# Reliable Broadcast Algorithms for HARTS

DILIP D. KANDLUR and KANG G. SHIN
The University of Michigan

The problem of broadcasting in point-to-point interconnection networks with virtual cut-through switching is considered. A simple extension of virtual cut-through is proposed, which provides good support for broadcasting in mesh-connected multicomputers. An implementation of this extension (termed a broadcast primitive) for a hexagonal mesh multicomputer called HARTS, that uses virtual cut-through switching, is also presented. Based on this primitive, a set of broadcast algorithms is developed for the hexagonal mesh topology. These algorithms deliver multiple copies of a message from a source node to every other node in the hexagonal mesh through disjoint paths. They can be used for broadcasting in the presence of faulty nodes/links, even when the identity of the faulty components is not known. The performance of these algorithms has been analyzed and compared with the performance of other possible broadcast algorithms.

Categories and Subject Descriptors: B.4.3 [**Input/Output and Data Communication**]: Interconnections—*topology*: C 2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network topology, circuit-switching networks*; C3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*HARTS*

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Broadcasting, fault-tolerant networks, virtual cut-through

## 1. INTRODUCTION

The availability of inexpensive and powerful microprocessors has fueled the design and development of multiprocessors and multicomputers with a large number of processors. Several topologies have been proposed for interconnecting these processors, including trees, hypercubes, and meshes. The hexagonal

The work reported here is supported in part by the Office of Naval Research under contracts N00014-85-K-0122 and N00014-85-K-0531, by NASA under grant NAG-1-296, and an IBM Graduate Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this report are those of the authors and do not necessarily reflect the views of the funding agencies.

Authors' addresses: K. G. Shin, Real-Time Computing Laboratory, Division of Computer Science and Engineering, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122 email: kgshin@dip.eecs.umich edu. D. D Kandlur, IBM T. J. Watson Research Center, Hawthorne, NY 10532 email: kandlur@watson ibm com

mesh topology [3, 7, 18] is one such topology, which offers some interesting properties. Meshes and other topologies that have a fixed, small-node degree have the advantage that it is easy to implement routing schemes such as virtual cut-through [13]. The Torus routing chip [5, 6] and the HARTS routing controller [8] are examples of such implementations. A small-node degree makes implementation easier because it is possible to employ techniques like time-division multiplexing in the switch. In the virtual cut-through-switching method, messages arriving at an intermediate node are forwarded to the next node in the route without buffering if a circuit can be established to the next node. Since messages do not necessarily get buffered at intermediate nodes, the delays encountered are smaller than those for packet switching.

This paper addresses the problem of broadcasting in mesh-connected multicomputer systems which use virtual cut-through switching. Although this operation is very simple for *broadcast* networks like the Ethernet and the Token Ring, where a message transmitted can be "seen" by every other node in the network, it is more involved for a point-to-point interconnection network. For this type of network, a simple nonredundant broadcast algorithm, which delivers a single copy of a message to every node, essentially constructs a spanning tree for the network graph rooted at the source node. It is desirable to minimize the height, and hence the number of store-and-forward communication steps, for the spanning tree. We present a simple primitive to support broadcasting efficiently in this type of network. This primitive is based on the virtual cut-through switching scheme and significantly reduces the number of required store-and-forward communication steps. We also present an implementation technique for this primitive for the HARTS routing controller.

Based on this primitive, we then develop broadcasting algorithms for meshes which are resilient to node/link faults. Motivation for this work is provided by its applicability in implementing algorithms for problems like clock synchronization and distributed agreement in the presence of faults [14, 15]. In these problems, it is necessary to ensure that a nonfaulty node can correctly deliver its private value to all other nonfaulty nodes in the system. This problem is difficult because (intermediate) faulty nodes can discard, corrupt, and possibly alter the information passing through them. Although on-line distributed diagnosis schemes are available for identifying faults, these schemes do not give 100 percent fault coverage unless the testing for the diagnosis goes on for a very long time. Therefore, it is highly desirable that these broadcast algorithms should work even when the identity of all the faulty processors is not known. This is accomplished by delivering multiple copies of the message through disjoint paths to every node in the system. The receiving nodes can then identify the original message from the multiple copies using a scheme which is appropriate for the fault model used, like majority voting.

Although the broadcast primitive can be used to develop similar algorithms for rectangular meshes, this paper presents algorithms for the wrapped hexagonal mesh topology, since these are more complex than those for a

rectangular mesh. The hexagonal mesh network, which is pivotal in the discussion that follows, is a regular, homogeneous graph in which each node has six neighbors. It can be defined succinctly as follows.

> A C-wrapped hexagonal mesh of size $n$ is comprised of $N = 3n(n-1) + 1$ nodes, labeled from 0 to $N-1$, such that each node $s$ has six neighbors $[s+1]_N$, $[s+3n-1]_N$, $[s+3n-2]_N$, $[s+3n(n-1)]_N$, $[s+3n^2-6n+2]_N$, and $[s+3n^2-6n+3]_N$, where $[a]_b$ denotes $a \bmod b$.

The graph can be visualized as a simple hexagonal mesh with wrap links added to the nodes on the periphery. A simple hexagonal mesh looks like a set of concentric hexagons with a central node, where each hexagon has one more node on each edge than the one immediately inside of it. The size of the hexagonal mesh is the number of nodes on any side of the hexagon. Figure 1a shows a simple hexagonal mesh of size 3, while Figure 1b illustrates the wrapping scheme for the nodes. The *diameter* of a C-wrapped hexagonal mesh of size $n$, which is the maximum distance between any two nodes in the mesh, is $n-1$. An analysis of some of the topological properties of this network,and its comparison with other topologies, can be found presented by Chen et al. [3]. The hexagonal mesh offers better connectivity, and thus better fault-tolerance, than a rectangular mesh. Compared to the hypercube topology, it has better scalability and the advantage of fixed-node degree. Also, for small systems (less than 100 nodes), it has better connectivity than a hypercube. A multicomputer with a C-wrapped hexagonal mesh topology, called HARTS, is currently being built at the Real-Time Computing Laboratory, The University of Michigan. Mayfly [7] is another system which uses the hexagonal mesh topology. The current version of HARTS has nineteen nodes, to be configured as a hexagonal mesh of size 3. The processors will be connected to the network through a custom-designed hardware component called the Network Processor, which is under development. The Network Processor uses the HARTS routing controller [8], which implements the virtual cut-through-switching scheme, as the front-end interface to the interconnection network.

To the best of our knowledge, this is the first reported work dealing with reliable broadcasting in point-to-point interconnection networks with virtual cut-through switching. In other related work, Chou and Gopal have recently presented some algorithms for linear broadcast routing [4]. The linear broadcast technique is similar in principle to the broadcasting primitive presented here. These authors, however, concentrate on the problem of finding optimal *simple* broadcast algorithms for general network topologies, and they have shown that the general form of this problem is NP-complete. A multiple-copy reliable broadcast algorithm for the hypercube topology is presented by Ramanathan and Shin [17]. Algorithms for total exchange and optimal broadcasting, again in hypercube multicomputers, can be found presented by Fraigniaud [10] and Johnson and Ho [12]. We presented [3] a point-to-point broadcast algorithm for the hexagonal mesh, which required $n+2$ communication steps in a mesh of size $n$. That algorithm, which is based on

