Reconstructing a distributed depth-first-search tree after network topology changes

S. A. M. Makki and George Havas*
Key Centre for Software Technology
Department of Computer Science
University of Queensland
Queensland 4072
Australia
sam@cs.uq.oz.au havas@cs.uq.oz.au

Abstract

We consider the problem of reconstructing a distributed depth-first-search (DFS) tree in an interconnected communication network in the presence of link failures or deletions and/or recoveries or additions. We describe an algorithm which efficiently reconstructs a distributed DFS tree after several links are deleted, with link additions also properly handled. Under standard assumptions, the required number of messages and time units in the worst case is bounded by $k \times (h + |V|(1 + r))$, where k is the number of link failures, h is the length of the longest simple path in the network, $|V|$ is the number of nodes in the network and $0 \leq r < 1$.

Keywords: Fault tolerance, depth first search, interconnection networks, distributed algorithms, communication graph.

1 Introduction

Faults in an interconnection network can cause severe performance degradation unless some kind of protection is provided which allows the system to operate in the presence of the faults. Reliability is of critical importance in situations where a computer malfunction could have catastrophic results. Therefore reliability techniques have become increasingly important.

Resilience in computing systems can be achieved through both *fault prevention* and *fault tolerance* techniques [1]. Fault tolerance enables the system to cope with the effects of faults in the state of the system by the incorporation of otherwise redundant components into the system.

*Supported in part by the Australian Research Council

Over the past decade many distributed DFS algorithms have been proposed, culminating in the algorithm described by us in [2], which includes comprehensive references to earlier work by others. None of these algorithms is designed to perform in the presence of failures (or deletions), possibly combined with recoveries (or additions) of links. We present a new, reasonably efficient algorithm which functions correctly in the presence of link failures and recoveries. The algorithm quickly reconstructs a DFS tree, as long as the graph remains connected, after any number of link failures/recoveries, which are allowed to occur at arbitrary times. (If the graph becomes disconnected, the algorithm constructs a DFS tree for that connected component which includes the root node, which may well be useful in various applications.)

It is inefficient to repair a disconnected tree caused by a single failure. Thus our algorithm does not even try to repair a broken tree, but rather replaces it with a newly constructed tree. It extends simply to handle multiple failures.

2 The model

An interconnection network can be modeled by an undirected communication graph $G = (V, E)$ where V and E are the set of sites and the set of bidirectional communication links of the communication network, respectively. Sites are autonomous in that they perform their computation and communicate with each other only by sending messages. They do not share a common memory. Each site has a unique identity and has local information, such as the identity of each of its neighbors. Message delivery is handled by the communication subsystem which delivers a message from sender to receiver in finite time without any alteration

or loss. A receiving site keeps messages in arbitrary order till they can be processed. The time for receiving and processing a message is negligible.

A link failure is simply the loss of connection between two nodes. It may result in the loss of or delay to a message. However when a message is delivered, it is delivered correctly. (Node failure can be modeled by the failure of all links adjacent to the node.) In a distributed environment failures cannot be predicted in advance. Here we study link failures. We assume that nodes do not behave in a malicious (Byzantine) manner. Our assumptions about a link are as follows.

1) A link is said to be up when a message can be transmitted in either direction, otherwise it is down.

2) Any link failure is detected by both end nodes in a finite time, not necessarily at the same time, but before the next status change (i.e. up, down).

3) After a link fails all messages in the link are lost. At link recovery time there are no messages in transit and none waiting to be sent over it.

Given such a network and a depth first search (DFS) tree, we want to maintain and update a DFS tree rooted at an arbitrary site so that we have a valid tree as soon as possible after failures and/or recoveries of links. We evaluate the complexity of our algorithm using standard complexity measures. The communication complexity is the total number of messages sent during the execution of the algorithm. The time complexity is the maximum time elapsed from the beginning to the termination of the algorithm, assuming that delivering a message over a link requires one unit of time.

3 Preliminaries

First, we briefly review some of the standard definitions from graph theory. A connected graph G is a union of two subgraphs T and $\overline{T}$, where T is a spanning tree and $\overline{T}$ is the collection of chords which complements T in G . A link is called a *tree link* if it belongs to the DFS tree, otherwise it is called a *non-tree link*. A simple path in a graph is a path each of whose nodes occurs exactly once in the path.

Our new fault-tolerant algorithm is built on the algorithm of [2], which we refer to as the basic algorithm. The basic algorithm is essentially sequential. It achieves better complexity than previous algorithms by incorporating the set of visited nodes in messages to reduce the number of probes and by using dynamic backtracking to reduce the amount of backtracking. The algorithm requires $|V|(1+r)$ messages and time units in the worst case, where $0 \leq r < 1$ and $|V|$ is the number of sites in the network. The value of r depends on the network topology and possibly on the path chosen. A complete description of the standard algorithm and a complexity analysis is presented in [2].

4 The resilient algorithm

Link failure and recovery in an interconnected network is inevitable, caused by such things as repairs, maintenance, computer crashes and addition of new components. Therefore a technique is needed to identify the events (failure or recovery) and to start the necessary routine to ensure reliable functioning in the presence of the topological changes. Thus a procedure which constantly checks for the occurrence of such an event resides in each node.

