resolve contention when two start packets cross on a link in opposite directions. The arbiters at opposite ends of each link initialize these variables to complementary values to ensure consistent contention resolution. Each arbiter also has a variable *pending*, which counts the number of start packets that have been sent to *linkport*, but not acknowledged (either implicitly or explicitly). Finally, each arbiter has a packet *ackp*, which is just a packet whose type field is set to **ack**.

The most subtle part of the algorithm is the part that deals with contention resolution across a link. The important thing here is that both arbiters recognize when start packets have crossed on the link (we call this a contention event). This can be tricky, since an arbiter may send several start packets before receiving an indication that any of the packets was received. The key to recognizing a contention event is some form of acknowledgment. It turns out that one need not acknowledge every start packet, only the ones that are not involved in contention events. A transition diagram for the arbiter is shown in Fig. 6 and a program defining the detailed logic in Fig. 7. The numbers labeling the guards in Fig. 7 correspond to the numbers labeling the arcs in Fig. 6. Notice that the algorithm does nothing when $state$ = **in_burst**, $p.typ$ = **start**, $p.\text{source} \neq current\_trans$ and $from = nodeport$. This case occurs when two or more bursts arrive at a node at about the same time. The burst that wins the contention is the one whose start packet exits the node *last*.

We use a slightly different arbiter for the terminals. The

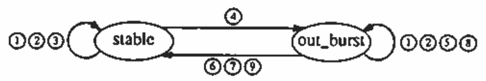

Fig. 8. Transition diagram for terminal arbiter.

```
do linkport?p →relay(p ,linkport,nodeport)
 | nodeport?p →relay(p ,nodeport,linkport);
od;

procedure relay(packet p , port from, port to)
(1)     if p. typ=data ∧ p. source=current_trans→
               to!p ;
(2)      | p. typ=end ∧ p. source=current_trans→
               to!p ;  current_trans=null ;
(3)      | state=stable ∧ p. typ=start ∧ from=linkport→
               current_trans := p. source;  nodeport!p ;  linkport!ackp;
(4)      | state=stable ∧ p. typ=start ∧ from=nodeport ∧ current_trans=null →
               current_trans := p. source;  linkport!p ;
               pending := 1;  state := out_burst;
(5)      | state=out_burst ∧ p. typ=start ∧ from=linkport ∧ pending>1→
               myturn := ¬ myturn ;  pending := pending−1;
(6)      | state=out_burst ∧ p. typ=start ∧ from=linkport ∧ pending=1 ∧ myturn →
               myturn := false;  state := stable;
(7)      | state=out_burst ∧ p. typ=start ∧ from=linkport ∧ pending=1 ∧ ¬ myturn →
               current_trans := p. source;  nodeport!p ;
               myturn := true;  state := stable;
(8)      | state=out_burst ∧ p. typ=ack ∧ from=linkport ∧ pending>1→
               pending := pending−1;
(9)      | state=out_burst ∧ p. typ=ack ∧ from=linkport ∧ pending=1→
               state := stable;
        fi;
end;
```

Fig. 9. Program for terminal arbiter.

terminal arbiter does not allow the terminal to start a burst if there is another burst already in progress. Since the terminal has only one incident link (and hence one arbiter), there is also no need to resolve contention among bursts arriving simultaneously at the terminal. Consequently, the arbiter can be slightly simpler, having only one transitory state, **out_burst**. A transition diagram for the arbiter is given in Fig. 8 and a program in Fig. 9. We refer to the algorithm implemented by the two arbiters just described as Algorithm 3.1. Fig. 10 illustrates the operation of Algorithm 3.1. The figure uses the same notation introduced earlier with a few additions. First, the notation $A_x(t)$ indicates an acknowledgment packet belonging to the burst with source x passing a given point in the channel at time t. The asterisks at one end of each link means that the *my_turn* variable belonging to the arbiter at the marked end of the link is initially true. The labels inside the internal nodes indicate the timing and sequence of events at those nodes. The sequence shown begins with a start packet sent by terminal b at time 0; this is received at the internal node on the left where it is replicated and sent out the other two incident links. It is also acknowledged on the link back to b. As the start packet crosses the link to terminal a, it passes a start packet coming from a. The start packet from a wins the contention on the link and consequently, propagates through the channel. At the internal node on the right side it contends with another start packet coming from c and loses the contention, allowing that start packet to propagate.