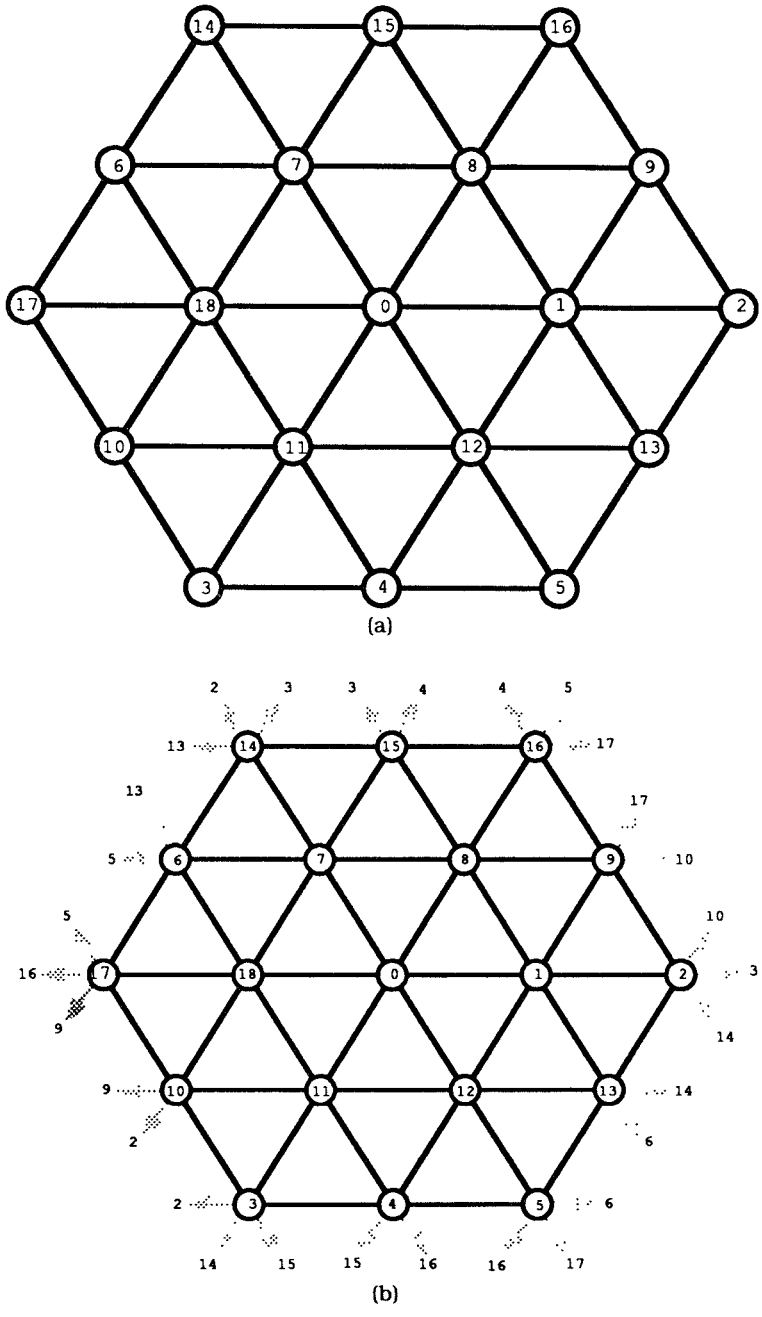


Fig. 1. A hexagonal mesh of size 3 (E-3).

traditional store-and-forward switching, does not consider possible hardware support for virtual cut-through switching, and it does not handle multiple-copy broadcasts. Protocols for reliable broadcasting, mainly for broadcast networks, can also be found in the literature [1, 2]. These protocols try to provide

a consistent delivery ordering among broadcast messages, but they do not consider the tolerance of malicious failures.

This paper is organized as follows. Section 2 describes the proposed broadcast primitive and its implementation for HARTS. In Section 3 we develop an algorithm for simple broadcasting based on this primitive. Broadcast algorithms, which deliver multiple copies of the message through node-disjoint paths to each node in the hexagonal mesh, are presented in Section 4. An analysis of these algorithms and their comparison with other broadcast algorithms is presented in Section 5. The paper concludes with Section 6.

## 2. THE BROADCAST PRIMITIVE

When we consider multicomputer systems with virtual cut-through switching, packet routing is typically handled by a front-end controller at each node. The normal operation of the controller is to compare the packet destination with the node address, and if they match, the packet is delivered to the processor. Otherwise, it is forwarded to the next node in the route. In many such systems dynamic routing is employed, in which case the controller also has to choose the next node on the route. For example, in HARTS, the possible routes of a message to the destination are described by three routing tags (which take positive and negative values) corresponding to the distances to be traversed in the six directions in the hexagonal mesh. The routing controller examines the tags in turn to check whether there are any nonzero values, and if so, whether the corresponding outgoing link is available. As the message is being routed toward the destination, its routing tags are also updated to reflect the new distance to the destination.

One of the principal advantages of this scheme is that the node processor does not have to examine and process all the packets going through the node. However, this advantage would be lost when broadcast messages are to be delivered using a simple store-and-forward broadcasting scheme. In addition to the larger delays caused by buffering, these messages could result in a substantial load on the processors. To facilitate efficient broadcasting, it is therefore necessary to support the operation at the link level. We propose to use the RELAY primitive, shown below in the form of a procedure, to accomplish this. This procedure shows the actions to be taken by the link controller when a packet arrives. It is assumed that the packet header contains the information required for handling broadcast messages like *type*, *distance*, *step*, and *tag*. The *type* field distinguishes a BROADCAST packet from an ordinary packet, while the *distance* gives the number of nodes to be traversed in a particular direction. The *step* and *tag* fields are used by the broadcast algorithms described later in this paper.

In the RELAY procedure, *deliver* corresponds to the delivery of the packet by the link controller to the processor. The procedure also shows that the link controller is responsible for updating the distance field in the packet header before delivering or relaying the packet. The packet is relayed to the next node in the same direction in which it arrived, i.e., on the link opposite to the

input one, using *send_on_link*.

```
procedure RELAY
begin
  receive_from_link(packet, from_direction)
  if (packet.type = BROADCAST)
    packet.distance := packet.distance − 1
    deliver(packet)
    if (packet.distance ≠ 0)
      send_on_link(direction = from_direction, packet)
    end
  else
    normal packet handling
  end
end
```

There are several reasons for choosing this primitive. First, it blends in easily with the existing dynamic routing algorithms. Second, the *deliver* and the *send_on_link* steps can be accomplished concurrently using a "tee" operation. Third, the operation is simple enough to be implemented at little additional cost in the link controller. Furthermore, we will show that this primitive can be used very effectively to develop broadcast algorithms for mesh-connected multicomputers.

The implementation of the "tee" operation can be described in more detail in the context of the HARTS routing controller [8]. The controller contains six receivers and six transmitters, corresponding to the incoming and outgoing links, connected to a single bus. This bus, called the *time-sliced bus*, also has interfaces to the packet buffer management unit in the node to accept and deliver packets. The bus is time-slotted and each receiver is thus guaranteed an access slot, which it uses to place the data that it receives on the bus. Most of the intelligence in the routing controller resides in the receivers. When a packet is received, the receiver examines the routing tags in the packet header to check whether the packet has reached its destination. If not, it checks the directions in which a packet can be forwarded and tries to reserve a transmitter in one of these directions. Note that when shortest-path routing [3] is used, the routing tags are such that at most two of the three routing tags are nonzero. In this case, the packet can be forwarded in at most two of the six directions. If the reservation succeeds, the transmitter accepts any data that is placed on the time-sliced bus by the receiver and transmits it. If the reservation attempts do not succeed, the receiver asserts a control line to request the buffer management unit to store the packet for later transmission.

To implement the RELAY primitive, the receiver operation can be modified to recognize packets of type BROADCAST. For this packet type, in addition to attempting a reservation for the transmitter in the same direction, it also asserts the control line to store the packet. Therefore, when the receiver places packet data on the bus, it can be forwarded to the next node (*send_on_link*) and dropped to the node (*deliver*) simultaneously. If the

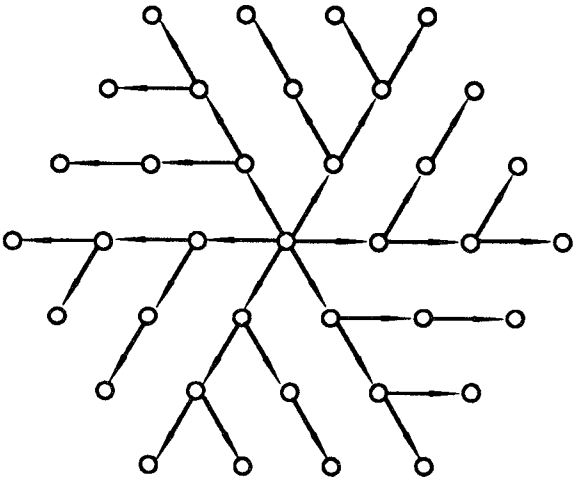

Fig. 2. Simple broadcast for an E-4 mesh (SBCAST).

reservation does not succeed, the packet is dropped to the buffer management unit as usual. In practice, only one packet is delivered to the buffer management unit even if the packet cannot cut through to the next node. The packet header is marked appropriately to inform the network processor about the status of the forward transmission. The HARTS routing controller is microprogrammable and these modifications have been implemented by changing the microprograms, without any change to the controller hardware.

