The Multi-Tree Approach to Reliability in Distributed Networks

Alon Itai
Computer Science Department
Technion, Haifa, Israel

Michael Rodeh
IBM Israel Scientific Center,
Technion City, Haifa, Israel.

ABSTRACT

Consider a network of asynchronous processors communicating by sending messages over unreliable lines. There are many advantages to restricting all communications to a spanning tree. To overcome the possible failure of $k' < k$ edges, we describe a communication protocol which uses k rooted spanning trees having the property that for every vertex v the paths from v to the root are edge-disjoint.

An algorithm to find two such trees in a 2 edge-connected graph is described that runs in time proportional to the number of edges in the graph. This algorithm has a distributed version which finds the two trees even when a single edge fails during their construction. The two trees them may be used to transform certain centralized algorithms to distributed, reliable and efficient ones.

1. INTRODUCTION

Consider a network $G=(V,E)$ of $n=|V|$ asynchronous processors (or vertices) connected by $e=|E|$ edges. The network may be used to conduct a computation which cannot be done in a single processor, since either the data is distributed over the network and it is too costly to move it all to a single processor, or the resources of the network, such as memory, are distributed among the processors.

Every processor in the network is a computer with random access memory (RAM) and a program of its own. We assume that each processor v with $deg(v)$ neighbors has $O(deg(v))$ cells of local memory each of $O(\log(n+MaxID))$ bits. Thus, no processor can have the entire network stored locally. The processors are distinguishable by distinct identification numbers (*id*). Each processor knows its own *id* and that of its neighbors and may use this information to make decisions. Other than this the local programs are independent of the network. Such programs are called *distributed programs*.

We assume that there is no shared memory so that processors may communicate only by transmitting messages, each of length $O(\log(n + Max\ id))$ bits (this bound is implied by the bound on the degree). Two neighbors may *pass* messages on the edge between them, while two remote processors have to *transmit* messages. Each message transmission may require several message passings. The time a message passes through an edge is unpredictable; moreover, if the edge is faulty the message might get lost. Until a message arrives, no processor is able to decide whether the edge is faulty or just slow. Edges are assumed to have infinite buffers; messages may accumulate in these buffers without disturbing the sending processor.

In the absence of any prior agreement concerning message routing, *broadcasting* may be used: To broadcast a message to v, u adds the address of v to the message and passes it to all its neighbors which upon receiving the message pass it further. Special provisions must be taken to ensure that the same message is not passed many times through the same edge. However, these provisions require a large amount of memory at each vertex. Even if each message is passed only once through each edge there are $\theta(e)$ message passings in order for u to transmit a message to v.

No such provisions are required for *tree networks* - networks that do not contain cycles. Therefore, to properly communicate in arbitrary networks, we could find a subnetwork which forms a spanning tree, and restrict all message passings to the edges of the tree.

The problem with tree networks is that they are unreliable - the failure of a single edge makes the network disconnected. In our model of faults, the effect of failures is the loss of messages. After a failure, edges may recover. We shall count the number of edges which failed rather than the number of messages lost.

2. THE 2-TREE PROTOCOL

To overcome the unreliability of tree networks more than one spanning tree may be used. We are after a communication protocol which is *1-edge resilient* — a protocol which resists the failure of any single edge.

2-edge-connectivity of the network is a necessary condition for the existence of 1-edge resilient protocols. However, a 2-edge connected graph (such as a cycle) need not contain two edge-disjoint spanning trees. Thus our protocols use trees which are not completely disjoint.

2.1. The 2-Tree Protocol for Edges

For a spanning tree T rooted at r let $T[v]$ denote the path in T from v to r.

Definition: Two spanning trees T_1 and T_2 with a common root r satisfy the 2-*tree condition for edges* if for *every* vertex v, $T_1[v]$ and $T_2[v]$ are edge-disjoint.

Lemma 1: If a graph G has two spanning trees T_1 and T_2 fulfilling the 2-tree condition for edges then G is 2-edge connected.

Proof: (Although the lemma follows from Theorem 2 below, we have chosen to give a direct proof too, since it sheds light on the following discussion.) For the sake of contradiction, let (u,v) be a separating edge. W.l.o.g. (u,v) separates v from r. Therefore, every path from v to r must pass through (u,v). In particular, $T_1[v]$ and $T_2[v]$ both use (u,v) and cannot be edge disjoint, thus violating the 2-tree condition.

Before we prove the converse of Lemma 1, let us see how to use an arbitrary pair of trees which fulfill the 2-edge condition to construct a 1-edge resilient protocol to transmit a message from any vertex u to any other vertex v.

The 2-Tree Protocol for Edges:

(1) u sends a message upwards to the root on both trees.

(2) When the root gets a message from one of its neighbors, it sends the message downwards on *both* trees.

Since every message is duplicated at the root, in the case of no failures, for every message sent from u, v receives four messages from r, and possibly two more messages from u. However the messages sent by the two tree protocol for edges are easier to manage than those used in simple broadcasting: Only the source and the target vertices of the message have to be aware of the duplication of the message. The other vertices just transmit the message through the appropriate tree. Thus none of the vertices need remember which messages have already passed.

