Distributed Fault-Tolerant Routing in Kautz Networks

Wei-Kuo Chiang and Rong-Jaye Chen

Department of Computer Science and Information Engineering
National Chiao-Tung University
Hsin Chu, Taiwan, R.O.C. 30050
E-mail: rjchen@csunix.csie.nctu.edu.tw

Abstract

For a Kautz network with faulty components we propose a distributed fault-tolerant routing scheme, called DFTR, in which each nonfaulty node knows no more than the condition of its links and adjacent nodes. We construct a rooted tree for a given destination in the Kautz network, and use it to develop DFTR such that a faulty component will never be encountered more than once. In DFTR, each node is attempting to route a message via the shortest path. If a node on the path detects a faulty node at the next hop, a best alternative path for routing the message around the faulty component is to be obtained. A best alternative path is first generated by the reduced concatenation of this node and the destination, and then is checked to make sure that it does not contain any of encountered faulty nodes. If it does, a new alternative path is generated as before. We invent an efficient approach in the checking step to reduce computational time. With slight modification, DFTR may adapt to de Bruijn networks[3] as well.

1 Introduction

The development of distributed systems improves resource utilization, reliability, and performance. A great many researchers proposed and analyzed different interconnection networks[1-16] for distributed systems. However, in a large network it is unrealistic to expect all the nodes along a specified path to be fault-free at all times. Thus it is necessary to incorporate fault-tolerant routing capabilities in the network. A number of important research results on the fault-tolerant routing in various networks have been available in the literature [2,9,12-14]. A few architectures are also proposed and shown to possess fault-tolerant routing capabilities[5,16].

In [9,12], a shortest fault-free path from a source node to a destination node is selected directly among the node-disjoint paths excluding the faulty components. In such cases, it is assumed that the source node has fault information of all other nodes in the network. On the other hand, in order to enable all fault-free nodes to correctly identify all faulty components in the network, various algorithms are proposed to broadcast the information about faulty components to all the other nodes in a network such that messages can be routed around the faulty components[12].

Clearly, each node can always find a shortest fault-free path for every message to its destination if the node contains the information on all faulty components. However, it is impractical to maintain and update such information, because it wastes traffic bandwidth to broadcast routing information to all other nodes and it will consume space by storing tables and directories at each node. In this paper, we will propose a distributed fault-tolerant routing scheme for Kautz networks based on Kautz digraphs[8]. With a little modification, the routing scheme may be adapted to de Bruijn networks[3], which has been studied quite extensively[1,4-7,9,12,16].

The paper is organized as follows. Definitions and properties of Kautz digraphs are presented in Section 2. We propose in Section 3 a distributed fault-tolerant routing algorithm. In Section 4 we compare the path length in our routing algorithm for faulty Kautz networks with that in another routing algorithm for faulty hypercubes in [2]. Finally, Section 5 concludes this paper.

2 Definitions and properties of Kautz digraphs

This section defines the Kautz digraph and summarizes properties of the Kautz digraph, which will be used later.

Definition 2.1 Let \(G=(V,A)\) be a digraph where \(V\) is a set of nodes and \(A\) is a set of (directed) arcs. For a node \(v\) in \(V\), the outdegree (indegree) is the number of nodes which are adjacent from (to) node \(v\). The degree of a digraph \(G\) is defined as the maximum among the outdegrees and indegrees of all nodes.

Definition 2.2 The distance from node \(u\) to node \(v\) is defined as the length of a shortest path from node \(u\) to node \(v\), where the length of a path is equal to the number of arcs encountered in the path. The diameter of a digraph \(G\) is defined as the maximum distance from any node to any other node. The connectivity of a digraph \(G\) is defined as the minimum number of nodes whose removal results in a disconnected or trivial digraph.

Definition 2.3 The Kautz digraph \(K(d,k)\) (defined in [8]) with degree \(d\) and diameter \(k\) is the digraph whose nodes are labeled with words \((u_1, ..., u_k)\), where \(u_i\) belongs to an alphabet of \(d+1\) letters \((0,1, ..., d)\), and \(u_i \neq u_{i+1}\), for \(1 \leq i \leq k-1\). The label of a node is also called the address of the node. For simplicity, we name a node with its address. There is an arc from a node \(u\) to a node \(v\) if and only if the
last \((k-1)\) digits of \(u\) are the same as the first \((k-1)\) digits of \(v\). In other words there is an arc from \((u_1 u_2 \ldots u_k)\) to \((u_2 \ldots u_x)\), where \(x\) can take any letter from the alphabet \(\{0,1,\ldots,d\}\) except \(u_0\). This digraph has \(d^k + d^{k-1}\) nodes. The Kautz digraph \(K(2,3)\) is shown in Figure 2.1.