We note without proof that Algorithm 3.1 satisfies properties P_1–P_5. P_1 is satisfied since packets are never delayed by the arbiters. Property P_2 is satisfied since the nodes transmit only well-formed bursts and the arbiters propagate only packets whose source field matches the variable *current trans*. Property P_3 is satisfied (with $k = 1$), since only packets with source field equal to *current_trans* are propagated. Property P_4 is satisfied because if no terminal transmits after time t, any terminal active at time t will still be active at $t + \Delta$, and if no terminal is active at time t, one of the set of contending terminals at time t (if any) will be active at $t + \Delta$, since the arbiters always make consistent decisions.

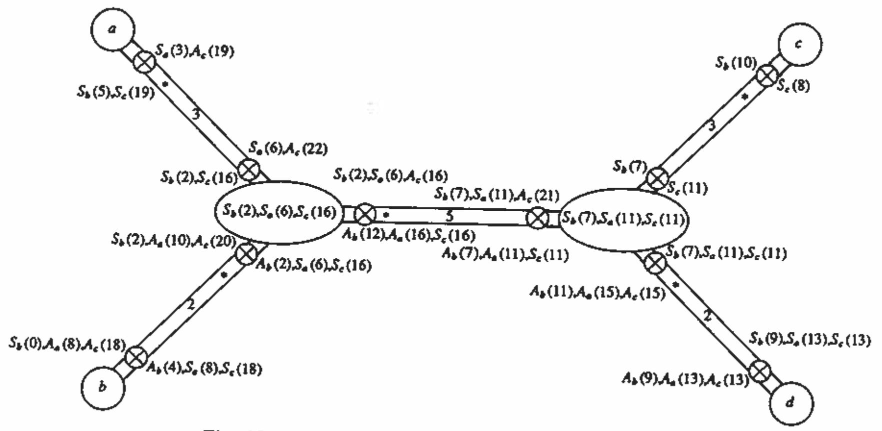

Fig. 10. Example of operation of Algorithm 3.1.

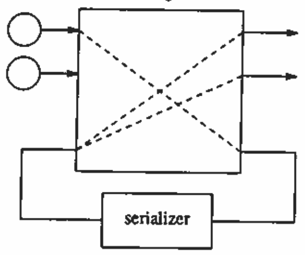

Fig. 11. Serializing start packets.

We close with a discussion of some of the basic assumptions made in this section and their implications for a practical realization of Algorithm 3.1. We first address the question of delays. We have assumed, for simplicity of description, that link delays are fixed and node delays are zero. Neither of these properties is essential for a practical algorithm. We do require that links process packets sequentially (that is, one packet cannot pass another on a link). We also require that all packets arriving at a node on a particular link and leaving on another, leave in the same sequence in which they arrived, and in the case of start packets, that every arbiter "see" start packets coming out of the node in the same sequence. This last property, can be implemented without difficulty. In a typical network, the node is implemented as a high speed packet switching fabric (see [6], for example). The start packets can be serialized by first sending them to a dedicated *serializer* port on the switch fabric, which is fed back and then broadcast to all ports in the channel. This is illustrated in Fig. 11. So long as the total volume of start packets is not too large, this solution can be effective.

We have also assumed that start, end and acknowledgment packets can be reliably transmitted. This is essential for correct operation and to maintain the synchronization of the state information at the opposite ends of each of the links. This reliance on perfect transmission and synchronization makes the algorithm rather fragile. A practical version would have to incorporate additional mechanisms to allow detection of and recovery from synchronization loss. Such additions would probably preclude a simple hardware implementation.

The intricacy of Algorithm 3.1 suggests that some simplification might be achieved by having a single arbiter per link rather than requiring two. Unfortunately, this observation is mistaken. Our requirement that one of a set of contending bursts eventually get access (expressed formally in P_4), requires that different arbiters in the connection make consistent decisions concerning any given set of bursts. If there were a single arbiter per link, that arbiter would still have to make consistent decisions with the arbiters on the links adjacent to it. This is no easier (and possibly more difficult) than the task faced by an arbiter pair on each link. Moreover, with an arbiter pair we can exploit the small time delay through the node to simplify arbitration at the node, limiting the more complex arbitration to the link where each arbiter need be concerned with only one other arbiter.

