Reconfiguration of Spanning Trees in Networks in the Presence of Node Failures

K. Ravindran, G. Singh and **P. Gupta**

Department of Computing and Information Sciences
Kansas State University
Manhattan, KS 66506

Abstract: Connectivity among a set of user entities in a network can be provided by a network level abstraction of acyclic graph (or spanning tree). The paper discusses the reconfiguration of a graph in the presence of failure of network nodes. A reconfiguration manifests as a *graph fragmentation* problem, whereby two or more disjoint subgraphs attempt to connect with one another to form a composite graph. Fragment interconnection requires contention resolution between fragments to avoid cycles. This paper presents two classes of contention resolution algorithms applicable for environments with potentially large number of fragments. They are based on pre-established ranking among fragments and random arbitration among fragments. The algorithms have been evaluated by simulation and compared. The algorithms are useful in supporting data multicasting across workstations and distributed computations involving data on different machines.

1 Introduction

One of the research problems on wide-area networks focuses on providing connectivity among users entities (or simply users) attached to various network nodes. The connectivity can be best provided by a network level abstraction of *acyclic graph* (or spanning tree) that has vertices in network nodes and edges over communication links between the nodes, with users residing in one or more vertices. All vertices and edges in a graph can participate in the flow of information across users. Typical applications are data multicasting across workstations in wide-area networks [1, 2], finding routes between transport hosts in a wide-area network [3, 4] and large scale parallel computations involving functions placed on different processors in a interconnected processor engine (*e.g.*, hypercubes).

A graph represents a logical topology of vertices and edges superimposed on the physical topology of nodes and links. This paper concerns with reconfiguration of a network graph, i.e., at the logical topology level, caused by node removals from the network due to failures (say, a crash). In this reconfiguration, the failure of a node containing a vertex causes execution of an algorithm by other nodes in the network that creates edges in the graph through rest of the network to circumvent the failed node and maintain connectivity among the remaining vertices. We assume that the network remains connected at the physical connectivity level in the presence of failures.

Typically, the failure of a node containing a vertex partitions the graph into one or more subgraphs not connected with one another, which we refer to as *fragments*. We model the reconfiguration problem as one of connecting two or more fragments together by creating edges. A reconfiguration algorithm is a distributed protocol executed by each fragment to connect to the other fragments. The fragment interconnection requires coordination among the concerned fragments so that the resulting graph composed from the fragments does not contain cycles. The coordination problem manifests itself as resolving contention among fragments attempting creation of edges simultaneously to determine which edges should be created and which should not be. The paper discusses the reconfiguration problem from this perspective of fragment coordination. The paper describes two classes of contention resolution protocols that are usable in an environment where the number of fragments can be potentially large (for instance, high speed network applications involving a large number of users):

- Based on a pre-established ranking among the fragments;
- Based on a random arbitration among the fragments.

The ranking based protocol results in a deterministic algorithm (i.e., an algorithm that terminates with known delay bounds). However, the algorithm needs to derive the ranking of fragments from node identifiers (id) in the underlying network. The random arbitration protocol results in a probabilistic algorithm. Here, the algorithm needs only a function to generate random numbers that allows the contending fragments to re-attempt interconnection after a random interval of time, but otherwise does not need any special support mechanism from the network. The performance of the two types of algorithms have been studied by simulation.

The algorithm deals with the case when there are no failures during a reconfiguration. However, the algorithm can be used as a building block in composing an extended

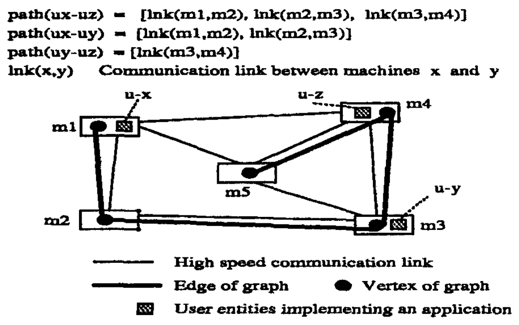
path(ux-uz) = [lnk(m1,m2), lnk(m2,m3), lnk(m3,m4)]
path(ux-uy) = [lnk(m1,m2), lnk(m2,m3)]
path(uy-uz) = [lnk(m3,m4)]
lnk(x,y) Communication link between machines x and y
u-x
u-z
m4
m1
m5
u-y
m2
m3
High speed communication link
Edge of graph
● Vertex of graph
▨ User entities implementing an application

