JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 20, 99–106 (1994)

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, Hsinchu, Taiwan, 30050, Republic of China

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 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 a slight modification, *DFTR* may adapt to de Bruijn networks as well. © 1994 Academic Press, Inc.

1. INTRODUCTION

The development of distributed systems improves resource utilization, reliability, and performance. A great many researchers proposed and analyzed different interconnection architectures [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 are 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 faults to all the other nodes in a network such that messages can be routed around the faulty components [12].

* To whom all correspondence should be addressed.

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 propose a distributed fault-tolerant routing scheme in 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 compute the path length at the worst case in our routing algorithm for faulty Kautz networks. 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.

DEFINITION 1. The *outdegree* (*indegree*) of a node is the number of nodes which are adjacent from (to) the node. The *degree* of a digraph G is defined as the maximum among the out-degrees and in-degrees of all nodes. A digraph G is called a *d-regular* digraph if the out- and in-degrees of every node are equal to d.

DEFINITION 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 3. The Kautz digraph *Kautz* (d, k) (defined in [8]) with in- and out-degree d and diameter k is the digraph whose nodes are labeled with words $(u_1 \ldots u_k)$, where u_i belongs to an alphabet of $d + 1$ letters $\{0, 1, \ldots, d\}$, and $u_i \neq u_{i+1}$, for $1 \leq i \leq k - 1$. The label of a node

0743-7315/94 $5.00

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 $(u_1u_2 \ldots u_k)$ to $(u_2 \ldots u_kx)$, where x can take any letter from the alphabet $\{0, 1, \ldots, d\}$ except u_k. This digraph has $d^k + d^{k-1}$ nodes.

THEOREM 1 [15]. *The Kautz digraph Kautz (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 $k+2$.*

The Kautz network is based on Kautz digraph [8], which possess the attractive features below for diverse applications.

1. Minimal maximum message routing delay: The lower bound of the diameter for d-regular digraph with n nodes is $\lceil \log_d(n(d-1)+1) \rceil - 1$. The Kautz digraph *Kautz* (d, k) has $d^k + d^{k-1}$ nodes; its diameter k reaches the lower bound $\lceil \log_d((d^k + d^{k-1})(d-1)+1) \rceil - 1$.
2. Optimal fault tolerance: It is known [4, 6] that Kautz networks of degree d are optimally fault-tolerant, and failure of any $(d-1)$ components is tolerated.
3. A very appropriate candidate to construct a large network: In designing network topologies using *Kautz* (d, k), d can be chosen to match the degree of fault tolerance required. And the number of nodes N is determined by the chosen diameter k, by the relation $N = d^k + d^{k-1}$.

3. DISTRIBUTED FAULT-TOLERANT ROUTING

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

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 on the way of the message to its destination.

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. Finally, we demonstrate the correctness of our algorithm and give an example to illustrate how it works.

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 generated at the node where the message is originated, and 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 obtain a best alternative path. A formal description of the algorithm is given below.

ALGORITHM *Distributed_Fault_Tolerant_Routing(DFTR).*

begin
step **1.** **If** the node address = the destination address in the routing record,
then accept the message;
else according to the routing record in the message header,
precisely determine which node it should forward the message to.
step 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)
step **2.1. call Subroutine *Route_Generation.***
step **2.2. call Subroutine *Path_Finding.***
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 concatenation of two nodes below.

DEFINITION 4. The *concatenation* of node $u = (u_1u_2 \ldots u_k)$ and node $v = (v_1v_2 \ldots v_k)$ is defined as $(uv) = (u_1u_2 \ldots u_kv_1v_2 \ldots v_k)$. For $1 \le p \le k$, the *p-reduced concatenation* of u and v is defined as $(uv)_p = (u_1u_2 \ldots u_kv_{p+1} \ldots v_k)$ if $u_{k-p+1}u_{k-p+2} \ldots u_k = v_1v_2 \ldots v_p$, i.e., a suffix of node u is equal to a prefix of node v. For convenience, we define $(uv)_0$ as (uv) if $u_k \neq v_1$. The *shortest concatenation* of u and v is defined as $(uv)^* = (uv)_p$ with the largest possible p.