![Figure 2.1 The Kautz digraph \(K(2,3)\).](image)

**Theorem 2.1** [15] The Kautz digraph \(K(d,k)\) has connectivity \(d\). Between any pair of nodes, there exist \(d\) node-disjoint paths, one of length at most \(k\), \(d-2\) of length at most \(k+1\) and one of length at most \(k+2\).

## 3 Distributed fault-tolerant routing

In this section we propose a distributed fault-tolerant routing scheme called DFTR for Kautz networks. Assume that each nonfaulty node is required to know no more than the condition of its own links and its adjacent nodes. Thus, the information of faulty components encountered before has to be added to the message in such a way as to route it around the faulty components.

Node failures are primarily considered here since the effect of a node failure is worse than a link failure. The paths affected by a link failure are a subset of the paths affected by the failure of one of the nodes connected by the link. Therefore, the routing and detour techniques will be adapted to link failures easily.

From Theorem 2.1, the Kautz \((d,k)\) network can tolerate up to \((d-1)\) node failures. Thus, if we can ensure a faulty node not to be encountered by a message more than once, and the number of faulty nodes is less than \(d\), then we are able to find a fault-free path between all pairs of fault-free nodes such that all nonfaulty nodes can communicate with each other in the network.

We sketch our fault-tolerant routing strategy as follows: Each node attempts to route messages via shortest paths first. Whenever the message encounters the faulty component and requires to be rerouted, we choose a best alternative path to be the shortest path among all the paths excluding the faulty components encountered before.

### 3.1 Main Algorithm

Self-routing can be accomplished in Kautz networks simply by the shifting of the digits of the destination label into the right end of the source label in the proper order[9]. The proposed routing technique can be implemented by using a distributed routing algorithm with the aid of a routing record. The routing record is attached to the message header. Any intermediate node on the path, by looking into the routing record, can precisely determine which node it should forward the message to.

When the message encounters a faulty node and needs to be rerouted, a more complicated procedure is required to produce the shortest path among all the paths excluding the faulty components encountered before on the way of the message to its destination. Then the message will be transmitted along the path. A formal description of the algorithm is given below.

**Algorithm DFTR:** When the message is received at node \(i = (i_1 i_2 \ldots i_d)\) on the path, including the source node, the following sequence of steps is carried out.

1. If the node address = the destination address then accept the message.
2. Detect the state of the next node
   - If the next node is fault-free then send the message to the next node; else (faulty)
     1. call Subroutine Route_Generation.
     2. call Subroutine Path_Finding.
3. end of the algorithm.

In Subsection 3.2 we demonstrate a procedure to generate routes in the order of paths by increasing length for Subroutine Route_Generation. In Subsection 3.3 we show how to keep the information about faulty nodes. The method is to be used in Subsection 3.4 to find a best alternative path for Subroutine Path_Finding.

### 3.2 Route generation

For ease of describing the distributed routing, we introduce the concatenation and the \(p\)-reduced concatenations of two nodes below.

**Definition 3.1** The concatenation of node \(u = (u_1 u_2 \ldots u_k)\) and node \(v = (v_1 v_2 \ldots v_k)\) is defined as \((uv)\). For \(1 \leq p \leq k\), the \(p\)-reduced concatenation of \(u\) and \(v\) is defined as \((uv)_p\) if \(u_k v_1 = 0\). The shortest concatenation of \(u\) and \(v\) is defined as \((uv)_{\infty}\).

**Definition 3.2** A route between \((u_1 u_2 \ldots u_k)\) and \((v_1 v_2 \ldots v_k)\) is defined as \(u_1 u_2 \ldots u_k v_{p+1} \ldots v_k\) for some \(p \geq 0\) and \(u_k v_{p+1} v_{p+2} \ldots v_k = v_1 v_2 \ldots v_p\) or \((u_1 u_2 \ldots u_k v_{p+1} \ldots v_k = v_1 v_2 \ldots v_k)\).

