MESSAGE ROUTING IN HARTS WITH FAULTY COMPONENTS *

Alan Olson Kang G. Shin

Real–Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122.

ABSTRACT

It is important to design a distributed system which is capable of delivering messages even in the presence of faulty components between their source and destination nodes. We develop a routing scheme in two steps for a wrapped hexagonal mesh, called **HARTS** (Hexagonal Architecture for Real–Time Systems), which assures the delivery of every message as long as there is a path between its source and destination.

The proposed scheme can also detect the non-existence of path between a pair of nodes in a finite amount of time. Moreover, the scheme requires each node in **HARTS** to know only the state (faulty or not) of each of its own links. The performance of the simple routing scheme is simulated for 3– and 5– dimensional H–meshes while varying the physical distribution of faulty components. It is shown that a shortest path between the source and destination of each message is taken with a high probability and a path, if exists, is usually found very quickly.

1 Introduction

Recently, distributed computing systems have received a great deal of attention, since not only do they provide a high degree of parallelism but also are capable of greater fault-tolerance. However, because distributed systems are more complex than uniprocessor systems, they are more likely to suffer a component failure. Therefore, if the system cannot reconfigure itself after a failure, its fault-tolerance will be lower than a uniprocessor system. Thus, a distributed system must be designed so that most operations can continue even in the presence of component failures.

Message routing is one aspect of a distributed system which can be severely affected by component failures. Any routing algorithm must be able to handle the case where the intended path of a message is blocked by link/node failures. Some work has already been done on message routing in a hypercube in the presence of link and node failures [1, 3, 6, 8, 10]. Also, some algorithms are proposed in [7] to broadcast the information about faulty components to all the other nodes in the system so that messages can be routed around the faulty components. Clearly, if each node is equipped with the information on all faulty components, then it can always determine a shortest fault-free path to route messages as long as the source and destination nodes are connected. However, it is usually too costly to equip every node with the entire network information, especially when the size of the system is large. Hence, it is important to develop routing schemes which require each node to keep only the minimal information essential for making correct routing decisions.

The goal of this paper is to develop such a routing scheme for the Hexagonal Architecture for Real-Time Systems (**HARTS**), currently under development at the Real-Time Computing Laboratory, The University of Michigan. (See [2, 9] for justifications of using a hexagonal mesh topology and its comparison with other topologies.) We will require each node to know only the condition (faulty or non-faulty) of its own links. Our routing scheme will deliver each message successfully as long as there is a path between its source and destination. Unlike others, it does not require any assumption on the number of faults or fault patterns. If there does not exist any path between the source and destination nodes, our routing scheme will be able to detect the non-existence of path in a finite amount of time. An addressing and routing scheme for **HARTS** *in the absence of* faulty components has been developed [2], and a routing controller chip has also been developed as the front-end communication interface for each **HARTS** node [4].

This paper is organized as follows. In Section 2, **HARTS** is briefly reviewed for completeness (see [2, 4] for a detailed account). Section 3 outlines a method for routing on an unwrapped mesh. In Section 4 the algorithm is extended to properly handle the wrapping of the mesh. In Section 5, the performance of the algorithm is simulated for several different mesh sizes and fault types. Finally, the paper concludes with Section 6.

2 Hexagonal Mesh Architecture

A simple hexagonal mesh (H-mesh) is a set of nodes laid out on a hexagonal grid such that there is a central node inside a series of nested hexagons. Each hexagon has one more node on each edge than the one immediately inside of it. The *dimension* of the mesh is defined to be the number of nodes on one side of the outermost hexagon. It has been shown in [2] that the number of nodes in an e–dimensional H-mesh is $3e^2 - 3e + 1$.

For a number of reasons, such as ease of task allocation and migration, it is advantageous for a processing surface to be homogeneous. A simple H-mesh is not homogeneous since the nodes on the outermost hexagon will have only three or four connections while the rest of the nodes will have six. To achieve homogeneity, the unused links on the perimeter are "wrapped" as described below.