Example 1: Consider the graph G depicted in Figure 1 with the spanning trees

$$T_1 = (r, v_1, v_2, \cdots, v_9) = G \setminus (v_9, r)$$

$$T_2 = (r, v_9, v_8, \cdots, v_1) = G \setminus (v_1, r)$$

Let $u = v_3$ and $v = v_7$. $u = v_3$ sends a message on T_1 to v_2, v_1 and r. r sends the message downwards on T_1 to $v_1, v_2, \cdots, v_9$ and on T_2 to $v_9, v_8, \cdots, v_1$.

$u = v_3$ also sends the message on T_2 to v_4, v_5 , ... , v_9, r. When r gets this message, it sends it downwards on T_2 to $v_9, v_8, \cdots, v_1$. In addition, r sends the message on T_1 to $v_1, v_2, \cdots, v_9$. Thus, if no edge fails v_7 intercepts the message five times: once from v_3 on T_2, twice from r on T_2 and twice from r on T_1.

However, if the edge (v_4, v_5) fails, then only the message upwards on T_1 gets to r, r sends it downwards on T_1 and on T_2, but only the propagation on T_2 reaches $v = v_7$. Thus, only the combination of the two trees ensures that the message arrives.

Theorem 2: Let T_1 and T_2 be two trees fulfilling the 2-tree condition for edges. Then the 2-Tree Protocol for Edges is 1-edge resilient. Moreover, only $O(n)$ message passings are required.

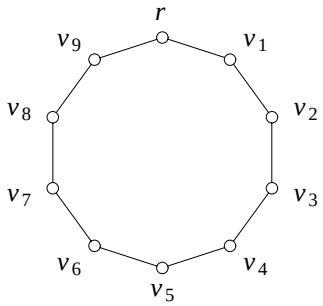


Figure 1

Proof: Let u be a vertex which wants to transmit a message to another vertex v. According to the 2-tree protocol for edges, u sends the message upwards to the root r on both trees. As $T_1[u]$ and $T_2[u]$ are edge-disjoint, at least one copy of the message will arrive at v even if a single edge fails.

When the message arrives at r, r sends it on both trees to v. Again, as $T_1[v]$ and $T_2[v]$ are edge-disjoint, at least one copy will arrive at v, even in the presence of a single edge-failure.

A message can pass through an edge of any of the two trees at most once in each direction; as the message is duplicated at the root, the maximum number of message passings is $3 \cdot 2 \cdot (n-1) = O(n)$.

The converse of Lemma 1, namely, that every 2-edge connected graph has two spanning trees fulfilling the 2-tree condition for edges is also true. In the rest of this section we develop an algorithm to construct two such trees.

2.2. s-t Numbering

To show that 2-edge connectivity implies the 2-tree condition for edges we follow a suggestion of Professor A. Lempel [L] (thus replacing a previous lengthier direct proof).

Let G be a 2-vertex connected graph, and (s,t) an edge of G, then $g: V \rightarrow \{1,2,\cdots,n\}$ is an *s-t numbering* if the following conditions are satisfied:

(1) $g(s) = 1$ and $g(t) = n$.

(2) Every vertex $v \in V \backslash \{s,t\}$ has two adjacent vertices u and w such that

$$g(u) < g(v) < g(w).$$

Thus if every edge is oriented from its low numbered end to its high numbered end then every vertex lies on a directed path from s to t. Lempel, Even and Cederbaum [LEC] have shown that every 2-vertex connected graph G has an s-t numbering and used it to test graph planarity. Even and Tarjan [ET] gave a linear algorithm based on DFS to find such a numbering. We follow the algorithm and proof as presented in [Ev].

To construct an s-t numbering we have to look more closely at the DFS algorithm.

Let D be a DFS tree rooted at t whose first edge is (t,s). Let $N(v)$ be the number given to v by the DFS. Thus $N(t) = 1$ and $N(s) = 2$. $(v_1,\cdots,v_m)$ is a *tree path* if v_i is the tree parent of v_{i+1}, $(i = 1,\cdots,m-1)$.

The edges of $G \backslash D$ are called *fronds.* $(v_1,\cdots,v_m,v_{m+1})$ is the *lowpoint path* from v_1 if $(v_1,\cdots,v_m)$ is a tree path, (v_m,v_{m+1}) is a frond and $N(v_{m+1})$ is the smallest number for which there exists such a path from v_1. Let $L(v_1) = \min\{N(v_1),N(v_{m+1})\}$ be the *lowpoint* of v_1. (If G is biconnected $N(v_{m+1}) < N(v_1)$ for every vertex v_1 other than the root.)

The following algorithm which is a slight modification of [ET, Ev] finds an s-t numbering, assuming that D has already been found, and that *lowpoint*(v) has been computed for every vertex v. The algorithm uses a stack to process vertices in the reverse order in which they were encountered. Initially, only t and s are on the stack (s on top). Vertices which were never on the stack are considered *new*.

