September 1987

Report No. STAN-CS-87- 1178
Also numbered KSL-87-44

A Dynamic, Cut-Through Communications Protocol with Multicast

by

G. T. Byrd, R. Nakano, and B. A. Delagi

Department of Computer Science

Stanford University
Stanford, CA 94305

Knowledge Systems Laboratory **August** 1987
Report No. KSL-87-44

A Dynamic, Cut-Through Communications Protocol with Multicast

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

KNOWLEDGE SYTEMS LABORATORY
Department of Computer Science
Stanford University
Stanford, CA 94305

A Dynamic, Cut-Through Communications Protocol with Multicast*

Greg Byrd[†]
Department of Electrical Engineering
Stanford University
Stanford, CA 94305

Russell Nakano[‡]
Department of Computer Science
Stanford University
Stanford, CA 94035

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

‘This work was supported by DARPA Contract F30602-85-C-0012, NASA Ames Contract NCC 2-220-S1, and Boeing Contract W266875.

‘Supported by an National Science Foundation Graduate Fellowship, with additional support provided by the Dept. of Electrical Engineering. Any opinions findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

*Author’s present address: Digital Equipment Corporation, 100 Hamilton Avenue UCO-1, Palo Alto, CA 94301.

Abstract

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

1 **Introduction** 1

2 Components 2

3 The Protocol 3
3.1 Packets 3
3.2 Packet Transmission 5
3.3 Flow Control 5
3.4 Deadlock Avoidance 7
3.4.1 Unicast Deadlocks 7
3.4.2 Multicast Deadlocks 7

4 Implementation 9
4.1 Operator 10
4.1.1 Sending a Packet 10
4.1.2 Receiving a Packet 10
4.2 **Fifo-buffer** 11
4.3 Net-Input 13
4.4 Net-Output 13
4.5 Router 14
4.5.1 Making a Connection 15
4.5.2 Multicast Transmissions 15

5 CARE Implementation 17
5.1 Operator 18
5.2 **Fifo-buffer** 19
5.3 Net-Input, Net-Output, **and** Router 21
5.4 Results. 22

6 Conclusion 22

References 22

List of Figures

1 Components of a CARE *site*. 3
2 Organization of a packet. 4
3 Network component interconnections. 6
4 Example of deadlock in a multicast. 8
5 Generic network component. 9
6 Fifo-buffer state diagram. 12
7 Simulated **fifo-buffer** output state diagram. 20

List of Tables

A Tags used by communications system. 4
B Flow-control signals. 5
C Packet terminators. 10
D Communication cycle phases. 14

1 Introduction

This is a revision of an earlier paper [1], in which we presented a high-performance point-to-point communications protocol with multicast capabilities. The protocol described here is essentially the same, but an effort has been made to describe the protocol in terms that more closely correspond to the intended hardware implementation.

The protocol described in this paper is designed to effectively 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 then 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 means that a routing decision is made on the fly, as **a** packet is received, rather than after buffering the entire packet. For example, in "virtual cut-through" routing **[3],** the packet is passed on a word at **a** time, until **a** desired channel is blocked, at which time the packet is buffered.' "Wormhole" routing **[5],** on the other hand, uses flow control signals to halt the packet flow, rather than buffering it. Cut-through routing offers reduced buffering requirements (since the packet need not be buffered at each node) and low latency. **[6,7]**

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 **[8].** 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 halted. The flow control mechanism is local, because actions are taken based on the state of the downstream component rat her 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

---

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

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 section 3. Section 4 presents a hypothetical hardware implementation of the protocol, while section 5 describes the implementation 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 communications 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 ***operat*** or 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 operator-evaluator 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.**[2] We will refer to the pair connected to the operator as the "local" net-input and net-output. 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, where a "word" is long enough to specify **a** target site.

