FAULT-TOLERANT ROUTING IN MESH ARCHITECTURES

Alan Olson Kang G. Shin

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

ABSTRACT

It is important for a distributed computing system to be able to route messages around whatever faulty links or nodes may be present. We developed a fault-tolerant routing algorithm which assures the delivery of every message as long as there is a path between its source and destination. The algorithm works on many common mesh architectures such as the torus and hexagonal mesh.

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 the system to know only the state (faulty or not) of each of its own links. The performance of the routing scheme is simulated for both square and hexagonal 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 if a path exists, is usually found very quickly.

Index Terms: Hexagonal mesh, distributed systems, routing, faulty links, cycles, incisions.

The work reported here is supported in part by the NSF under Grants MIP-9203895 and by the Office of Naval Research under Grant N00014-91-J-1115. 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.

1 Introduction

The processors of a distributed computing system communicate by sending messages over a network. Faults in the network can prevent the delivery of messages unless the network provides fault-tolerant routing. However, most distributed systems pay little attention to this potential problem. While they provide simple and efficient routing algorithms, the algorithms usually will not work properly if faults are present in the network. In this paper, we propose a simple and efficient fault-tolerant routing algorithm which can be used for many mesh-type distributed system architectures.

An obvious way to handle fault-tolerant routing is for each node to keep track of all faults in the system. A node can be expected to know the state (failed or not) of its own links, and some algorithms are proposed in [6] to broadcast information about faulty components to all other nodes in the system. With this information messages can always be routed by shortest paths. There are two main problems with this approach. First is the amount of memory that may be needed to store all this information, especially if the system is large. Second is the overhead it induces. The standard routing algorithms of most systems allow routing decisions to be made by simple circuitry using only information on the message header. This allows optimizations like virtual cut-through [4], which speed up message delivery by avoiding buffering at intermediate nodes. If other information must be consulted, the message must be buffered, and message delivery is delayed. We therefore restrict ourselves to the situation where each node knows only the state of its own links.

Some work has already been done on fault-tolerant routing in the hypercube [1, 3, 5, 7]. These algorithms either take advantage of the specific mathematical properties of the hypercube and are therefore inapplicable to meshes, or use some form of global information, which we want to avoid. The authors of [1] present an algorithm which does not use global information, but relies on properties specific to hypercubes and cannot tolerate more than n faults, where n is the dimension of the hypercube.

To date, we know of no other fault-tolerant routing strategy for mesh-type distributed systems systems. 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. 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 this in a finite amount of time.

This paper is organized as follows. In Section 2, we briefly describe two representative mesh architectures. Section 3 outlines a fault-tolerant routing algorithm which works in most cases.

In Section 4 the cases where the algorithm will not work are examined, and a fix is presented. In Section 5, the performance of the algorithm is simulated for several mesh types. The paper concludes with Section 6.

2 Homogeneous Mesh Architectures

Mesh architectures provide a number of advantages over other distributed architectures such as the hypercube. In a mesh, the number of links per node is constant and does not increase as the mesh size increases. The number of links per node in a hypercube increases with the number of nodes, and the high number of communication links and corresponding high through-traffic can overload the nodes of large systems. Also, the number of nodes in a mesh is a quadratic function of the mesh dimension, whereas that in a hypercube is an exponential function of the hypercube dimension.

The two mesh architectures we will consider in this paper are the torus and the hexagonal mesh. These meshes have several characteristics, common to most mesh architectures, that we must have in order for our routing algorithm to work. The mesh must be regular, if unwrapped it must be planar, and if wrapped it must be homogeneous. Each link has an associated vector, and one must be able to change the order of links (vectors) in a path without changing the destination. That is, one will arrive at the same node regardless of whether one takes two hops in the x direction followed by one hop in the y direction, or one hop in the x direction followed by one hop in the y direction followed by one hop in the x direction.