Set i:=1.
while the stack is not empty **do begin**

(1) Remove the top vertex v from the stack.

(2) $g(v)$:=i; i:=i+1.

(3) For all tree edges (v,w) to a new vertex w, let $(w=w_1,\cdots,w_m,w_{m+1})$ be the lowpoint path from w; push $w_m,w_{m-1},\cdots,w_1$ onto the stack (w_m first).

(4) For all paths $(u_0,u_1,\cdots,u_m,v)$ from some old vertex u_0 to v such that $u_1,\cdots,u_m$ are new, (u_i,u_{i+1}) are tree edges and (u_m,v) is a frond, push $u_1,\cdots,u_m$ onto the stack (u_1 first).

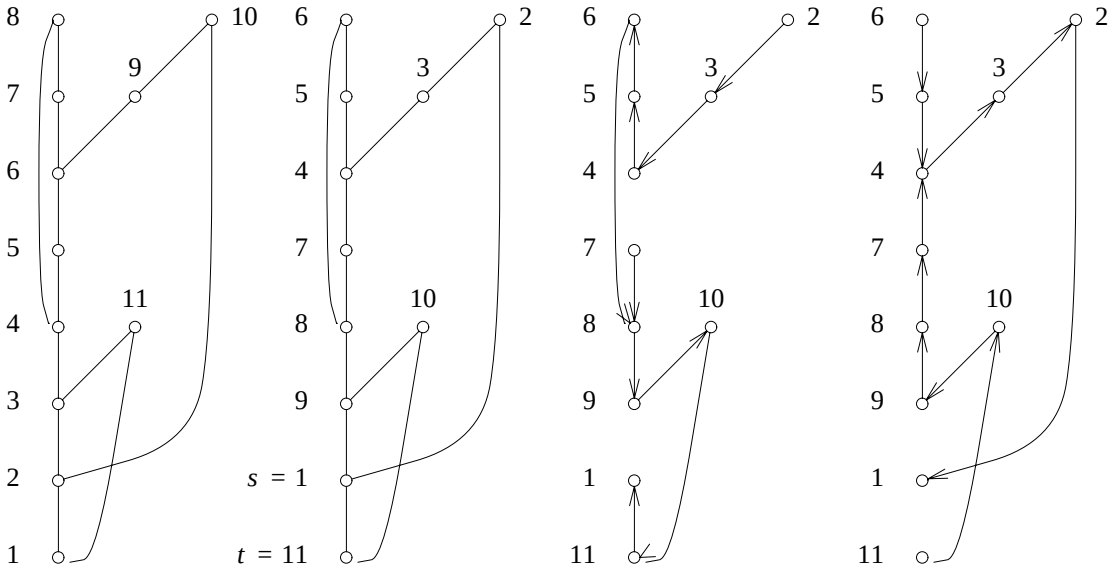
end.

Example 2: Figure 2.*a* shows a graph with a DFS numbering – $N(v)$ and Figure 2.*b* shows $g(v)$ an s-t numbering of that graph.

Even and Tarjan [ET] have shown that this algorithm finds an s-t numbering in $O(e)$ time.

2.3. Constructing the Two Trees

To construct two trees S and T rooted at r which satisfy the 2-tree condition for edges first assume that G is 2-vertex connected, and that g is an s-t numbering with $s = r$. To construct S choose for every vertex $v \neq s$ an edge (u,v) such that $g(u) < g(v)$ (for t choose an edge other than (s,t)). T consists of the edge (s,t) and an edge (v,w), $g(v) < g(w)$ for every vertex v *nmem* $\{s,t\}$. The 2-tree condition is easily verified since $S[v]$ consists of vertices u with $g(u) < g(v)$ but cannot contain the edge (s,t), while $T[v]$ consists of the edge (s,t) and vertices w with $g(v) < g(w)$. Figures 2.*c* and 2.*d* depict two trees constructed for the s-t numbering of Figure 2.*b*.



DFS-numbering s-t numbering *T* *S*

(a) (b) (c) (d)

Figure 2

In the general case, G is 2-edge connected but not 2-vertex connected. Let G_1 , ... , G_m be the *blocks* (2-vertex connected components) of G. Each G_i ($i>1$) is connected to some G_j ($j<i$) by an articulation point s_i. The 2-vertex connected components and articulation points can be found by a single DFS starting from any vertex t_1 of G_1. Let s_1 be the second vertex visited by the DFS. Without loss of generality, $s_1 \in G_1$. For each block G_i, let T_i and S_i be two spanning trees of G_i which are rooted at s_i and were constructed by the above procedure when applied to G_i. Then let

$$S = \bigcup_{i=1}^{m} S_i \qquad \text{and} \qquad T = \bigcup_{i=1}^{m} T_i .$$

It is easy to verify that for every vertex v, $T[v]$ and $S[v]$ are edge-disjoint. This algorithm requires $O(e)$ time. Note that any vertex may serve as the common root s_1. We get:

Theorem 3: Let G be a 2-edge-connected graph and r a vertex of G. Then there exist two spanning trees T and S such that for all vertices v $S[v]$ and $T[v]$ are edge-disjoint. Moreover, S and T can be found in $O(e)$ time.