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 **destina-**tions) 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 ***fifo-buffers*** queues packets between the operator and local net-input and net-output. The ***upstream*** fifo-buffer queues packets from the network to

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

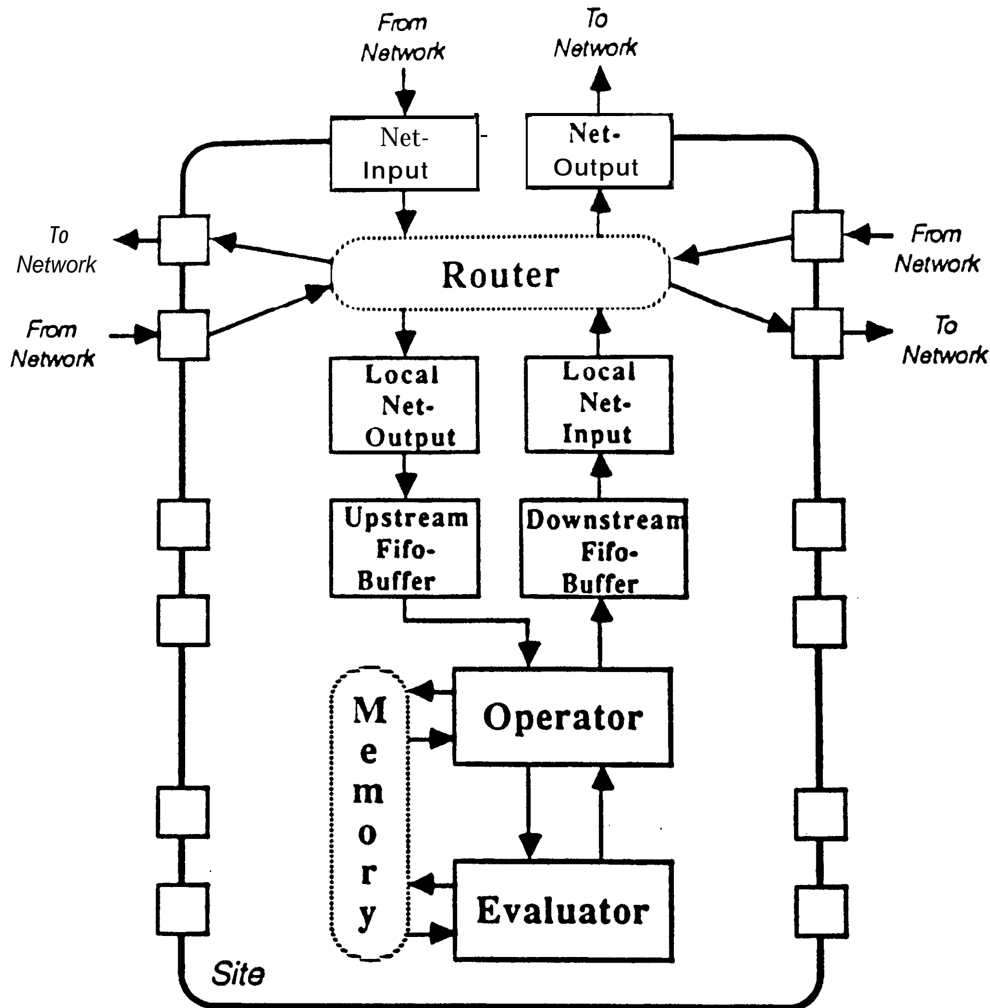


Figure 1: Components of a CARE ***site.***

the operator; the ***downstream*** **fifo-buffer** queues packets from the operator to the network.

3 The Protocol

3.1 Packets

Figure 2 shows the organization of a packet. The first part of a packet is devoted to the ***target*** *entries.* Each entry specifies a target site, as well as other information that will be used when the packet arrives at the site. Following the target entries are zero or more words of ***data*** and a one-word ***packet terminator.*** The operator determines the status of a packet by examining its **terminator.**[3]

Each word in a packet is tagged, so that target entries may be differentiated

[3] **AS described in section 4.1.**

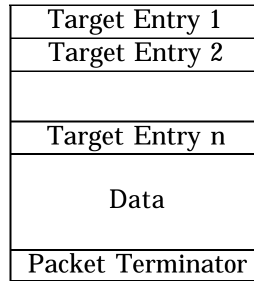


Figure 2: Organization of **a** packet.

from data. There are two types of tags used for specifying **a** target site-one which indicates that there is only one target for this packet (i.e., ***unicast*), and** one which indicates that there may be more than one (i.e., ***multicast*).** This allows the router to handle unicasts efficiently, without the extra mechanisms required for multicasts described later. There are also **a tags for the** other words in a target entry, which do not specify a site.