Any node in the simple mesh will have six oriented directions, one corresponding to each of the six links. Without loss of generality, the link pointing horizontally to the right can be thought of as the *x direction*, the link 60 degrees counter-clockwise as the *y direction*, and the link 120 degrees counter-clockwise as the *z direction*. The

*The work reported here is supported in part by the NASA under Grant NAG–1–296 and by the Office of Naval Research under Contract N00014–85–K–0122. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

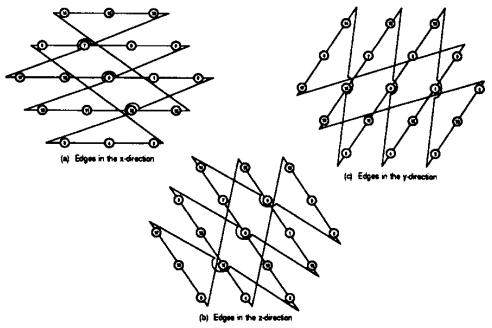

Figure 1: Wrapping for a mesh of dimension 3

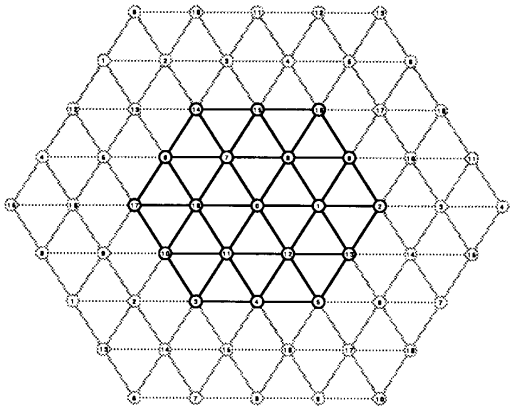

Figure 2: Hexagonal mesh of dimension 3

remaining three links point in the $-x$, $-y$, and $-z$ directions, respectively. A simple H-mesh of dimension e can be partitioned into $2e-1$ rows with respect to each of the x, y, and z directions. Using this, "C-type wrapping" is defined as follows [2, 9].

C-type wrapping: For each of the directions x, y, and z, connect the last processor in row i to the first processor in row $(i+e-1)$ mod $(2e-1)$, where the rows are numbered $0,\ldots,2e-2$.

An illustration of the row partitions and wrap connections for each direction on a mesh of dimension 3 is given in Fig. 1.

C-type wrapping leads to an elegant labeling scheme for the nodes. Since the mesh is homogeneous, all nodes are topologically equivalent. Therefore, any node can be viewed as being in the center of the mesh. Choose one node to serve as the origin and label all other nodes with their respective distance (number of hops) from the origin in the x direction. An example of the labeling for an H-mesh of dimension 3 is given in Fig. 2, where the greyed area of the mesh is used to show more clearly how the links are wrapped.

Routing with the above labeling scheme is simple. An elegant $\Theta(1)$ algorithm is presented in [2] to give the shortest path between any two nodes. It returns three integers, m_x, m_y, and m_z, each of which represents the distance to be traveled in the corresponding direction (a negative offset indicates movement in the corresponding negative direction). At least one of the offsets is guaranteed to be zero, and $|m_x|+|m_y|+|m_z| \leq e-1$ where e is the dimension of the mesh. These offsets are included in the message header, and as each node forwards the message, it updates the offsets appropriately. If the message reaches a node with $m_x = m_y = m_z = 0$, the message has reached its destination. This routing scheme is a significant improvement over the one in [9] because it handles the mesh wrapping transparently.

3 Simple Routing

The goal of this paper is to develop a routing algorithm which can successfully deliver any message as long as the destination is reachable given that each node only has knowledge of the state of its own links. And, if there is no path between the source and destination of a message, this fact should be detected in a finite amount of time.

The structure of the H-mesh provides an obvious method for detouring around a single faulty link. Each link forms one side of a triangle whose other two sides form a convenient detour should the link fail. As an example, consider Fig. 2. If the link from 18 to 0 has failed, a message could detour around the failed link using the path 18 to 7 to 0 or the path 18 to 11 to 0. This can be recursive; for example, if the link from 18 to 7 has also failed, the detour could be 18 to 6 to 7 to 0.

This algorithm is easy to implement. The message starts in a *free* mode in which message routing is much the same as in a fault-free mesh, until all links of a node on the shortest path to the destination are found to be faulty. The message is then placed in *detour* mode. In detour mode a message "rotates" counter-clockwise through the node, checking each link in turn and sending the message on the first non-faulty link found. The message starts checking with the link counter-clockwise of the links on the shortest path if it has just been placed in detour mode, or with the link counter-clockwise of the link by which the message entered the node if the message entered the node while in detour mode. A message exits detour mode when it reaches a node closer to the destination than the node where the message entered detour mode.

Accounting for this algorithm is simple. The message need only keep track of the destination node, the current state (free or detour), the offsets to the destination, and the distance to the destination when the message entered detour mode. The offsets are adjusted each time the message is sent to another node, but are never "recalculated" with respect to the destination. That is, wrapping is ignored when the offsets are updated so that the offsets may not represent the shortest path to the destination. This simple-minded approach will, at times, lead to poor routing decisions, but will be a great aid when it comes to determining when a destination is unreachable.