## 3. SIMPLE BROADCASTING

The algorithm for simple broadcasting is shown below in the procedures BCAST_INIT and SBCAST_RELAY. An example of its operation is given in Figure 2 for a hexagonal mesh of size 4 (denoted by E-4). In this algorithm, and in the algorithms described later in this section, the size of the hexagonal mesh is $n$, and the directions referred to are labeled in a counterclockwise sense, as illustrated in Figure 3a. With reference to the definition of the C-wrapped hexagonal mesh, direction 0 corresponds to the link from a node $s$ to the node $[s + 1]_{3n^2-3n+1}$. The term *principal axis* is used frequently in the explanation of the algorithms. This refers to an imaginary line connecting the center of the hexagon to one of the six corners. Since the C-wrapped hexagonal mesh is a homogeneous structure, any node can be considered to be at the center of the mesh, and the algorithms can be described by placing the broadcasting node at the center. It is also useful to define directions relative to the direction in which the packet arrived into a node, as in Figure 3b. Hence, *left* corresponds to the absolute direction (in + 1)mod 6, *right* corresponds to direction (in − 1)mod 6, and so on.

The procedure BCAST_INIT is executed by the node which initiates the broadcast, and is common to all broadcasting algorithms. This node plays no further part in the broadcast process. In BCAST_INIT, the distance is set to

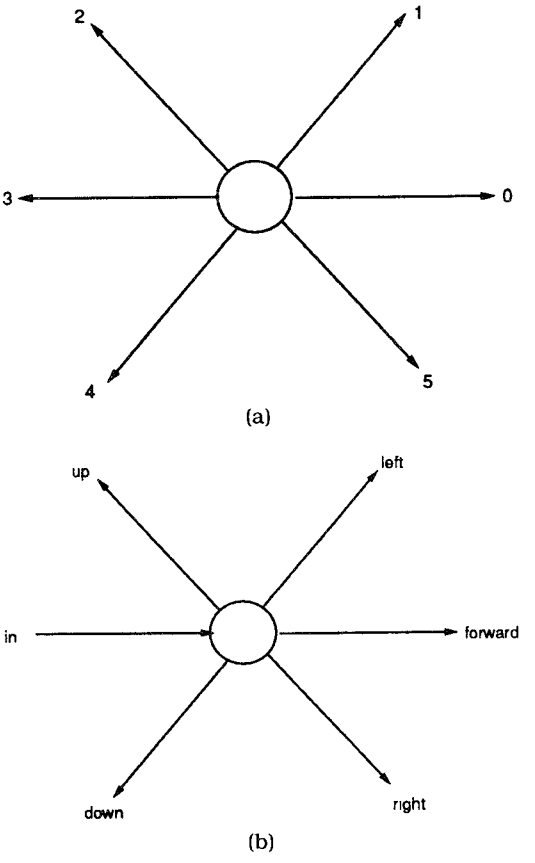


Fig. 3. Direction labeling.

$n - 1$ because this is the diameter of a hexagonal mesh of size $n$. The *send_packet* function, which is also used in other algorithms, is a nonblocking *send* and only initiates the transmission of the packet. Actual packet transmission can proceed in parallel on the six outgoing links after the initiation. The other procedure, SBCAST_RELAY, is not specific to a particular node and describes the overall operation for the system. It is activated whenever a broadcast packet is received at a node. The actions taken by a node are driven by the information that it receives in the broadcast packet. It is noted that the step number, *step*, and the algorithm type are a part of the state information that is contained in the broadcast packet. Moreover, the information about the direction from which a particular packet was received (denoted as *from_direction*), is available to the receiving node. This is indicated by the *receive* operation in SBCAST_RELAY. Based on this information, the processor at an intermediate node can determine the next step in the broadcast as per the SBCAST_RELAY algorithm. The algorithm terminates after step 2 because packets with a step value of 2 do not branch any further. Note that *send_packet* and *receive* are processor-level operations,

distinct from the link-level operations shown in the RELAY primitive.

```
procedure BCAST_INIT
begin
  packet.type := BROADCAST
  packet.step := 1
  send_packet(packet, direction = 0, distance = n − 1)
  send_packet(packet, direction = 1, distance = n − 1)
  send_packet(packet, direction = 2, distance = n − 1)
  send_packet(packet, direction = 3, distance = n − 1)
  send_packet(packet, direction = 4, distance = n − 1)
  send_packet(packet, direction = 5, distance = n − 1)
end

procedure SBCAST_RELAY
begin
  receive(packet, from_direction)
  if (packet.step = 1)
    packet.step := 2
    if (packet.distance ≠ 0)
      direction := (from_direction + 1)mod 6
      send_packet(packet, direction, packet.distance)
    end
  end
end
```

The correctness of this algorithm can be explained based on Figure 2, which shows the paths taken by a broadcast packet. The broadcast packet is delivered to all nodes on the six principal axes by the BCAST_INIT operation. The "distance" field in the broadcast packet header is decremented, as shown in the RELAY primitive, at each intermediate node and at the receiving node. Hence, a node on the principal axis which is $m$-hops away from the source node sees a value of $(n - 1 - m)$ in the *packet.distance* field. In SBCAST_RELAY, this value is used in the forwarding, so the forwarded packet travels a total distance of $m + (n - 1 - m) = n - 1$ from the source node. Since the nodes on the periphery of the hexagonal mesh are $n - 1$ hops from the center, the forwarded packet will reach the peripheral node.

## 4. MULTIPLE COPY BROADCASTS

While simple broadcasting is sufficient for many applications, it is susceptible to message loss, possibly due to data corruption and/or link and node failures. There are several applications which require more resilient broadcast mechanisms, like the clock synchronization algorithm described by Ramanathan et al. [16]. For this type of application, we have developed a family of efficient and elegant algorithms, called *k-reliable* broadcasts, to deliver $k$ copies of a message to each node using node-disjoint paths. The algorithms can also be used to guard against message loss in applications which require reliable message delivery, in place of the conventional acknowledgment-retry mechanism.

In the clock synchronization algorithm, for example, to tolerate $m$ arbitrary (Byzantine) faults, it is necessary for a node to transmit $2m + 1$ copies of its local clock to every other node in the system through disjoint paths.

From the values received, a node can determine the value that was sent by the originator using the technique described by Yang and Masson [19]. Therefore, using a 5-reliable broadcast, it is possible to achieve clock synchronization in a hexagonal mesh in the presence of up to two Byzantine faults. In this application, and in other applications of reliable broadcasting, there are two aspects to a reliable broadcast: the *delivery* mechanism and the *reception* mechanism.[1] The delivery mechanism consists of algorithms that deliver multiple copies of a message to all other nodes, through disjoint paths. It is noted that in the presence of faults, some of these copies may be corrupted or lost. The reception mechanism involves algorithms which interpret and assimilate information from the different copies received at a node. These are strongly dependent on the fault model used. Different reception mechanisms are discussed by Ramanathan and Shin [17]. These mechanisms are not dependent on the hypercube topology, so they can also be used for the hexagonal mesh.

This section presents the message delivery algorithms, starting from the two-copy algorithm and progressing toward the six-copy version. The delivery algorithms have a common broadcast initiation procedure, BCAST_INIT, which was described in Section 3.

### 4.1 2-Reliable Broadcast (2-BCAST)

The algorithm 2-BCAST shown below delivers two copies of the message to each node. Its operation is illustrated in Figure 4 for an E-4 mesh, and for clarity, only the actions of nodes on two of the principal axes are shown. Also, the links that wrap around are shaded and labeled. This algorithm is similar to SBCAST_RELAY, except that the message is forwarded in two directions. Using the explanation presented earlier for the simple broadcast algorithm, it is observed that nodes which are not on the principal axes get two copies of the message, as shown in the figure. Also, nodes on the extremities of the principal axes ( *packet.distance* = 0) use the wrap links to send the message to nodes on another axis. For example, in the figure, the wrap links (a) and (b) are used to deliver messages to two other principal axes. This ensures that nodes on the principal axes also get two copies of the message, and through disjoint paths.

```
procedure 2-BCAST
begin
  receive(packet, from_direction)
  if (packet.step = 1)
    packet.step := 2
    if (packet.distance ≠ 0)
      send_packet(packet, direction = (from_direction + 1)mod 6, packet.distance)
      send_packet(packet, direction = (from_direction − 1)mod 6, packet.distance)
```

[1]The terminology used here is taken from [11].