3. DISTRIBUTED DFS

3.1 Distributed DFS in unreliable networks

DFS has a distributed nature (though sequential, see [R]): It has a single center of activity which we designate by a *token.* The token moves from a vertex to one of its neighbors. When the token is at v, v inspects its neighbors to find a vertex which does not yet belong to the tree. If such a vertex exists the token is moved there. Otherwise, the token backtracks to the parent of v. If the token cannot backtrack since v has no parent (i.e. v is the root), the algorithm terminates.

In the distributed context, every transfer of the token, as well as the inspection of the neighboring vertices, requires communication. Thus, $\theta(e)$ communications are involved. A naive implementation in the unreliable setup is to replace each communication by a broadcast, which if traverses each edge at most once, yields an $O(e^2)$ message passing algorithm. To make sure that each message indeed traverses each edge only once, each vertex may hold a log of all the messages which it had sent. Need-

less to say, this mechanism requires $\theta(e)$ memory at each vertex (one entry for each DFS message).

The sequential nature of DFS allows for a mechanism which is more space efficient: Each message is prepended by a *message number* — the ordinal number of that message. This number is known to the vertex which issued the message since when a vertex issues a message it must hold the token and the number of messages issued before receiving the token is equal to the message number prepended to the message by which the token was received.

Since a message with number M is issued only after all previous messages have already reached their destination, after receiving the M'th message, a vertex need not forward any message whose message number is less than or equal to M. Thus, at the cost of one memory cell per vertex, each message is not passed through any edge more than once.

To improve the performance of DFS in unreliable networks we consider the following variant of DFS: Whenever the token arrives at a vertex v for the first time, v notifies all its neighbors that it has joined the DFS tree. The idea is that when a vertex joins the tree, it should already know which of its neighbors belong to the tree. Thus the inspection of the neighbors is replaced by notifications.

A slight problem is caused if a vertex gets the token before the notifications from all its neighbors have arrived. (Some message might have traveled slowly.) To remedy this, each notification should be a *send/acknowledge* type of communication. The token is transferred from v only after receiving acknowledgement from all its neighbors. This variant also requires at most $O(e)$ message passings in reliable networks.

This version of the algorithm might fail if an edge fails. As indicated before, each message is replaced by a broadcast. There are two types of messages: Token passing and neighbor notification. Since the token is passed at most $O(n)$ times, this requires $O(ne)$ message passings. The send/acknowledge part of the distributed DFS uses $O(e)$ messages. Replacing each by a broadcast yields an $O(e^2)$ algorithm.

A closer look into the send/acknowledge pattern leads to the following modification [Shr]: When a vertex v gets the token for the first time, it sends a send/acknowledge message concurrently to all its

neighbors which do not send the message further but send an acknowledgement back on the edge connecting them with the token. The vertex v holding the token now waits until $deg(v)-1$ acknowledgements arrive. Then v notifies the last vertex by a broadcast and waits for an acknowledgement, which should also be sent by broadcasting. This algorithm requires $O(ne)$ message passings.

3.2. A Different Model

We can dispense of the send/acknowledge transmissions and still have an $O(ne)$ algorithm if we restrict the types of failures and the order in which messages pass through the edges. Namely, in this subsection we assume that malfunctioning edges never recover, and messages are passed through the edges in a strict first-in first-out fashion.

The new algorithm is similar to the previous one, each broadcast message is prepended by the message number and no vertex forwards messages whose number is lower that the maximum encountered message number. When a vertex joins the tree it sends a broadcast message notifying this fact to the entire graph, also if there is a neighbor not on the tree the token is passed (by a broadcast) to such a neighbor, otherwise the token is returned (by a broadcast) to the DFS-tree parent of the vertex.

If no messages are lost, every vertex eventually discovers which of its neighbors belongs to the tree. There remains, however, the problem of synchronization — a vertex w might have received the token before the message "u joined the tree" from its neighbor v has arrived . Thus, w does not know which of its neighbors already belong to the tree. Moreover, the message number mechanism and an edge failure might have caused some messages not to be forwarded, thus we need to show that the status of the neighbors is indeed known to the holder of the token. In the remainder of this subsection we show that under this restricted type of failures, this algorithm is correct.

Consider Figure 3 where we trace the journey of the token in the graph. The dashed arrows indicate logical transfers of the token according to the DFS order, while the solid lines trace the actual message transfers required to pass the token. I.e., the token starts at the root u_1, which passes it on to u_2, until it finally arrives at $u_m=u_1$ ($m<2m$). However, the token does not pass from u_j to u_{j+1} directly. In fact, it may visit several vertices along a path P_j (see Figure 3) before arriving at u_{j+1}. u_{j+1} receives

