January 1987

Report No. STAN-CS-87-1146
Also numbered KSL-87-02

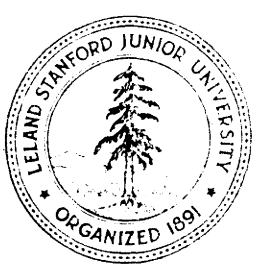
A Point-to-Point Multicast Communications Protocol

by

Gregory T. Byrd, Russell Nakano, Bruce A. Delagi

Department of Computer Science

Stanford University
Stanford, CA 94305



Knowledge Systems Laboratory
Report No. KSL-87-02

January 1987

A Point-t o-Point Mult icast Communications Protocol

Gregory T. Byrd†
Department of Electrical Engineering
Stanford University
Stanford, CA *94305*

Russell Nakano
Department of Computer Science
Stanford University
Stanford, CA 94305

Bruce A. Delagi
Worksystems Engineering Group
Digital Equipment Corporation
Maynard, MA 01754

Abstract

Many network topologies have been proposed for connecting a large number of processor-memory pairs in a high-performance multiprocessor system. In terms of performance, however, the communications protocol decisions may be as crucial as topology. This paper describes a protocol to support point-to-point interprocessor communications with multicast. Dynamic, cut-through routing with local flow control is used to provide a high-throughput, low-latency communications path between processors. In addition. multicast transmissions are available, in which copies of a packet are sent to multiple destinations using common resources as much as possible. Special packet terminators and selective buffering are introduced to avoid deadlock during multicasts. A simulated implementation of the protocol is also described.

Contents

List of Figures

List of Tables

1 Introduction

Many network topologies have been proposed for connecting a large number of processor-memory pairs in a high-performance multiprocessor system [1]. These topologies are often evaluated in terms of the average number of hops traversed by a packet, for example. However, the network performance may depend as much on its communication protocol as on its physical topology. For example, suppose the average number of hops in a network is M and the average packet length is N. In a store-and-forward network, the transmission time of a packet would be proportional to M x N. If cut-through switching is used, however, the transmission time would be proportional to M + N, a significant difference for relatively large values of M or N. An appropriate communications protocol, then, is crucial if the full benefits of a topology are to be realized.

The protocol described in this paper is designed to fully utilize network resources. Dynamic, cut-through routing with local flow control is used to provide a high-throughput, low-latency communications path between processors. In addition, a multicast facility is provided, in which copies of a packet are sent to multiple destinations, using common resources as much as possible.

Dynamic routing means that the communications channel to be used is chosen at transmission time, based on what channels are available. The alternative, static routing, would prescribe a specific channel for every destination-if that channel were not available, the transmission would be blocked. Dynamic routing, by adapting to current channel usage, attempts to balance the network load. It is especially useful when the communications traffic is unpredictable or variable over time [2]. Balancing the load allows more of the communications resources of the system to be well used throughout a computation.

Cut-through routing [3] means that a routing decision is made on the fly, as a packet is received, rather than first buffering the entire packet and then deciding what to do with it.[1] This reduces buffering requirements in the system, since the packet does not need to be stored at intermediate points in the transmission.[2] Kernami and Kleinrock [5] demonstrate that the cut-through approach outperforms both circuit switching and message switching (store-and-forward) when the communication paths are short, network utilization is relatively high, and messages are fairly small.

Flow control, in general, is any mechanism which attempts to regulate the flow of information from a sender to match the rate at which the receiver can accept it [6]. In this protocol, a transmission may be blocked and resumed in the event of network congestion. If an output channel becomes blocked, the sender stops sending data and halts the flow of data from upstream. When the channel becomes unblocked, the transmission is continued from where it was

[1] A related concept is staged circuit switching, described in [4].

[2] Cut-through switching as described in [3] requires that the packet be completely buffered if the output channel is blocked. In this protocol, no further data will be received from downstream until the channel becomes available. Thus, packet buffering is not required.

halted. The flow control mechanism is local, because actions are taken based on the state of the downstream component rather than global information about the entire network.

Multicast transmissions in a point-to-point network allow a packet to be sent to multiple destinations, using common resources as much as possible. The packet is replicated as needed, and subsets of the original target list are assigned to the copies. Thus, "virtual busses" are available precisely as and when they are needed. Selective buffering and special packet terminators allow potential deadlock conditions in multicasts to be detected and avoided.

The network components which define the protocol are introduced in Section 2, and the protocol itself is described in Sections 3 and 4. Finally, Section 5 describes an implementation of the protocol in the CARE simulation system.

2 Components

This section defines the network components used by the protocol. The protocol is defined by the behavior of these components and the values that are passed among them. Of course, these components do not necessarily correspond to distinct physical entities in a machine which implements this protocol-they are merely a useful means of specifying the functional behavior of such a machine.

The site component corresponds to a processor-memory pair in the target machine. In particular, a site contains an operator, an evaluator, a router, some local storage, and some network interface components, which are called net-inputs and net-outputs (see Figure 1).

The *evaluator* is the part of the site which executes application code. The evaluator can request network activity, but otherwise has no role in the network behavior of the system, so very little will be said about it in this paper.

The *operator* is responsible for handling system-level activity, including communication. In the target machine, it would create packets to be sent over the network and accept transmissions destined for its associated processor. The operator and evaluator communicate through shared local memory. The details of this communication will not be addressed in this paper.

The site components which interface directly to the network are called *net-inputs* and *net-outputs*. On each site, there is a net-input/net-output pair connected to the operator, for local packet origination and delivery, as well as a pair for every communication channel to the network.[3] We will refer to the pair connected to the operator as the "local" net-input and net-output.

The net-input is responsible for accepting a packet, making connections (using the router) to one or more net-outputs, and sending it on its way. The net-output is concerned with delivering the packet to a particular location, either the local operator or the next site on the transmission path. Note that,

[3] The exact number of net-inputs/net-output pairs required by a site depends on the network topology.

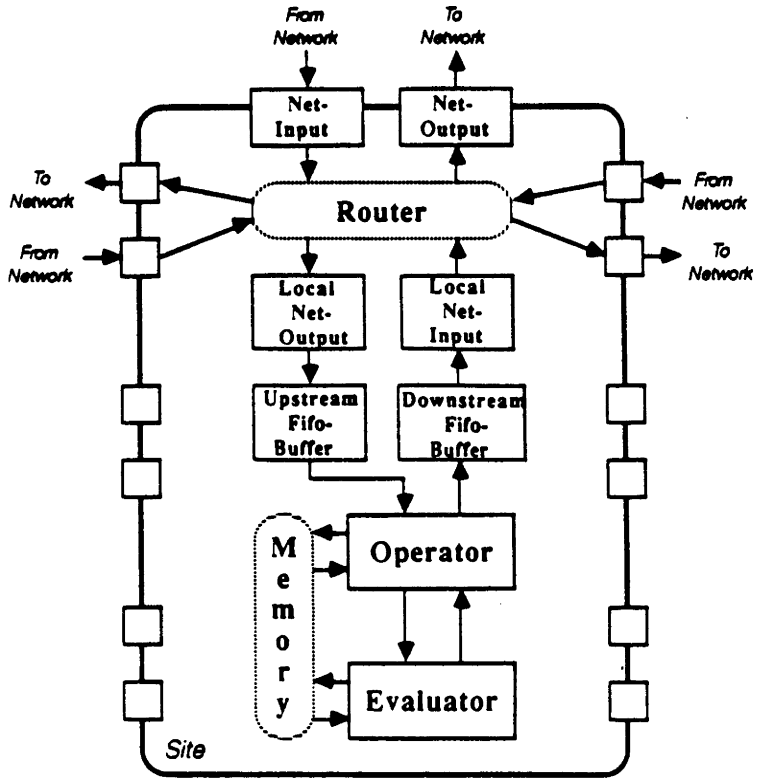


Figure 1: Components of a CARE *site.*

because of cut-through routing, net-inputs and net-outputs are only required to have enough storage for one word of a packet, rather than the entire packet.

The *router* connects all the net-inputs on a site to all the net-outputs. When it receives a packet from a net-input, it determines the destination (or destinations) and makes the connection to the appropriate net-output (or net-outputs). Also, flow control information from the net-outputs are relayed by the router to the appropriate net-input.