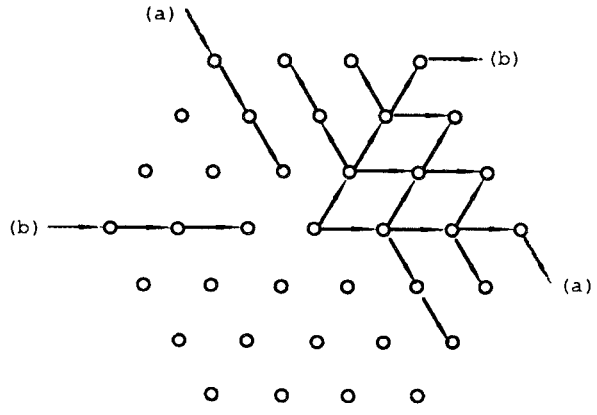


Fig. 4. 2-BCAST for an E-4 mesh.

```
    else
      send_packet(packet, direction = (from_direction − 1)mod 6, distance = n − 1)
    end
  end
end
```

### 4.2 3-Reliable Broadcast (3-BCAST)

The 3-reliable broadcast algorithm can be explained through a transformation of the 2-BCAST algorithm. In going from 2-BCAST to 3-BCAST, it is necessary to deliver one more copy of the packet to each node. This is accomplished by arranging for the delivery of the third copy using the wrap links from nodes that are diametrically opposite with respect to the broadcasting node. There are two modifications required to the 2-BCAST algorithm, intermediate nodes on the principal axes now transmit the packet *left* for $n - 1$ hops to reach nodes in the opposite sextant. Also, the extreme nodes on the axes transmit the packet $n - 1$ hops to the *left*. The results of these modifications can be seen in Figure 5, where the labels indicate nodes connected by a wrap link.

```
procedure 3-BCAST
0.  begin
1.    receive(packet, from_direction)
2.    if (packet.step = 1)
3.      packet.step := 2
4.      if (packet.distance ≠ 0)
5.        send_packet(packet, direction = (from_direction + 1)mod 6,
            distance = n − 1)
6.        send_packet(packet, direction = (from_direction − 1)mod 6,
            packet.distance)
7.      else
8.        send_packet(packet, direction = (from_direction + 1)mod 6,
            distance = n − 1)
```

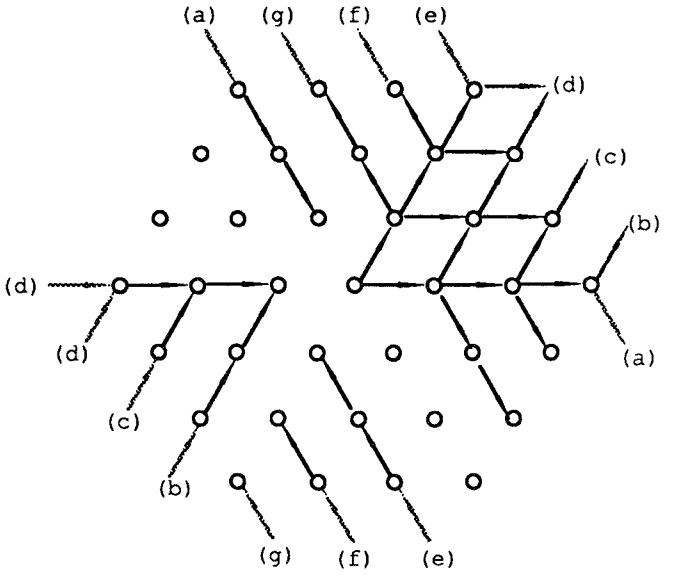


Fig. 5. 3-BCAST for an E-4 mesh.

```
9.          send_packet(packet, direction = (from_direction − 1)mod 6,
              distance = n − 1)
10.       end
11.     end
12. end
```

### 4.3 6-Reliable Broadcast (6-BCAST)

We choose to describe 6-BCAST before 4-BCAST and 5-BCAST because the latter two can be treated as restricted forms of 6-BCAST. This algorithm, which is presented in procedure 6-BCAST, creates the broadcast tree shown in Figure 6. This figure shows the packets generated by 6-BCAST from a single principal axis. As compared to 3-BCAST, this algorithm is more complicated because it is necessary to use an additional forwarding step at some of the nodes. We employ a *tag* field in the broadcast packet header to ensure that only the relevant nodes will execute the additional step. This tag field takes different values, and it is interpreted by the receiving node to forward the packet in the appropriate direction (lines 26–38 of 6-BCAST). Nodes on the extremities of the principal axes use tags 'A' and 'B' to forward the packet to two sextants. This is shown in the first part of Figure 6, where the labels mark nodes that are connected by wrap links. The immediate neighbors of the broadcast source node, with *packet.distance* of $n - 2$, use tags 'C' and 'D' to reach nodes on the adjacent principal axes. In the second part of Figure 6, the graph is redrawn by repositioning the nodes reached by wrap links. The initiating node is now at the bottom left corner of the hexagon. The figure demonstrates that the broadcast tree generated is quite regular, which is not immediately apparent from procedure 6-BCAST.

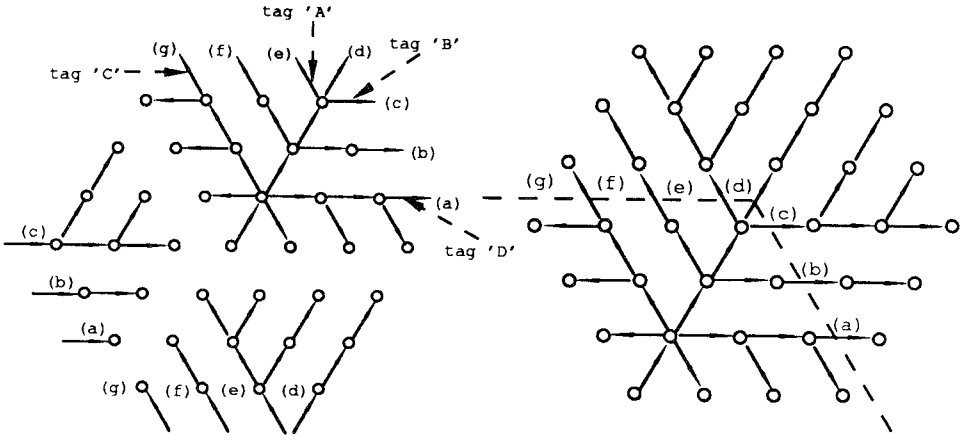


Fig. 6. Packets generated in a 6-BCAST (from one direction).

It can be seen that the packets reach all the nodes in the hexagonal mesh (except the initiator node). Similarly, the nodes also receive packets originating from each of the other five principal axes. Hence, each node receives six copies of the packet. However, it is not obvious from the figure that these six copies would be received through *node disjoint* paths. We will show later that this is indeed the case. Given that the node degree of all nodes of the hexagonal mesh is six, this is the maximum number of disjoint paths possible. *This algorithm shows that the hexagonal mesh is 6 connected in terms of node connectivity.*

```
procedure 6-BCAST
0.  begin
1.    receive(packet, from_direction)
2.    if (packet.step = 1)
3.      packet.step := 2
4.      if (packet.distance = 0)
5.        packet.tag := 'A'        /* tag = NONE for 4-BCAST and 5-BCAST */
6.        send_packet(packet, direction = (from_direction + 1)mod 6,
            distance = n - 1)
7.        packet.tag := 'B'        /* this send is excluded for 4-BCAST */
8.        send_packet(packet, direction = (from_direction - 1)mod 6,
            distance = n - 1)
9.        packet.tag := NONE  /* this send is excluded for 4-BCAST and
                                 5-BCAST */
10.       send_packet(packet, direction = from_direction, distance = n - 1)
11.     else if (packet.distance = n - 2)
12.       packet.tag := 'C'
13.       send_packet(packet, direction = (from_direction + 1)mod 6,
            distance = n - 1)
14.       packet.tag := 'D'
15.       send_packet(packet, direction = (from_direction - 1)mod 6,
            distance = n - 1)
16.       packet.tag := NONE
17.       send_packet(packet, direction = (from_direction + 2)mod 6,
            distance = 1)
```

```
18.        send_packet(packet, direction = (from_direction − 2)mod 6,
             distance = 1)
19.      else
20.        packet.tag := NONE
21.        send_packet(packet, direction = (from_direction + 1)mod 6,
             distance = n − 1)
22.        send_packet(packet, direction = (from_direction − 1)mod 6,
             distance = n − 1)
23.      end
24.    else if (packet.step = 2) AND (packet.distance ≠ 0)
25.      packet.step = 3
26.      case (packet.tag) of
27.        'A':
28.          send_packet(packet, direction = (from_direction − 1)mod 6,
               packet.distance)
29.        'B':
30.          send_packet(packet, direction = (from_direction + 1)mod 6,
               packet.distance)
31.        'C':
32.          send_packet(packet, direction = (from_direction + 1)mod 6, 1)
33.        'D':
34.          send_packet(packet, direction = (from_direction − 1)mod 6, 1)
35.        NONE:
36.      end
37.    end
38. end
```