Finally, there must be some way of recognizing when the routing algorithm cannot deliver a message. This is done by showing that any message which does not reach its destination must eventually cycle.

Lemma 1 *On any finite H-mesh, if a message never reaches its destination, it will eventually reach a point after which it will never return to free mode.*

The proof is trivial if one keeps in mind that the algorithm as described above ignores the wrapping of the mesh.

Lemma 2 *Given message m, for any non-faulty link o of node n there will be a link i of node n such that if m enters n while in detour mode and remains in detour mode while at n, m will leave n by link o if and only if it entered by link i.*

Proof: Let i be the first non-faulty link clockwise of o. The rest of the proof then follows trivially from the description of the algorithm. □

Theorem 1 *On any finite H-mesh, if a message does not reach its destination, it will eventually be in a cycle.*

Proof: By Lemma 1, a message that does not reach its destination will eventually reach a point after which it will never return to free mode. Consider a message that will not reach its destination and has entered detour mode permanently. From Lemma 2 it is clear that knowing the current node and the outbound link is enough to determine the future behavior of the message. The outbound link not only determines the next node, but the inbound link for the next node, and therefore the outbound link for the next node. This in turn determines the outbound link for the next node, and so on. Call the current node and the selected outbound link the *state* of the message. From Lemma 2 it is clear that given the current state s, not only can the sequence of future states be determined, but that it will always be the same whenever the message is in state s. Since there are only a finite number of nodes and at most six possible outbound links per node, there are only a finite number of states. A message that does not reach its destination will pass through an infinite number of states. It must therefore pass through some state s twice within a finite amount of time. This clearly constitutes a cycle as the message has returned to s, and by the above observation not only will it return to s, but will go through the same sequence of states in doing so. Further, it will continue to return to s indefinitely since the sequence of states following s will always be the same. □

Now the algorithm needs some method for detecting the cycle. The following theorem provides it.

Theorem 2 *If a message is in a cycle, the cycle must include the node at which the message was last in free mode, and will exit that node via the same link as it did when it first entered detour mode.*

Proof: Assume message m has entered detour mode permanently and is in a cycle. Any node could appear more than once in a tour of the cycle, but if a node appears more than once, then each time the message reaches the node it must exit the node via a different link than it used before. Therefore, each step in the cycle may be specified by the node and the outgoing link. Pick any such node–link pair in the cycle. Find the first occurrence of this pair after the last free node, call it (n_1, ℓ_1). Since the message is in detour mode, Lemma 2 shows that the node–link pair (n_2, ℓ_2) preceding (n_1, ℓ_1) will precede all subsequent occurrences of (n_1, ℓ_1). Therefore, (n_2, ℓ_2) is in the cycle. The same argument applies to (n_3, ℓ_3), (n_4, ℓ_4), ..., (n_k, ℓ_k), where n_k is the last free node. The last free node and the link by which it was exited are therefore in the cycle. □

From Theorem 2, the above algorithm will be unable to deliver a message if it returns to the last node where it was in free mode, and exits via the same link it did previously.

4 Handling Mesh Wrapping

The algorithm described in the preceding section is fairly close to what is desired. Its main problem is its inability to take into account the wrapping of the mesh. One can better understand how the algorithm functions if one imagines the mesh in Fig. 2 extended infinitely far. Then each node would have infinitely many "copies" of itself. Each copy of the destination node represents a distinct family of paths to the destination. If the destination is reachable, then at least one of these copies, not necessarily the closest one, must be reachable. The algorithm as it has been described so far will only attempt to reach the closest copy of the destination.

In this section we will show that using the above algorithm a message will reach any copy of the destination it tries to reach, provided that copy is reachable. And, will derive a "fix" for the above algorithm so that it will always reach a reachable destination.

4.1 Cycles and Reachability

Since a message which fails to reach its destination will cycle, it is important to detect cycles, and to note their characteristics. Cycles are detected by checking the current node and outgoing link against the last free node and its outgoing link. From this perspective, there are only two types of cycles.

The first kind, called a *circle* is characterized by the message returning to the exact same node at which the cycle started. Fig. 3 shows an example set of failed links that would cause a circle. If, in the case of Fig. 3, a message should be sent from node 0 to node 11, it would travel along the perimeter of the isolated mesh component and return to node 0. The existence of a circle indicates the H-mesh has become disconnected.