A pair of buffers, called ***fifo-buffers***, queue packets between the operator and local net-input and net-output. The *upstream* fifo-buffer queues packets from the network to the operator; the *downstream* queues packets from the operator to the network.

3 Protocol Overview

3.1 Packets

Figure 2 shows the organization of a packet. The first part a packet is devoted to the *target entries.* Each entry contains a target address, a pointer to data within the packet, and flags indicating the last target in the list.

Following the target addresses are zero or more words of ***data*** and a **one-word** *packet terminator.* There are three distinct packet terminators, as shown in Table A, which are used by the operator to determine the status of the packet .[4]

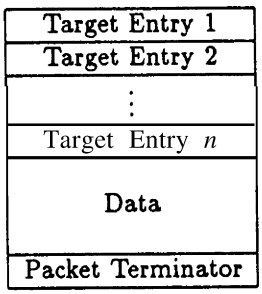


Figure 2: Organization of a packet.

Terminator	*Meaning*
:end-of-packet	Normal packet termination.
:abort-packet	Packet is to be discarded by operator.
:local-end-of-packet	Treat as :end-of-packet, except ignore all packet targets other than the local site.

Table A: Packet terminators.

3.2 Packet Transmission

The transmission path of a packet is shown in Figure 3. First, an evaluator requests a packet transmission. The operator then sends the packet (through a buffer) to the local net-input. For the moment, assume that there is only one target for the packet. (This is called a *unicust* transmission.) The router then decides which net-output should receive the packet, based on the target address and the availability of net-outputs, sets up a connection between the local net-input and the selected net-output, and begins the transfer of the

[4] As described in Subsection 4.3.

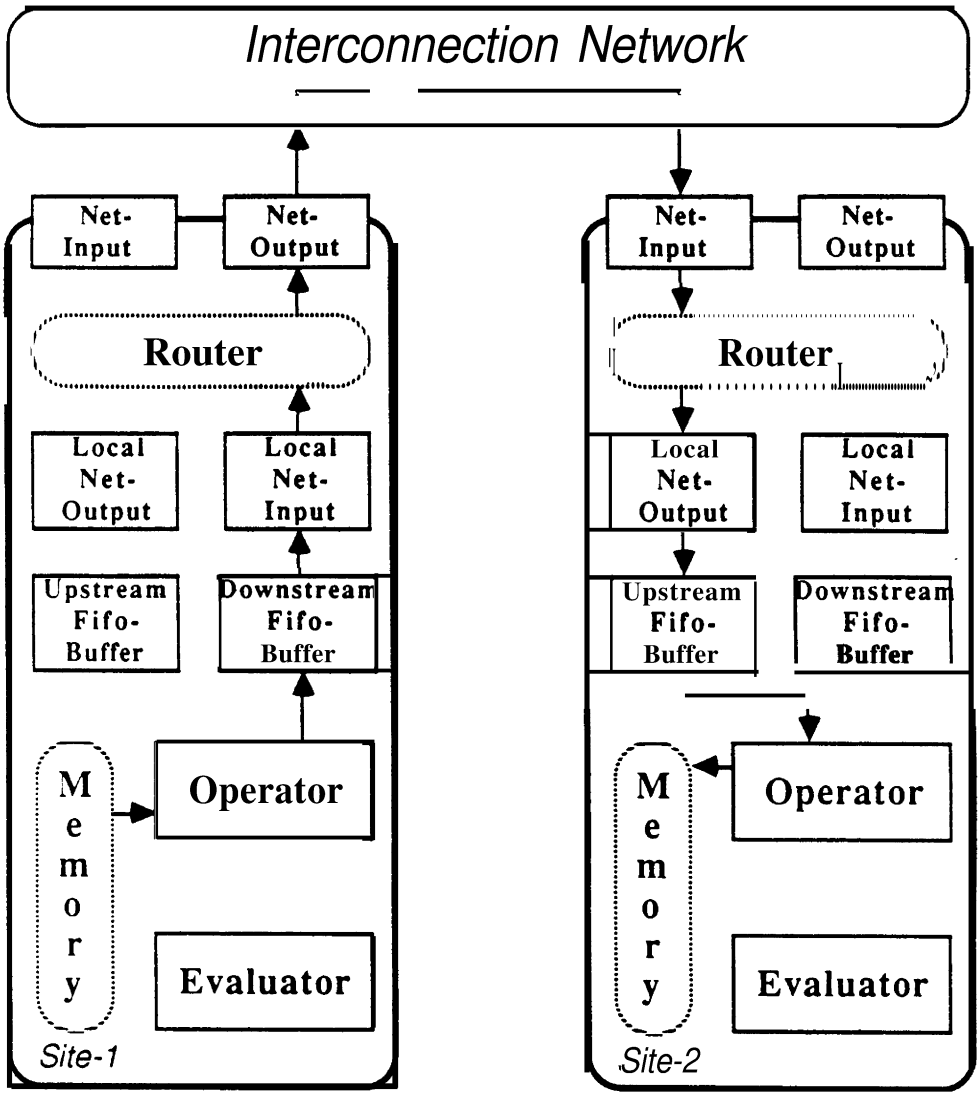


Figure 3: Network component interconnections. Packets travel in the direction marked by arrows. Flow control information flows in the opposite direction.

packet. Each non-local net-output is physically connected to a net-input on a (logically) neighboring site. When available, this net-input accepts the packet, and its router sends the data to the local net-output, if the target has been reached, or to another net-output, if not. This continues until the target has been reached, where the local net-output delivers the packet to the operator (through a **fifo-buffer).** The operator can then perform whatever operation is specified by the packet, such as storing the value in memory or queueing some operation for the evaluator, for example.

If the packet has more than one target, the router may split it-that is, it may send (essentially) the same packet to several net-outputs. This is called a ***multicast*** transmission. Each transmitted packet contains a distinct ***subset*** of the targets of the original packet. The copying operation is done during transmission, one word at a time, as opposed to buffering the entire packet and making copies. If one branch of the multicast is blocked, the net-input sends pad characters down the other branches until valid data may be sent down all the paths. The pad characters are thrown away when received by a fifo-buffer..

3.3 Flow Control

Flow control information, in the form of status signals, flows in the direction opposite to packet transmission. There are four distinct status signals, **as** shown *in* Table B. The status signals are used to indicate to the upstream component whether the packet or packet terminator can safely be transmitted.

A 'free signal means that the component is not currently involved in a transmission and is ready to receive data. An 'open signal is used when the component is involved in a transmission and is ready to receive the next word of the packet. If the transmission becomes blocked for some reason, a 'wait signal is sent upstream to temporarily halt the flow of data. Finally, the **'abort-**request signal indicates that a potential deadlock condition has been detected and the transmission may be aborted. Details about how these signals are generated and interpreted will be presented in Section 4.

Status	*Meaning*
'free	Available to receive packet.
'open	Packet header has been received; available to receive more data.
'wait	Busy or network is blocked; do not send more data.
'abort-request	Potential deadlock **detected.**[a]

[a] **Only** a **fifo-buffer** may originate the 'abort-request signal.

Table B: Flow-control signals.

I Component I	Odd Phase	Even Phase
Net-Input	Latch status from downstream and conditionally open data latch to allow data to flow downstream.	Open status latch to allow status information to flow upstream.
Net-Output	Open status latch to allow status information to flow upstream.	Latch status from downstream and conditionally open data latch to allow data to flow downstream.

Table C: Communication cycle phases.

A *communication* cycle consists of two major phases[5] (see Table C). During one phase, a component latches the status signal from downstream. Based on that signal, it may open its data latch to allow data from upstream to flow downstream. Otherwise, it holds the previously latched data. During the other phase, the component opens its status latch to allow status information (perhaps modified by the component) to flow upstream. The cycles of adjacent network components (e.g., net-inputs and net-outputs) are arranged so that one component is latching the status information while the downstream component is determining the status for the next cycle. Thus there cannot be a race between the latching of data and the status signal which controls it.

3.4 Deadlock Avoidance

The existence of packet multicasts introduces the possibility of deadlock. A packet traveling through the network acquires the use of network resources (e.g., net-inputs and net-outputs) and simultaneously excludes the use of those resources by other packets. Without special attention paid to the possibility of deadlocks, it is possible that resources are consumed to perform the multicast, but completion of the multicast is not possible because the resources acquired are insufficient.