In addition, we assume that routing is done on a hop-by-hop basis, with routing decisions made by intermediate nodes, instead of having a path chosen by the node where the message originated. We assume the message system is based on a store-and-forward approach, which may employ techniques such as virtual cut-through [4] when possible in order to avoid the overhead of buffering the message at each individual node. Messages will be buffered at intermediate nodes if the outgoing links they wish to use are busy, and we assume each node has enough buffer space so that deadlock is not a problem. Our algorithm could be used for a circuit-switched environment, but we do not consider this possibility in this paper.

2.1 The Torus

An unwrapped square mesh of dimension $i \times j$ will contain ij nodes laid out in a rectangular grid, i nodes along the horizontal edge, and j nodes along the vertical edge. When wrapped, the square mesh becomes a torus: the right link of a node on the right edge of the mesh is connected to the node on the left edge which is in the same row, and the downward link of a

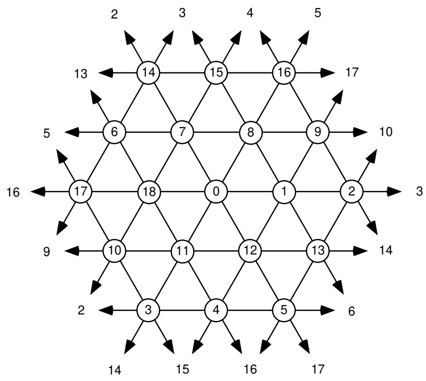


Figure 1: Hexagonal mesh of dimension 3

node on the bottom edge of the mesh is connected to the node on the top edge of the mesh which is in the same column.

Any node in the mesh will have four oriented directions. One can think of the right link as being the x *direction*, and the downward as the y *direction*. The left and top links then become the $-x$ and $-y$ directions. The set of shortest paths between any two nodes can then be expressed as offsets in these two directions. For example, the shortest paths between nodes 10 and 7 would be defined by offsets of 2 in the x-direction and -1 in the y-direction. Given these offsets, a message can be routed simply by sending it along a link which will reduce one of the offsets.

2.2 Hexagonal Mesh

An unwrapped 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 *edge dimension*, e, 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 $3e^2 - 3e + 1$.

Any node in an unwrapped H-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 remaining three links point in the $-x$, $-y$, and $-z$ directions, respectively. An H-mesh of edge dimension e can be wrapped using "C-type wrapping" [2], which produces a homogeneous mesh. An example of a C-wrapped H-mesh of edge dimension 3 is given in Fig. 1.

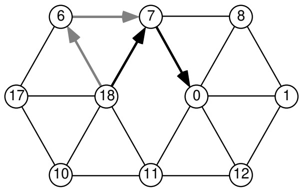


Figure 2: Detouring in a hexagonal mesh

A $\Theta(1)$ algorithm is presented in [2] to give all shortest paths between any two nodes in a C-wrapped H-mesh. It returns three integers, m_x, m_y, and m_z, each of which represents the distance to be traveled in the corresponding 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. As for the torus, routing can be done by simply sending the message along a link which will reduce one of the offsets.

3 Detouring

The structure of the system is an important factor in any routing algorithm. Fault-tolerant routing algorithms for the hypercube take advantage of that system's unique mathematical properties, it is only logical that we should take advantage of the simple, regular structure of mesh systems. We will only consider link failures, since a node failure can be modeled as the failure of all of its links.

The torus and H-mesh provide an obvious method for detouring around a single faulty link. Each link forms one side of a square or triangle whose other sides form a convenient detour should the link fail. As an example, consider the hexagonal mesh fragment in 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.

A message is in one of two modes: *free* mode for when no faults are obstructing the path, and *detour* mode for when the message's path is blocked by faults. The routing done at each node is as follows:

Algorithm FTRoute

1. If the message is in detour mode, and if the current node is closer to the destination than the node where it entered detour mode, put the message in free mode.

2. If the message is in free mode, select the set of links along shortest paths to the destination.

If the message is in detour mode, select the link immediately counter-clockwise of the link by which the message entered the node.