Figure 1: Graph abstractions in network

algorithm to handle the case of such failures. The algorithmic framework of interconnecting graph fragments can also be generalized into a reconfiguration algorithm that deals with vertices being added and/or removed from a graph, whereby a node joining or leaving the network can also participate in the algorithm [6].

The paper is organized as follows: Section 2 describes the model of a graph and the concept of fragmentation and reconnection. Section 3 describes a base algorithm. Section 4 describes the ranking based contention resolution. Section 5 describes the random arbitration based contention resolution. Section 6 evaluates the two classes of algorithms. Section 7 summarizes the paper.

2 Reconfiguration Model

The network consists of a set of nodes $\mathcal{V}$ interconnected through point-to-point links (e.g., data switches linked by fiber optic cables in a high speed network, processors linked by interconnection bus in a hypercube). The user entities implementing an application may be attached to one or more network nodes $\mathcal{U}$, and are end-points of communication. The network has sufficient connectivity so that every node is reachable from every other node. See Figure 1.

A graph $\mathcal{G}(\widehat{\mathcal{U}}, \widehat{\mathcal{E}})$ provides a communication path (i.e., logical connectivity) among the users through a set of network nodes $\widehat{\mathcal{U}}$, where $\mathcal{U} \subseteq \widehat{\mathcal{U}} \subseteq \mathcal{V}$ and edges $\widehat{\mathcal{E}}$ map one-to-one to the intervening links. $\mathcal{G}$ may be viewed as an unrooted spanning tree for users, whereby the user residing in a node $u \in \mathcal{U}$ may execute algorithms for traversing a tree $\mathcal{T}_u(\mathcal{G})$ with root at u and leaves at the nodes $(\mathcal{U} - u)$. For instance, the user may be a transport level host in a network and the tree traversal over the edges may allow identification of data routes to the other hosts [4]. $\mathcal{G}$ thus embeds a set of trees $\{\mathcal{T}_u(\mathcal{G})\}_{\forall u \in \mathcal{U}}$.

When a node containing a vertex u' fails (say, due to a crash), a reconfiguration algorithm in the network creates a set of edges to interconnect the various vertices $\in \widehat{\mathcal{U}}$ that are adjacent to u'. Such an algorithm basically causes recomputing the data paths in $\mathcal{G}$ through the newly created edges.

2.1 Graph fragmentation

A fundamental problem in logical topology reconfigurations is the handling of *fragmentation* in a graph, whereby two or more subgraphs not connected with one another (or fragments) need to be connected together by creating edges and vertices over appropriate links and nodes respectively in the physical topology to form a composite graph. Typically, the failure of a node containing a vertex fragments the graph at this vertex so that these fragments reconnect with one another circumventing the failed node. See figure 2. Prior to failure, node S_9 had graph edges to its neighbors S_6, S_7, S_8, S_{10} and S_{11}. After the failure, the sets of nodes $\{S_1, S_3, S_4, S_7\}$, $\{S_2, S_5, S_6\}$, $\{S_{10}, S_{12}\}$, $\{S_8\}$ and $\{S_{11}\}$ constitute the fragments.

2.2 Fragment Interconnection

A fragment is identified by the id of one of its constituent nodes, namely a neighbor of the failed node that has a vertex and detects the failure (e.g, in Figure 2, $\{S_{10}, S_{12}\}$ is identified by the node id of S_{10}). The node which detects the failure starts the reconfiguration activity on behalf of its fragment to find physical paths to connect to other fragments. Often, there can be several neighbors of a failed node which may detect the failure and each of these nodes may initiate a reconfiguration activity.