If only unicast transmissions were allowed, this kind of deadlock would not occur. Assuming that a packet cannot be infinitely long, a blocked unicast packet will eventually either acquire the network connection that it needs or be (temporarily) stored at the local site (freeing up any upstream resources for

[5] Any necessary signal serialization would occur within a major phase.

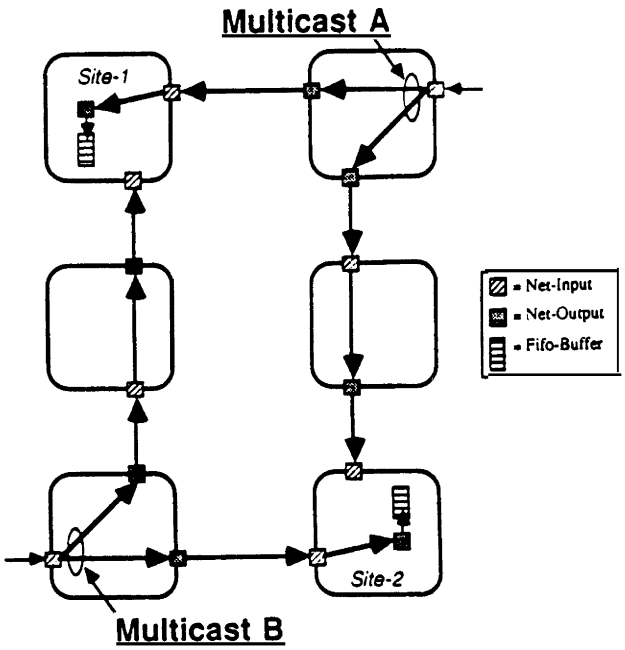


Figure 4: Example of deadlock in a multicast.

this packet). . In other words, any resource that is acquired will eventually be released.

Figure 4 illustrates an example of how multicast deadlock can arise. Suppose we have two multicast transmissions, call them A and B, with common destinations, *site-1* and site-2 Suppose that one of the packets from multicast A has already gained access to the local net-output on *site-l.* A packet from multicast B has similarly gained access to the local net-output on site-b For multicast A to continue, it needs to gain access to the local net-output of *site-2*;[6] for B to complete, it needs to gain access to the local net-output on *site-l.* Also, neither of the multicasts can release the resources it has already required until the transmission is completed. Since each multicast has acquired a resource that the other needs, a deadlock results.

In order to recover from such a situation, the system must:

- Detect a potential deadlock condition, such as the situation described **above;**
- Back out of the unsafe condition (by aborting one or more transmissions, thereby releasing some set of resources); and

[6] **The** transmission cannot continue because the net-input cannot send any words until all branches of the multicast are ready to receive it. Since the branch waiting for the local net-output of site-2 is blocked, none of the branches may proceed.

- Retransmit the aborted packets later, when the network is (hopefully) less congested.

Whenever a packet is split for multicast, the protocol requires that a copy of the original packet (with a complete target list) be sent to the local net-output. This packet will then be stored in a **fifo-buffer**, so that it may be retransmitted in the **case** that the current multicast must be aborted due to deadlock.

The packet terminator has two roles in deadlock avoidance. First, a **fifo-buffer** can detect a potential deadlock if the packet terminator has not been received in a "reasonable" amount of time.[7] Second, the packet terminator indicates to all operators which received the packet what should be done with it. For example, a multicast is aborted by sending the **:abort-packet terminator** downstream-all operators which receive a packet with this terminator will ignore the packet. Also, the operator which receives the copy of the original packet can tell whether it needs to be retransmitted by looking at its terminator. More details will be presented in the next section.

These actions are sufficient to prevent persistent deadlock during multicasts. However, since there is finite storage in the system, a scenario can be constructed in which all the storage becomes committed and no packets can be delivered. The protocol does not prevent this type of resource exhaustion. The assumption is made that the designed capacity of the system is sufficient for its applications.

4 The Protocol

This section provides a detailed description of the behavior of each of the network components. First, however, we present the details of the deadlock avoidance mechanisms, so that the behavior of individual components can be understood in the context of an overall transmission.

4.1 Deadlock Avoidance Mechanisms

The protocol mechanisms which allow deadlocks to be detected and avoided are as follows:

1. If a packet has multiple targets, before a router can split the packet for multicast, the local net-output must be available. This is to insure that a connection to the **fifo-buffer** is possible, so that the packet may be stored for possible retransmission.

   (a) The local net-output is sent a copy of the packet which contains a complete target list (rather than a subset). This assures that the packet may be retransmitted to all of its targets if the multicast is aborted.

[7] See Subsection 4.1.

(b) If the local net-output is unavailable, then the packet may be sent, but only to a single target. The intent is that a packet sent in this fashion will either visit each target site individually, or will eventually reach a site with an available local net-output and be multicast to the remaining sites on the packet target list.

2. Upon receiving the front end of a packet, the **fifo-buffer** starts a timeout **procedure.**[8] If the timeout occurs before the packet terminator is received, the **fifo-buffer** asserts the 'abort-request signal upstream on the flow control line.

   (a) When a net-input currently engaged in a multicast receives an 'abort-request (from a downstream **fifo-buffer)** before it sends the packet terminator, the net-input goes into abort *mode.*

   (b) Net-inputs which are not involved in a multicast ignore the **'abort-**request signal; net-outputs merely pass an 'abort-request upstream.

3. In *abort mode,* the net-input performs several actions:

   (a) All connected non-local net-outputs are sent the **:abort-packet** terminator, and they **are** disconnected from the net-input. This signals any downstream operator to **ignore** the packet when it is received. At this point, only the connection to the local net-output is active.

   (b) The 'open flow control signal is sent upstream to unblock the packet transmission.

   (c) When the packet terminator arrives at the net-input, the packet terminator that is received is passed on to the local net-output. The **:abort-packet terminator causes the local operator to discard** the packet. The **:end-of-packet** terminator will result in retransmission, if the original target list contained remote (not local) sites.

4.2 Generic Component Description

Next we describe the behavior of individual components. Most of the components are described as finite state machines which have input ports, output ports, and internal state variables. The input and output ports are used to pass packets and flow control information-packets flow downstream, flow control signals flow upstream. The ports and their functions are shown in Table D. Figure 5 shows a "generic" network component, with its input and output ports.

[8] **The** intent is to determine when the packet terminator has not arrived in a "reasonable" amount of time. This might actually be a timer, where the interval is some function of the expected packet length, or it might be some threshold limit for the number of consecutive pad characters a **fifo-buffer** will accept. The details are not specified by the protocol documented here.

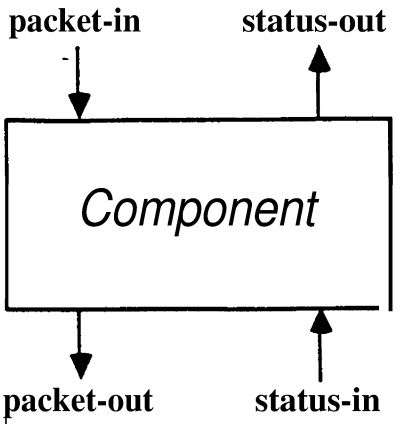


Figure 5: Generic network component.

Port	Function
packet-in	Packet data from upstream component.
packet-out	Packet data to downstream component.
status-in	Flow control from downstream component.
status-out	Flow control to upstream component.

Table D: Input and output ports.

The behavior of most of the components can be described in terms of states and transitions between those states (i.e., a state machine). It is often useful to illustrate the states and transitions in a state transition diagram, as in Figure 6. The transitions are labelled with the condition used to trigger the transition, and the status signal to be sent upstream (through the status-out port) when the transition is made.

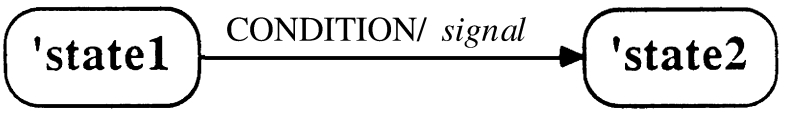


Figure 6: A state transition diagram.