The second kind of cycle, called an *incision*, is characterized by the message reaching a *copy* of the node at which the cycle started. Fig. 3 also contains an incision. If a message should be sent from node 11 to node 8, it would head leftward through nodes 10, 17, 16, 4, and back to 11. The existence of an incision does not necessarily indicate the H-mesh has become disconnected.

As was shown above, in the case of either cycle type the destination may still be reachable. Throughout the rest of this paper, the terms *circle failure* and *incision failure* will refer to messages going in to the respective cycle type when the destination was reachable, and *circle fault* and *incision fault* will refer to messages going into the respective cycle type when the destination was not reachable.

We have to show that if the message is in a cycle, the copy of the destination that it was attempting to reach was not reachable. The following definitions are useful in the discussion to follow.

Definition 1 *Given a failed link ℓ, the* block *B containing ℓ is the set S of failed links such that (i) $\ell \in S$, and (ii) for every $\ell' \in S$ $\exists \ell^* \in S$ such that ℓ' and ℓ^* connect to the same node and are neighboring links on that node.*

A block is simply a collection of failed links through which no message may pass. Fig. 4 contains examples of blocks, each of which is circled.

Definition 2 *The* perimeter *nodes of a block B is the set of all nodes, each of which has one or more links belonging to B.*

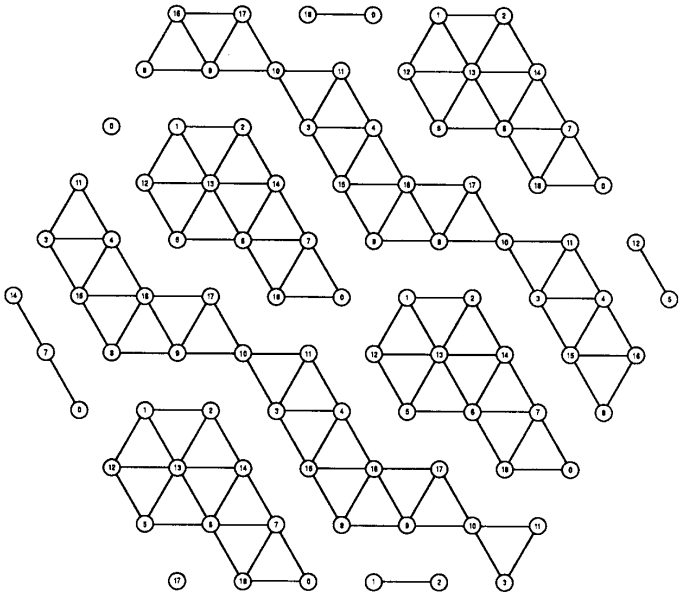

Figure 3: Circle and incision.

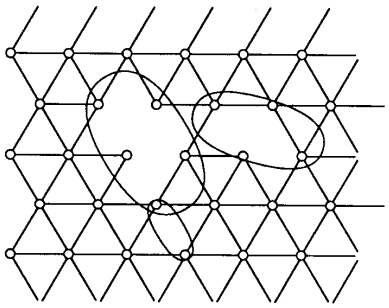

Figure 4: Examples of blocks.

Using these definitions, we prove the following lemma, which states that if the message is in a cycle, there is only one block that is responsible for the cycle.

Lemma 3 *If a message is in a cycle, all failed links that the message encountered will belong to the same block.*

Proof: By induction. From the definition of a block it is clear that all failed links encountered while the message is at the node where it was forced into detour mode belong to the same block.

Given that all failed links encountered while at a node A are in the block, there are two possible cases. The first case is shown in Fig. 5a. The message exits node A via link 2 after finding link 1 has failed. Upon reaching node B, it finds link 3 has failed. Link 3 will

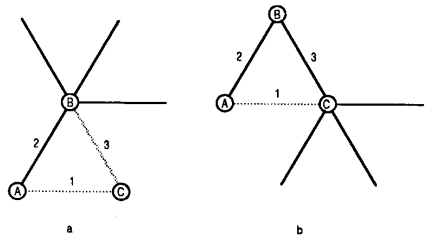

Figure 5: Perimeter nodes.

be in the same block as link 1 because it is immediately adjacent to link 1 on node C. (Note that link 1 is in the block by the inductive assumption.) Further, any more failed links encountered while at node B will be in the block because link 3 is in the block. The other case is that shown in Fig. 5b. In this case the message reaches node B via link 2 and immediately proceeds to node C via link 3. At node C the first link examined will be link 1, which is in the block, and therefore all other faulty links examined while at node C will be in the block.