When a set of fragments attempt to connect with one another, it is necessary to prevent cycles from being formed in the composite graph as a result of the reconfiguration. Preventing of cycles basically requires the fragments to coordinate with one another in some form. Consider, for instance, two fragments F_1 and F_2 attempting reconfiguration. If F_1 and F_2 connect independently with each other, the resulting graph can contain cycles. To prevent this, a possibility is for F_1 and F_2 to coordinate with each other so that only one them connects to the other. We model this coordination problem as one of resolving contention among the fragments to elect a coordinator which then creates edges to interconnect the various fragments. Refer to Figure 2. Suppose the fragment $\{S_{10}, S_{12}\}$ becomes the coordinator after contention with other fragments. The nodes S_{10} and S_{12} execute a protocol to create edges between the pairs of nodes (say) $[S_4, S_6]$, $[S_7, S_8]$, $[S_6, S_{10}]$ and $[S_{11}, S_{12}]$.

Thus, there are two basic components of a reconfiguration algorithm, viz., the election and the creation of edges. A simple but less efficient method may be to execute these components as distinct activities in that sequence. In our approach however, we have chosen to integrate these components into the algorithm as inseparable activities. This is to reduce the message overhead and the reconfiguration latency, i.e., time for reconfiguration to complete. In this approach, each fragment attempts to grow in size by creating edges towards other fragments until it meets another fragment. A contention between the fragments results at the point of meeting, whereupon one of the fragments wins the contest (using some contention resolution mechanism).

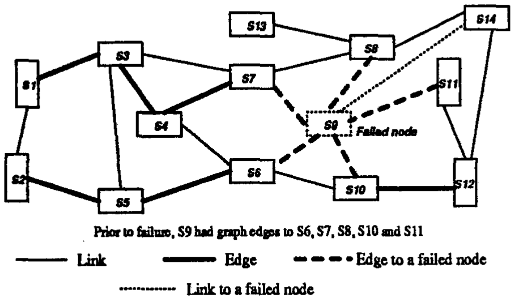

Figure 2: Fragmentation of graph

The winning fragment then expands by acquiring the losing fragment and attempts to further grow in size. The algorithm eventually completes by generating a single winner, with all fragments in the reconfiguration set connected together as a composite fragment acquired by the winner.

3 A base level algorithm

This section discusses the various aspects of our algorithmic realization of a reconfiguration activity.

3.1 Network model

The physical connectivity information is maintained in a node internal table. Consider a node S. Let $\{S^j\}_{j=1,2,\ldots,K}$ be a set of nodes that are neighbors to S, i.e., nodes with a link from S. Then $\{(id(S^j), port_j)\}$ constitutes the physical connectivity table of S, where $id(S^j)$ is the networkwide unique id of S^j and $port_j$ refers to a node level port (*i.e.*, network interface) through which S is linked to S^j. The network implements a function **Port_To**(S^j) to extract $port_j$ from the table and a function **Set_of_ports** to extract the list of ports to all neighboring nodes. With a unique id for every node in the network, the tables at all nodes specify the physical connectivity in the network. S may execute a separate protocol whereby S can detect the removal of a neighbor (say due to failure) and the addition of a node as a neighbor, and update the table accordingly.

A function **Assign_Edge**($port_j$) is provided that creates a directed edge from S originating through $port_j$ and incident on the neighbor S^j so that S becomes adjacent to S^j in the graph. A function **Get_Port** is also provided that generates a list of ports through which S has edges to its adjoining vertices.

We assume that the network provides reliable FIFO transmission of messages over a link and that transmission delay over a link is bounded. We also assume that the network diameter $\mathcal{D}$ is finite.