3. If any of the selected links are non-faulty, send the message along that link.

4. If all links out of this node have already been selected and tested, **halt**. Otherwise, select the link counter-clockwise of the set of previously selected links, and go to step 3.

Accounting for this algorithm is simple. The message need only keep track of the destination node, the current state (free or detour), and the distance to the destination when the message entered detour mode. One problem with the algorithm is that it will only halt if the message reaches its destination, or the current node has no non-faulty links. There is no way to detect an unreachable destination. Intuitively, it seems that a message with an unreachable destination will get into a cycle.

The following theorems show that if a message is attempting to reach an unreachable destination, it will form a cycle. They also show that the cycle will take place entirely in detour mode, and the cycle will include the node where the message last entered detour mode *and* the link by which it left that node. So detecting a cycle is as simple as remembering at what node the message entered detour mode and by which link it exited that node. If the message does not leave detour mode, returns to the node where it entered detour mode, and is about to leave that node by the same link it did before, it is in a cycle and will not reach its destination.

Lemma 1: Given message m, for any non-faulty link o of node n there will be non-faulty 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, if there is only one non-faulty link at this node then $i = o$. The rest of the proof then follows trivially from the description of the algorithm. □

Lemma 2: On any finite mesh, if a message never reaches its destination, it will eventually reach a point after which it will never return to free mode.

Proof: Any hop which a message makes while in free mode or which results in a transition from detour mode to free mode will reduce by one the minimum distance the message has been from its destination. Call such a hop a *reducing hop.* Any hop that is not a reducing hop must be made in detour mode. As the message starts a finite distance from the destination there will only be a finite number of reducing hops made by any message. A message that does not reach its destination will, in theory, undergo an infinite number of hops. Thus, there must be a last reducing hop in the path. All hops made by the message after the last reducing hop are not reducing hops, the message must be in detour mode. □

Given this, we can now show that such a message must cycle.

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

Proof: By Lemma 2, 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 1 it is clear that knowing the current node and the outbound link is enough to determine the future routing 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 1 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 a finite number of available 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 we show that the cycle must include the node where the message entered detour mode.

Theorem 2: If a message is in a cycle, it will return to the node at which the message entered detour mode, and will exit that node via the same link it did before.

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 1 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 node where the message entered detour mode and ℓ_k is the link by which it exited it. The theorem follows. □

With Theorem 2, we can now redefine step 3 of **FTRoute** to detect cycles and halt:

3. If all the selected links are faulty, go to step 4. If the message is in detour mode, entered detour mode at this node, and later left this node by the selected link, the message is in a cycle, **halt**. Otherwise, send the message on any selected, non-faulty link.

We should point out that while our fault-tolerant routing algorithm may cause message cycles, the presence of these cycles does not increase the likelihood of deadlock. There are several reasons for this. First, a message that "runs into its tail" will fit the conditions of Theorem 2, and will therefore be considered undeliverable and be discarded. Later we will consider modifications of our routing algorithm that will try to deliver some of the messages which fit the conditions of Theorem 2. Even with these modifications deadlock will still not be a problem since the message system is based on a store-and-forward system (but may employ virtual cut-through), so any message which runs into its tail may be buffered to allow its tail to catch up.

4 Cycles

Theorems 1 and 2 show that the routing algorithm will always terminate. It remains for us to show that it will not fail to deliver messages to reachable destinations. In this section we will consider the types of cycles which exist and what they imply with regards to the reachability of the destination.

4.1 Types of Cycles

Cycles are detected by checking the current node and outgoing link against the node where the message entered detour mode and the corresponding outgoing link. However, this tells us little about the cycle. More information can be gained by exploiting the directional properties of the links.

As was mentioned in Section 2, each link has a corresponding vector. Each vector will correspond to a distance (usually 1) in one of the mesh's available directions. For example, in a torus the right link will have a vector of length one in the x direction, and in an H-mesh the link 30 degrees counter-clockwise of vertical will have a vector of length one in the z direction.