Also, tags are used to implement several special **characters** required for the protocol. There are two types of pod ***characters*:** one for denoting a null target entry, and one for indicating that there is no word available for transmission. Finally, there are three distinct packet **terminators-:end-of-packet, :local-end-of-packet,** and **:abort-packet.** The uses of these special characters will be further explained as the protocol is described.

Table **A** summarizes the tags needed to implement target entries and special characters.

Target Sites	**:unicast-site** **:multicast-site**
Pad Characters	**:null-target** **:null-transmission**
Terminators	**:end-of-packet** **:local-end-of-packet** **:abort-packet**

Table **A:** Tags used by communications system.

3.2 Packet Transmission

The transmission path of a packet is shown in figure 3. First, an evaluator requests a packet transmission. For the moment, assume a unicast transmission (only one target). The operator then sends the packet (through a fifo-buffer) to the local net-input. The router decides which net-output should receive the packet, based on the target site 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 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 site has been reached, or to another net-output, if not. This continues until the target site 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.**[4] The copying operation is done during transmission, one word at a time, **as** opposed to buffering the entire packet and making copies. If any branch of the multicast is blocked, the **net-input** sends **:null-transmission** characters down the other branches until valid data may be sent down all the paths. The pad characters (either **:null-target** or **:null-transmission)** 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 three distinct status signals, as

[4] **Each copy of the packet as it is transmitted will have the same number of target entry "slots," but some of them will contain null entries.**

Status	**Meaning**
'open	Available to receive **data.**
'wait	Busy or network is blocked; do not send more **data.**
'abort-request	Potential deadlock detected.=

"Only a fifo-buffer may originate the 'abort-request signal.

Table B: Flow-control signals.