298
(u_3...u_kv_{p+1}v_{p+2}) \rightarrow ... \rightarrow (u_k...u_kv_{p+1}...v_k) = (v_1...v_k) or (u_1u_2...u_k) \rightarrow (u_2...u_kx_1) \rightarrow ... \rightarrow (x_qv_1...v_{k-1}) \rightarrow (v_1v_2...v_k).

Due to the structure of the Kautz network, any path in a Kautz network can be expressed as a route. And any reduced concatenation of node u and node v can specify a route \([u,v]\).

**Definition 3.3** For 0 \leq p \leq k, the \(p\)-reduced route of u and v is defined as \([u,v]_p = [u_1u_2...u_kv_{p+1}...v_k]\), which is the route induced by \((uv)_p\). Similarly, the shortest route of u and v is defined as \([u,v]^*\), which is the route induced by \((uv)^*\).

Using the terminologies above, we describe the subroutine below which generates the routes in the order of paths by increasing length.

**Description of Subroutine Route_Generation**

For each Kautz network, we may generate the routes with the concept of the reduced concatenations of any source-destination pair. A naive way to find an integer \(p\) such that the last \(p\) digits of the destination node are the same as the first \(p\) digits of the source node is as follows. We match the destination address \((t_1t_2...t_k)\) against the source address \((s_1s_2...s_k)\) by starting at the first digit of source address that matches \(t_1\) and continuing (comparing to \(t_2\) and so on) until we either complete the match or find a mismatch. In the latter case, however, we have to go back to the place from which we started and start again. We may have to backtrack and compare again a substantial number of times, leading to \(O(k^2)\) number of comparisons in the worst case. Notice that a lot of the work is redundant.

Our algorithm is obtained by modifying the string-match algorithm developed by Knuth, Morris, Pratt [10,11]. The preprocessing of destination address is the essence of the algorithm. We will study all the repeating patterns of destination address and devise a backtracking table to handle mismatches. There is an entry in the table for each digit in destination address corresponding to the amount of backtracking required when there is a mismatch involving this digit. This is important because we now can take advantage of all the matches done; none of them will be repeated. Additionally, when the matching is finished, we can consult the backtracking table to know the other \(p\)'s for \(p\)-reduced routes. The following is the formal description of the subroutine.

**Subroutine Route_Generation(RG)**

Input: Source \((s_1s_2...s_k)\) and Destination \((t_1t_2...t_k)\).

Output: All the indices \(p\)'s, where \(p\)-reduced route exists.

begin
1. preprocess the destination address to compute the backtracking table.
   Table[] := max \(j\) s.t. \(t_{j-1}t_{j-1}...t_{j-1} = t_1t_2...t_3; 2. j := 1;
   for \(i := 1\) to \(k\) do
   if \(s_i = t_j\) then \(j := j + 1\), \(i := i + 1\)
   else \(j := Table[j] + 1\);
   /* if mismatch, consult the table to backtrack */
endfor;
3. \(i := 1;\)
   \(j := j - 1;\)
   while \(j > 1\) do
   \(P[i] := j;\)
   /* all possible \(p\)'s for \(p\)-reduced routes generated */
endwhile
if \(P[i-1] \neq 1\) then \(P[i] := 0\) /* if \(s_k \neq d_1^*\*/
end of subroutine.

Note that the algorithm takes \(O(k)\) comparisons, where \(k\) is the digit number of the address in the network.

**3.3 How to keep information about faulty nodes**

In order to determine whether there exists the address of any encountered faulty node inside the new route, a naive way is to match the route against each faulty address from left to right digit-by-digit. By using the string-match algorithm in [11], this procedure needs \(O(kf)\) number of comparisons, where \(f\) is the number of the faulty nodes.

There is, however, a more effective method to determine the result. We will show how to construct a rooted tree for a given destination node to explain the method. We denote the rooted tree for destination node \(t\) by \(RT(t)\), which is described as follows. To avoid ambiguity, we call a node in a rooted tree "vertex", while calling a node in a Kautz network "node".