B. A Transparent Algorithm

The algorithms considered up to now all incorporate explicit control packets to coordinate the access arbitration. As we have seen, this leads to a number of complications since the operation of these algorithms depends critically on reliable transmission of control packets. In this section, we modify the algorithm of the previous section in three ways. First, we eliminate the use of explicit control packets and instead rely on contention among data packets. This provides a transparency that makes the algorithm simpler to use and eliminates (in part) the need for perfectly reliable transmission of control packets. The second change involves the method used to pick a winner, when two bursts contend across a link. The algorithm considered here, uses a fixed priority scheme that sacrifices fairness for the sake of simplicity. These changes make possible one final simplification, involving the arbitration method used to resolve contention at a node.

The algorithm (called Algorithm 3.2) treats the channel as a directed tree with one particular node designated as root. All links then connect a parent node to a child node, with parents having priority. What this means is that if two nodes start to transmit at about the same time, the node whose packet first reaches the nearest common ancestor of the two nodes is given priority. Once a burst reaches the root of the connection, it is guaranteed to be successful, since the root is a common ancestor of all the nodes. While this scheme favors nodes that are close to the root, it appears to allow higher throughput than the fair scheme and is much simpler to implement. In the following, we describe the single transmitter version, but in this case, the extension to multiple transmitters is straightforward.

Two different types of arbiters are required for the algorithm. For every link, the arbiter nearest the root gives priority to packets coming from the node and the arbiter furthest from the root gives priority to packets coming from the link. We refer to these two arbiters as *near-end* and *far-end* arbiters, respectively.

The state diagram of a near-end arbiter is given in Fig. 12, and the corresponding program is in Fig. 13. The arbiter has three states: **free**, **up**, and **down**. When in the **free** state, there is

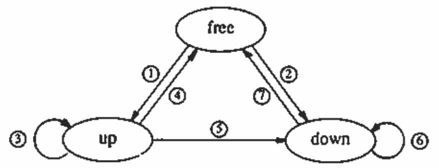

Fig. 12. Transition diagram for near-end arbiter.

```
do linkport?p →relay(p ,linkport)
 | nodeport?p →relay(p ,nodeport);
 | timeout →relay(p ,timeout );
od;

procedure relay(packet p , port from)
(1)    if state=free ∧ from=linkport→
                p.lno := lno ; nodeport !p ;
                reset (timer ); state := up;
(2)     | state=free ∧ from=nodeport→
                linkport!p ; reset (timer ); state := down
(3)     | state=up ∧ from=linkport→
                nodeport!p ; reset (timer );
(4)     | state=up ∧ from=timeout →
                state := free;
(5)     | state=up ∧ from=nodeport ∧ p.lno < lno →
                linkport!p ; reset (timer ); state := down
(6)     | state=down ∧ from=nodeport→
                linkport!p ; reset (timer );
(7)     | state=down ∧ from=timeout →
                state := free;
        fi;
end;
```

Fig. 13. Program for near-end arbiter

no active burst, when in the **up** state, there is an active burst coming from the link and when in the **down** state, there is an active burst coming from the node. When in the **down** state, packets received from the link are discarded. When in the **up** state, packets received from the node can preempt the currently active burst and cause a transition to the down state if the burst comes from a link whose link number is smaller than the number of the arbiter's link (identified by the variable *lno*). The arbiter on the link connecting to the node's parent (which is a far-end arbiter) always puts a link number of 0 in its packets, allowing it to preempt bursts from other links. The elimination of explicit control packets requires that some implicit mechanism be used to define the end of a burst. We adopt a simple timeout mechanism for this purpose. In the near end arbiter, the timeout causes a transition from either the **up** or **down** states back to the **free** state. The timer is reset whenever a packet is sent. The state diagram of a far-end arbiter is given in Fig. 14, and the corresponding program in Fig. 15. In this case, there are just two states: **up** and **down**. A packet received from the link always causes a transition from **up** to **down**. If the timeout expires, the link reverts to the **up** state.