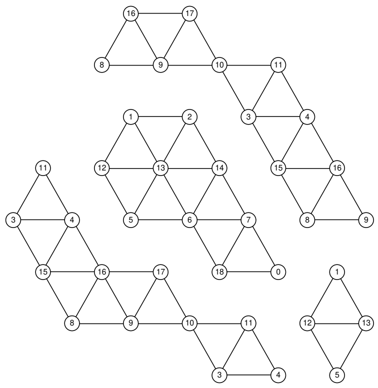


Figure 3: A circle

We define the *mesh vector sum* of a path as the sum, in each of the available directions, of the vectors of the individual links in the path, reducing where appropriate. For example, in the torus a path that goes right twice, up twice, and then left will have a mesh vector sum of $(1x, -2y)$. Reducing the sum is important in the H-mesh where, for example, $(1x, 0y, 1z)$ is equivalent to $(0x, 1y, 0z)$. If a mesh vector sum of the traversed links is kept with the message, cycles can be divided into two types.

The first kind of cycle is characterized by a zero mesh vector sum when the message returns to the node at which the cycle started. This kind of cycle is called 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. A circle indicates the mesh has become disconnected. It will be shown later that in the case of a circle the destination is not reachable.

The second kind of cycle is characterized by a non-zero mesh vector sum when the message returns to the node at which the cycle started. This kind of cycle is called an *incision*. Fig. 4 shows an example of an incision. If a message is sent from node 0 to node 1, it would head upward through nodes 8, 9, 17, 5, 12, and back to 0. An incision does not necessarily indicate the mesh has become disconnected. As this example shows the destination may in fact be reachable, the path $0 \to 7 \to 14 \to 2 \to 1$ contains no faults. Incisions are not possible in unwrapped meshes as the mesh vector sum of any cycle in a plane must be zero.

For the rest of this section we assume that the cycle encounters all faulty links, i.e., all faulty links are "checked" at least once during the course of the cycle. The presence of other faults doesn't affect the course of the cycle, nor will "fixing" them make any reachable destination unreachable.

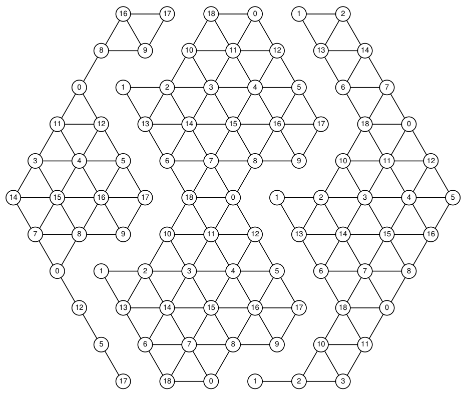


Figure 4: An incision

4.2 Circles

A circle indicates that the mesh has become disconnected and the message is tracing the perimeter of one of the connected components. It is easier to work with planar graphs, but if the graph is wrapped it may no longer be planar. However, we can take advantage of the fact that the unwrapped graph is planar, and the wrapped graph is homogeneous, to "unroll" the graph to form an infinite planar graph. The result is much like Fig. 4, only extended infinitely in all directions.

Theorem 3: If a message is in a circle, the destination is not reachable.

Proof: A Jordan curve is a continuous non-self-intersecting curve whose origin and terminus coincide. The union of the edges of a cycle in a planar graph will form a Jordan curve. The path of a circle will thus form a Jordan curve, except in cases where the same link is traversed twice. We can derive a true Jordan curve by connecting the midpoints of the faulty links in the order in which they are tested by the routing algorithm. This curve will run alongside the path of the cycle, and since it doesn't intersect itself it will be a true Jordan curve. It also has the property that all the links it intersects are faulty, and it does not pass through any nodes.