4.3 Operator

The operator sends and receives packets through the network and through the memory it shares with the evaluator. Thus, it has more than one set of ports for

packet communication. To avoid confusion, the ports it uses to communicate with the network are **prefixed** network- (e.g., network-packet-in), while the ports used for communication with the evaluator are prefixed evaluator- (e.g., evaluator-packet-in). Only network communication will be discussed in this paper.

With respect to the network, both the upstream and downstream components of an operator are fifo-buffers. The upstream **fifo-buffer** queues packets from the local net-output and sends them to the operator. The downstream **fifo-buffer** queues packets from the operator and sends them to the local **net-**input.

Two state variables are used by the operator for network communications:

1. network-buffer: Used to temporarily store an incoming packet from the network.

2. network-buffer-status: Indicates whether the packet in the **network-**buffer has been serviced ('new or **'old).**

4.3.1 Sending a Packet

The operator **has** a queue of operations, or requests, which it services in order of arrival. If the **head** of this queue is a packet to be sent out into the network, and network-status-in is 'free, indicating that the downstream **fifo-buffer** is ready to accept a packet, the operator sends the packet (with an **:end-of-packet** terminator) through the network-packet-out port.

4.3.2 Receiving a Packet

A packet arrival at the operator is signalled by the appearance of data on the network-packet-in port. The network-status-out port is set to 'open, which signals to the upstream **fifo-buffer** to keep sending packet data until the packet terminator **arrives.** The packet data is stored in the network-buffer.

The arrival of an **:end-of-packet** signifies that the packet transmission was successful. Network-buffer-status is set to 'new, signifying that the data in the temporary buffer should be looked at. At some later time, the operator services the packet and sends a 'free signal to the incoming fifo-buffer (through network-status-out), indicating that another packet may be received, and network-buffer-status is set to 'old, so that the packet is not serviced twice.

If the operator notices that some or all of the target addresses of the received packet do not correspond to its own address, the packet is sent back out into the network. This might happen for one of the following reasons:

1. During a unicast transmission, a net-input could not make a connection to the desired net-output. The packet is forced into the local fifo-buffer, so that the operator may resume the transmission at a later time, freeing up the net-input and its upstream components.

2. A multicast transmission was aborted. The local fifo-buffer received a copy of the packet with a complete target list, so that the packet could be retransmitted in case of an abort.

A **:local-end-of-packet** terminator instructs the operator to accept the packet, as in the case of :end-of-packet, but to ignore any non-local target addresses. This indicates that a multicast was successful, and so does not have to be retried.

The arrival of an **:abort-packet** terminator instructs the operator to discard the packet. The operator then asserts 'free on network-status-out, indicating that another packet may be received, without setting network-buffer-status to 'new-that is, the packet data in the temporary buffer is never serviced.

4.4 Fifo-buffer

Each site has two fifo-buffers, which have identical behavior but perform slightly different functions. One fifo-buffer is upstream with respect to the operator, and the other is downstream.

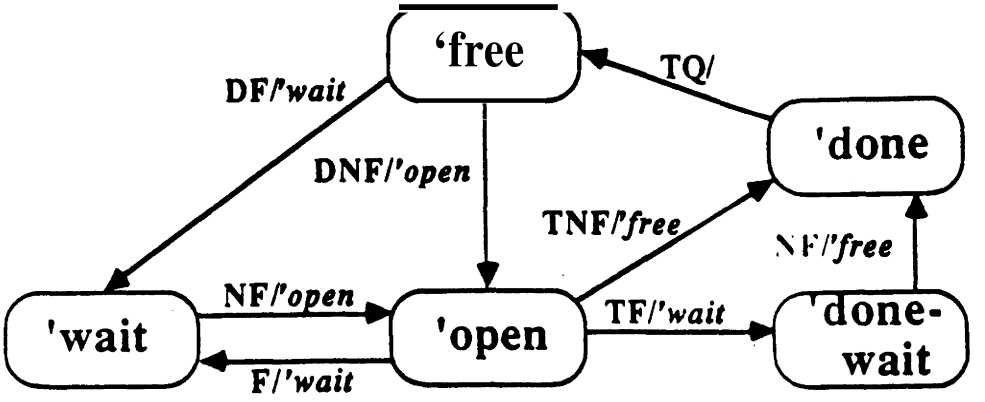
On its output side, the upstream **fifo-buffer** is connected to the operator, while the downstream flfo-buffer is connected to the local net-input. If the queue is not empty, the **fifo-buffer** responds to a 'free or 'open signal on the status-in port by removing the oldest item from the queue and sending it through the packet-out port. If a 'wait signal is received, the transmission is temporarily halted until a non-'wait signal appears.

On its input side, the upstream fifo-buffer is connected to the local **net-**output, and the downstream **fifo-buffer** is connected to the operator. The **fifo-**buffer needs to keep track of (1) whether the packet data and terminator have been received and (2) whether they have been placed in the queue. The state diagram of the input side is shown in Figure 7, and the states are described in Table E.

State	*Meaning*
I 'open	Readv for more data: terminator not received.
'wait	Queue full; terminator not received.
'done	Terminator received, but not yet queued.
'done-wait	Terminator received, but queue full.
'free	Terminator aueued, **readv** for next **packet.**

Table E: Input states for fifo-buffer.

The **fifo-buffer** begins in the 'free state. Whenever data arrives on the packet-in port, if the queue is not full, the 'open state is entered and 'open is asserted on status-out. If the queue is full, the 'wait state is entered and 'wait is asserted; when space becomes available in the queue, the 'open state



Condition	Meaning
DF	Data arrives, and queue full.
DNF	Data arrives, and queue not full.
F	Queue full.
NF	Queue not full.
TF	Terminator arrives, and queue full.
TNF	Terminator arrives, and queue not full.
TQ	Terminator queued.

Figure 7: Fife-buffer state diagram.

is entered and 'open is asserted. If the queue becomes full at any point in the transmission, the 'wait state is entered and the 'wait signal is asserted on status-out, so that no more data will be sent from upstream. When space becomes available, the 'open state is re-entered, and 'open is sent upstream to resume the flow of data.

When a packet terminator arrives, if the queue is not full, the 'done state is entered and 'free is asserted on status-out. If the queue is full, the 'done-wait state is entered first, which asserts 'wait until space is available in the queue. Then the 'done state may be entered. When the terminator is actually in the queue, the 'free state is entered, and the fifo-buffer is ready to receive another packet.

Not shown in the state diagram is the timeout procedure mentioned in Subsection 4.1. This is because the details of the timeout procedure are dependent on the implementation. The intent of the timeout is to indicate when the fifo-buffer has been waiting an unusually long time for the packet terminator. When

a timeout occurs, the 'abort-request signal is sent upstream through **status-**out. The fifo-buffer behavior then continues **as** described above.

4.5 Net-Input

The downstream component from a net-input is a router, but the values on the status-in port are actually originated from a downstream net-output and are passed through the router. If the net-input is local (connected to an operator), its upstream component is a fifo-buffer; otherwise, its upstream component is a net-output (on a logically neighboring site). The states of the net-input are shown in Table F, and the transitions are illustrated in Figure 8. A state variable, connection, is used to save the type of the current downstream connection.