**Figure 3.1** The rooted tree for destination \((t_1t_2...t_k)\).

Given any destination node \(t = (t_1t_2...t_k)\) in a Kautz\((d,k)\), we configure \(RT(t)\), beginning with \((t_1t_2...t_k)\) as the root, as shown in Figure 3.1. Any vertex \(u = (u_1u_2...u_k)\) in the \(RT(t)\) is connected to \(d\) children which can be represented as \((xu_1u_2...u_{k+1})\) where \(x\) is any integer from 0 to \(d\) other than \(u_1\). Note that the arrowheads of the arcs are pointing upward, because the children of a vertex
in the RT(t) are the predecessors of that node in the corresponding Kautz network.

Definition 3.4 The level of a vertex in a rooted tree is defined as the length to the root.

Lemma 3.1 Each node in Kautz(d,k), is represented by at least a vertex at level \( \leq k \) in RT(t).

Proof: From the diameter \( k \) of Kautz(d,k), we know that there exists at least one path of length at most \( k \) from any source to any destination. It is easy to see that for any node in a Kautz(d,k), the shortest distance between its corresponding vertex and the root of the tree is equal to or less than \( k \).

From the construction of the RT(t), the following lemmas can be easily derived, and hence, the proof is omitted.

Lemma 3.2 For each vertex in RT(t), its children are all distinct.

Lemma 3.3 In RT(t) where \( t=(t_1 t_2...t_k) \), the last \( (k-n) \) digits of each vertex at level \( n \), \( 1 \leq n \leq k-1 \), are \( t_1 t_2...t_k \).

Theorem 3.1 In RT(t), for each \( m \leq k \), all the vertices at level \( m \) present the distinct nodes in Kautz(d,k).

Proof. We prove it by contradiction. Suppose to the contrary that there exists one node appearing at least twice at the same level in RT(t). Without any loss of generality, let us assume that the root of the RT is \( (t_1 t_2...t_k) \) and node \( (x_1 x_2...x_i t_1 t_2...t_k) \) appears in two distinct places at level \( i \), \( 1 \leq i \leq k \). For convenience, we denote the two vertices at different places by \( (x_1 x_2...x_i t_1 t_2...t_k)^1 \) and \( (x_1 x_2...x_i t_1 t_2...t_k)^2 \). Based on Lemmas 3.2 and 3.3, the fathers of the two vertices are identical, but they do not appear at the same places in the tree, which can be represented as \( (x_2...x_i t_1 t_2...t_k t_{i+1})^1 \) and \( (x_2...x_i t_1 t_2...t_k t_{i+1})^2 \). By repeating the same reason, we find that there are two identical vertices at level \( 1 \), which contradicts Lemma 3.2.

Any node in a Kautz network appears more than once in the corresponding RT(t). In order to simplify our representation, we give the following definition.

Definition 3.5 In RT(t), the representing vertex of a node which is nearest to the root is called the best vertex of the node.

To reduce the amount of information added to each message for routing around the faulty components, we will give the concept of relative position between two nodes in the RT(t). We first introduce some terminologies used to represent the relative positions.

Definition 3.6 For destination \( t \), the left tag of node \( u \) is defined as the \([u,t]^*\) excluding the part of node \( t \). That is, if \([u,t]^*=[u_1 u_2...u_k p_1 t_1 t_2...t_k]\), the left tag of node \( u \) is \([u_1 u_2...u_k p]\). The digit number of the left tag of a node is equal to the level number of its best vertex in the RT(t). Similarly, for any two nodes \( u \) and \( v \), the right tag of node \( v \) with respect to node \( u \) is defined as the \([u,v]^*\) excluding the part of node \( u \). That is, if \([u,v]^*=[u_1 u_2...u_k v_{p_1} v_{p_2} v_{p_3}...v_k]\), the right tag of node \( v \) with respect to node \( u \) is \([v_{p+1} v_{p+2}...v_k]\). The digit number of a right tag represents the distance from node \( u \) to node \( v \).

From Theorem 3.1, the best vertex of each node is unique in the RT(t). We can use the left tags of the faulty nodes to represent the relative positions of their best vertices with respect to the root in the RT(t). Furthermore, for the case the faulty node on the route is at its best vertex in the RT(t), it is more efficient to check the new route to see whether it contains any faulty node encountered before, for we need to compare only the corresponding digits of the route with their left tags as follows.