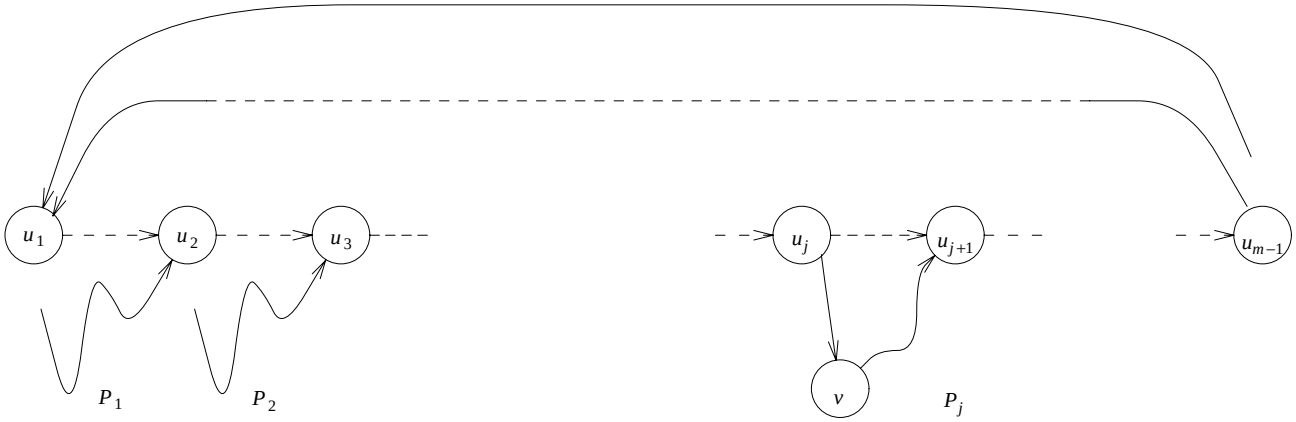


Figure 3

control of the token when it first receives the message $\mu_j = <j,u_j,u_{j+1},m_j>$. ($j$ is the message number, u_j the source of the message, u_{j+1} its destination and all the remaining information is contained in m_j.)

Lemma 2: Let $v \in P_j$ and i,k satisfy $1 \le i < k \le j$. Then v received the message $\mu_i = <i,u_i,u_{i+1},m_i>$ before it received (or issued) the message $\mu_k = <k,u_k,u_{k+1},m_k>$.

Proof: By double induction. The outer induction is on the path number, and the inner one is on the location of v on the path.

Basis: For $j=1$ there is not such i, so the lemma is vacuously true.

Induction step: Assume that the lemma holds for $P_1,P_2,\ldots,P_{j-1}$. Let $P_j = (u_j=w_1,\ldots,w_q=v,\ldots,w_k=u_{j+1})$. We now conduct an induction on q.

Basis $q=1$: $w_1 = u_j = v$, the claim holds for $k < j$ since u_j also belongs to P_{j-1}. As for $k = j$, u_j did not issue μ_j until μ_{j-1} was received, and that occurred after any μ_i ($i < j$) was received.

Induction Step: By induction w_{q-1} received μ_i before it received μ_k, therefore it passed μ_i on $(w_{q-1},w_q=v)$ before it passed μ_k on the same edge. By definition of P_j, μ_j managed to pass on $(w_{q-1},w_q=v)$. Because of our restrictive model, this edge could not have failed before μ_j passed, thus both μ_i and μ_k passed on $(w_{q-1},w_q=v)$ successfully.

Since our model stipulates a strict first-in first-out discipline, μ_i arrived at $w_q=v$ before μ_k.

Theorem 4: If failed edge do not recover and the messages passed through each edge obey the FIFO discipline then the above algorithm finds a DFS tree in $O(ne)$ messages.

4. REDUCING THE COST OF COMMUNICATION

4.1. Broadcasting in *O(n)* Message Passings

In this section we consider broadcasting protocols, which like the algorithms of Section 3, avoid sending any message through any edge more than once. However, even in this case each single message might traverse the entire network, thus requiring $\Omega(e)$ message passings. Here, we show how to reduce the cost of such broadcasting to $O(n)$ by restricting broadcasts to a 2 edge-connected spanning subgraph.

Let $D=(V,E_D)$ be a DFS tree (see Section 3) and E_L the set of *lowpoint fronds* of D (fronds leading from a vertex v to the vertex w such that $N(w)=L(v)$). Let $G' = (V,E_D \cup E_L)$. G' is 2-edge connected if and only if G is. Thus if all broadcast messages are restricted to G' a single broadcast will cost only $|E_D|+|E_L| \leq 2n-2 = O(n)$. Thus, after finding G' broadcasting becomes cheap. (G' was used previously in [IR, ILPR].)

It remains to find G' distributedly. Since each vertex knows which edge leads to its DFS parent and which to its children, it knows the tree edges adjacent to it. The remaining edges are fronds. However, v must know its lowpoint, $L(v)$, in order to figure out which of the fronds is its lowpoint frond. The lowpoint satisfies:

$$L(v) = \min(\{L(s) : s \text{ is a DFS child of } v\} \cup \{N(w) : (v,w) \in E \text{ and } w \text{ is not the parent of } v\})$$

Just before the DFS backtracks from v, v can figure out $L(v)$ by asking for $L(s)$ from all its children and for $N(w)$ from all its other neighbors. By employing a send/acknowledge protocol similar to the one presented in Section 3.1 (wait for all but the last neighbor and then send the message to the last neighbor by a broadcast) we get an $O(ne)$ scheme. Thus, finding G' costs no more than the DFS.