Since all failed links encountered at the first node are in the block, and given all failed links encountered at the current node are in the block, all failed links encountered at the next node where failed links were encountered before will be in the block, the lemma follows by induction. □

The following is a formal definition of the intuitive notion of what it means to have two paths cross one another.

Definition 3 *A path P_1 is said to* cross *another path P_2 if $\exists$ nodes n_1 and n_2 in both P_1 and P_2 such that the paths use the same links between n_1 and n_2, at node n_1 P_1 has a link (either inbound or outbound) which is counter-clockwise of P_2's inbound link and clockwise of P_2's outbound link, and at n_2 P_1 has a link clockwise of P_2's inbound link and counter-clockwise of P_2's outbound link.*

It is worth noting that n_1 and n_2 may in fact be the same node as that would trivially satisfy the condition that the paths overlap in between. With the above definition, we can now prove the following.

Lemma 4 *No fault-free path can cross the path of the message in detour mode.*

Proof: For a fault-free path to cross the path of a message there must be a node with a non-faulty link counter-clockwise of the message's inbound link and clockwise of the message's outbound link. This is not possible if the message is in detour mode. □

Up to now, we have shown that if a message is in a cycle there is exactly one block that is responsible. Now we have to "confine" the block by showing that it has a very definite location with respect to the cycle.

Definition 4 *A* segment *induced by a path P is the set of nodes not on P such that all paths between any two nodes in the segment cross P an even number of times.*

Lemma 5 *All perimeter nodes of the block which caused a circle will be either in the cycle or in one of the segments induced by the cycle, and none of the nodes in the segment containing the perimeter nodes will be reachable.*

Proof: Consider any two perimeter nodes. Since the perimeter nodes and the links in the block will form a connected subgraph, there should be a path between the two perimeter nodes that uses only links in the block. From the definition of a block it is clear that there will exist such a path where at each node the path will exit the node either via the link by which it entered, or by one of the links immediately clockwise or counter-clockwise of the entry link. Such a path could not cross the cycle. If the two nodes were not in the same segment, the path would have to cross the cycle. Therefore the two nodes must be in the same segment.

Call the segment containing the perimeter nodes the *block segment*. Assume all links not in block B have not failed and there is some reachable node in the block segment. Then there must be a non-faulty link ℓ from some node c in the cycle to some other node n in the block segment. Link ℓ must be clockwise of c's inbound link and counter-clockwise of c's outbound link. From node c follow along the cycle until a faulty link ℓ' is found counter-clockwise of the inbound link and clockwise of the outbound link. Such a link must exist or the message would not be in a cycle. Link ℓ' will connect from some node on the cycle to some perimeter node n'. Since the path just constructed (starting at n and ending at n') crosses the cycle exactly once, n' must not be in the block segment. Since n' is a perimeter node, it cannot be in any other segment. Therefore, n' must also be on the cycle and is reachable. Since ℓ' connects two nodes which are already reachable, ℓ' can be made non-faulty without affecting the reachability of any nodes. Once ℓ' has been made non-faulty a new, smaller cycle can be formed by returning the message to the last free node and letting the routing algorithm start. This new cycle will still contain c, so the same operation can be performed. In fact, it can be repeated until there are no more faulty links along the cycle. This is clearly impossible; if no faulty links are present, the cycle would never exist. Therefore, link ℓ and node n cannot exist. □

Now that the block that is responsible for the cycle has been confined to a single segment, we show that the message will stick close to the perimeter of the block.

Lemma 6 *A message that is forced into a circle by block B will pass through every reachable perimeter node of B.*

Proof: This follows from Lemma 5. Any perimeter node must either be in the block segment or on the cycle. If it is in the block segment, by Lemma 5 it is not reachable. Therefore any reachable perimeter node must be on the cycle. □

Finally, we can show that the algorithm will only fail to reach a destination if the destination is in fact unreachable.

Theorem 3 *If a message is in a cycle, the copy of the destination the message is attempting to reach is not reachable.*

Proof: This proof has two parts: the first for circle, and the second for incisions.

Suppose the cycle is a circle. Consider the node on the cycle which is closest to the destination. When the message first entered this node it was placed in free mode. It was prevented from leaving free mode by the presence of one or more failed links, which by Lemma 3 must be in the block which caused the circle. At the other end of one of these failed links must be a perimeter node which is closer to the destination. This node must be in the block segment, and since it is closer to the destination than any other node on the cycle, the destination must be in the block segment. Therefore, by Lemma 5 the destination is not reachable.