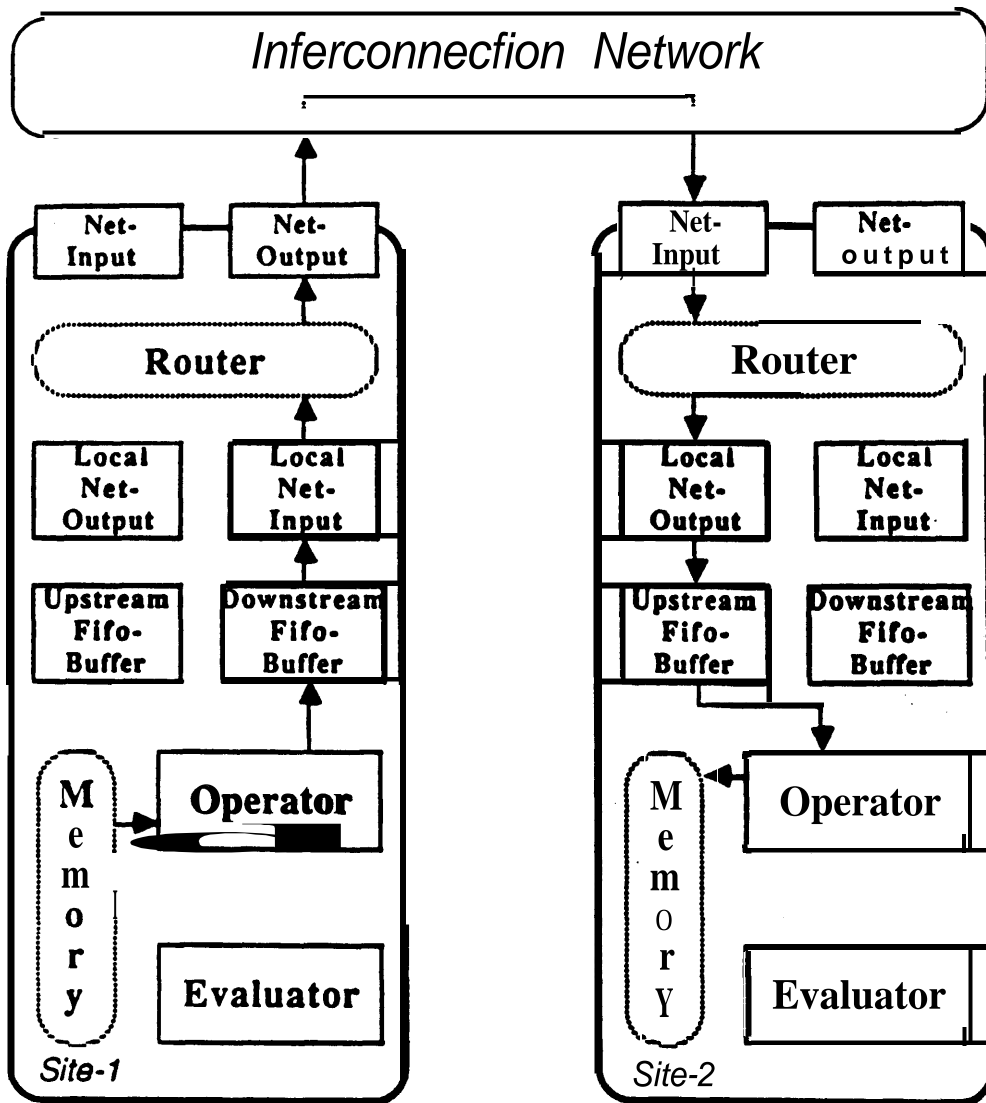


Figure 3: Network component interconnections. Packets travel in the direction marked by arrows. Flow control information flows in the opposite direction.

shown in Table B. The status signals are used to indicate to the upstream component whether data can safely be transmitted.

An 'open signal is used to indicate that the component 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 multicast deadlock condition has been detected and the transmission may be aborted.

3.4 Deadlock Avoidance

3.4.1 Unicast Deadlocks

Dally and Seitz [5] have developed a deadlock-free unicast transmission scheme for **wormhole** routing, based on virtual channels. Our strategy is different-if progress cannot be made, **a** packet may be temporarily buffered at an intermediate site. In this way, at least one of the packets responsible for a deadlock will be removed from the network, so that the other packets **may** make progress. Thus, this protocol is **a** compromise between virtual cut-through [3], in which the packet is always buffered when it is blocked, and **wormhole** routing [5], in which the packet **is *netter* buffered.**

More specifically, if the number of connection attempts for an acceptable net-output exceeds a threshold, then the local net-output is considered as **a** potential target. If the local net-output becomes available before the desired **net-**output, the packet is buffered, freeing its upstream channels. When the operator examines the packet and discovers that the packet was targetted for another site, it will retransmit the packet. Assuming packets cannot be infinitely long, either the local net-output or **an** acceptable remote net-output will eventually become free, so that deadlocks can be avoided, as long as there is sufficient space in the site at the front edge of the transmission.

3.4.2 Multicast Deadlocks

The existence of packet multicasts introduces the possibility of another type 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 transmission is not possible because the resources acquired are insufficient.

Figure 4 illustrates an example of how multicast deadlock can arise. Suppose we have two multicast transmissions, call them d 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-t.** For

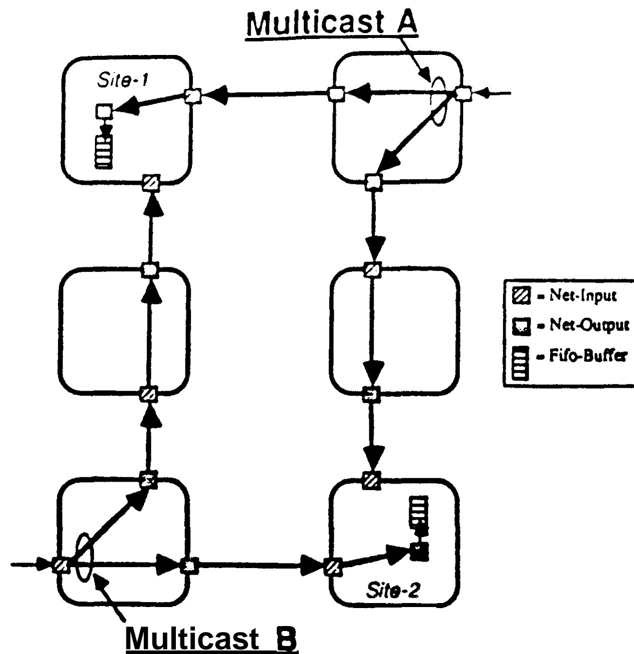


Figure 4: Example of deadlock in a multicast.

multicast *A* to continue, it needs to gain access to the local net-output of *site-2*;[5] 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
- 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.

[5]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.