### 4.4 5-BCAST and 4-BCAST

Both 5-BCAST and 4-BCAST can be realized as restricted forms of 6-BCAST. In 5-BCAST, we eliminate the additional forwarding step for packets that were tagged 'A' in 6-BCAST. We can accomplish this by setting *packet.tag* to NONE on line 5 in procedure 6-BCAST, and by excluding the *send* operation on line 10. The resulting broadcast tree in one direction is shown in Figure 7. To get 4-BCAST from 5-BCAST, we further eliminate the *send* operation on line 7 of 6-BCAST. The broadcast tree for 4-BCAST is shown in Figure 8.

### 4.5 Correctness of the Algorithms

The correctness of the simple broadcast algorithm can be shown using the figure of the complete broadcast tree (Figure 2). This technique can also be used for 2-BCAST to show that each node receives two copies, and the paths used are node disjoint. For the more complicated algorithms like 6-BCAST, we can show that all nodes receive the required number of copies using the broadcast tree. To show that the paths are node disjoint, we consider the case of 6-BCAST in some detail and examine the paths generated from the source node to a particular destination node. There are two main cases to be considered: (1) the destination node is on one of the principal axes, (2) the destination node is between two principal axes. Figure 9 shows the paths in these two cases.

*Case* 1. Path 0 is created by the BCAST_INIT operation. Paths 1 and 5 are created using tag 'D' and tag 'C' packets (lines 13 and 15), with

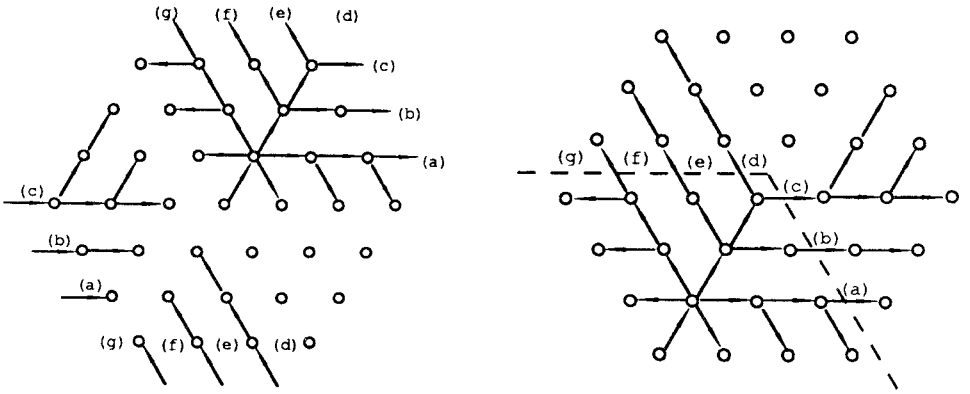


Fig. 7. Packets generated in a 5-BCAST (from one direction)

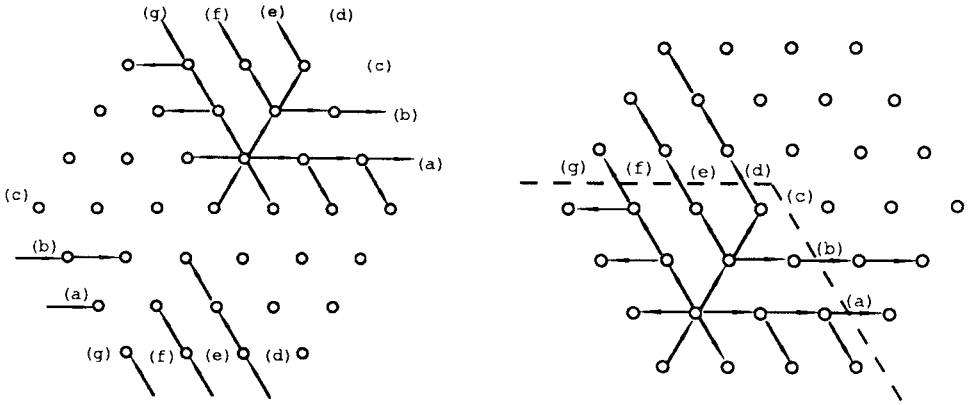


Fig. 8 Packets generated in a 4-BCAST (from one direction).

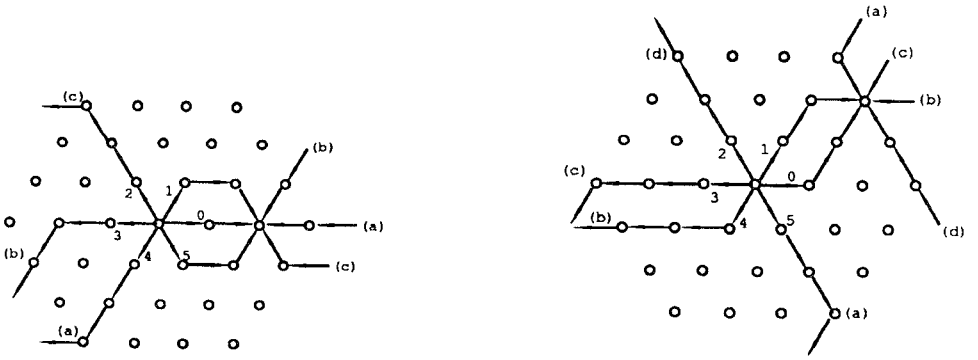


Fig. 9. Disjoint paths in a 6-BCAST.

subsequent forwarding. Path 2 is created using a tag 'A' forwarding on lines 6 and 28, whereas path 3 is created on line 21. Path 4 is created by the send operation on line 6.

*Case* 2. Paths 0 and 1 are created from the axes which bound the sextant containing the destination node. They are created essentially by the operations on lines 21 and 22. Versions of these operations for nodes adjacent to the sender (distance = $n - 2$) and at the end of the principal axes (distance = 0) can be found on lines 13, 15 and 6, 8 respectively. Paths 3 and 4 are similar, and they are created from the axes opposite to the axes which bound the sextant containing the destination node. The send operation on line 10 creates path 2. Path 5 is created using tagged forwarding on line 6.

We can see that the 6 paths generated in each case are node disjoint. 4-BCAST and 5-BCAST are restricted forms of 6-BCAST, so it follows that the paths generated by these algorithms are also node disjoint. In 5-BCAST, one sextant of the hexagon is left uncovered in each direction. From the structure of the broadcast tree, we can see that a different sextant is left uncovered in each direction. Hence, the number of copies received by each node is five. Similarly, in 4-BCAST, the number of copies received is four for the 4-BCAST, and the paths are disjoint.

We have also used an enumeration technique to independently verify these algorithms. We developed a program which implements the broadcast algorithms and generates broadcast trees for a fixed source node. From the broadcast tree of the $k$-BCAST, this program traces the paths from the source node to each node in the system. It then verifies that each node is visited by $k$ paths and that the paths are node disjoint. We used this program to verify that the algorithms performed correctly, for mesh sizes[2] from 3 to 15.