Due to the structure of the Kautz network, any path in a Kautz network can be expressed as a route. Any reduced concatenation of node u and node v can specify a route $[u, v]_p$ if $(uv)_p$ exists.

DEFINITION 5. A *route* is defined as $[u_1u_2 \ldots u_kv_{p+1} \ldots v_k]$ for some $p \ge 0$ and $u_{k-p+1}u_{k-p+2} \ldots u_k = v_1v_2 \ldots v_p$ or $[u_1u_2 \ldots u_kx_1x_2 \ldots x_qv_1v_2 \ldots v_k]$ for some $q \ge 1$, which represents a path from node u to node v as $(u_1u_2 \ldots u_k) \rightarrow (u_2 \ldots u_kv_{p+1}) \rightarrow (u_3 \ldots u_kv_{p+1}v_{p+2}) \rightarrow \cdots \rightarrow (u_{k-p+1} \ldots u_kv_{p+1} \ldots v_k) = (v_1v_2 \ldots v_k)$ or $(u_1u_2 \ldots u_k) \rightarrow (u_2 \ldots u_kx_1) \rightarrow \cdots \rightarrow (x_qv_1 \ldots v_{k-1}) \rightarrow (v_1v_2 \ldots v_k)$.

DEFINITION 6. For $0 \le p \le k$, the *p-reduced route* of u and v is defined as $[u, v]_p = [u_1u_2 \ldots u_kv_{p+1} \ldots v_k]$, 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 $(uv)^*$.

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. Our method 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 $(t_1t_2 \ldots t_k)$ against the source address $(s_1s_2 \ldots s_k)$ by the algorithm which is obtained by modifying the string-match algorithm developed by Knuth and co-workers [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. When the matching is finished, we get the largest p for the shortest route. Additionally, we can consult the backtracking table to know the other p's for p-reduced routes. The following is the formal description of the algorithm.

SUBROUTINE *Route_Generation* (*RG*).

Input: Source $(s_1s_2 \ldots s_k)$ and Destination $(t_1t_2 \ldots t_k)$.
Output: All the indices p's, where p-reduced route exists, $p \ge 0$. The values (p's) are stored in $P[i]$.

```
begin
  step 1. preprocess the destination address to compute the backtracking table.
          Table[i] := max j such that t_{i-j}t_{i-j+1} ... t_{i-1} = t_1t_2 ... t_j.
  step 2. j := 1;
          for i := 1 to k do      //* k is the digit number of the address *//
            if s_i = t_j          //* if matches then continues *//
            then j := j + 1, i := i + 1
            else j := Table[j] + 1; //* consult the table to backtrack *//
          endfor;
  step 3. i := 1; j := j - 1;
          while j > 1 do      /*all p-reduced routes generated by consulting the table*//
            P[i] := j; i := i + 1;
            j := Table[j + 1];
          endwhile
            if P[i - 1] ≠ 1 then P[i] := 0      //* if s_k ≠ d_1*//
end of subroutine.
```

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

3.3. How to Keep Information about Faulty Nodes

From Theorem 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. To ensure a faulty node not to be encountered by a message more than once, we store information about encountered faulty nodes into the message.

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

There is, however, a more effective method to determine the result. First, we show how to construct a rooted tree for the 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."

Given any destination node $t = (t_1t_2 \ldots t_k)$ in a *Kautz* (d, k), we configure $RT(t)$, beginning with $(t_1t_2 \ldots t_k)$ as the root, as shown in Fig. 1. Any vertex $u = (u_1u_2 \ldots u_k)$ in the $RT(t)$ is connected to d children which can be represented as $(xu_1u_2 \ldots u_{k-1})$ 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 7. The *level* of a vertex in $RT(t)$ is defined as the length to the root.