A potential deadlock is detected by the fifo-buffer when the number of consecutive **:null-transmission** characters exceeds a threshold. This indicates that one or more branches of the multicast have been blocked for a long time, which implies the possibility of deadlock. When the threshold is exceeded, the **fifo-buffer** asserts an **'abort-request** signal upstream, so that the router may abort the transmission if necessary.

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.

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 Implementation

This section provides **a** detailed description of the behavior of each of the network **components** in **a** hypothetical hardware implementation. Figure 5 shows **a** "generic" network component, with its input and output ports. The **input and** output **ports** are used to pass packets and flow control information-packets **flow down&ream, flow control** signals flow upstream. The packet-in port accepts **data from upstream, and** the **packet-out port** sends data downstream; the **status-in port accepts** flow control signals from downstream, and the **status-out** port sends flow control signals upstream.

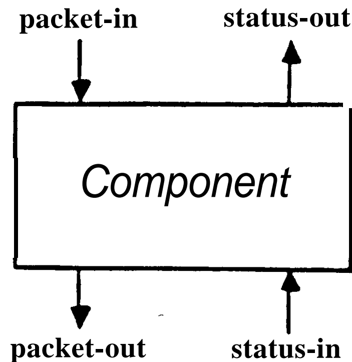


Figure 5: Generic network component.

4.1 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.

4.1.1 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 **'open,** 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.

4.1.2 Receiving a Packet

A packet arrival at the operator is signalled by the appearance of a **target** entry word on the **network-packet-in** port. The **network-status-out port is set,** to **'open,** which signals the upstream **fifo-buffer** to keep sending packet words, and the packet is stored in a temporary buffer.

The action taken by the operator when the packet is completely received depends on the type of **packet terminator.** There are three types of terminators, shown in Table C, and their interpretations are given below.

The arrival of an **:end-of-packet** signifies that the packet transmission was successful. The operator sends **'wait** to the upstream fifo-buffer (through **'network-status-out)** until the **packet** is serviced (e.g., an evaluator operation

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 C: Packet terminators.

is queued). When the operator is ready to receive the next packet, it asserts 'open.

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.[6] 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 (originated locally) 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 (i.e., no retransmission). This indicates that a multicast was successful and does not have to be retried.

The arrival of an **:abort-packet** terminator instructs the operator **to ignore** the packet. In other words, the temporary buffer holding the packet is released without servicing the packet.

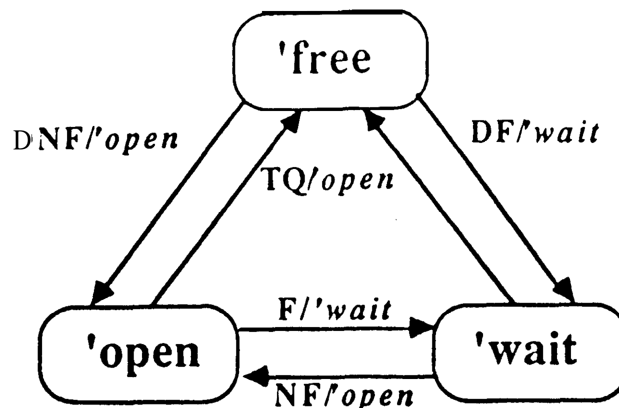
4.2 **Fifo-buffer**

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. The **fifo-buffer** can be thought of as three distinct parts: the ***input,*** the ***queue,*** and the ***output.***

The queue is **a** simple FIFO queue, with one-word input and output ports. It responds to a 'take signal from the output by placing the oldest item in the queue on the output ports. It responds to **a 'put** signal from the input by placing the incoming data at the tail of the queue. It also presents a ***queue-status*** signal to both the input and output, which can be **'empty, 'some, or 'full.** If the queue is empty, it sends **a** pad character to the output in response to **a 'take** signal.

On its output side, the upstream fifo-buffer is connected to the **operator,** while the downstream fifo-buffer is connected to the local net-input. The output interprets an **'open** signal on **status-in** by sending **'take** to the queue and sending the resulting output downstream. Nothing is removed from the queue if **status-in** is **'wait.**

[6] **If any of the targets are local, the operator keeps a copy of the packet and strips the local targets from the retransmitted packet.**



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

Figure 6: Fifo-buffer state diagram.