Henceforth, we assume that all broadcasts are done in G' and that each broadcast costs $O(n)$.

4.2. Communication on Trees

We have used spanning trees for communication, and have circumvented the problem that not every 2-edge-connected graph has 2-edge-disjoint spanning trees by replicating messages at the root. This has an adverse effect of increasing the traffic at the root. If, however, the graph has 2-edge-disjoint spanning trees then we may send messages directly on the 2-trees. To reduce the number of message passings, we may use the following heuristic which finds the shortest tree-path between two vertices without increasing the memory requirements. Thus sometimes shortening the path a message travels (though not improving the worst case).

Suppose each vertex w knew $C(w,c)$, the set of vertices of the subtree rooted at the child c of w. When w gets a message destined to v it passes it to c if $v \in C(w,c)$, if no such c exists the message is transmitted to the root.

To store $C(w,c)$ compactly, we conduct a depth-first search (DFS) on the spanning tree starting from the root. Let $N(x)$ be the number assigned to x by the DFS. We now observe that $N(C(w,c)) = \{N(x) : x \in C(w,c)\}$ consists of consecutive numbers, i.e.

$$N(C(w,c)) = \{i : \min N(C(w,c)) \leq i \leq \max N(C(w,c))\}.$$

Thus, it suffices that w store only $\min N(C(w,c))$ and $\max N(C(w,c))$ for each child c. Now, $N(v)$ is used as the network address of v.

5. CONSTRUCTING THE TWO TREES DISTRIBUTEDLY

In this subsection we develop a distributed version of Even and Tarjan's centralized algorithm to find an s-t numbering [ET], thus giving an example of transforming a centralized algorithm into a distributed one.

5.1. Distributed Construction of an s-t Numbering.

The s-t numbering algorithm is very similar to DFS, and a distributed version can be easily derived. The only problem is that of keeping a stack of length $O(n)$ in a network, each vertex of which

has limited memory. The stack, therefore, cannot be kept at any single vertex. Rather, each vertex on the stack knows only its stack predecessor, and the top vertex knows that it is on top. (This works since the s-t numbering algorithm of Section 2.2 pushes each vertex onto the stack at most once.)

Now the distributed version is very similar to the centralized one. The token starts at s. When a path is constructed, the token moves with the frontier of the path. When the path ends, the token is broadcasted to the top of the stack. Since the moves of the token correspond to adding or deleting a vertex from the stack, there are $O(n)$ broadcasts costing a total of $O(n^2)$ message passings.

5.2. Finding the Two Trees

Once each vertex u has its s-t number $g(u)$, it broadcasts it on G'. The neighbors of u record u's s-t number. Thus each vertex v knows the s-t number of all its neighbors. Then, v selects an incoming edge (u,v), $(g(u)<g(v))$ for S and an outgoing edge (v,w) $(g(v)<g(w))$ for T. After selecting the edges, v should broadcast this information (on G'), so that u knows that $(u,v) \in S$ and w knows that $(v,w) \in T$. Thus, every vertex discovers which of its adjacent edges belong to S and which to T. All these additional broadcasts (on G') cost $O(n^2)$ message passings.

6. COMPUTATIONS ON TREES

To understand the full power of the 2-tree protocol let us consider functions whose data is distributed over the network and whose value is independent of the order of the data. The algorithms we consider use trees as their communication subnetwork. A typical example is the Σ operation which computes the sum of a multi-set of values which are distributed over the various processors of the network.

A straightforward implementation of the Σ operation uses an arbitrary spanning tree and traverses it from the leaves to the root. This scheme does not work if edges might fail. However, replacing each message passing by a broadcast yields an $O(n^2)$ algorithm. An $O(n \log n)$ algorithm is described below.

One of the nice properties of the Σ operation is that it may be carried out on any computation tree. This allows one to choose the tree *dynamically.* In general, this tree will be different from both S and T. To this end, we need an operation called *split*, which is initiated by the root, takes an argument

$1 \le i \le n$ and yields the following two subtrees:

$$S_i = \{(u,v) \in S \mid g(u) \le g(v) \le i\}$$

$$T_i = \{(u,v) \in T \mid g(v) \ge g(u) > i\} \cup \{(s,t)\}.$$

The *split* operation has an interesting distributed implementation:

(1) The number i is broadcasted on both S and T. Therefore, every processor will eventually get the message even if a single edge fails.

(2) By inspecting $g(v)$ and comparing it to i every processor v can decide whether it belongs to S_i or to T_i.

(3) By inspecting the s-t number of its neighbors, every processor $v \in S_i$ can decide which of its neighbors also belongs to S_i. In particular, v can decide whether it is a leaf of S_i. (The same applies to T_i.)

Note: This algorithm assumes that each vertex knows the s-t number of its neighbors. This may be accomplished in $O(n^2)$ message passings (see section 5.2).