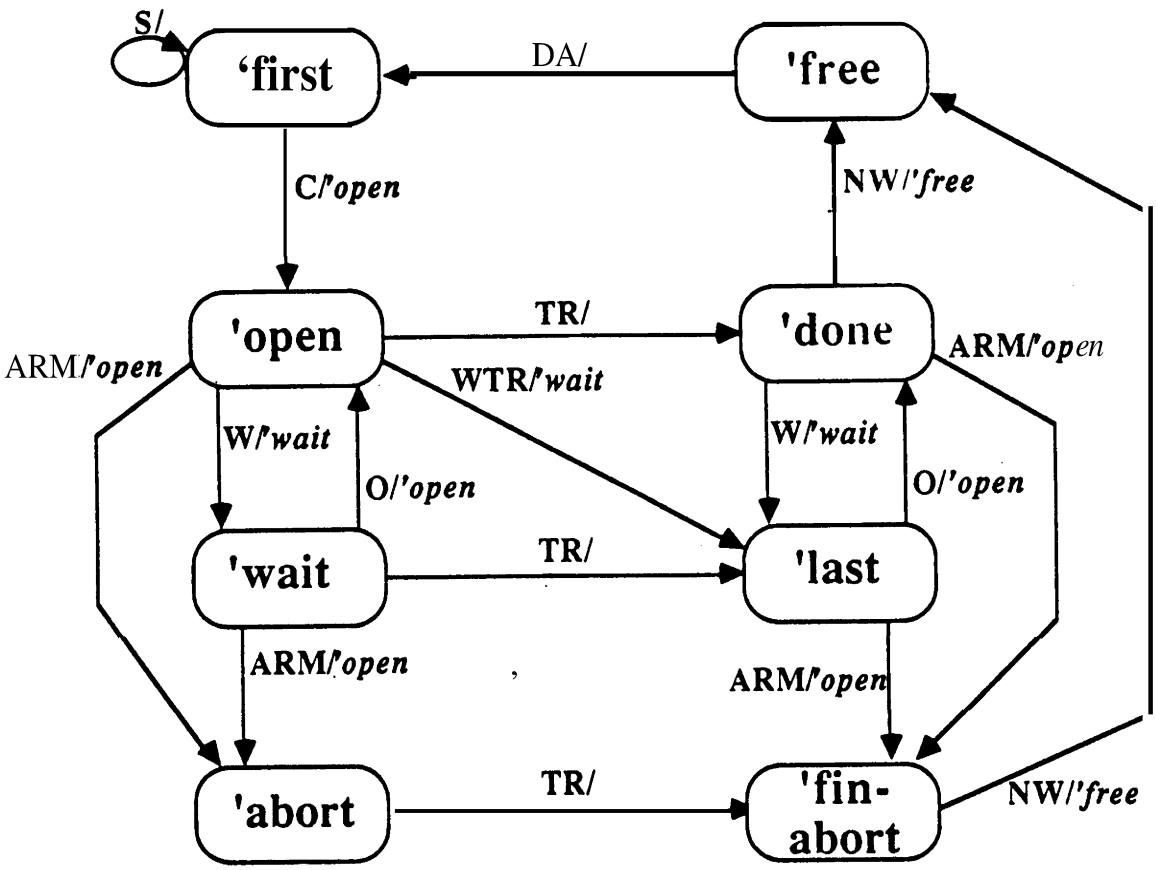
Value	*Meaning*
'first	Packet received, but net-input not yet connected to the network.
'open	Connected to network and packet transmission in progress.
'wait	Downstream requested wait after transmission started.
'done	Terminator received, but not sent.
'last	Downstream requested wait after terminator received, but before it was sent.
'abort	Abort requested from downstream.
'fin-abort	Abort requested, and terminator received.
'free	Idle—remains in this state until the network connection goes free *and* a new packet is received.

Table F: States for net-input.

The net-input begins in the 'free state, with all its downstream connections free. When the front end of a packet arrives on packet-in, it is sent directly to the router, which attempts to make the proper connection based on the packet's target list. If the router is successful, it makes the appropriate connections, begins transmission of the packet to the connected net-output(s), and returns one of the following values on connection, which indicates the type of connection that was made:

'unicast All targets of the packet reside on a single site.

'passthru The packet has multiple sites in its target list, but has only been sent to a single net-output. This type of connection indicates that the local **fifo-buffer** was not available to accept a copy of the packet.



Condition	Meaning
DA	Data arrives.
S	'Seek returned (try again).
C	Connection obtained.
W	'Wait rec'd on status-in.
0	'Open rec'd on status-in.
ARM	'Abort-request rec'd & this is a multicast
TR	Terminator received.
WTR	Terminator and 'wait received.
NW	Non-wait signal rec'd on status-in.

Figure 8: Net-input state diagram.

'all-remote The packet has multiple sites in its target list, and the router has made connections to multiple net-outputs. The packet's target list contained only non-local sites.

'some-local The packet has multiple sites in its target list, and the router has made connections to multiple net-outputs. The packet's target list included the local site.

If the connection attempt is unsuccessful, because of busy channels, for example, the router returns 'seek, which prompts the net-input to try again. If the number of unsuccessful attempts exceeds a threshold, the router sends the packet to the local **net-output**—the local operator will retransmit the packet if any destination in the target list is not local.

A successful connection causes the net-input to enter the 'open state and to assert 'open on status-out. At this point, several possible transitions can occur. We will first consider the *commit* case, where no 'abort-request is received and the net-input successfully delivers the packet. Later, we will consider the ***abort*** case.

4.5.1 Commit Mode

Ignoring 'abort-request for the moment, two possible events can occur: (1) the packet terminator arrives on the packet-in port, or (2) one or more downstream. net-outputs send 'wait over the status-in port. The 'wait state is entered if a 'Gait signal is received; the 'done state is entered if the packet terminator is received; the 'last state is entered if both are received. Figure 8 shows the possible transitions among these states. Whenever a 'wait is received from downstream, 'wait is asserted on status-out to halt the information flow from upstream, as well. The wait condition is cleared when an 'open signal appears on status-in. This indicates that all the downstream net-outputs are ready to receive the packet terminator and causes a transition from 'wait to 'open, or from 'last to 'done.

If the net-input is in the 'done state and 'open is received from downstream, the appropriate packet terminators are sent according to the type of connection:

'unicast ***or*** 'passt hru: An :end-of-packet is sent to the single downstream net-output (local or remote).

'all-remote: An :end-of-packet is sent to all the non-local connected **net-**outputs; **:abort-packet** is sent to the local net-output, because the operator should discard the packet rather than attempt to re-send it.

'some-local: An :end-of-packet is sent to all non-local connected net-outputs; :local-end-of-packet is sent to the local net-output, so that the operator will ignore the remote addresses in the packet's target list.

After the packet terminator has been sent out, all connections to net-outputs are released, the 'free state is entered, and the net-input is available to receive the next packet.

4.5.2 Abort Mode

Abort mode is entered if an 'abort-request is received from downstream before the packet terminator is sent downstream, and the current transmission is a multicast ('all-remote or 'some-local). ('Abort-request is ignored on a non-multicast transmission. From this point, we will assume a multicast transmission.)

If the 'abort-request is received before the packet terminator (i.e., while in 'open or 'wait), the 'abort state is entered. When the packet terminator arrives, the net-input enters the 'fin-abort state. Alternatively, the **'abort-packet** could arrive after the packet terminator, in which case 'fin-abort is entered directly from 'done or 'last.

Whenever abort mode is entered, the net-input sends an **:abort-packet** to all non-local connected net-outputs and disconnects them. They will, in turn, pass the terminator downstream when possible. The only connection retained is to the local net-output. When the local net-output is ready to receive the packet terminator (i.e., 'open is received on status-in), the net-input passes on whichever type of terminator it received. The two cases are as follows:

:end-of-packet No upstream packets have been aborted, so it is the responsibility of this site to abort the downstream transmissions and to re-transmit the packet. Upon receiving the **:end-of-packet,** the operator will notice some non-local addresses in the packet's target list and will send it back out into the network.

:abort-packet Some upstream site is aborting the multicast and will eventually resend the packet. The operator on this site, then, is instructed to ignore this packet.

The net-input then enters the 'free state and releases the local connection, ready to receive the next packet.

4.6 Router

The router is responsible for the following:

- Determining to which net-outputs a packet should be sent, based on its list of target addresses, the system routing strategy, and the current availability of net-outputs; and
- Creating, maintaining, and deleting the connections between a net-input and a set of net-outputs, including transmitting data and flow control signals between them.

The router, unlike the other components, is not modelled as a finite state machine-it is conceived as a priority network (implemented in combinational logic, for example). Information about routing and active connections can be thought of as residing in the tables shown in Table G.

Table	*Contents*
preference- table	For each logical output direction, a sorted list of net-outputs to be considered.
input-connection-table	For each net-input, a list of connected net-outputs.
output-connection-table	For each net-output, its connected net-input.
output-status-table	For each net-output, its transmission status.

Table G: Routing tables.

The first words of the packet are always the target list. As each target is received, the router makes an appropriate connection to a net-output and sends that address downstream. The routing (for'each target address) takes place in a single communication cycle,[9] so there'is no additional delay introduced by the router.