Suppose the cycle is an incision. Call a copy of the last free node and the copy of the destination node the message is attempting to reach an arbitrary *source–destination pair*. Consider a source–destination pair past which the message will travel after it has been in the cycle for some time. No path can be longer than $3e^2 - 3e$ hops, and if the message has been in the cycle for more than $3e^2 - 3e$ hops, any path between this local source–destination pair must cross the path of the cycle, which is not allowed. Since no path exists for an arbitrary source–destination pair, the copy of the destination the message is attempting to reach must not be reachable. □

4.2 Dealing with Cycles

Theorem 3 assures us that the simple algorithm will reach a reachable copy of the destination. It does not, however, say that the destination is not reachable. If the message is in a cycle, one of the other copies of the destination might still be reachable. But, the type of cycle the message is in gives a few clues as to where some reachable copies may lie.

4.2.1 Circles

The only way the destination could still be reachable if a circle has been encountered is when the situation is like that in Fig. 3. In this case a message going from node 0 to node 13 will "circle" the

destination. This can be fixed without too much trouble at the expense of a slightly more complicated algorithm.

A circle will create a family of finite, connected components in the unwrapped mesh. For a destination to be reachable, there must be copies of both the source and destination in the same connected component. Further, since in an e–mesh no two nodes are more than $e-1$ hops away from one another [1], all nodes in any connected component will be within $e-1$ hops from some perimeter node of the component (assuming no faults internal to the component force a longer path). Therefore, if a message is in a circle and there is a reachable copy of the destination, since by Lemma 6 the cycle will include all perimeter nodes of the component, the copy must be within $e-1$ hops of some node on the cycle. The algorithm need only make note of all such copies of the destination and attempt to reach each of them.

This extension will require some extra storage space to be allocated for the message, but it will rarely require much. In a 3–mesh, the message will find no more than 4 copies of the destination, and finding more than one or two is very unlikely.

4.2.2 Incisions

The following theorem provides an idea on how to handle incisions.

Theorem 4 *In the presence of an incision, if a destination is reachable, then either the closest copy is reachable, or if a cycle occurs, then a reachable copy of the destination must lie within* $2e-1$ *hops of both the last free node and the node where the cycle was detected.*

Proof: By Theorem 3, if the message is in a cycle, the copy of the destination the message is attempting to reach is not reachable. By Lemma 1 a message must either reach its destination or cycle. Therefore, if the message does not cycle it must reach its destination and the closest copy of the destination is reachable.

Otherwise, the message cycles. Consider the last free node. It is a perimeter node of the incision, and there are reachable copies of it $2e-1$ hops away in either direction along the incision. Since two copies of the perimeter node at the other end of any failed link tested while at the last free node will be within $2e-2$ hops of the last free node, another copy of the incision lies no more than $2e-2$ hops away. Therefore, by moving $2e-1$ hops away from the last free node, the message will still be within $2e-1$ hops of some reachable copy of the last free node. So, nothing is to be gained by moving more than $2e-1$ hops from the last free node, and a copy of any reachable node must lie within $2e-1$ hops.

Call the last free node c. There are six copies of c within $2e-1$ hops of c. If the destination is reachable, one of the copies of the destination local to one of these six copies of c must be reachable. Let c' be the copy of c where the cycle is detected. Node c' is one of the six copies, but its local copy of the destination is not reachable since it is in the same position as c. The same goes for c'', the copy of c in the opposite direction along the incision of c'. It is simply c seen from c''s perspective. There are two copies of c which are $2e-1$ hops away from both c and c'. The last two copies of c lie $2e-1$ hops away from both c and c''. These two pairs are equivalent since the copies between c and c'' are the copies between c and c' as seen from c''s perspective. Therefore, if any of the copies of the destination are reachable, one of the copies must be within $2e-1$ hops of both the last free node and the node where the cycle was detected. □

The preceding theorem shows that if the destination is reachable, there is a reachable copy within $2e-1$ hops of wherever you happen to be. It also provides a way of eliminating a number of the possibilities, since it shows that a reachable copy will lie within $2e-1$ hops of both the last free node and the node where the cycle was detected. There can be at most 2 such nodes.

4.2.3 Final Algorithm

Using what has been said above, it is possible to specify the complete algorithm which can properly handle circles and incisions, and to therefore deliver a message whenever the destination is reachable.

We assume that there is a field on the message, message.offsets, which contains the offsets along the x, y and z axes to the current copy of the destination. We also need the array message.alt_offsets[] which contains the offsets to alternate copies of the destination. Lastly, we define message.circle which is TRUE if a circle has been detected, and message.incision which is TRUE if an incision has been detected.