Let the new route be represented as \([r_1 r_2...r_k-r_x...r_{k-x} t_1 t_2...t_k]\), and the left tag be \([l_1 l_2...l_i]\), \( 1 \leq i \leq k-1 \). Then we compare \([r_1 r_2...r_k-r_x...r_{k-x} l_1 l_2...l_i]\) from right to left digit-by-digit. If they match, we continue to compare the next corresponding digits. If they do not match, we check with the left tag of another detected faulty node. The above steps are called left_mark_checking.

However, if there is an encountered faulty node on the alternative path, but it is not at its best vertex in RT(t), then the fault cannot be detected by using only the left tag of the node. It is because that the left tag of a node records only the best vertex of the node.

In order to solve the above problem, we will continue to develop some characteristics of Kautz networks from RT(t) for the case that the position of a detected faulty node on the new route is not at its best vertex in the RT(t).

For any node \( u \) in Kautz(d,k), although node \( u \) appears at distinct positions in the RT(t), yet the subtrees below these positions are identical because the subtree below vertex \( u \) can be viewed as RT(u) beginning with node \( u \) as the root. Hence, for any node \( u \) in Kautz(d,k), the relative position of any of its descendants with respect to node \( u \) is fixed whether or not node \( u \) is at its best vertex in any RT(t).

During a message routing, if the next node is detected being faulty, the current node is forced to reroute the message around it. To facilitate our presentation, the current node is called the rerouting node.

From the above observations, we know that regardless that the faulty node encountered before is either at its best vertex or not in the RT(t), the right tag of the detected faulty node with respect to the rerouting node is fixed at all times. However, these right tags computed at the previous rerouting node become out of date when the rerouting node changes from one to another. Thus, we add only the left tag(s) of the encountered faulty node(s) to the message to keep track of such information.

3.4 Finding a best alternative path

Let us denote the sets of left and right tags by Left_Tags and Right_Tags respectively. First, we propose an approach to reduce computation time in checking the new route with Left_Tags. The idea is to check the route with Left_Tags in the order of left tags by increasing the digit number. Recall the concept of left tag
in the foregoing. The digit number of a left tag represents the
distance from that faulty node to the root, i.e., the
level of its best vertex. We can find that the shorter the
distance from a node to the root (destination) the worse
effect results when the node is faulty, because the subtree
below the node cannot be used to route the message.
Likewise, we check the new route with Right Tags from
a right tag which has the minimum digit(s). But we
compare the corresponding digits of the route with the
right tag from left to right digit-by-digit. These steps are
called right_mark_checking.

Description of Subroutine Path-Finding(PF)
The subroutine is invoked when the node detects that
the next node for the routing message is faulty. According
to Definition 3.6, it extracts the left tags of the
encountered faulty nodes, the left and right tags of the next
directly from the routing record. Next, we consider
how to obtain a best alternative path.

From Subroutine Route_Generation, we can obtain
routes of the possible paths in the order of paths by
increasing length. Note that the new successful route
must not be the shortest path. Thus, this checking
begins with the second shortest path. We compare the
new route with Left_Tags and current Right_Tags by
left mark checking and right mark checking, respectively.
By the method we can determine if the message
will encounter any detected node on the route. If not, we
can attach new Left_Tags to the routing record of the
message, and then send the message to the next node
along the new route; otherwise, we need to extract the
right tag of the detected faulty node directly from the
route, adding it to Right_Tags. Then the above steps for
checking other routes should be repeated until we can get a
route excluding any detected faulty node.

If there is no more reduced concatenation in the above
step to produce an alternative path, we need insert a digit
between the labels of the rerouting node $(t_1 t_2... t_k)$ and
the destination node $(t_1 t_2... t_k)$, where $x$ can take any value
from the alphabet $\{0, 1,..., d\}$ except $d$, and. From
Theorem 2.1, in order to successfully route messages as
long as the number of faulty nodes is less than $d$, in the
worst case we may use the similar method to insert two
digits between them. More formally, this subroutine is
described in algorithmic form as follows.