## 5. ALGORITHM ANALYSIS

The *latency* of a broadcast can be defined as the elapsed time between the initiation of the broadcast and the delivery time of the last packet in the broadcast. This latency can vary depending upon the system load and the number of cut-through routes. To analyze broadcast algorithms, one metric is the latency for the algorithm in the best case, that is, when the network is otherwise idle. This latency can be computed based on a model that is commonly used for point-to-point communications. The time required to transmit a packet of length $M$ can be modeled as $S + rM$, where $S$ is the packet set-up time and $r$ is the transmission rate on the link. When a packet *cuts through* a node, the delay experienced is essentially the time taken to receive and examine the packet header, a small constant $d$. Hence, if a packet cuts through $i$ nodes, the time elapsed between the start of transmission and the end of reception is $S + rM + id$.

Consider the simple broadcast algorithm presented in Section 3. Assuming that the network is otherwise idle, and the node can concurrently transmit

[2]A hexagonal mesh of size 2 is a complete graph of seven nodes, which can be treated as a special case.

messages on multiple links (as in the HARTS routing controller), the packet can be delivered to all nodes on the principal axes in a single transmission. The second step, which completes the operation, can also be accomplished by a single packet transmission. Hence, in the best case, the broadcast operation can be completed using *two* packet transmissions. The longest path in this broadcast is $n - 1$ hops, which is the diameter of hexagonal mesh of size $n$, and the packet is buffered and relayed only once on this path. Hence, the number of nodes that are cut through is $n - 3$ and the best-case maximum message latency for the broadcast is $2S + 2rM + (n - 3)d$.

The latency for the other broadcast algorithms can be determined in a similar fashion, using the number of message transmissions and the maximum path length. Note that the *send_packet* operations in all the algorithms can be performed in parallel since they do not have any common links. The paths traced by these algorithms do not result in any contention for the links because each link has to carry at most one packet. Moreover, as mentioned earlier, it is assumed that a node can transmit packets on more than one outgoing link simultaneously, as is the case for the HARTS routing controller [8]. The latencies of these algorithms are shown in Table I. 2-BCAST and 3-BCAST have a minimum number of *two* message transmissions, whereas the other three algorithms require three transmissions because of the additional forwarding step.

A comparison based only on the best-case latency is not satisfactory because the latency is very sensitive to the number of times that a message gets buffered. In the C-wrapped hexagonal mesh, the RELAY primitive can also be used to send a packet to all nodes using *send_packet* with distance set to $3n(n - 1)$, since this traces out a Hamiltonian cycle along any one of the six directions emanating from the broadcasting node. The latency for packet delivery for this algorithm (called Algorithm A) would be $S + rM + (3n(n - 1) - 1)d$ in the best case. This shows that in the best case, for certain values of $S$, $r$, and $d$, Algorithm A can perform better than SBCAST.

However, the probability of a packet getting buffered increases with the length of the path, which results in a larger latency. For example, consider a packet that is to be delivered to a node which is $m$ hops away. If the packet cuts through all the intermediate $m - 1$ nodes, the latency is $S + rM + (m - 1)d$. However, if the packet gets buffered at $i$ of the $m - 1$ nodes on the path, the latency experienced would be $(i + 1)(S + rM) + (m - 1 - i)d$, where the average queueing delay experienced before a packet is serviced is included in the setup time $S$. The average number of times that a packet gets buffered can be determined using the queueing network model, like the one used by Ilyas and Mouftah [11] and Kermani and Kleinrock [13]. Assuming that the network is uniformly loaded and the utilization of each link is $\rho$, the probability that a message gets buffered waiting for a link is $\rho$. For brevity, we call this the probability of buffering for a link. Since the probability of buffering for a link is independent of the probability of buffering for any other link, the number of times that a message gets buffered follows a binomial distribution. Hence, if a message has to traverse $m$ links, the average number of times that it will get buffered is $m\rho$.

Table I. Latency for Different Broadcast Algorithms

| Type | Best-Case |
|---|---|
| SBCAST | $2(S + rM) + (n - 3)d$ |
| 2-BCAST | $2(S + rM) + 2(n - 2)d$ |
| 3-BCAST | $2(S + rM) + 2(n - 2)d$ |
| 4-BCAST | $3(S + rM) + (n - 3)d$ |
| 5-BCAST | $3(S + rM) + (2n - 5)d$ |
| 6-BCAST | $3(S + rM) + (2n - 5)d$ |

Table II. Comparison of Simple Broadcast Algorithms

| $\rho$ | | SBCAST (no cut-through) | Algorithm A |
|---|---|---|---|
| $n = 3$ | 0.00 | $2(S + rM)$ | $(S + rM) + 17d$ |
| | 0.05 | $2(S + rM)$ | $1.85(S + rM) + 16.15d$ |
| | 0.10 | $2(S + rM)$ | $2.70(S + rM) + 15.30d$ |
| | 0.15 | $2(S + rM)$ | $3.55(S + rM) + 14.45d$ |
| | 0.20 | $2(S + rM)$ | $4.40(S + rM) + 13.60d$ |

Based on this model, the average-case broadcast message latency for Algorithm A is $(l + 1)(S + rM) + (3n^2 - 3n - 1 - l)d$, where $l = (3n(n - 1) - 1)\rho$. It is difficult to compute a similar expression for SBCAST because it has many parallel transmissions and because the latency is determined by the delivery time of the last (slowest) message. However, it is possible to compute the latency for SBCAST assuming that the broadcast message gets buffered at each intermediate node. A comparison of these two algorithms for different values of $\rho$ in a hexagonal mesh of size 3 is given in Table II. It is noted that the setup time $S$ is an increasing function of the utilization $\rho$ because it also includes the queueing delay. From this table, it is clear that SBCAST outperforms Algorithm A as the utilization $\rho$ increases. Also, we can show that as $n$ increases, SBCAST performs better even for very small values of $\rho$.

### Simulation Results

In order to study the performance of our broadcast algorithms, we have simulated these algorithms using a discrete-event simulator. This simulator was originally developed by Dolter et al. [9] to study the behavior of virtual cut-through in HARTS, and it models the routing hardware, its interface to the buffer management unit, and the network processor of each node. It has been modified and extended to implement the RELAY primitive in the routing hardware and the broadcast algorithms in the network processor.

The simulator accurately models the delivery of each message by emulating the routing hardware along the route of a packet at the microcode level. It also captures the internal bus access overheads experienced by packets as

they pass through an intermediate node. For example, when a transit packet arrives at an intermediate node, the following sequence of events is initiated. First, the receiver for the link on which the packet arrived waits for the packet header to become available. It then examines the packet header to determine the packet type. For a BROADCAST packet, the receiver tries to schedule two events: one to reserve the transmitter in the same direction to forward the packet, the other to the buffer management unit to receive the packet. Lastly, the receiver schedules events to signal the completion of the packet at this node. The simulator collects detailed statistics for different types of messages. In addition to supporting exponentially distributed packet lengths, the simulator can also use a discrete distribution of packet lengths in which the user specifies different types of messages, their lengths, and the probability of generation of each type of message.

One of the objectives of the simulation experiments was to get an estimate of the performance of the broadcast algorithms under different network load conditions. The traffic generated for the network was uniform across all the nodes and consisted of two types of packets: regular and broadcast. At each source node, packets were generated using a Poisson arrival process and they were assigned type *regular* or *broadcast* with probability 0.999 and 0.001, respectively. For the base nonbroadcast traffic, destination nodes were chosen such that the probability of communicating with a node was inversely proportional to the distance from the source. Both types of packets were assigned lengths of 64, 128, and 512 bytes, with probability 0.3, 0.5, and 0.2, respectively. The link load used for plotting is the ratio of the packet generation rate to the peak I/O rate of the routing hardware. Currently, the peak I/O rate that can be supported by the routing hardware is 4 MBytes per second.

Figure 10 shows the latency of message delivery for the SBCAST algorithm. The units for latency in this and other figures is hardware "clock cycles"; in the current routing controller hardware this is 1.5 microseconds. The three curves shown in the figure are for meshes of size 5, 7, and 9. It can be seen that the latency increases with mesh size and with link load. For a particular link load the increase is close to linear when we move from one mesh size to another. The latency increases super-exponentially for link loads beyond 0.5, indicating that the network is close to saturation at that point. The reason for this "early" saturation is that link access overheads are not included in the computed peak I/O rate. For example, the routing hardware imposes a forced idle time corresponding to 8 bytes between the transmission of successive packets on any link. These access overheads decrease the effective throughput of the routing hardware, so saturation (for regular messages) sets in for link loads greater than 0.7. Since the broadcast message has to traverse multiple links, and the latency is determined by the last packet to be delivered, the latency increases rapidly with increasing link load.