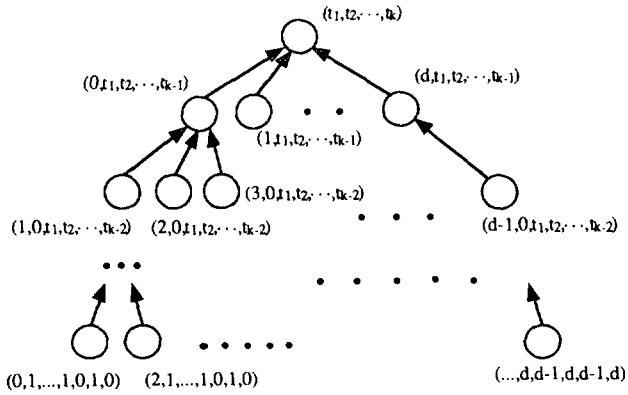

FIG. 1. The rooted tree for destination $(t_1t_2 \ldots t_k)$.

The following lemmas can be easily derived, and hence, the proof is omitted.

LEMMA 1. *For each vertex in* $RT(t)$, *its children are all distinct.*

LEMMA 2. *In* $RT(t)$ *where* $t = (t_1t_2 \ldots t_k)$, *the last* $(k - n)$ *digits of each vertex at level* n, $1 \le n \le k - 1$, *are* $t_1t_2 \ldots t_{k-n}$.

THEOREM 2. *In* $RT(t)$, *for each* $m \le k$, *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 $(t_1t_2 \ldots t_k)$ and node $(x_1x_2 \ldots x_it_1t_2 \ldots t_{k-i})$ appear in two distinct places at level i, $1 \le i \le k$. For convenience, we denote the two vertices at different places by $(x_1x_2 \ldots x_it_1t_2 \ldots t_{k-i})^1$ and $(x_1x_2 \ldots x_it_1t_2 \ldots t_{k-1})^2$, respectively. Based on Lemmas 2 and 1, the fathers of the two vertices are identical, and they also appear in two distinct places in the tree, which can be represented as $(x_2 \ldots x_it_1t_2 \ldots t_{k-i}t_{k-i+1})^1$ and $(x_2 \ldots x_it_1t_2 \ldots t_{k-i}t_{k-i+1})^2$. By repeating the same reason, we find that there are two identical vertices at level 1, which contradicts Lemma 1. ■

COROLLARY. *Given any two nodes* u *and* v *in a Kautz* (d, k), *for every* $m \le k$, *there exists at most one path of length* m *from* u *to* v.

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

DEFINITION 8. 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 9. For destination t, the *left tag* of node u is defined as the $[u, t]^*$ excluding the part of node t. That is, if $[u, t]^* = [u_1u_2 \ldots u_{k-p}t_1 \ldots t_k]$, the left tag of node u is $[u_1u_2 \ldots u_{k-p}]$. 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 $[u, v]^* = [u_1u_2 \ldots u_kv_{p+1} \ldots v_k]$, the right tag of node v with respect to node u is $[v_{p+1}v_{p+2} \ldots v_k]$.

From Theorem 2, 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 the left tags of these encountered faulty nodes as follows.

Let the new route be represented as $[r_1r_2 \ldots r_{k-x}t_1t_2 \ldots t_k]$, and the left tag be $[f_1f_2 \ldots f_i]$, $1 \le i \le k-1$. Then we compare $r_1r_2 \ldots r_{k-x}$ with $f_1f_2 \ldots f_i$ 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 a faulty node encountered before on the new route, and it is not at its best vertex in the corresponding $RT(t)$, then the fault can not 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.

Now we will develop some characteristics of Kautz networks from RT to solve the above problem. 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 $RT(t)$. In other words, the right tag of node u with respect to any of its descendants is fixed at all times.

During a message routing, if the next node is detected being faulty, the current node which is forced to reroute the message around the fault 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 faulty node(s) encountered before 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 the left tag of a node represents the distance from its best vertex to the root. Additionally, we can find that the shorter the distance from a vertex to the root (destination) the worse effect results when the node is faulty, because the subtree below the vertex 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 by Algorithm *DFTR* when the message encounters a faulty node at the next hop. According to Definition 9, it loads the left tags of the faulty nodes encountered before directly from the routing record of the message, and extracts the left and right tags of the next node. Next, we consider how to obtain a best alternative path.