If there is only one target, the router makes the connection (see below) and returns 'unicast. If there is more than one target, the router checks the status of the local net-output. If the status is 'free, then the appropriate connections are made and either 'all-remote or 'some-local is returned. If the local net-output is not 'free, then a single connection is made based on the first target on the list (ignoring the other targets), and the returned connection value is 'pass thru.

Making a connection involves determining the logical "direction" (e.g., up or down) of the target from the local site, then determining which net-output should be used for that direction, and finally updating the connection tables and starting the packet transmission.

Determining the logical direction depends on the network topology and is usually straightforward. For example, a grid or torus requires only some arithmetic comparisons between the target address and the local address to get Up, Down, Right, Left, or some combination of these. A hypercube, on the other hand, requires an exclusive-OR operation to see which bits in the destination address are different than the local address. Equally simple operations can be envisioned for most other network topologies, as well.

[9] See Subsection 3.3.

Once the logical direction is determined, the router looks in the **preference-table** for *a* list of net-outputs to consider. This table implements the system routing strategy and is determined when the system is built. It lists, in decreasing order of preference, all the net-outputs that might be used to send a packet in a given logical direction. The router checks all the status of each of these, in turn, until an available net-output is found. If none is found, then the connection fails, and 'seek is returned to the **net-input**.[10] Examples of routing strategies which may be implemented by the routing table are (1) try all net-outputs, starting with the closest to the target, (2) try only one net-output (static routing), and so forth.

During the transmission, the router is responsible for passing flow control information from the net-outputs to their connected net-inputs. If a net-output, for example, asserts 'wait on its status line, the router must relay that signal to the net-input which is connected to it. Also, the router cannot pass the net-input an 'open signal until *all* of its downstream net-outputs are in a **non-wait** state. The input-connection-table, output-connection-table, and output-status-table are useful for these types of operations.

4.7 Net-Output

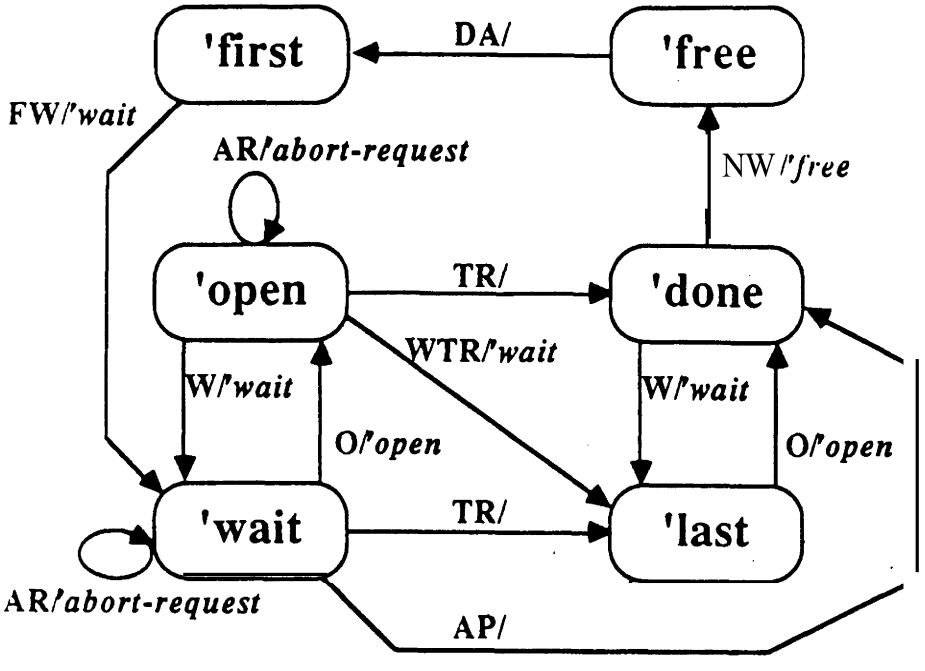
The upstream component of a net-output is always a net-input. On the downstream side, the local net-output is connected to the **fifo-buffer** which delivers packets to the operator, while a non-local net-output is connected to a net-input on a logically neighboring site. The net-output states are listed in Table H, and the transitions are illustrated in Figure 9.

State	*Meaning*
'first	Packet received, but not yet sent.
'open	Packet transmission in progress.
'wait	Downstream requested wait.
'done	Terminator received, but not sent.
'last	Downstream requested wait after terminator received, but before it was sent.
'free	Terminator sent, ready to receive next packet.

Table H: States for net-output.

The net-output is initially in the 'free state. When a packet arrives on packet-in, it enters the 'first state. If its downstream component (either a

[10] **Note** that, in the case of a multicast, partial **finds** (in **which** only a subset of the targets can be assigned to net-outputs) **must** be forced to fail (by sending an **:abort-packet** terminator over the connections made thus far), or the operator would not know which parts of a **multicast** to retransmit in case of an abort.



Condition.	Meaning
DA	Data arrives.
FW	'Free or 'wait rec'd on status-in.
W	'Wait rec'd on status-in.
0	'Open rec'd on status-in.
AR	'Abort-request rec'd on status-in.
TR	Terminator received.
WTR	Terminator and 'wait received.
AP	:Abort-packet terminator received.
NW	Non-wait signal rec'd on status-in.

Figure 9: Net-output state diagram.

net-input or a **fifo-buffer)** has placed 'wait on the status-in port, the **net-**output asserts 'wait on status-out, which inhibits the upstream net-input from sending anything else. When the downstream component becomes ready to accept the packet, it will assert 'free.

When a 'free signal is received from downstream, the net-output transmits the packet and enters the 'wait state, asserting 'wait on status-out. The net-output remains in the 'wait state until an 'open signal is received from downstream.

The net-output then enters the 'open state, sending an 'open signal to the upstream net-input (via the router). Things then continue much the same **as** in the net-input. 'Wait is entered if the downstream component requests a wait and the packet terminator has not arrived. 'Done is entered when the packet terminator arrives; 'last is entered if a wait is requested from downstream after the terminator arrives. If an 'abort-request is received from downstream before the packet terminator arrives, it is relayed to the upstream net-input. If the packet terminator has already arrived, then the 'abort-request was premature and is ignored.

Then the net-output sends the packet terminator, when the downstream component is ready to accept it, and enters the 'free state. When the downstream net-input accepts the packet terminator and responds by asserting 'free, the net-output asserts 'free on its status line. The upstream net-input will then release the connection, and the net-output becomes available to receive the next packet.

5 CARE Implementation

In this section, we provide an overview of the implementation of the protocol in the CARE simulation system. CARE is a library of functional modules and intrumentation built on top of an event-driven simulator **[7]**, which is used to investigate parallel architectures. The typical CARE architecture is a set of processor-memory pairs (sites) connected by some communications network, though it can also be configured to represent a system of processors communicating through shared memory. The behavior and relative performance of CARE modules can easily be changed, and the instrumentation is flexible and useful in evaluating the performance of an architecture or in observing the execution of a parallel program.

CARE is implemented using Flavors-an object-oriented extension of **Zetal-**isp **[8]**. Roughly speaking, each component described in Section 2 is implemented as an object (an instance of a flavor). (One notable exception is the router-its functions and tables are assumed by the site object, rather than implemented **as** a separate component. Also, the memory at a site is not explicitly represented as an object, but exists implicitly in the simulator.) Associated with each object is a set of instance variables, used to hold state information, and

a set of ***methods***, procedures used by the object to respond to messages from other objects.[11] The instance variables loosely correspond to the ports and state variables used to describe the protocol in Section 4. In particular, each of the components which are described in terms of a state machine has a instance variable, packet-status, which hold the current state of the component.

These objects communicate through shared structures called ***vias***, which represent unidirectional data paths. These are the "wires" which connect the components' "ports." Asserting a value on the sending end of the via immediately (in simulated time) triggers an event for the object at the other end. Therefore, a via can be considered a zero-delay wire which can transmit any arbitrary value (not just single bits).

The simulation is **functional**,[12] rather than circuit-level, and event-driven, rather than clock-driven, because cycle-by-cycle simulation of a parallel machine would be extremely time-consuming, especially when the number of processors is large. For this same reason, we do not wish to model the transmission of *a* packet one word at a time. Instead, a packet is represented by two distinct parts, one representing the contents of the packet, and the other representing the packet terminator. In the following discussion, ***packet*** will refer to the first part (representing the front edge of a "real" packet), and ***packet terminator*** will refer to the terminator part.