Figure 11 shows the performance of the multiple copy broadcast algorithms compared to the SBCAST algorithm for a hexagonal mesh of size 7. The graph shows the percent increase in latency of the multiple copy broadcasts

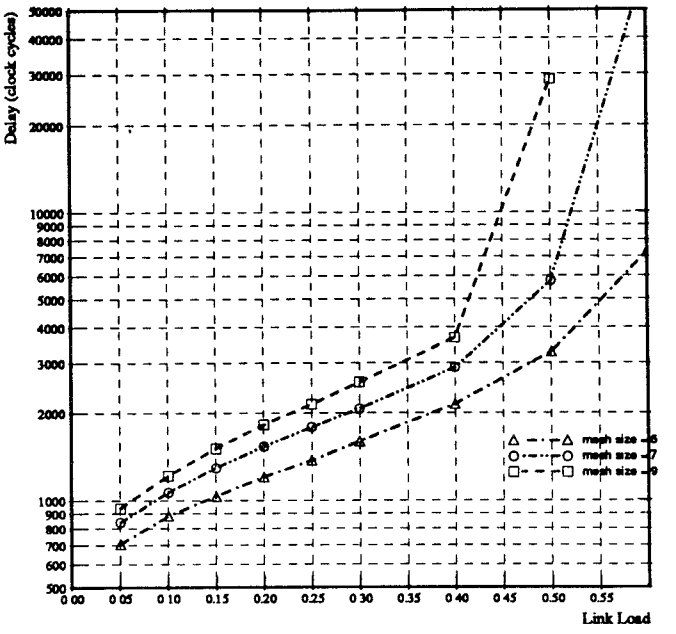


Fig. 10. Performance of simple broadcast.

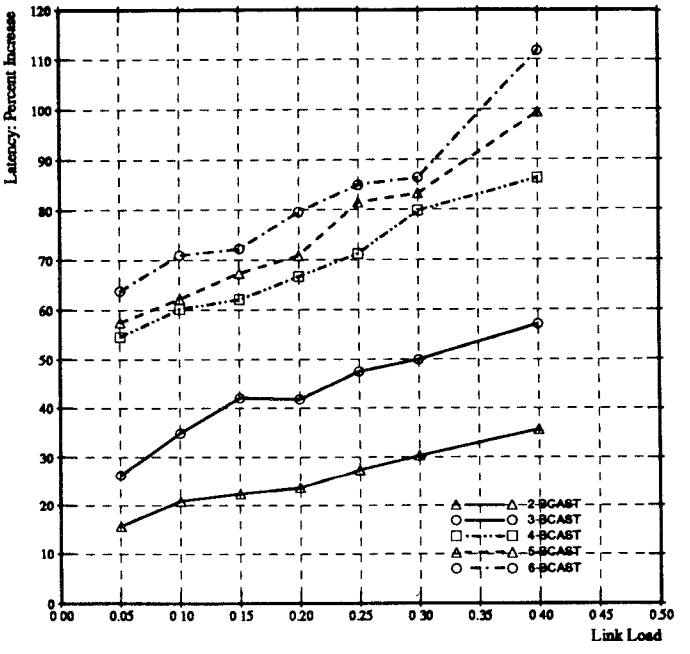


Fig. 11. Performance of multiple copy broadcasts (mesh size = 7).

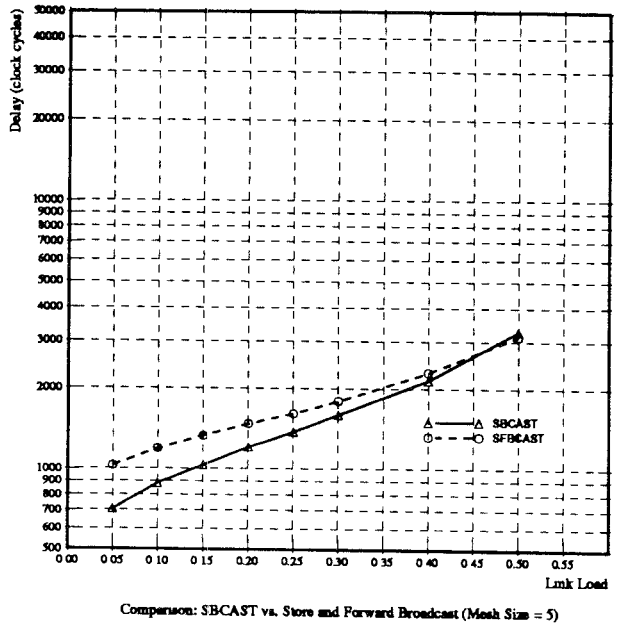


Fig. 12. Performance comparison of simple broadcast algorithms

compared to the SBCAST latency for different link loads. It is observed that the percentage increases with increasing link load, showing that the latency of multiple copy broadcasts increases more rapidly than SBCAST. Part of the reason is that there are more messages to be delivered, and the latency is determined by the slowest message. The cost of 4, 5, and 6 copy broadcasts is quite close, because these three algorithms all use the wrap links to deliver messages and have a long path length.

The other objective of the simulation is to study quantitatively the benefits of using the virtual cut-through broadcast primitive. This is achieved by comparing the performance of our SBCAST algorithm against an algorithm which uses only store-and-forward packet switching. This algorithm, termed SFBCAST, uses the same broadcast tree as SBCAST (see Figure 2). Two performance metrics are used for the comparison: one is *latency*, and the other is *mean delivery time* which is defined to be the average of the delivery time of a broadcast message over all the nodes in the network. This second metric is necessary because latency, which is defined to be the delivery time for the last packet in the broadcast, does not consider the shorter delivery times of other packets and may be skewed by large queueing delays on some path. It is noted that the simulator does not count processing overhead in the nodes, but ignoring this overhead this tends to favor SFBCAST since packets which cut through do not incur the processing overhead in any case. This means that in practice the SBCAST algorithm would perform better, when compared to SFBCAST, than what is shown in the simulations.

The latency of SFBCAST is substantially higher than that of SBCAST for low link loads, but it increases more gradually with the link load. This can be clearly seen in Figure 12, which plots latency of the two algorithms for a

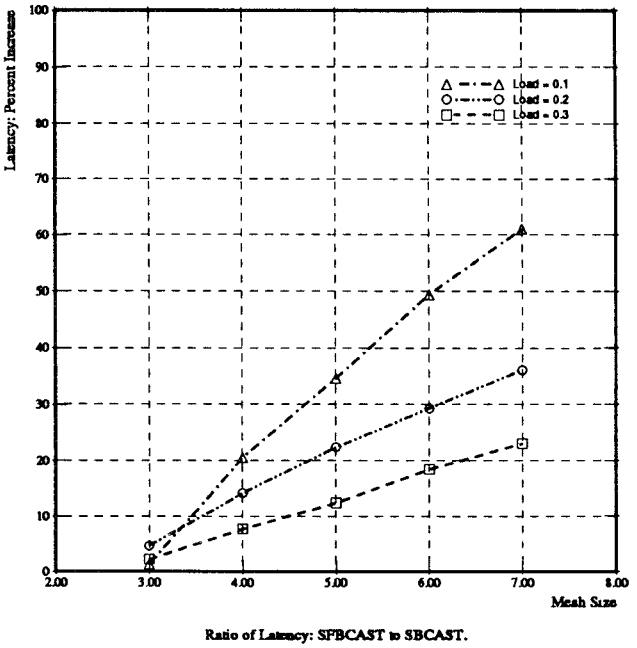


Fig. 13. Comparison of SBCAST and SFBCAST with varying mesh size.

mesh of size 5. Figure 13 and 14 show the effects of varying the mesh size on the relative performance of SBCAST and SFBCAST. Figure 13 plots the percent increase in latency of SFBCAST over SBCAST for different link loads and mesh sizes. This figure shows that the difference in latency reduces as the link load increases, which is expected because the number of packets which cut through reduces with increasing link load. For a fixed link load, the difference in latency increases with increases in mesh size. The results of comparing the average delivery time (Figure 14) are similar, except that the percent difference is substantially higher in this case. These results reflect the advantages of virtual cut-through switching over store-and-forward packet switching.