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 whether the terminator for the current packet has arrived, because of the multicast abort procedure needed for deadlock avoidance, so we describe the input handler as a finite state machine, whose state diagram is shown in figure 6. The labels on the arcs represent the condition which caused the transition and the status signal asserted on status-out as a result.

The fifo-buffer input begins in the 'free state. Whenever new 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 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 reentered, and 'open is sent upstream to

resume the flow of data.

When the fifo-buffer is in the **'open** state, a "time-out" may occur, which indicates that number of consecutive **:null-transmission** characters has exceeded a threshold. When this happens, it remains in the 'open state and asserts 'abort-request on the status-out port.

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

4.3 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 neighboring site).

The net-input serves as **a** one-word data buffer and relays flow control information to its upstream component. It has a two-phase operation:

1. During phase one, the status **latch is opened,** and the current value of **status-in** flows upstream. This **value** will either be **'open** or 'wait-the router will not allow an **'abort-request** signal to ever **reach** the net-input. The data latch (fed by **packet-in)** is closed during this **phase**, and the stored value is output on **packet-out.**

2. During phase two, the net-input closes the status latch and examines the latched signal. If the signal is **'open,** it opens the data latch, allowing data to flow downstream. If the signal is **'wait,** the data latch remains closed. In any case, the data latch is closed at the end of this phase.

4.4 . 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 operation of the net-output is the same as the net-input, except that the phases are reversed. The net-output conditionally latches data during phase one, and allows flow control signals to flow upstream during phase two. The only other difference is that the **'abort-request** signal may be passed upstream.

Table D summarizes the net-input and net-output operations during the two communication phases.

Component	Phase One	Phase Two
Net-Input	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.
Net-Output	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.

Table D: Communication cycle phases.

4.5 Router

The router connects the **net-inputs and** net-outputs of a site, and is responsible for:

- Determining to which net-outputs ♋ packet should be sent, based on **the packet's** target addresses, the system routing strategy, and the current availability of net-outputs;
- Creating, maintaining, and deleting the connections between ♋ net-inputs and sets of net-outputs, including transmitting data and flow control signals between them; and
- Sending appropriate pads and packet terminators, in order to implement the deadlock avoidance mechanism.

For **a** unicast transmission, the function of the router is quite simple. Upon examining the **packet** target, it selects **a** net-output (possibly the local one) to continue the transmission, based on the location of the target site relative to its own and on the availability of net-outputs. If no connection can be made, a **'wait** signal is sent to the requesting net-input until a net-output becomes available. After **a** net-output is selected, the router maintains the connection by sending **data** from the net-input to the net-output and sending flow control signals from the net-output to the net-input. When the packet transmission is completed, the net-output becomes available to accept another connection.

During **a** multicast transmission, the packet targets are read one at a time, and the connections to net-outputs are made as the targets are read. For each

net-input the router keeps track of the type of its current connection. There are three possible connecton types:

'unicast The packet is being transmitted to only one target, either because there was only a single target in the packet, or because the packet is being "passed through" because the local net-output was not available.

'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.

In the next two sections, we present further details about how connections are made and how multicasts are handled.

4.5.1 Making a Connection

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.

The protocol does not prescribe a particular routing policy for the network. Instead, information about possible connections is "hard-wired" into the router in the form of a priority network. Conceptually, we model the priority network as a preference fable-for every logical direction, we provide a prioritized list of net-outputs that may be considered. Examples of routing strategies which may be implemented in this way are (1) try all net-outputs, starting with the closest to the target, (2) try only one net-output (static routing), and so forth.

Given a direction, the router checks the status of each net-output in the preference table, in turn, until an available net-output is found. If none is available, then the connection fails, and **'wait** is sent upstream to the net-input.

4.5.2 Multicast Transmissions

When a multicast packet arrives, the router makes a connection for each packet target, one at a time. If the connection for a target has already been made (in response to an earlier target), the target entry is merely transmitted downstream

to that net-output. Whenever a target entry is transmitted, **:null-target** characters are sent down all of the other connections. In this way, the target list is partitioned along several paths. When the packet data is received by the router, it is transmitted to all the connected net-outputs. If any of the downstream paths becomes blocked, **:null-transmission** characters are transmitted down all the other paths.