A Jordan curve will partition the nodes of a planar graph into two disjoint sets, internal nodes and external nodes, and any path between an internal node and an external node must cross the Jordan curve. For the unwrapped mesh we have a single Jordan curve, and two disjoint sets of nodes. For the wrapped mesh we have an infinite number of Jordan curves. For each curve, define the internal nodes to be the finite set of nodes, while the external nodes are those in the rest of the plane. Because all the links that cross a Jordan curve are faulty, no fault-free path can cross a Jordan curve. Thus, there are no fault-free paths between an internal node, and a node in the corresponding set of external nodes.

It only remains to show that the nodes on the cycle path, and the destination node, are in different sets. Assume the nodes on the cycle are internal nodes (the case when they are external nodes is similar). Consider the node on the cycle closest to the destination. Any link connected to this node on a shortest path to the destination must be faulty (this is what started the cycle). The nodes on the other ends of these links will be external nodes, since they will be on the other side of the Jordan curve. They will also be closer to the destination than any node on the cycle. Consider a shortest path from one of these nodes to the destination. There can be no faulty links in the path, if there were, one of the nodes connected to the edge must be in the cycle (since all faulty links are checked during the course of the cycle), and would therefore be closer to the destination than the node on the cycle which is closest to the destination. Since there exists a fault free path from an external node to the destination, the destination must be an external node, and there can be no fault-free path from any node on the cycle to the destination. □

4.3 Incisions

Theorem 3 assures us that if the message is in a circle there is no path to the destination. As we noted before, incisions are not possible in an unwrapped mesh, and in such systems **FTRoute** will always deliver a message to a reachable destination. As the example of Fig. 4 shows, this is not true for wrapped meshes. For such systems we must modify the algorithm to try alternate routes to the destination. The main difficulty is deciding which alternate routes should be tried. In this subsection we consider a method for selecting these alternatives.

Definition 1: The *endpoint vector* of a path is the mesh vector sum of the path. The *endpoint distance* of a path is the length of the endpoint vector.

Definition 2: The *characteristic vector* of an incision is the endpoint vector of a single trip through the cycle.

It may be easier to think of the characteristic vector as an actual vector (direction and length) in the infinite plane representation we used when discussing circles.

Lemma 3: If two incisions have different characteristic vectors, they will cross.

The proof of this lemma is obvious.

For the next theorem we need to introduce the concept of a subdivision of a graph. Graph G contains a subdivision of graph H if each node of H has a corresponding node in G and there exists a set of node-disjoint paths P in G such that for every link between two nodes of H, there is $p \in P$ where p is a path between the corresponding nodes in G. Essentially, the nodes of H are replaced by nodes in G, and the links of H are replaced by non-intersecting paths in G.

Theorem 4: If the faulty links that caused the message to go into an incision are removed from the graph, the resulting graph will be planar.

Proof: A graph is planar if and only if it does not contain a subdivision of either K_5 (the complete graph with 5 vertices), or $K_{3,3}$ (the complete bipartite graph with two sets of 3 vertices each). We will show that any subdivision of K_5 or $K_{3,3}$ that exists in the original mesh will cross the incision.

Consider K_5 first. At least two of the paths corresponding to edges of K_5 must use the wrap links of the mesh. In fact, there must be two incisions with non-parallel characteristic vectors in any subdivision of K_5. To see why, take any subdivision of K_5 in the complete mesh, and cut selected links until the resulting graph is an unwrapped mesh. Choose which links to cut so that the fewest possible links from the subdivision of K_5 are cut. The resulting graph will be an irregular mesh fragment much like that in Fig. 5, only without the wrap links. Because the resulting graph is unwrapped, no incisions are possible. However, because we cut as few links of the K_5 subdivision as possible, and circles are inherently planar, any circles in the subdivision of K_5 will be untouched. Our cutting has therefore disrupted only the incisions. A single incision, or a number of parallel ones, can be restored by wrap links similar to those in Fig. 5, while the graph remains planar. Since K_5 is non-planar, there must be two non-parallel incisions in the subdivision. A similar argument holds for $K_{3,3}$, there must be two non-parallel incisions in any subdivision of $K_{3,3}$.