ALGORITHM A

```
while current_node_address ≠ message.destination

  if message.mode = FREE
    or message.distance < message.last_free_distance then
    message.mode := FREE
    outbound_links := all links on the shortest path
  else
    outbound_links := link 60° counter-clockwise of inbound link
    if message.distance > n − 1
      and not (message.circle or message.incision) then
      save offsets to nearest copy of destination
      in message.alt_offsets[ ]
    endif
  endif

  if not (outbound_links subset failed_links) then
    L := any element of (outbound_links — failed_links)
  else
    alternate := link 60° counter-clockwise of outbound_links
    while L ≠ alternate do
      if not (alternate in failed_links) then
        L := alternate
      else
        alternate := link 60° counter-clockwise of alternate
        if alternate in outbound_links then
          there is no way out of this node
          exit
        endif
      endif
    enddo
    if message.mode = FREE then
      message.mode := DETOUR
      message.last_free_distance := message.distance
      message.last_free_node := current_node_address
      message.last_free_link := L
    else
      if message.last_free_node = current_node_address
        and message.last_free_link = L then
        if not (message.circle or message.incision) then
          if message.last_free_distance = message.distance then
```

```
            message.circle := TRUE
          else
            message.incision := TRUE
            find all copies of the destination within
            2e − 1 of the current node and the last free
            node, put offsets to them in message.alt_offsets[ ]
          endif
        endif
        if message.alt_offsets[ ] empty then
          the destination is unreachable
        else
          Get a new destination from message.alt_offsets[ ]
          endif
        endif
      endif
    endif

    update message fields to reflect traversing link L
    send message on link L
  enddo
```

5 Simulation

Simulation was used to evaluate the algorithms performance for several different mesh sizes and fault types. Simulation was done using a simplified version of the algorithm which halted when a cycle was detected, instead of switching to an alternate destination. This was done because it was felt that the most important result of the simulation would be to determine how common incisions and circles were. Also, proper handling of cycles adds a great deal of complexity to the algorithm and a great deal of space to the message header, we felt that circle and incision faults might not be common enough to justify the extra expense.

5.1 Design of the Simulator

The simulator is designed to determine the average performance of the algorithm for a fixed sized mesh with a fixed number of faults. It could be set to simulate only link failures or only node failures. For each run the simulator was recompiled for the desired mesh size, number of faults, and number of fault configurations. For each fault configuration the simulator selected at random the specified number of links (or nodes) to be removed, ran Floyd's algorithm [5] on the resulting graph to determine the shortest path between any two nodes, then simulated the sending of a message between each pair of nodes. The number of hops each message took, whether or not it was delivered, and the types of cycles (if any) were noted. These results were then compared with the results of the shortest path algorithm to determine extra hops, whether the message was actually deliverable, etc.

5.2 Simulation results

Simulations were run for both a 3-mesh and a 5-mesh with both link failures and node failures. Space restrictions preclude including all results, but the results from the 3-mesh with link failures are plotted here. Fig. 6 shows the number of messages which were deliverable, delivered, and undeliverable. Fig. 7 shows the percentage of undelivered messages which were undeliverable, or which were deliverable but failed to reach their destinations due to circle failures or incision failures. Figs. 8 and 9 show the average difference and

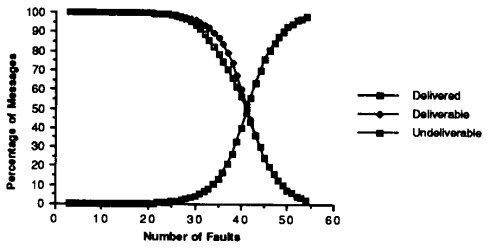

Figure 6: Messages vs. number of faults.

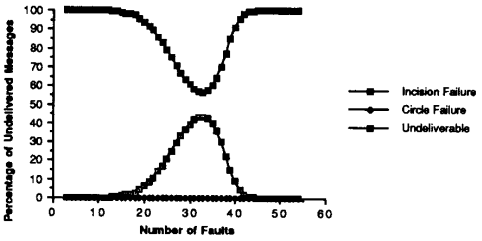

Figure 7: Undelivered messages vs. number of faults.

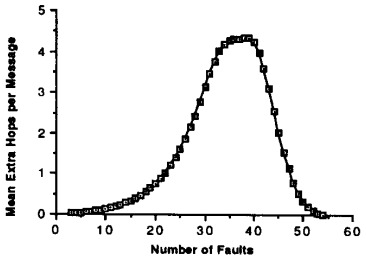

Figure 8: Mean extra hops vs. number of faults.