There is an additional complication for the router, however, since the local net-output must be sent a copy of the packet to be buffered, in case the transmission is aborted and must be retried. Because of the special **:unicast-site** tag, the router knows immediately whether a packet should be treated as a multicast or unicast. Note, however, that since the router only looks at one address at a time, the router cannot determine when the ***lust*** target occurs for **a** particular branch of the multicast. Thus downstream routers may mistakenly interpret a packet with only one target as **a** multicast. As **a** result, unnecessary local copies of this packet will be **made as it makes** its way to its target **site.**[7]

When the first target of a multicast is received, the router tries to connect to the **local** net-output, as well as the net-output specified by the preference table. If the local net-output is not available, then the packet is not split at this site. Instead, the entire packet is sent down the remote connection. In this way, the packet will either sequentially visit each target, on the list or will finally reach a site where it may be split.

If at any time during the connection process, **a** desired net-output is not available, a 'wait is sent upstream to the net-input to halt the flow of additional targets. While waiting for a net-output to become free, the router must send target pad characters down the established connections. Unlike in the unicast case, we cannot decide to divert this target to the local net-output, since then there would be no way to tell which targets were actually serviced and which were diverted. Therefore, to avoid the possibility of deadlock during target processing, the local net-output must be sent ***data*** pad characters, so that the downstream fifo-buffer can time out, if appropriate, and the multicast can be aborted.

If the transmission completes successfully (i.e., is not aborted), the received packet terminator is passed on to all the remote (non-local) net-outputs, but the local net-output may be sent a modified terminator, as follows. If the received terminator is **:abort-packet,** it is sent as is, instructing the local operator to ignore the packet. If the received terminator is **:end-of-packet,** the terminator sent to the local net-output depends on the connection type:

'all-remote: An **:abort-packet** is sent, since the packet should not be **retransmitted,** and may be ignored.

[7]The router **could be optimized to notice when an *all-remote connection only uses a single connection—an :abort-packet could then be sent to the local fifo-buffer, since there is no possibility of deadlock and thus no retransmission will be necessary.**

'some-local: A :local-end-of-packet is sent, instructing the operator to accept the packet for the local targets, but to ignore the remote targets (i.e., do not retransmit).

If, during the multicast transmission, the router receives an 'abort-request signal from the local net-output (generated by the downstream fifo-buffer), the router aborts all the remote connections for the connected net-input by forcing the net-outputs to latch an :abort-packet terminator. An **'open** signal is passed upstream to the net-input, and the transmission proceeds as if it were a unicast transmission destined for the local operator. When the packet terminator is received, it is passed directly to the local net-output. Note that an **:end-of-packet** will cause the packet to be retransmitted by the operator,[8] since there are non-local targets, and an **:abort-packet** will cause the packet to be discarded.

5 CARE Implement at ion

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 instrumentation built on top of an event-driven simulator [9], 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 Zetalisp [10]. 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 ***insfance variables,*** used to hold state information, and a set of ***methods,*** procedures used by the object to respond to messages from other objects.[9] The instance variables loosely correspond to the ports and state variables used to describe the protocol in section 3. 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.

[8] If there are local targets, a copy of the packet will be kept and the local targets will be removed from the target list upon retransmission.

'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.

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,[10] 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. The transmission time of a packet is the time between arrival of its front edge and its terminator. In this way, 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, the operator delays an appropriate time between sending a packet **and** sending its terminator. When the transmission time of the packet has elapsed, the terminator is sent as soon as an **'open** signal is received from the **fifo**-buffer. This is a simplified model, since there can be arbitrary delays involved in freeing up space in **a** full buffer, but the fifo-buffer output module ensures that the proper space is inserted between packet and terminator in the network.

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.

[10]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 communications system is simulated in terms of register transfers, while the execution of (uniprocessor) application code by the evduator 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 **"red" machine.**

5.2 Fifo-buffer

In the simulator, the amount of storage in the fifo-buffer may be set at run time.[11] Each packet or packet terminator takes up one space in the buffer, no matter what its actual size.

Since we do not simulate each word of a packet transmission, the fifo-buffer cannot count pad characters to detect a potential multicast deadlock. Instead, the simulated fifo-buffer uses a time-out procedure: when the packet is received, the fifo-buffer schedules a wake-up event at random time in the future, based on the packet size (for example, between 1.5 and 3 times the packet transit time). If the packet terminator has not arrived by that time, the fifo-buffer asserts **'abort-request.** This is not a viable option for actual implementation, since a real packet header contains no information about the packet size.