Since there are two non-parallel incisions in any subdivision of K_5, at least one of these incisions must be non-parallel to the message incision. By Lemma 3 the message incision and the incision from the subdivision of K_5 must cross. Similarly for any subdivision of $K_{3,3}$, one of its incisions must cross the message incision.

Since any subdivision of K_5 or $K_{3,3}$ crosses the message incision, they must all contain faulty links. When the faulty links are removed, these subdivisions will no longer be possible. Since no subdivisions of K_5 or $K_{3,3}$ will be present in the mesh once the faulty links are removed, the resulting mesh must be planar. □

A planar embedding of the graph with the incision shown in Fig. 4 is shown in Fig. 5.

Theorem 4 assures us that if the message is in an incision, then the system graph is planar. It does not tell us the diameter of the graph. The resulting graphs will be very much like that in

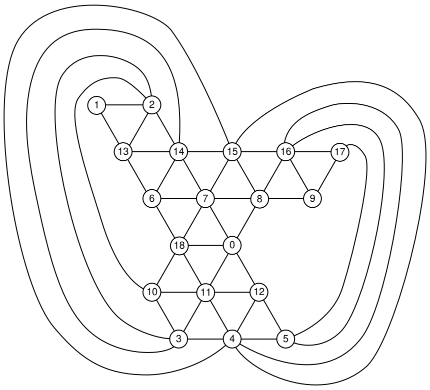


Figure 5: Planar embedding of a faulty H-mesh

Fig. 5, an irregular mesh fragment surrounded by wrap links. One "edge" of the mesh fragment is formed by the incision, the distance between the two edges may vary, but must be bounded by properties of the original wrapped mesh.

Definition 3: The *width* of a mesh with respect to incision i is the length of the shortest incision not parallel to i.

The width with respect to an incision is easy to calculate. In an $x \times y$ mesh it will be either $x + 1$ or $y + 1$, depending on which is smaller, and whether the incision is parallel to the x-axis, y-axis, or neither. In an H-mesh the width is always $2e - 1$, where e is the edge dimension of the mesh.

It is easily verified that the distance between the two edges of the mesh fragment will always be less than the width of the mesh with respect to the incision. So, if the width with respect to the incision is w, and there is a path to the destination, there is a path to the destination from one of the nodes on the cycle which has endpoint distance less than w. We can now modify step 3 of **FTRoute** to properly handle incisions.

3. If all the selected links are faulty, go to step 4. If the message is not in detour mode, or did not enter detour mode at this node, or did not later leave this node by the selected link, send the message on any selected, non-faulty link. Otherwise, the message is in a cycle. If the cycle is an incision, and the message has not been in an incision before, then:

   (a) Compute w, the width of the mesh with respect to the incision.

   (b) For each node on the cycle, find all endpoint vectors of length less than w corresponding to paths to the destination.

(c) Determine how many unique alternatives are contained within these vectors, that is, eliminate vectors which point to the same destination.

(d) Replicate the message, sending one copy to each alternative.

Since these copies will not be taking the shortest path to the destination they should not recompute the shortest path, but stick to the one they have.

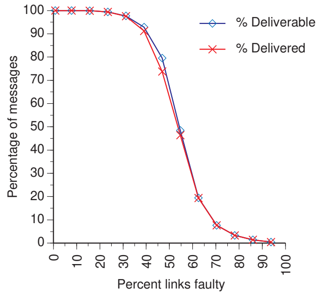
This modification guarantees all messages will reach their destination if the destination is reachable. However, this comes at a price. While the previous version of the algorithm can usually be implemented by dedicated hardware at the network interface, the work involved in the above modification will usually require the attention of the node's main processor. This causes considerable overhead and delay. Also, another flag must be maintained to tell whether or not the message is the original or a copy. This flag will affect both how the message is routed, and whether or not the message is discarded if it runs into an incision. In practice, it is unlikely that properly handling incisions will be worth the extra cost. As we show in the next section, incisions are rare, and occur only when a large number of failures are present.