The algorithm provides the best performance if the root node is as close as possible to the geographical center of the connection. We close with a brief sketch of an efficient distributed algorithm for finding the center. Let u be a node with neighbors $v_1, \cdots, v_d$, $(d \geq 2)$. Let x_i be the length of a longest path starting from u and passing through v_i. Let y_i be the length of a longest path starting from v_i that does not pass through u. Let y^* be the largest of the y_i (for simplicity, assume it is unique), let x_1^* and x_2^* be the largest two of the x_i and let v^* be the neighbor on the path of length x_1^*. We note that assuming all links have strictly positive delay, $2y^* < x_1^* + x_2^*$ implies that u is on the longest path (the diameter) of the tree and there must be some u on the diameter that satisfies this inequality. Also, if u is on the diameter than $x_1^* \leq x_2^* + \delta(u, v^*)$ if and only if u is a center node. Note that there may be two center nodes.

Thus, we can identify the center nodes of the tree if each node can learn the values of the x_i and y_i. This, however, is easy to

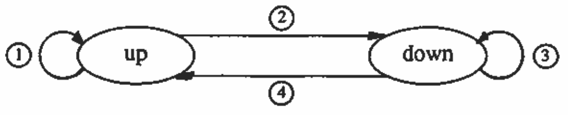

Fig. 14. Transition diagram for far-end arbiter.

```
procedure relay(packet p , port from)
(1)    if state=up ∧ from=nodeport→
                linkport!p
(2)     | state=up ∧ from=linkport→
                nodeport!p ; state := down;
(3)     | state=down ∧ from=linkport→
                p.lno := 0; nodeport!p ; reset (timer );
(4)     | state=down ∧ from=timeout →
                state := up;
        fi;
end;
```

Fig. 15. Program for far-end arbiter.

do, so long as each node knows the delay across each of its incident links. The algorithm is initiated by the terminal nodes, each of which simply sends a packet to its neighbor. The packet contains a field, which is used to carry a delay estimate and the terminals initialize this field to zero. When an internal node u receives a packet from a neighbor v_i, the delay value in the packet is equal to y_i. The node computes x_i by adding $\delta(u, v_i)$. Once u has received messages from $d - 1$ of its neighbors, it sends a packet to the remaining neighbor, containing the largest y_i value computed so far. Later, when it receives a packet from the last neighbor, it sends a packet to every other neighbor v_j containing the largest value in $\{y_1, \cdots, y_d\} - \{y_j\}$. The algorithm requires exactly $2(n - 1)$ messages to compute the x_i and y_i for all the nodes where n is the number of nodes in the channel. By the remarks in the previous paragraph, once this is done, the center nodes can identify themselves. An arbitrary tie-breaking rule can be used to pick a unique center node and for all other nodes u, the neighbor v_i for which x_i is maximum is on the path from u to the center.

C. Multiple Transmitters and Priorities

The next algorithm we consider offers a simple implementation supporting multiple transmitters, plus the option of prioritized access. It can be implemented with explicit start and end packets or without them. For simplicity we describe the version with start and end packets and assume initially that the start and end packets are transmitted reliably. At the end of the section we point out how the requirement for reliable transmission of start and end packets can be eliminated. The maximum number of simultaneous transmitters is k. We provide priorities for each burst by including a priority field in each start packet. Once a burst becomes active, it can be interrupted only by a higher priority burst. If two bursts of the same priority contend with one another, an arbitrary but fair decision is made to select the winner.

We let $\pi(b)$ denote the priority of a burst b. Priorities have nonnegative integer values with smaller numbers corresponding to "higher priority." We define a partial ordering on bursts, which we denote by the symbol $<_t$; if b_1 and b_2 are bursts, then $b_1 <_t b_2$ if $\pi(b_1) < \pi(b_2)$ or $\pi(b_1) = \pi(b_2)$ and $b_1 \in \alpha(t)$ and $b_2 \notin \alpha(t)$. We define a prioritized access arbitration algorithm by properties P_1-P_4 given earlier together with one new property:

P_5' If no terminal transmits any packet after time t, then given any two bursts b_1, $b_2 \in \alpha(t) \cup \gamma(t)$, $b_1 \notin \alpha(t + \Delta) \wedge b_1 <_t b_2$ implies $b_2 \notin \alpha(t + \Delta)$.

```
do linkport?p →relay(p ,nodeport)
 | nodeport?p →relay(p ,linkport);
od;

procedure relay(packet p , port to)
        if p. typ=start ∧ |B |<k →
                to!p ;  B := B ∪{(p. prio,p. rand,p. source};
        | p. typ=data ∧ (*,*,p.source ) ∈ B →
                to!p ;
        | p. typ=end ∧ (*,*,p.source ) ∈ B →
                to!p ;  B := B −{(*,*,p. source)};
        | p. typ=start ∧ |B |=k →
                x := max B ;  y := (p. prio,p. rand,p. source);
                if y<x →B := B ∪{y }−{x }; to!p ; fi;
        fi;
end;
```