Let $\Sigma(R)$ denote the Σ operation applied to the tree R. $\Sigma(R)$ is implemented by traversing R from the leaves to the root.

When the root initiates a $split(i)$ operation, each vertex finds out to which tree it belongs, and the leaf vertices (of both trees) initiate the Σ operation. Since S_i and T_i are edge-disjoint, either $\Sigma(S_i)$ or $\Sigma(T_i)$ or both terminate successfully. Moreover, the root knows which operation succeeded. If both operations succeed then the root can compute Σ since $\Sigma = \Sigma(S_i)+\Sigma(T_i)$. Since the root cannot know whether both operations will succeed, it cannot wait for the second operation to terminate, and therefore it should continue immediately after receiving the first result.

To compute Σ of the entire network, the root proceeds in a binary search fashion: Initially, the root has the value of $\Sigma(T_n)=0$ and that of $\Sigma(S_1)$.

```
begin
  i := 1;  j:=n;
  R_S := Σ(S_1);  R_T := 0;
```

while $i < j$ **do begin**
  $m := \lfloor \frac{i+j}{2} \rfloor$;
  split (m) and initiate $\Sigma(S_m)$ and $\Sigma(T_m)$;
  wait until $\Sigma(S_m)$ or $\Sigma(T_m)$ finishes;
  if $\Sigma(S_m)$ finished first *then begin*
    $i := m+1$; $R_S :=$ the value of $\Sigma(S_m)$ *end*
  else {$\Sigma(T_m)$ finished first} *begin*
    $j := m$; $R_T :=$ the value of $\Sigma(T_m)$ *end*
end;
return $(R_S + R_T)$
end

Theorem 5: The algorithm computes Σ in $O(n \log n)$ message passings.

Proof: First note that for all m either $\Sigma(S_m)$ or $\Sigma(T_m)$ succeed. Thus the algorithm does not wait forever.

Next, let j_0 be the last time $\Sigma(T_m)$ has succeeded. From then on $m < j_0$ and $\Sigma(S_m)$ succeeds. Thus in each iteration m increases. The algorithm terminates when $i = m+1 = j_0$.

Each iteration requires $O(n)$ message passings. There are $O(\log n)$ iterations since each time $j-i$ decreases by a factor of 2.

The above scheme has further applications than the Σ operation. For more examples see [Shr].

7. EXTENSIONS

7.1. The k-Tree Protocol for Edges

Some of the previous results can be extended for more than two trees to obtain algorithms resilient to the failure of more edges. Obviously, the edge-connectivity of the network is an upper bound. Namely, if the edge-connectivity of the network is exactly k then there exits no communication protocol resilient to the failure of k edges. On the other hand, broadcasting is $k-1$-resilient, though inefficient in terms of communication and memory. As before we restrict the communication to trees to obtain more efficient algorithms.

Definition: A graph G satisfies the *k-Tree Condition for Edges* if for all vertices r there exist spanning trees $T_1, \cdots, T_k$ such that for *every* vertex v and $1 \le i < j \le n$, the two tree-paths $T_i[v]$ and $T_j[v]$ from v to r are edge-disjoint.

We have not been able to generalize Theorem 3 to $k>2$ and we state it as a conjecture

The k-Tree Conjecture for Edges: If G is a k-edge-connected graph then G satisfies the k-Tree Condition.

The converse, a counterpart of Lemma 1, is easy.

We now show some weaker results for $k \ge 3$. The orientation theorem of Nash-Williams [NW] and the branching theorem of Edmonds [Ed, T, Shi] show that every k-edge-connected graph has $lfloork/2\ rfloor$ edge-disjoint spanning trees. Thus k-edge connectivity implies the condition for $lfloork/2\ rfloor - 1$. The following theorem yields a stronger result ($lfloork/2\ rfloor$) for odd k.

Theorem 6: Every k-edge-connected graph has k spanning trees such that the removal of any $lfloor\frac{k-1}{2}\ rfloor$ edges leaves at least one tree intact.

Proof: Let $\vec{G}$ be the directed graph obtained from G by replacing every undirected edge (u,v) by two directed edges (u,v) and (v,u). $\vec{G}$ is k-edge-connected. Thus, by using the branching theorem of Edmonds [Ed, T, Shi] we can find k edge-disjoint spanning trees $\vec{T}_1, \vec{T}_2, \cdots, \vec{T}_k$. Let $T_1, T_2, \cdots, T_k$ be the corresponding trees in G. The theorem follows since every undirected edge belongs to at most two of the trees.

There still is an advantage in using disjoint spanning trees since no edge will carry the traffic of more than one tree, thus preventing possible congestion at that edge. The theorem implies that 3-edge-connectivity is sufficient to obtain 1-resilence and 5-edge-connectivity is sufficient for 2-resiliency.

If $T_1, \cdots, T_k$ satisfy the k-tree condition then u can transmit a message to v using the following *k-Tree Protocol:*

(1) u sends a message upwards to the root on *all* k trees.