From Subroutine *Routes_Generation*, we can obtain all the routes of length at most k in the order of paths by increasing length. Since the new successful route must not be the shortest path, the checking begins with the second shortest route. 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 new route includes the faulty nodes encountered before. 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 method to produce an alternative path, we need to insert a digit x between the labels of the rerouting node $(r_1r_2 \ldots r_k)$ and the destination node $(t_1t_2 \ldots t_k)$, where x can take any value from the alphabet $\{0, 1, \ldots, d\}$ except r_k and d_1. From Theorem 1, in order to successfully route messages as long as the number of faulty nodes is less than d, at 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 at the rerouting node $(r_1r_2 \ldots r_k)$.

begin

step **1** {obtain information about faulty nodes encountered before.}

1.1 load *Left_Tags* from the routing record of the message.

1.2 extract the left and right tags of the next node in the old route, and add them to *Left(Right)_Tags*.
$i := 1$;
step **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 the new route := $[r_1 r_2 \ldots r_k d_{\text{position}+1} \ldots d_k]$;
else the new route := $[r_1 r_2 \ldots r_k + x$(or $x_1 x_2$ if necessary) $+ d_1 \ldots d_k]$;
step **3** {checking for the new route.}
call *left_mark_checking* ;
if Found(Faulty) **then** added to *Right_Tags*, **goto** step **2;**
else **call** *right_mark_checking* ;
if Found(Faulty) **then goto** step **2;**
else new *route* & *Left_Tags* attached to the message;
send to the next node along the new route.
end of the subroutine.

Proof of the Correctness of Our Algorithm

THEOREM 3. *If there is a faulty node encountered before existing on the new route, it will be detected by Left_Tags or current Right_Tags which have been obtained up to now at this rerouting node.*

Proof. We prove it by contradiction. Suppose to the contrary that there is a faulty node encountered before on the new route cannot be detected, then its corresponding vertex must not be at its best vertex and its right tag has not been obtained. Therefore, there exists at least a vertex nearer to the root (destination) in the corresponding $RT(t)$ 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 shorter 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. Link 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 the faulty link, we regard node v as a faulty node.

3.5. An Example

The following example illustrates how Algorithm *DFTR* works in the presence of faults. Let node (120) and node (310) be the source and destination for a message in *Kautz* (3, 3). Suppose nodes (031) and (231) are faulty. We construct the $RT(310)$ for explanation in Fig. 2.

In the beginning, we have the shortest route [120310]. Then Algorithm *DFTR* transmits the message along the shortest path (120) → (203) → (031) → (310). Unfortunately, node (031) prior to the destination is faulty. Hence, we use Subroutines *RG* and *PF* to find a best alternative path at node (203). 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 produce the new route by inserting one digit between nodes (203) and (310). We may choose the insertion digit $x = 0$ or 1. But these routes [2030310] and [2031310] including the encountered faulty node (031) will be detected after all by *left_mark_checking* and *right_mark_checking*, respectively. Thus, $x = 2$ is the

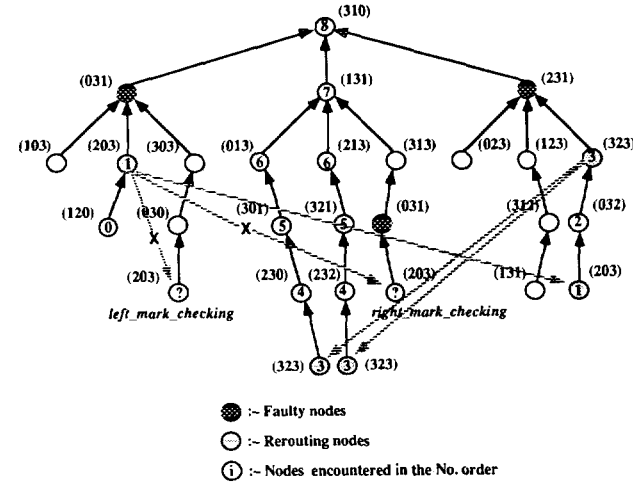

FIG. 2. The successive steps of our routing algorithm for the example.