Fig. 16. Program for internal arbiter of Algorithm 3.3.

```
do linkport?p →relay(p ,linkport ,nodeport)
 | nodeport?p →relay(p ,nodeport ,linkport);
od;

procedure relay(packet p , port from, port to)
        if p.typ=start ∧ |B |<k →
                to!p ;  B := B ∪{(p. prio,p. rand,p. source};
        | p.typ =data ∧ (*,*,p.source ) ∈ B →
                to!p ;
        | p. typ=end ∧ (*,*,p.source ) ∈ B →
                to!p ;  B := B −{(*,*,p. source)};
        | p. typ=start ∧ |B |=k ∧ from=linkport→
                x := max B ;  y := (p. prio,p. rand,p. source);
                if y<x →B := B ∪{y }−{x }; to!p ; fi;
        | p. typ=start ∧ |B |=k ∧ from=nodeport→
                (x ,*,*) := max B ;
                if p.prio <x →B := B ∪{y }−{x }; to!p ; fi;
        fi;
end;
```

Fig. 17. Program for terminal arbiter of Algorithm 3.3.

As indicated above, start packets now must have a priority field in addition to the *source* field that is part of every packet. In addition, we require a third field called *rand* to ensure fair contention resolution; this field if filled with a randomly selected integer by the terminal arbiter when a start packet is transmitted from a node.

We now describe an algorithm that implements prioritized access, which we refer to as Algorithm 3.3. The basic idea is to use the natural numeric ordering on the triples (*prio*, *rand*, *source*) to resolve contention in a consistent way. There are two arbiter types, *internal arbiters* at all the internal nodes and *terminal arbiters* at all the terminals. Each internal arbiter monitors the traffic passing through it and if more than k bursts attempt to pass through it at once, it will cease propagating the burst with the largest triple. Since the priority field is treated as most significant, high priority burst are treated preferentially; the random field provides fair treatment of bursts at the same priority level and the source field eliminates the possibility of ties, ensuring consistent contention resolution at all arbiters.

Each internal arbiter maintains a set B containing triples; one for every source that is currently authorized to transmit. Each triple includes the values (*prio*, *rand*, *source*) transmitted in the start packet initiating the burst. The arbiter monitors the bursts passing through it, updates the set B as necessary and discards packets from sources that are not currently authorized to transmit. A program implementing such an arbiter is given in Fig. 16. In this program, * is used to indicate a "don't care" field. Terminal arbiters are slightly different, in that they must prevent a terminal from preempting another burst of the same priority. A program implementing a terminal arbiter appears in Fig. 17. We note without proof that the algorithm satisfies properties P_1–P_4 and P_5'. P_1 is satisfied since packets are never delayed by the arbiters. Property P_2 is satisfied since the nodes transmit only well-formed bursts and the arbiters propagate only packets belonging to bursts in B. Property P_3 is satisfied since $|B| \leq k$ at all times and the packet is propagated only if it belongs to a burst in B. Property P_4 is satisfied since the arbiters on a link always make consistent choices and never discard contending bursts unless already propagating k bursts. Property P_5' is satisfied because the arbiters preempt a burst if and only if a higher priority burst must be served.

Algorithm 3.3 can be efficiently implemented in a practical multipoint communication network. The packet processors that implement the arbiters must maintain a copy of the set B for every channel passing through them. When a packet is received, the appropriate set must be retrieved from memory, used to make decisions and possibly updated, then written back to memory. The main hardware cost is the memory, which amounts to roughly from 5 to $10k$ bytes per channel (where k is the number of simultaneous transmitters).