## 6. CONCLUDING REMARKS

In this paper, we developed a broadcast primitive which is applicable to interconnection networks with virtual cut-through switching. The primitive is well suited for broadcasting in mesh-connected multicomputers, where a simple broadcast can be achieved using this primitive with only two message transmissions in the best case. We also presented a family of broadcast algorithms based on this primitive, which deliver $k$ ($k = 1, \ldots, 6$) copies of a message through node-disjoint paths to each node in the hexagonal mesh. The salient features of these algorithms are simplicity and the efficient use of virtual cut-through. The algorithms are particularly relevant to real-time systems, where the time overhead of identifying all the faulty processors

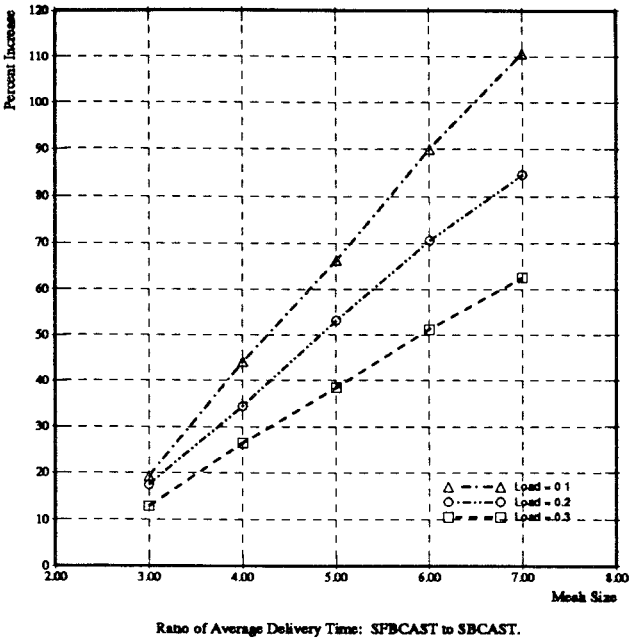


Fig. 14. Average delivery time comparison with varying mesh size.

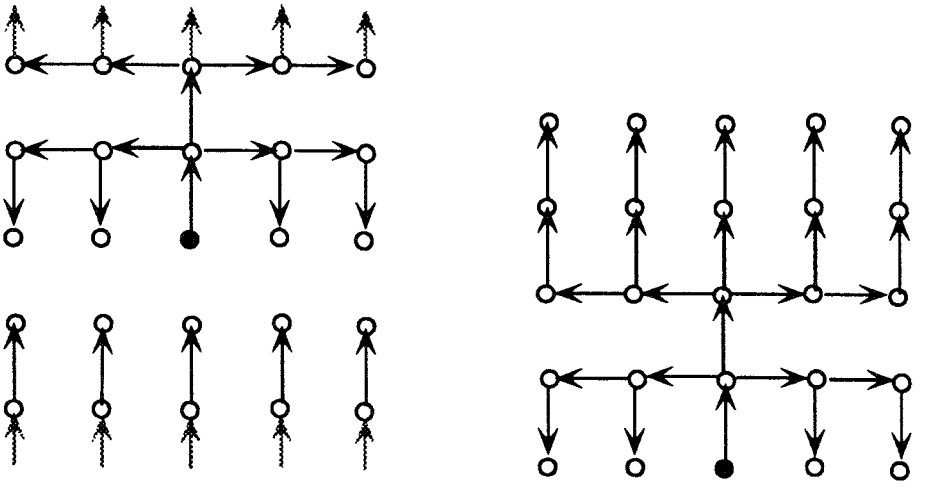

Fig 15. Broadcast tree for a wrapped rectangular mesh (in one direction).

on-line cannot be tolerated. Although the algorithms have been described separately, the relay procedures for the six algorithms can be combined into a single function which makes decisions based on the type of broadcast in progress. We note that similar algorithms can also be developed for wrapped rectangular meshes. The broadcast tree for a 4-reliable broadcast algorithm

in a wrapped rectangular mesh is shown in Figure 15. It can be seen that this tree is similar in structure to the 6-BCAST tree in Figure 6.

In HARTS, the RELAY primitive has been incorporated into the VLSI routing controller chip. The broadcast algorithms will be implemented on the network processor (NP) card, which is currently being developed. The NP has hardware support for time-stamping messages, and one immediate application of reliable broadcasting will be the establishment of a global time-base for HARTS, based on the clock synchronization scheme described by Ramanathan et al. [16].

## ACKNOWLEDGMENT

The authors would like to thank P. Ramanathan, James Dolter, and Sunggu Lee for their comments and suggestions on this work. Thanks also to the anonymous referees whose comments led to significant improvements in the presentation of this paper.

## REFERENCES

1. Birman, K. P. , and Joseph, T. A. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5*, 1 (Feb. 1987), 47–76.
2. Chang, J.-M., and Maxemchuk, N. F. Reliable broadcast protocols. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 251–273.
3. Chen, M.-S., Shin, K. G., and Kandlur, D. D. Addressing, routing, and broadcasting in hexagonal mesh multiprocessors. *IEEE Trans. Comput. C-39*, 1 (Jan. 1990), 10–18.
4. Chou, C.-T., and Gopal, I. S. Linear broadcast routing. *J. Algorithms 10*, 4 (Dec. 1989). 491–517.
5. Dally, W. J., and Seitz, C. L. The torus routing chip. *J. Distrib. Syst 1*, 3 (1986), 187–196.
6. Dally, W. J., and Song, P. Design of a self-timed VLSI multicomputer communication controller. In *Proceedings IEEE International Conference Computer Design: VLSI in Computers* (Boston, Oct. 1987), pp. 230–234.
7. Davis, A., Hodgson, R., Schediwy, B., and Stevens, K. Mayfly system hardware. Tech. Rep. HPL-SAL-89-23, Hewlett-Packard Corp., Apr. 1989.
8. Dolter, J. W., Ramanathan, P., and Shin, K. G. A microprogrammable VLSI routing controller for HARTS. In *Proceedings IEEE International Conference on Computer Design: VLSI in Computers* (Boston, Oct. 1989), IEEE, New York, pp. 160–163.
9. Dolter, J. W., Ramanathan, P., and Shin, K. G. Performance analysis of message passing in HARTS: A hexagonal mesh multicomputer. *IEEE Trans. Comput. C-40*, 6 (June 1991), 669–680.
10. Fraigniaud, P. Asymptotically optimal broadcast and total-exchange algorithms in faulty hypercube multicomputers. Tech. Rep. 89-05, Ecole Normale Superieure de Lyon, 46, Allee de Lyon, May 1989.
11. Ilyas, M., and Mouftah, H. T. Towards performance improvement of cut-through switching in computer networks. *Perform. Eval. 6*, 2 (July 1986), 125–133.
12. Johnsson, S. L., and Ho, C.-T. Optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Comput. C-38*, 9 (Sept. 1989), 1249–1268.
13. Kermani, P., and Kleinrock, L. Virtual cut-through: A new computer communication switching technique. *Comput. Networks 3* (1979), 267–286.
14. Lamport, L., and Melliar-Smith, P. M. Synchronizing clocks in the presence of faults. *J. ACM. 32*, 1 (Jan. 1985), 52–78.
15. Lamport, L., Shostak, R., and Pease, M. The Byzantine generals problems. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.

16 RAMANATHAN, P., KANDLUR, D. D , AND SHIN, K G. Hardware-assisted software clock synchronization for homogeneous distributed systems. *IEEE Trans. Comput. C-39*, 4 (Apr 1990), 514–524.
17 RAMANATHAN, P., AND SHIN, K. G. Reliable broadcast in hypercube multicomputers. *IEEE Trans. Comput. C-37*, 12 (Dec. 1988), 1654–1657
18. STEVENS, K. S. The communication framework for a distributed ensemble architecture AI Tech. Rep 47, Schlumberger Research Laboratory, Feb. 1986.
19. YANG, C. L., AND MASSON, G. M. A distributed algorithm for fault diagnosis in systems with soft failure. *IEEE Trans Comput. C-37*, 11 (Nov. 1988), 1476–1480.

Received August 1990; revised June 1991; accepted August 1991