Subroutine Path-Finding(PF): This subroutine is
invoked by Algorithm DFTK at the rerouting node
$(t_1 t_2... t_k)$ when the routing message encounters a faulty
node at the next hop.

begin
1. (information about encountered faults.)
   1.1 load Left_Tags from the routing record
   1.2 extract the left and right tags of the next node,
      and add them to Left(Right)_Tags.
2. (produce the new route in the order of
   paths by increasing length.)
   $i := i + 1$; // the $i$-th shortest route*/
   position := P[i]; // P[i]-reduced route*/
   if position < 0 then
     // to insert digit(s) is required */
     if one insertion digit $x$ is used
       then position := $-1$;
     else insert two digits $x_1 x_2$, position := $-2$;
   endif
   endif
   if position > 0 then
     new route := $r_1 r_2 ... r_k^{\text{position} + 1} - d_k$
   else new route := $r_1 r_2 ... r_k + x (x_1 x_2) + d_1 ... d_k$;
3. (checking for the new route.)
   call left_mark_checking;
   if Found(Faulty)
     then added to Right_Tags, goto 2;
   else call right_mark_checking;
     if Found(Faulty) then goto 2;
     else new route & Left_Tags attached to the
         message;
     endif
   endif
end of the subroutine.

Proof of the correctness of Main Algorithm
Theorem 3.2 If there is an encountered faulty node
existing on the new route, it will be detected by Left_Tags
or current Right_Tags which have been obtained up to
now in this rerouting node.

Proof. We prove it by contradiction. Suppose to
the contrary that there is a faulty node encountered before
cannot be detected, then this node must not be at its best
vertex and its right tag hasn't been obtained. Therefore,
there exists at least a vertex nearer to the destination (root)
in the corresponding RT(i) for the node, such as the best
vertex. However, an alternative path is generated in the
order of paths by increasing length. This implies that the
route which contains the same faulty node had been
chosen before. Thus the right tag of this node must have
been obtained, which contradicts the assumption. □
As far as computational complexity is concerned, the
algorithm may not promote efficiency very much. But
we use the tree structure of Kautz digraphs to save much
redundant checking, thereby reducing computational time.

Additionally, the algorithm can be used easily for the
case of link failures. Links faults can be treated the same
way as node faults; i.e., if the link from node u to v is
faulty, node u is regarded as a faulty node to record the left
tag of the link. However, when the algorithm needs to
obtain the right tag of a faulty link, we regard node v as a
faulty node.

3.5 An example
The following example illustrates how our routing
algorithm works in the presence of faults. Let node (120)
and node (310) be the source and destination nodes for a
message in Kautz(3, 3). Suppose the faulty nodes are node (031) and node (131). We construct the corresponding RT(310) for explanation in Figure 3.2.

In the beginning, we have the shortest route [120310] from (120) to (310). Then Algorithm DFTR transmits the message along the shortest path (120) → (203) → (031) → (310). Unfortunately, node (031) is fault, hence, we use Subroutines RG and PF to find a best alternative path at node (203). Additionally, the left and right tags of the faulty node (031), (0) and (1), are added to Left_Tags and Right_Tags. Since there is no more reduced concatenation, we will produce the new route by inserting one digit between the addresses of the rerouting and destination nodes. We may choose the insertion digit x = 0 or 1. But these new routes [20302110] and [2031310] including the detected node (031) will be detected after all left mark checking and right mark checking, respectively. Thus, x = 2 is the only candidate, and the route of the best alternative path is [2032310]. The message will be routed from node (032) to the destination node (310) via the nodes (323) and (231). However, since node (231) is faulty, node (323) needs to reroute the message around the fault.

Again, we need to find a best alternative path at the new rerouting node (323). As mentioned earlier on, we try to choose insertion digit x to generate the new route, but we find that it is impossible to construct a best alternative path if the number of insertion digit is one. Thus, we must insert two digits between two nodes (323) and (310). Finally, we can obtain two best alternative paths [32301310] and [32321310], which can be represented as below:

Path 1: (323) → (230) → (031) → (131) → (310), and
Path 2: (323) → (232) → (231) → (213) → (131) → (310).

We may send the message along the Path 1 or Path 2, and then the message will reach the destination node (310). Figure 3.2 shows the successive steps.

Figure 3.2 The successive steps of our routing algorithm for the example.

4 Performance analysis

In this section we will compare the length of the path in Algorithm DFTR for faulty Kautz networks with that in Algorithm AI for faulty hypercubes[2]. Algorithm DFTR ensures a faulty node not to be encountered more than once, and thus the length of the resulting path is bounded as follows.

Theorem 4.1 Suppose there are f faulty nodes in Kautz(d, k) network, where d ≥ 3, k ≥ 2 and 1 ≤ f ≤ d−1, and Algorithm DFTR are used for routing a message from any source node to any destination node. Then, the length of the resulting path in the worst case is as follows.