As with Algorithm 3.1, this algorithm requires reliable transmission of start and end packets. However, the consequences of packet loss are less severe; a lost start packet lead to a lost burst, a lost end packet leads to a temporary loss of a portion of the channel bandwidth. We can reduce the probability of these events by simply transmitting several start and end packets, to avoid packet loss due to link errors, and giving control packets higher priority to avoid their loss due to buffer overflows. Another approach is to eliminate explicit control packets altogether. In this scheme, we include the *prio* and *rand* values in all data packets and use a timeout in place of an explicit end packet. The arbiters extend the stored records, to include the time that the most recent packet was received from that source. When a packet is handled the arbiter first scans B, throwing out any entries that are too old. It then proceeds in the normal way to handle the burst.

IV. Conclusion

In this paper, we have introduced the problem of access arbitration in tree-structured communication channels with long link delays. This is an interesting problem and one of some importance for communication networks supporting general multipoint communication. We have introduced two general approaches to solving the problem, and described five specific algorithms. We have omitted detailed discussions of correctness and performance, as our primary purpose is to introduce the problem and survey several candidate solutions. We close with a few comments on the relative merits of these solutions.

The token-based algorithms have a built-in latency associated with token circulation that limits their throughput. This is most problematical for traffic consisting of short bursts. The contention-based algorithms avoid this latency, but it's not entirely clear if this translates into a real difference in throughput.

All but Algorithm 3.1 can be implemented in a practical way. The critical dependence of Algorithm 3.1 on perfect transmission of control packets probably makes it unworkable in most practical settings. Algorithms 2.1, 3.2, and 3.3 admit simple hardware implementations, while 2.2 is most reasonably implemented using a programmable processor; while such an implementation is workable, it would be more costly and have limited throughput. There is a wide variance in the sensitivity of the various algorithms to the reliability of the underlying packet transmission. Algorithm 3.1 is the most delicate while 2.1 and 3.2 are the most robust, operating effectively even in the presence of fairly high packet loss rates. Algorithm 3.3, while not inherently robust can be made robust by transmission of multiple control packets or incorporation of control information in data packets along with introduction of timeouts. Finally, we note that of the contention-based algorithms, only 3.3 offers the full functionality of multiple transmitters, although 3.2 can be extended to accommodate this.

Based on this preliminary assessment, we conclude that Algorithms 2.1, 3.2, and 3.3 show the most promise and bear further study. There are several possible directions for future research. Perhaps the most important is to formulate a reasonable performance model to use in assessing the throughput and delay characteristics of the token passing approach versus the contention-based approach. In the case of token passing, one must consider alternative token circulation strategies. In the case of contention-based algorithms, we need to distinguish different kinds of throughput; one that counts only bursts received by all terminals and another that gives "partial credit" to bursts received by some subset of the terminals. We note that in general, the most interesting performance questions arise in the context of short bursts.

References

[1] A. Albanese, "Star network with collision-avoidance circuits," *Bell Syst. Tech. J.*, vol. 62, pp. 631-638, 1983.

[2] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, pp. 453-456, 1975.

[3] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666-677, 1978.

[4] W. Stallings, *Data and Computer Communications.* New York: Macmillan, 1985.

[5] P. P. Treis, "Collision-free star broadcast medium," *Automatika i Vychislitel' naya Tekhnika*, vol. 19, pp. 42-46, 1985.

[6] J. S. Turner, "Design of a broadcast packet switching network," *IEEE Trans. Commun.*, vol. 36, pp. 734-743, 1988.

Riccardo Melen received the degree in electrical engineering from Politecnico di Torino, Torino, Italy, in 1981.

He joined CSELT in 1983, where he was involved in research concerning the design and performance evaluation of interconnection networks for multiprocessors. Since 1985 he has been working on telecommunication systems, especially dealing with broad-band packet switching systems. During 1987 he has been working as a visiting researcher at Washington University in St. Louis. His current research interests are in the areas of broad-band networks, ATM switching, interconnection network theory, and switching systems control.

Jonathan S. Turner (M'77-SM'88-F'90) received the M.S. and Ph.D. degrees in computer science from Northwestern University in 1979 and 1981, respectively.

He is an Associate Professor of Computer Science at Washington University, where he has been since 1983. His primary research interest is the design and analysis of switching systems, with special interest in systems supporting multicast communication. He has been awarded a dozen patents for his work on switching systems and has a number of widely cited publications in this area. His research interests also include the study of algorithms and computational complexity, with particular interest in the probably performance of heuristic algorithms for NP-complete problems.