In the simulation environment, **explicit** packet terminators allow us to (1) implement the deadlock avoidance mechanisms described earlier, and (2) model the transmission of a packet through the network in terms of its front edge and its back edge. In this way, if the time between the transmission of the packet (front edge) and its terminator in the simulator is the same as the transmission time of the packet in a real machine, we can accurately model the transmission of the packet without explicitly representing every word.

In the following subsections, we describe how the protocol is implemented in terms of objects, packets, and packet terminators.

5.1 Operator

The time required to transfer *a* packet from the operator to a **fifo-buffer** (one word at a time) would be proportional to the size of the packet. To model this,

[11] Objects and messages are only a software tool used by the simulator. Sending ***messages*** between objects in the simulator has no particular correspondence to sending ***packets*** between components in the target machine.

[12] **The** simulation is functional, in the sense that not every aspect of the hardware is simulated in detail. Some aspects are simulated by register transfer level behavior, while other aspects have only a functional description. For example, the execution of application code by the evaluator is not simulated at all-it is directly executed by the host machine. However, timing information for the execution of application code, based on measurements and estimates, is used to **assure** that the simulation is reasonably faithful to the execution of a "real" machine.

the operator delays an appropriate time between sending a packet and sending its terminator.

Because storage in the simulated fifo-buffer is in terms of packets, rather than bytes[13], there will be no wait signals received from the downstream **fifo-buffer.** Therefore, merely delaying for a time proportional to packet size is sufficient.

A CARE operator receives a packet as described in the protocol. Note that the time between receiving the packet and its terminator is dependent on the size of the packet plus any delays encountered on its transmission path.

5.2 Fifo-buffer

In the simulator, the amount of storage in the **fifo-buffer** may be set at run time.[14] Each packet or packet terminator takes up one space in the **buffer,** no matter what its actual size. In particular, the buffer cannot fill up in the middle of accepting a packet, so the 'wait state will never be entered. Thus the operator, which feeds data into a **fifo-buffer,** does not have to deal with any waiting time in the middle of transmitting a packet, **as** described above. This simplifies the implementation of the protocol, at the expense of a slight loss of fidelity in the simulation.

On the output side, however, the simulated fifo-buffer is more complex than the protocol indicates. If a packet is being output from the queue, the **fifo-buffer** must introduce a delay between the packet and its terminator to model the packet transit time. However, the transit time is not merely proportional to packet size, because downstream blocking could cause arbitrary delays in the transmission.

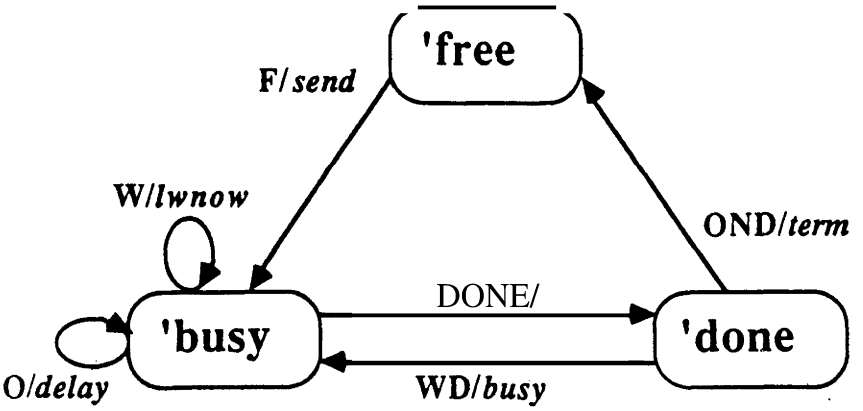
The simulated fifo-buffer output transitions are shown in Figure 10. In this case, the transitions are **labelled** with conditions and *actions,* rather than flow control signals. Some additional instance variables for the **fifo-buffer** are required to implement the **output function.** They are:

1. transmission-status: State of packet output.
2. delay: Accumulated time spent waiting.
3. last-wait: Event time when last 'wait was received.

Initially, transmission-status is 'free. If the downstream component requests data (status-in goes to 'free) and the queue is not empty, the top of the queue, which must be a packet, is placed on the packet-out via, **delay** is set to zero, and transmission-status goes to 'busy. Also, transmission-status is scheduled to go to 'done at a time that is proportional to packet size.

[13] **See** subsection 5.2.

[14] By setting the **care:***buffer-size***** variable to any positive integer, or to nil, which means "unbounded."



Condition	Meaning
F	'Free rec'd on status-in.
W	'Wait rec'd on status-in.
0	'Open rec'd on status-in.
DONE	'Done event.
WD	'Wait rec'd and [delay nonzero OR last-wait non-nil}.
OND	'Open rec'd and [delay = 0 AND last-wait = nil].

Action	Meaning
send	Send packet, schedule 'done for now + transmission-time.
lwnow	Last-wait = now.
delay	Delay = delay + (now - last-wait); Last-wait = nil.
busy	Schedule 'done for now + delay; Last-wait = nil.
term	Send terminator.

Figure 10: Implemented fifo-buffer output state diagram.

If no 'wait signals are received from downstream while the transmission is **'busy,** then the transmission will be done after the packet transit time has **elapsed,** and the **packet terminator** will be sent as soon as the downstream component **is ready** to receive it.

However, if 'wait is received during 'busy, last-wait is set to the current time and waiting is set to t. If 'open is received during **'busy,** the time spent waiting is added to delay and waiting is set to **nil.**

If 'open is received when transmission-status is 'done, and delay is non-zero, then 'busy is entered again, 'done is scheduled for the current time plus the accumulated delay, waiting is set to nil, and **delay** is set to zero. **Alternatively, if** waiting is t and delay is zero, then 'done has occurred in the middle of a wait; 'busy is entered, waiting is set to nil, and 'done is scheduled for the current time plus the difference between now and last-wait.

Finally, when 'transmission-status **is 'done, delay** is zero, and waiting is nil, the top item of the queue (which must be a packet terminator) will be sent. Then transmission-status becomes 'free, and the **fifo-buffer** is ready to respond to the next **data** request.

All of this is to ensure that the time between the packet and its terminator is dependent on the packet size plus any network delays along its path. The other components, net-inputs and net-outputs, do not require this added complexity on the output side. They will either maintain the current time separation or add to it due to downstream blockages, so there is no chance of their sending the packet terminator prematurely.