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 may choose insertion digit to generate the new route, but it is impossible to construct a best alternative path if the number of insertion digit is one. Thus, we need to insert two digits between nodes (323) and (310). Finally, we obtain two best alternative paths [32301310] and [32321310]. We may send the message along one of the two best alternative paths, and then the message will reach the destination node (310). Figure 2 shows the successive steps.

4. PERFORMANCE ANALYSIS

In this section we compute the resulting path length at the worst case in our algorithm for faulty Kautz networks. The worst case occurs when the number of faulty nodes is equal to which the network can tolerate up to. The resulting path length in Algorithm *DFTR* is bounded as follows.

THEOREM 4. *Suppose there are f faulty nodes in Kautz (d, k) network, where $d \geq 3$, $k \geq 2$ and $1 \leq f \leq d - 1$, and Algorithm DFTR are used for routing a message from any source node to any destination node. Then, the length of the resulting path at the worst case is as follows.*

(case 1) $1 \leq f \leq d - 2 : (k - 1) * (f - 1) + 2k - 1.$

(case 2) $f = d - 1 : (k - 1) * d + 2.$

Proof. From Theorem 1, between each pair of nodes, there are d node-disjoint paths, one of length at most k, $d - 2$ of length at most $k + 1$ and one of length at most $k + 2$. 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 $k + 1$ 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. Now we can figure out the steps needed at the worst case as follows.

(case 1) $1 \leq f \leq d - 2$: $(k - 2) + (k - 1) * (f - 1) + (k + 1) = (k - 1) * (f - 1) + 2k - 1$;

(case 2) $f = d - 1$: $(k - 2) + (k - 1) * (d - 2) + (k + 2) = (k - 1) * d + 2$. ■

THEOREM 5. *Suppose there are f faulty nodes in Kautz (d, k) network, $d = 2$, $k \geq 2$ and $f = 1$, and Algorithm DFTR are used for routing a message from any source node to any destination node. Then, the length of the resulting path at the worst case is as follows.*

(case 1) $k = 2$: 3.

(case 2) $k \geq 3$: $2k$.

Proof. We prove it by considering the two separate cases.

(case 1). $k = 2$. Suppose the source and destination nodes are (s_1s_2) and (d_1d_2). Without any loss of generality, assume that $s_2 \neq d_1$ and the node (s_2d_1) is faulty. Then the message needs to be rerouted by using Algorithm *DFTR* at the source node (s_1s_2). Additionally, we get the left and right tags $[s_2]$ and $[d_1]$. Since the cardinality of the set $\{s_2, d_1\}$ is 2, there must be a choice for the insertion digit, i.e., $x \in \{0, 1, 2\} - \{s_2, d_1\}$. Thus, the length of the resulting path is 3.

(case 2). $k \geq 3$. We use the similar method to prove case 2. Suppose the source and destination nodes are $(s_1s_2 \ldots s_k)$ and $(d_1d_2 \ldots d_k)$. Without any loss of generality, assume that the route of the shortest path is $(s_1s_2 \ldots s_kd_1d_2 \ldots d_k)$, and hence, s_k and d_1 are not identical. If the node $(s_kd_1d_2 \ldots d_{k-1})$ is faulty, then we need to reroute the message at the node $(s_{k-1}s_kd_1d_2 \ldots d_{k-2})$. The left and right tags are $[s_k]$ and $[d_{k-1}]$ for the faulty node. Assume there exists a new route $(s_{k-1}s_kd_1d_2 \ldots d_{k-2}yd_1d_2 \ldots d_k)$, which can send the message successfully to reach the destination node. Now we know that $y \neq d_{k-2}$, $y \neq d_1$, $y \neq s_k$, and $y \neq d_{k-1}$. Since the cardinality of the set $\{s_k, d_1, d_{k-2}, d_{k-1}\}$ is 3 if $\{s_k, d_1\} \neq \{d_{k-2}, d_{k-1}\}$, y does not exist, which contradicts the assumption. Thus, the length of the resulting path at the worst case may be $(k - 2) + (k + 2) = 2k$. ■

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 faulty component encountered before by checking only the corresponding digits on the new route with these relative tags.