5 Simulation

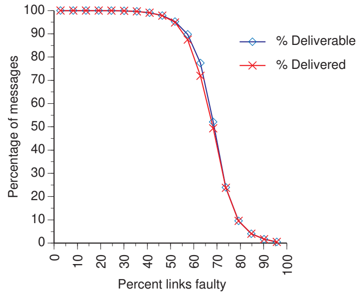
Simulation was used to determine the performance of **FTRoute**. Simulations were done for both the torus and the C-wrapped hexagonal mesh. The routing algorithm used was the simpler version, no effort was made to find alternate routes in the case of an incision. This was done partly because proper handling of incisions adds a great deal of complexity to the algorithm (and therefore to the simulator), and partly because of an intuitive notion that incision failures would be extremely rare. This intuition was borne out by our simulation results.

Our simulations were of link failures, and we assumed a uniform random distribution. While a node failure is equivalent to the failure of all of its links, the resulting distribution of link failures is non-uniform. We did some test simulations with node failures, and the results improved. This is not unexpected, with node failures the resulting mesh is more regular, with fewer of the dead ends and blind alleys found with link failures.

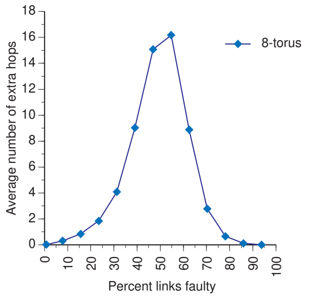
Simulation results for the 8-torus and 5-dimensional H-mesh are plotted in Fig. 6. The 8-torus is a 64 node system with 128 links. The 5-H-mesh is a 61 node system with 183 links. Figs. 6a and 6b plot the percentage of messages deliverable and the percent actually delivered against the percentage of links which were faulty. The upper line is the percentage of messages which were deliverable, i.e., the destination was reachable. The curve just below it is the messages which were actually delivered by the simple routing algorithm. Most of



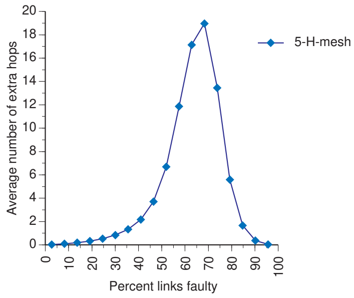
(a) Messages vs. percent faulty — 8-torus



(b) Messages vs. percent faulty — 5-H-mesh



(c) Mean extra hops vs. percent faulty — 8-torus



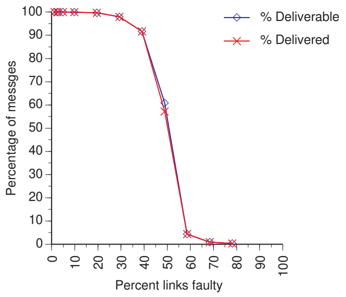
(d) Mean extra hops vs. percent faulty — 5-H-mesh

Figure 6: Algorithm performance on small systems

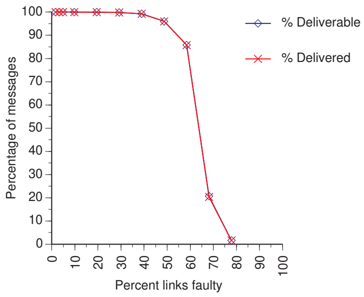
the time it is indistinguishable from the deliverable curve. Figs. 6c and 6d show the average difference in length between the path the routing algorithm took and the shortest path against the percentage of links which were faulty. It remains small for reasonable numbers of failures.