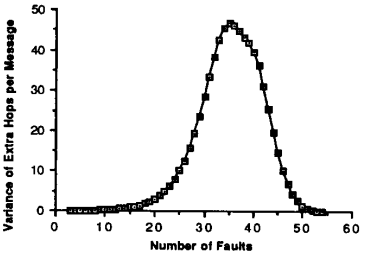

Figure 9: Variance of extra hops vs. number of faults.

the variance of the average difference between the length of the path the message took and the shortest path.

Results for the 5-mesh with link failures are much the same. The mean and variance of extra hops both increased (from maxima of 5 and 50 to 20 and 600), but the "bumps" in Fig. 7 are considerably smaller. For node failures on both mesh sizes, both mean and variance decrease drastically from the link failures case, as do the "bumps" in Fig. 7.

The results indicate that while incisions are the most common of the two, neither circles nor incisions are common, especially for small number of faults or large meshes. In light of this it may be practical to not implement the procedures for properly handling cycles. Care must be taken however, the existence of a single incision can make one quarter or more of the reachable nodes unreachable if incisions are not properly handled. But, if the number of failures is small, say less than one-fifth of the total number of components, the probability of an incision or a circle is extremely small.

Most worrisome is the high variance in the extra number of hops a message took in reaching its destination. This clearly violates the real-time constraints of the applications for which **HARTS** was designed. It is our feeling that any algorithm operating with the constraint that only local information will be available will have a high variance in path length when a large number of faults are present. When a large number of faults are present, the mesh has deteriorated to the point where little of its original structure remains. It is difficult to make intelligent routing decisions under these conditions when one cannot see beyond the borders of the current node.

6 Conclusion

In this paper a fault-tolerant routing algorithm for **HARTS** was developed and shown to deliver any message provided the destination is reachable.

The algorithm could be simplified greatly at the expense of some messages not being delivered to reachable destinations. Simulation showed that such cases are rare, especially for small numbers of faults. If a large number of faults may be present and the high variance can be tolerated, or it is imperative that all deliverable messages be delivered, then the extra complexity of the full routing algorithm may be justified.

It is worthwhile to investigate:

- How good is the non-fault-tolerant routing algorithm which just gives up as soon as all links along the shortest path are found to be faulty.
- How much improvement will be seen if while in free mode and the message has a choice of two links it picks one which will not reduce any offsets to zero.
- How much improvement can be gained from an algorithm which can check links in both the clockwise and counter-clockwise direction. The algorithm would only do one or the other while in detour mode, but when it first enters detour mode it checks both ways and takes the first free link it finds.

All of these questions are topics of further research and will be considered to some degree as **HARTS** is implemented.

References

[1] M. S. Chen and K. G. Shin, "Message routing in an injured hypercube," In *Proc. Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 312–317, Los Angeles, January 1988.

[2] M. S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, routing and broadcasting in hexagonal mesh multiprocessors," *IEEE Trans. on Comput.*, 1989. (in press).

[3] E. Chow, H. S. Madan, J. C. Peterson, D. Grunwald, and D. Reed, "Hyperswitch network for the hypercube computer," In *Proc. of 15th Annual Int'l Symp. on Computer Architecture*, pp. 90–99, 1988.

[4] J. W. Dolter, P. Ramanathan, and K. G. Shin, "A microprogrammable VLSI routing controller for HARTS," Technical Report Technical Report CSE–TR–12–89, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, 1989.

[5] R. W. Floyd, "Algorithm 97 : Shortest path," *Comm. ACM*, vol. 5, pp. 345, 1962.

[6] C. K. Kim and D. A. Reed, "Adaptive packet routing in a hypercube," In *Proc. Third Conf. on Hypercube Concurrent Computers and Applications*, Los Angeles, January 1988.

[7] J. G. Kuhl and S. M. Reddy, "Distributed fault tolerance for large multiprocessor systems," In *Proc. 7-th Annual Int'l Symp. on Computer Architecture*, pp. 23–30, 1980.

[8] T. C. Lee and J. P. Hayes, "Routing and broadcasting in faulty hypercube computers," In *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, pp. 625–630, Los Angeles, January 1988.

[9] K. S. Stevens, "The communication framework for a distributed ensemble architecture," Technical Report AI Technical Report 47, Schlumberger Research Lab., February 1986.

[10] A. Varma and C. S. Raghavendra, "Fault-tolerant routing of permutations in extra-stage networks," In *Proc. 6-th Int'l Conf. on Distributed Computing Systems*, pp. 54–61, 1986.