When this procedure detects a link failure it initiates corrective action. Some of this is purely local, but sometimes it requires distributed work. Thus, the end nodes should change local variables such as link status and possibly, in the context of a DFS tree, parent/child relationships. When parent/child relationships are broken, the DFS tree becomes disconnected, requiring reconstruction.

At any time, there may be several such events in different parts of the network, with consequent changes to the network topology. Here first we consider the effect of link recovery (or addition) on the network and then the effect of link failure (or deletion).

Link recovery or addition.
Recovery or addition of a link to the network does not change the structure of an existing DFS tree. The existing DFS tree remains valid. However recovery or addition has a potential effect on the structure of a DFS tree constructed after the recovery or addition. Thus, after link recovery or addition, it suffices that appropriate local variables be set correctly in the nodes which the link connects. For our purposes, it suffices for each of these nodes to add its thus connected node to its set of neighbours.

Link failure or deletion.
Consider a single link failure between two arbitrary nodes. If the nodes do not have a parent/child relationship then the link failure does not have any effect on an existing DFS tree. This means we can treat it in an analogous way to link recovery. In this case, to handle the potential effect on a later constructed DFS tree, each disconnected node deletes the thus unlinked node from its set of neighbours.

If however the deleted link connected nodes which have a parent/child relationship in an existing DFS tree, that DFS tree is no longer valid. It is theoretically possible to repair the broken tree, if the graph is still connected, but this is an undesirable exercise.

The broken DFS tree is now divided into two subtrees, which we call lower and upper. The lower subtree contains the child node and its descendants. The upper subtree contains all other nodes, including the previous parent node and its ancestors which are on the DFS tree simple path back to the DFS tree root. If the graph is still connected, these two subtrees can be connected by a replacement (currently non-tree) link, giving a new tree satisfying the DFS property.

To do so correctly we must explore the lower subtree to find all nodes in it, since proper connection may involve any one of those. Then we must backtrack from the previous parent through its ancestors to find the proper connection node. We have to resolve the communication problem of sending the lower subtree information to the old parent, but that link is down. Without additional routing information, there is no efficient solution to this problem. (This becomes more complicated again if we allow multiple failures.)

So, instead, we use a simpler process. We notify the root that the DFS tree is broken and the root initiates a new DFS tree construction. In view of the efficiency of the basic algorithm, this leads to a reasonably efficient reconstruction method.

Once we accept this strategy, we must decide how to notify the root of the failure. We could choose either to broadcast the information or to transfer it sequentially from the parent node of the failed link back up the DFS tree. In principle, broadcasting a failure is faster than sequential message passing [3, 4]. However the sequential method has the advantage of using the existing DFS tree path, which reduces the number of messages which are sent.

It is easy to extend the method to handle multiple failures. For example, if a failure message encounters a break in the DFS tree on its way back to the root, the message may be abandoned, since the encountered break will itself have initiated its own failure message which will suffice to reconstruct the tree. Furthermore, various savings can be made in situations where a node receives a failure message after it has received a reconstruction message. Details of these and other enhancements are beyond the scope of this paper.

5 Complexity analysis

Theorem 1. For a single failure the required number of messages and time units is less than $h + |V|(1 + r)$ in the worst case, where h is the length of the longest simple path in the network, $|V|$ is the number of nodes in the network and $0 \leq r < 1$.

Outline of Proof. For a single failure the associated parent node sends a failure message to the root through the existing DFS tree path. The root node then reconstructs the DFS tree using the basic algorithm. Under these circumstances, all messages are sent sequentially. In the worst case, the failure message reports the failure of a link which connects a leaf node to the DFS tree, when the failure message needs to travel almost the full height of the DFS tree, which is less than h, the length of the longest simple path in the network. So less than h messages are sent to transport the failure message from the parent node back to the root. This is followed by the basic algorithm, which has worst case $|V|(1+r)$ messages. Hence the message and time complexity for the worst case is bounded by the sum of these, giving the required result.

Theorem 2. For k link failures the required number of messages and time units in the worst case is less than $k \times (h + |V|(1 + r))$.

Outline of Proof. In the worst case all k failures will be of tree links and all k failure messages will be sent to the root. Furthermore, the DFS tree will be reconstructed k times. So the total number of messages and time units for k link failures is bounded by k times the bound for a single failure.

6 Conclusion

We have proposed a reasonably efficient resilient algorithm for reconstructing a distributed DFS tree after a number of link failures or deletions and link recoveries or additions in the network. The algorithm enables the system to function correctly soon after changes in the network topology. The worst time after which reconstruction is complete is bounded linearly by the product of the number of failures times the size of the network.

References

[1] T. Anderson (ed.), *Resilient computing systems,* Collins, London, 1985.

[2] S. A. M. Makki and G. Havas, "Distributed algorithms for constructing a depth-first-search tree," *Proceedings 1994 International Conference on Parallel Processing*, CRC Press, Boca Raton.

[3] P. Feldman, "Fault tolerance of minimal path routings in a network," *Proceedings 17th Annual ACM Symposium on Theory of Computing*, 1985, pp. 327–334.

[4] D. Dolev, J. Y. Halpern, B. Simons and H. R. Strong, "A new look at fault-tolerant network routing," *Information and Computation* **72**, 1987, pp. 180–196.