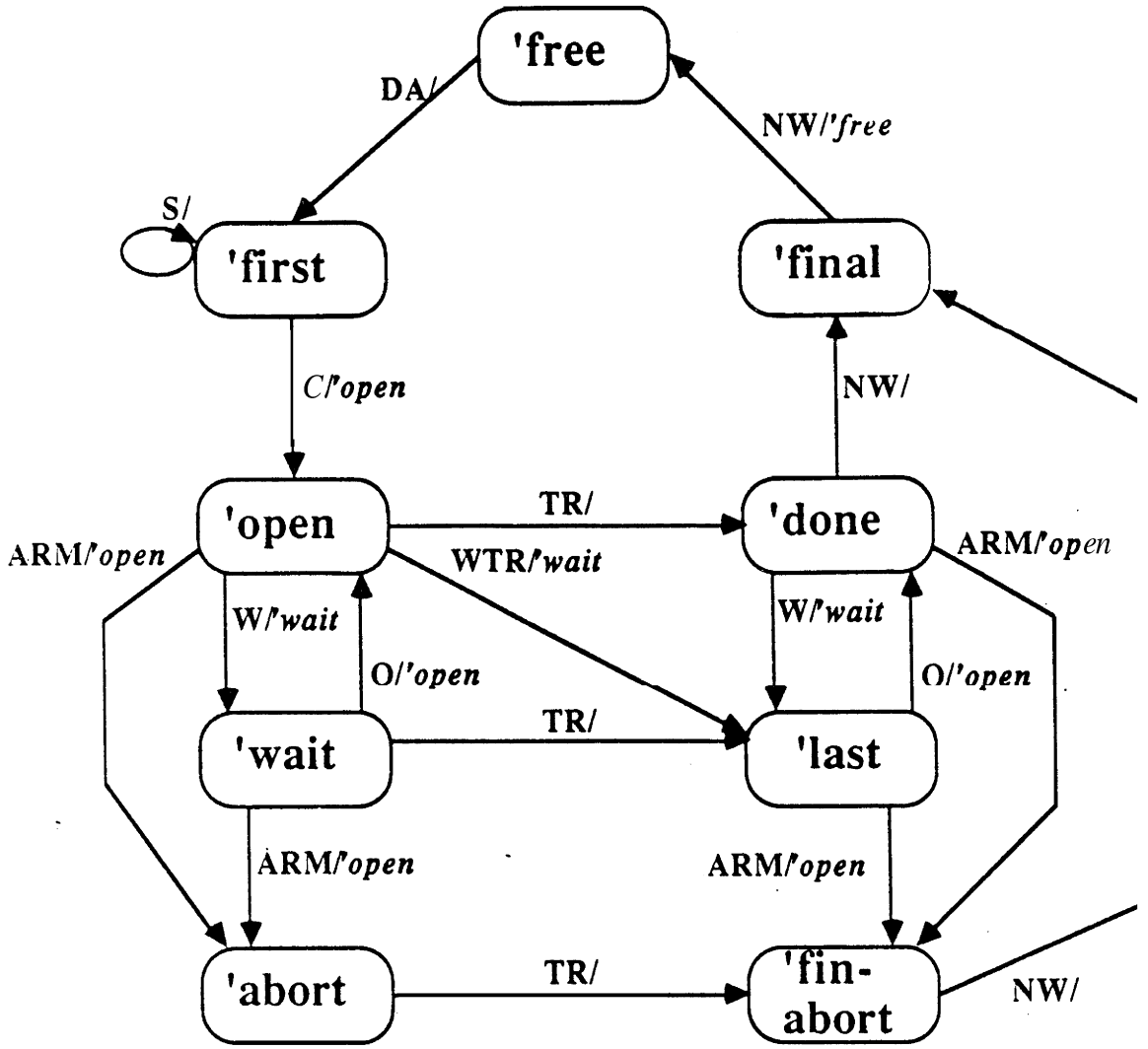
5.3 Net-Input

The main differences between the implementation and protocol concerning the net-input stem from the fact that there is no explicit router in **CARE.** Each **net-**input, then, communicates with the ***site*** which owns it (see Section **2),** rather than with *a* downstream router. The communication is done by passing Flavors messages, rather than asserting **data** on vias-thus, there is no packet-out instance variable, and status-in is not *a* **via.**[15]

To connect to net-outputs, the net-input sends a **:connect message** to the site. The site then performs the routing and makes the connections as described in Subsection 4.6, returning either 'seek or the type of connection made. Also, the site relays flow control information from the connected net-outputs by setting status-in.

Other site methods used by the net-input include **:disconnect-remote,** which releases the connections to all net-outputs except the local one, and **:send-all,** which transmits *a* packet or terminator to all connected **net-**outputs. **(Send-local and :send-remote** transmit to a subset of connected

[15] **Vias** must connect two distinct objects; status-in may be connected to any group of net-outputs at **a** given time, so using a via **is** not appropriate.



Condition	Meaning
DA	Data arrives.
S	'Seek returned (try again).
c	Connection obtained.
W	'Wait rec'd on status-in.
0	'Open rec'd on status-in.
ARM	'Abort-request rec'd & this is a multicast
TR	Terminator received.
WTR	Terminator and 'wait received.
NW	Non-wait signal rec'd on status-in.

Figure 11: Implemented net-input state diagram.

net-outputs.)

There is a potential software race in the simulator, which is avoided by adding an additional state in the net-input state machine description. If the net-input is in the 'done state and notices that none of the downstream **net-**outputs has asserted 'wait, it sends the packet terminator. However, there might be a simulation event scheduled for the same time slot in which one of the net-outputs receives a 'wait and propagates it upstream. In a real machine, this means that the terminator would not have been sent, but there is no way to "undo" the first action by the simulator.

Thus, instead of sending the terminator from the 'done state, the net-input schedules a transition to the 'final state two event-times later. This allows time for all the possible 'wait signals to be handled during the same event. When the 'final state is entered, the state of the connected net-outputs is again examined. If none of them are blocked, the packet terminators are sent immediately (in simulation time), and the 'free state is entered. Any 'wait signal which could arrive at that same instant would be too late to block the transmission in a real machine. The implemented version of the net-input state machine is illustrated in Figure 11.

5.4 Router

As mentioned earlier, there is no explicit router object in the CARE implementation. There are, however, site functions and methods which perform routing in response to a **:connect** message sent by a net-input.

The **:find-direction** method determines the logical direction of a target, given its address. This is defined as a method, rather than a function, because this operation is topology-dependent. In Flavors, we can define a specialized ***site*** object for a particular topology by changing this one method and inheriting the remaining behavior from the generic site definition.

The setup-targets function examines the target list, makes the connections, and copies the packet, as needed. Finally, the make-connections function is resposible for actually setting up connections and sending the packet downstream.

5.5 Net-Output

In the CARE implementation of the net-output, there is no explicit **status-**out instance variable for sending flow control information upstream. Instead, messages are sent to the ***site***, as above, which updates the status table for the particular net-output and relays the **information to** the connected net-input. There are **:wait**, **:open**, **:abort-request** and **:free** methods defined for the site for this purpose. Also, because packet input can come from any of the net-inputs on the site, packet-in is not implemented as a via.

Finally, on the initial transition into the 'wait state (from 'first) the **net-**output sends a **:first-wait** message, which updates the status table but does not trigger an event for the upstream net-input. This prevents unnecessary simulator events used to propagate the 'wait signal upstream; they are unnecessary because the net-input will not send anything else until the net-output sends an 'open signal.

5.6 Results

Variants of this protocol have been used for many CARE simulations over the course of several months. Though the performance has not been extensively measured, the protocol appears to offer reasonable performance over a range of network loads. Deadlocks and lost packets do not occur, even when the network is extremely congested. Thus, our experience with the protocol indicates that it offers efficient and robust one-to-one and one-to-many interprocessor communication.

6 Conclusion

A protocol for high-performance **interprocessor** communication has been presented. This protocol supports dynamic, cut-through routing with local flow control, which allows high-throughput, low-latency transmission of packets. In addition, multicast transmissions are allowed, in which a packet is sent to several targets using common resources as much as possible.

The protocol also prescribes mechanisms for detecting and avoiding deadlock conditions due to resource conflicts during multicast. In particular, .a copy of the packet is saved before it is split, special packet terminators are used to abort transmissions and trigger **retransmissions,** and random timeout intervals are used to detect potential deadlock conditions.

Finally, the implementation of this protocol in the CARE simulation system is described. Explicitly representing a packet as the front edge and the terminator allows accurate modelling of word-by-word packet transmission in a functional, event-driven simulator. Also, the success of the implementation indicates that this is a reasonable protocol for interprocessor communication.

References

[1] Tse-yun Feng. A survey of interconnection networks. Computer, 12-27, December 1981.

[2] V. Ahuja. Design *and Analysis of Computer Communication Networks.* McGraw-Hill, 1982.

[3] P. Kernami and L. Kleinrock. Virtual cut-through: a new computer communication switching technique. *Computer Networks,* 3:267, 1979.

[4] M. Arango, H. Badr, and D. Gelernter. Staged circuit switching. *IEEE Transactions on Computers,* G-34(2):174-180, February 1985.

[5] P. Kermani and L. Kleinrock. A tradeoff study of switching systems in computer communication networks. *IEEE Transactions on Computers,* C-29:1052, December 1980.

[6] Richard W. Watson. Distributed system architecture model. In *Distributed Systems-Architecture and Implementation,* chapter 2, pages 10-43, . . Springer-Verlag, 1981.

[7] Bruce A. Delagi, Nakul Saraiya, Sayuri Nishimura, and Greg Byrd. *An Instrumented Architectural Simulation System.* Technical Report KSL-86-36, Knowledge Systems Laboratory, Stanford University, January 1987.

[8] Sonya Keene and David Moon. Flavors: object-oriented programming on Symbolics computers. In *Common Lisp Conference,* 1985.