REFERENCES

1. Bermond, J.-C., Homobono, N., and Peyrat, C. Large fault-tolerant interconnection networks. *Graphs Combin.* **5,** (1989), 107–123.
2. Chen, M.-S., and Shin, K. G. Adaptive fault-tolerant routing in hypercube multicomputers. *IEEE Trans. Comput.* **C39,** (1990), 1406–1416.
3. de Bruijn, N. G. A combinatorial problem. *Proc. Academic Van Wetenschappen.* A49, 1946, 758–764.
4. Du, D. Z., and Hwang, F. K. Generalized de Bruijn digraphs. *Networks,* **18,** (1988), 27–38.
5. Esfahanian, A. H., and Hakimi, S. L. Fault-tolerant routing in deBrijn communication networks. *IEEE Trans. Comput.* **C34,** (1985), 777–788.
6. Homobono, N., and Peyrat, C. Connectivity of Imase and Itoh digraphs. *IEEE Trans. Comput.* **C37,** (1988), 1459–1461.
7. Imase, M., Soneoka, T., and Okada, K. Connectivity of regular directed graphs with small diameters. *IEEE Trans. Comput.* **C34,** (1985), 267–273.
8. Kautz, W. H. Bounds on directed (d, k) graphs, *Theory of cellular logic networks and machines.* AFCRL-68-0668 Final Report, (1968), pp. 20–28.
9. Kumar, V. P., and Reddy, S. M. A class of graphs for fault-tolerant processor interconnections. *IEEE 1984 Int. Conf. Distributed Computing Systems.* 1984, pp. 448–460.
10. Knuth, D. E., Morris, J. H., and Pratt, V. R. Fast pattern matching in strings. *SIAM J. Comput.* **6,** (June 1977), 323–350.
11. Manber, U. String matching, *Introduction to Algorithm: A Creative Approach.* Addison-Wesley, Reading, MA 1989, pp. 148–155.
12. Pradhan, D. K., and Reddy, S. M. A fault-tolerant communication architecture for distributed systems. *IEEE Trans. Comput.* **C31,** (1982), 863–870.
13. Pradhan, D. K. Fault-tolerant multiprocessor link and bus network architectures. *IEEE Trans. Comput.* **C34,** 33–45 (1985).
14. Raghavendra, C. S., Gerla, M., and Avizienis, A. Reliable loop topologies for large local computer networks. *IEEE Trans. Comput.* **C34,** (1985), 46–54.
15. Reddy, S. M., Kuhl, J. G., Hosseini, S. H., and Lee, H. On digraphs with minimum diameter and maximum connectivity. *Proc. 20th Annual Allerton Conference.* (1982), pp. 1018–1026.
16. Sengupta, A., Sen, A., and Bandyopadhyay, S. Fault-tolerant distributed system design. *IEEE Trans. Circuits Syst.* **C35,** (1988), 168–172.

Received December, 1991; revised July 7, 1992; accepted August 10, 1992

WEI-KUO CHIANG was born in Taiwan in 1967. He received the B.S. and M.S. degrees in Computer Science from National Chiao-Tung University, Taiwan, in 1989 and 1991, respectively. His research interests include fault-tolerant computing, interconnection networks, and parallel algorithms.

RONG-JAYE CHEN was born in Taiwan in 1952. He received a B.S. in Mathematics from National Tsing-Hua University, Taiwan, in 1977, and Ph.D. degree in Computer Science from University of Wisconsin at Madison, in 1987. He is now an associate professor in Department of Computer Science and Information Engineering in National Chiao-Tung University. Dr. Chen is also a member of IEEE. His research interests include parallel computation, algorithms, mathematical programming, and computer networking.