(2) When the root gets a message from one of its children, it sends the message downwards on *all* k trees.

Since every message is replicated k-fold at the root, in the case of no failures, v will receive a message sent by u as many as k^2+k times (at most k times downwards and k times on each tree).

7.2. Vertices

Protocols resilient to the failure of a vertex cannot route all information through any single vertex since when it fails the entire algorithm fails with it. However, efficient algorithms do exist for the special case where there exists a vertex r (the *root*) which does not fail. The *k-tree condition for vertices and root r* states that there are k spanning trees $T_1, \ldots, T_k$, such that for all vertices v, the paths $T_i[v,r]$ $(i=1,...,k)$ are vertex-disjoint ($T[u,v]$ denotes the path on the tree T from u to v).

The *k-tree conjecture for vertices and a single root* states that k-vertex connectivity implies that for all $r \in V$ the k-tree condition for vertices and root r holds. For $k=2$ our proof for edges using the s-t numbering proves the conjecture also for vertices. Zehavi [Z,ZI] has extended these techniques to $k=3$. For $k>3$ the conjecture remains open. When the k-tree condition for vertices and root r holds we may apply the k-tree protocol.

In the more general case when we cannot assume that any vertex is immune to failure we have not been able to design an efficient protocol for $k>2$. For $k=2$ our construction for edges implies that for any edge (s,t) there exists two trees S and T rooted at s and t respectively, such that for every vertex v, the paths $S[v,s]$ and $T[v,t]$ are vertex-disjoint. For such S and T a vertex u can transmit a message to the vertex v using the following 2-*Tree Protocol for Vertices:*

(1) u sends a message upwards to the roots of *both* trees (the message on S is called the *S-message* and that on T the *T-message).*

(2) When s gets an S-message from one of its children, it sends the message downwards on S and sends an additional copy, the S-message, on (s,t) to t.

(3) When t gets a T-message from one of its children, it sends the message downwards on T and sends an additional copy, the T-message, on (s,t) to s.

(4) When s gets a T-message it sends its downwards on S.

(5) When t gets a S-message it sends its downwards on T.

As with edges, v will receive any message at least once and no more than five times.

As mentioned before we have not been able to construct algorithms resilient to the failure of vertices since we were not able to overcome the following difficulties:

(1) In our model the processors cannot come to an agreement which depends on the course of the algorithm [FLP].

(2) The algorithm might terminate prematurely because it started at a single vertex which failed before sending any messages.

(3) When a vertex fails, all the information stored there is lost.

(4) If a vertex which is supposed to send a message fails, all the other vertices might wait forever for that message.

Therefore, constructing reasonable models and devising algorithms for them remains an interesting open problem.

REFERENCES

[Ed] J. Edmonds, Edge disjoint branchings, in *Combinatorial Algorithms,* R. Rustin, ed., Algorithmics Press, New York, (1972), pp. 91-96.

[Ev] S. Even, *Graph Algorithms,* Computer Science Press, Potomac, Maryland (1979).

[ET] S. Even and R. E. Tarjan, Computing an s-t numbering, Th. Comp. Sci., 2 (1976), pp. 339-344.

[FLP] M. J. Fischer, N. A. Lynch, M. S. Paterson, Impossibility of distributed consensus with one faulty process, Proc. of the ACM Sym. on Principles of Database Systems, March 1983, pp. 1-7.

[IR] A. Itai and M. Rodeh, Covering a graph by circuits, Proceedings of the fifth colloquium on Automata, Languages and Programming, Udine, Italy, (1978), pp. 289-299.

[ILPR] A. Itai, R. J. Lipton, C. H. Papadimitriou and M. Rodeh, Covering a graph by simple circuits, SIAM J. on Computing, 10 (1981), pp. 746-750.

[L] A. Lempel, private communication.

[LEC] A. Lempel, S. Even and I. Cederbaum, An algorithm for planarity testing of graphs, in *Theory of graphs, international symposium,* Rome, July 1966, P. Rosentiehl, ed., Gordon and Breach, N.Y., 1967, pp. 215-232.

[NW] C. St. J. A. Nash-Williams, On orientations, connectivity and odd vertex pairing in finite graphs, Canad. J. Math., 12 (1960), pp. 555-567.

[R] J. Reif, Depth first search is inherently sequential, Aiken Computation Lab., Division of Applied Science, Harvard University, November 1983.

[Shi] Y. Shiloach, Edge disjoint branching in directed multigraphs, Inform. Proc. Letters, 8 (1979), pp. 24-27.

[Shr] L. Shrira, Ph.D. Dissertation, Computer Science Dept., Technion, Haifa, Israel, (1986).

[T] R. E. Tarjan, A good algorithm for edge disjoint branchings, Inform. Proc. Letters, 3 (1975), pp. 51-53.

[Z] A. Zehavi, Ph.D. Dissertation, Computer Science Dept., Technion, Haifa, Israel, 1986.

[ZI] A. Zehavi and A. Itai, Three tree connectivity, TR #406 (1986) and TR #462 (1987), Computer

Science Dept., Technion, Haifa, Israel.