1) \(1 \leq f \leq d-2\) : \((k-1) \cdot (f-1) + 2k - 1\).
2) \(f = d - 1\) : \((k-1) \cdot d + 2\).

Proof. From Theorem 2.1, between each pair of nodes, there are d node-disjoint paths, one of length at most \(d-2\) of length at most \(k+1\) and one of length at most \(k+2\). Thus, the message passes at most \(k-2\) nodes before the first faulty node is encountered. After this, the message needs at most \(k-1\) steps to encounter any other faulty node. Finally, we can route the message to the destination through at most \(k+f\) nodes if the number of faulty nodes \(f\) is less than or equal to \(d-2\). However, if the number of faulty nodes encountered before is \(d-1\), the message needs at most \(k+2\) steps to reach the destination node.

From the above observation, we can figure out the steps needed at most as follows.

1) \(1 \leq f \leq d-2\) :
   \((k-2)+(k-1)+(f-1)+(k+1)-(f-1)+(f-1)+2k-1\);
2) \(f = d - 1\) :
   \((k-2)+(k-1)+(d-2)+(k+2)-(k-1)+d+2\).

The hypercubes are interesting because they have nice properties like symmetry, easy routings, and high fault tolerance. The n-dimensional hypercube has \(N=2^n\) nodes. In [2], M-S Chen and K.G. Shin proposed a distributed fault-tolerant routing scheme for a hypercube, called Algorithm AI, in which each node is required to know only the condition of its own links and its adjacent nodes just as the assumption of our algorithm.

The following theorem, derived in [2], bounds the path length in Algorithm AI for faulty hypercubes.

Theorem 4.2 Suppose \(f\) faulty nodes are encountered for routing a message from any source node to any destination node by Algorithm AI. Then, the length of the resulting path is at most \(m + 2f\), where \(m\) is the length of the shortest path between the two nodes.

Since both the connectivity and diameter of the hypercube are \(n\), from Theorem 4.2 we know that the path length in the worst case is \(n + 2(m-1)\).

We now compare Algorithm DFTR for faulty Kautz networks with Algorithm AI for faulty hypercubes in terms of the path length in the worst case. The worst case occurs when the number of faulty nodes is equal to which the network can tolerate up to.
The result of comparison is given in the right hand-side of Table 4.1. Since both hypercubes and Kautz networks have the restriction on the number of nodes, we cannot compare them on the same size of the network. Hence, we choose a Kautz network with the number of nodes and connectivity of Kautz network equal to or more than those of a hypercube. We find the length of the resulting path in the worst case by our algorithm in Kautz networks is less than that by Algorithm $A1$ in hypercubes even when the number of nodes in Kautz networks are much more than that in hypercubes.

Table 4.1
Comparisons of Kautz Networks and Hypercubes.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Connectivity</th>
<th>Diameter</th>
<th>Path length at worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypercube</td>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Kautz</td>
<td>20</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Hypercube</td>
<td>32</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Kautz</td>
<td>42</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Hypercube</td>
<td>64</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Kautz</td>
<td>72</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Hypercube</td>
<td>128</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Kautz</td>
<td>292</td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td>Hypercube</td>
<td>256</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Kautz</td>
<td>576</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>Hypercube</td>
<td>512</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Kautz</td>
<td>810</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Hypercube</td>
<td>1024</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Kautz</td>
<td>1100</td>
<td>3</td>
<td>22</td>
</tr>
</tbody>
</table>

5 Conclusion

Although Kautz digraphs lack incremental expandability, yet possess many desirable properties suitable for distributed computing systems. The Kautz digraph always gives the largest connectivity and the smallest diameter. In this paper, we present a distributed fault-tolerant routing algorithm for Kautz networks with faulty components. The routing algorithm does not require any table lookup mechanism.

In order to ensure a faulty component not to be encountered more than once, a novel method is proposed to obtain relative tags of faulty components. As a result, it becomes much more efficient to determine whether an alternative path contains any encountered faulty component by checking only the corresponding digits on the new route with these relative tags.

We also compare the length of the path in Algorithm $DFTR$ for faulty Kautz networks with that in Algorithm $A1$ for faulty hypercubes[2]. And the comparison shows our algorithm has shorter path length in the worst case.

References