On its output side, 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** 7. 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.

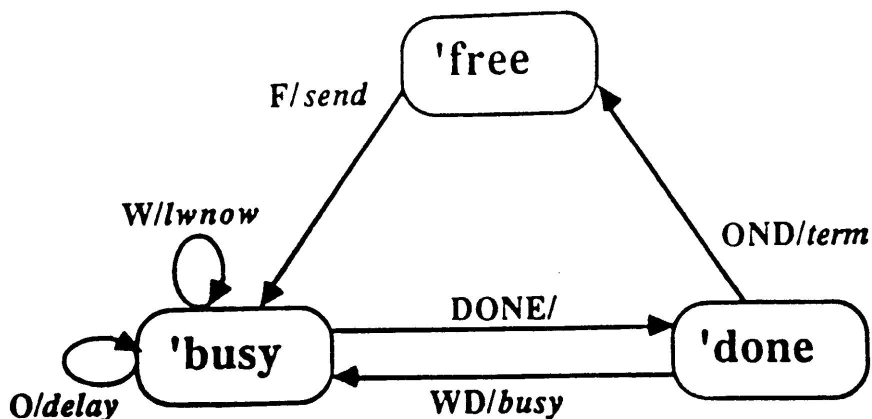
Initially, **transmission-status** is 'free. If the downstream component **re**quests data **(status-in goes** to **'open)** and the queue is not empty, the top of the queue, which must be a packet, is placed on the **packet-out** via, **delay** is set to zero, and **transmission-status goes** to **'busy.**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

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

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



Condition	Meaning
F	'Fret 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.
detay	Delay = delay + (now - last-wait); Last-wait = nil.
busy	Schedule 'done for now + delay; Last-wait = nil.
term	Send terminator.

Figure 7: Simulated **fifo-buffer** output state diagram.

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. Since they merely pass packets and terminators from one point to the next,[12] the flow control signals ensure that they will maintain the proper separation between a packet and its terminator.

5.3 Net-Input, Net-Output, and Router

As mentioned earlier, the router is not an explicit object in the simulation. Instead, the ***site*** object performs its operations. Net-inputs and net-outputs communicate with it by passing messages (in the Flavors **sense) rather than** making assertions on vias. Likewise, the site **updates** net-input and net-output "ports" by setting instance variables.

To connect to net-outputs, the net-input sends a **:connect** message to the site, which then attempts to make the appropriate connections. The result is stored in the **connection** instance variable of the net-input. If no connection could be made, **'seek** is returned; otherwise, the type of connection (unicast, **all**-remote, or some-local) is returned. If only some of the desired connections could be made, the unsuccessful targets are placed in the **pending-connections** instance variable. The net-input keeps sending **:connect** messages to the site until all the targets are satisfied.

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 net-outputs.)

Similarly, the net-output uses the **:wait, :open,** and **:abort-request** methods to relay flow control signals to the site, which then makes the appropriate assertions to the connected net-input.

In the router, 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

[12] **This is in contrast to the fifo-buffer, which must *insert* th, packet and terminator into the network at the proper time.**

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.

5.4 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] Gregory T. Byrd, Russell Nakano, and Bruce A. Delagi. ***A Poinf-fo-poinf Multicast Communications Protocol.*** Technical Report KSL-87-02, Knowledge Systems Laboratory, Stanford University, January 1987.

[2] V. Ahuja. Design ***and Analysis of CompuferCommunication 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,* C-34(2):174-180, February 1985.

[5] William J. Dally and Charles L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computem, C-36(5):547-553,* May 1987.

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

[7] William J. Dally. Wire-efficient VLSI multiprocessor communication networks. In Paul Losleben, editor, ***Advanced Research*** in VLSI--Proceedings ***of the 1987 Stanford Conference,*** pages 391-415, MIT Press, 1987.

[8] Richard W. Watson. Distributed system architecture model. In B. W. Lampson, M. Paul, and H. J. Siegert, editors, ***Distribufed Systems—Architecture and Implementafion,*** chapter 2, pages 10–43, Springer-Verlag, 1981.

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

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