Results show that the simple algorithm works surprisingly well. For most of their length there is little difference between the %Deliverable and %Delivered curves, agreeing with our intuition that incision failures would be rare. The path lengths are encouraging also. The curve does not begin to climb steeply until nearly 20% of the links have failed. Note the advantage gained by the H-mesh with its 50 percent more links per node, the deliverable curve stays higher longer, and the extra hops curve stays lower longer than the corresponding curves for the torus.

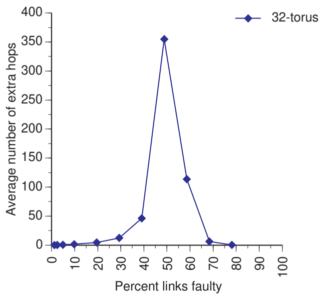
The results in Fig. 6 are good, but are for relatively small systems. We also ran simulations for a 32-torus (1024 nodes) and a 19-H-mesh (1027 nodes). The results of these runs are plotted in Fig. 7.



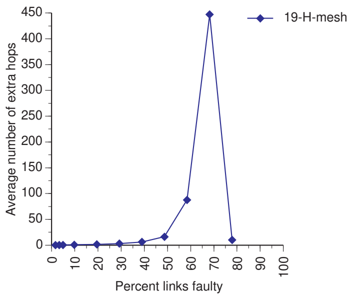
(a) Messages vs. percent faulty — 32-torus



(b) Messages vs. percent faulty — 19-H-mesh



(c) Mean extra hops vs. percent faulty — 32-torus



(d) Mean extra hops vs. percent faulty — 19-H-mesh

Figure 7: Algorithm performance on large systems

The graphs are much the same as for the smaller systems, and in some ways there is improvement with the larger systems. In Figs. 7a and 7b the %Deliverable and %Delivered curves are nearly indistinguishable, the slight gap between curves seen in Figs. 6a and 6b is not present. Also, it can be seen that the %Deliverable and %Delivered curves reach 0 faster than they did for the smaller systems. This is largely a result of mesh diameter, in larger systems there are more paths to the destination, but the paths are much longer, and therefore more likely to contain faults. The greater number of paths is more than offset by the increased probability that each path is faulty. In Figs. 7c and 7d we see that the peak of the extra number of hops curve has increased faster than the mesh size. This is not unexpected, since in a larger mesh there are more opportunities to turn down blind alleys, and is offset somewhat by the fact that the extra hops curves for the large meshes do not begin their rapid climb until much later.

6 Conclusion

In this paper we presented a fault-tolerant routing algorithm for use on multicomputers with mesh type interconnections. It works for both wrapped and unwrapped meshes, and simulations show that messages are delivered by near-minimal paths even in the presence of large numbers of link failures. In almost all cases a message will reach its destination if the destination is reachable, and in all cases if the message will not reach its destination the algorithm will determine this within a finite amount of time.

In some rare cases on wrapped meshes a message will not be delivered when its destination is reachable. This will not happen unless a fairly large number of faults are present. The algorithm can be extended, at some considerable expense in complexity, to properly handle these cases.

While we considered only square and hexagonal meshes in this paper, the algorithm should work in most other mesh types. We require only that the unwrapped version of the mesh be a planar graph, and that the wrapped mesh be homogeneous.

References

[1] M. S. Chen and K. G. Shin, "Adaptive fault-tolerant routing in hypercube multicomputers," *IEEE Trans. on Computers*, vol. 39, no. 12, pp. 1406–1416, December 1990.

[2] M. S. Chen, K. G. Shin, and D. D. Kandlur, "Addressing, routing and broadcasting in hexagonal mesh multiprocessors," *IEEE Trans. on Computers*, vol. 39, no. 1, pp. 10–18, January 1990.

[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] P. Kermani and L. Kleinrock, "Virtual cut-through: A new computer communication switching technique," *Computer Networks*, vol. 3, pp. 267–286, 1979.

[5] 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.

[6] 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.

[7] 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.