3.2 State variables

The information structures maintained by the algorithm are distributed, so each vertex only knows its adjacent vertices. In this sense, we provide a decentralised structure of reconfiguration algorithm. In our algorithm, each node maintains the following variables:

1. *Coord_so_far*: This is the local view of a node as to which fragment is coordinating. When a node detects the failure of a neighbor and starts a reconfiguration, it sets $Coord\_so\_far$ to its own id my_id.
2. *Port_to_coord*: This is the port in a node through which the fragment identified in *Coord_so_far* can be reached. When $Coord\_so\_far = my\_id$, the *Port_to_coord* is set to $NULL$.
3. *status*: This indicates either IDLE or WAIT, meaning respectively that a node is not participating or otherwise in a reconfiguration.
4. *recd_reply*[ ]: This gives information about a port as to whether a response has been received through that port indicating completion of the reconfiguration activity in that direction or otherwise.

3.3 Propagation of reconfiguration information

A node S sends a **Reconfig**(*node_list*, $frag\_id$) message through one or more of its ports, where *node_list* is a list of nodes — sorted in last-in-first-out order — through which the message has already traversed, and *frag_id* refers to the id of a neighbor that detected the failure of node. If S is the initiator of the reconfiguration, *node_list* and *frag_id* are each set to *my_id*, and **Reconfig** is sent through all the ports. Otherwise, the **Reconfig** is triggered by the arrival of a **Reconfig** at S from a neighbor; in this case, *my_id* is appended to *node_list*, and the **Reconfig** is forwarded through all ports except the port through which the **Reconfig** arrived. The flooding of messages is to propagate the information through the network that *frag_id* contends to become the coordinator for reconfiguration.

A node S' is initially in IDLE state. Upon receiving a **Reconfig** message, S' node sets $Coord\_so\_far = frag\_id$ and $Port\_to\_coord$ = **Port_To**(sender_of(**Reconfig**)) and moves into WAIT state, awaiting completion of the reconfiguration in the downstream direction (i.e., away from the coordinator).

S' generates a response message to sender_of(**Reconfig**) when there is no contention in the downstream direction from S'. The response message, referred to as **No_contention**, implies that a new edge cannot be created along the path it traverses. The message is generated by default in the case when S' does not have a port to forward the **Reconfig**, i.e., **Set_of_ports** = {**Port_To**(*sender_of*(**Reconfig**))} + Q, where Q is the set of ports (if any) along which either fragments have already been acquired or contention does not exist. The above actions are given by the following code skeleton:

```
receive Reconfig($node\_list, frag\_id$)
  if ($status$ = IDLE)
    Generate No_contention if no more ports
      to forward Reconfig
    $Coord\_so\_far := frag\_id$;
    $status$ := WAIT;
    $Port\_to\_coord$ := Port_to (sender_of (Reconfig));
    $\forall p \in$ (Set_of_Ports $-$ $Port\_to\_coord$)
      send Reconfig($node\_list + my\_id, frag\_id$)
            through $p$.
```

3.4 Detection of potential cycles

Since a reconfiguration wave propagates through the network by message flooding, the **Reconfig**($node\_list, frag\_id$) message received by S' may be due to cycles in the physical topology, which is detected and handled as below:

1. The **Reconfig** message is a copy of the message received by S' earlier, which may arise due to the sending of this message by some node S'' through its ports. A potential cycle that may exist in this case is indicated by the condition $Coord\_so\_far = frag\_id$. If the **Reconfig** message is received over a graph edge, S' infers that the message has entered the fragment to which S' originally belonged at two different points resulting in the $frag\_id$ colliding with itself over the edge. So S' needs to resolve the contention so that its fragment gets connected to $frag\_id$ at one point only. If the **Reconfig** message is not received over a graph edge, S' responds with a **No_contention**.
2. The **Reconfig** message is a copy of the message sent by S' earlier through a port but now received through another port. This message loop is indicated by the condition $my\_id(S') \in node\_list$. In this case, S' sends a **No_contention** through sender_of(**Reconfig**).

The actions are given by the following code skeleton:

```
receive Reconfig($\ldots, frag\_id$)
  if ($status$ = WAIT)
    $e$ := Port_to($sender\_of$(Reconfig));
    if ($frag\_id = Coord\_so\_far$ && $e \notin$ Get_Port)
      send No_contention through $e$;
      exit;
    if ($my\_id \in node\_list$)
      send No_contention through $e$;
      exit;
    resolve contention between $Coord\_so\_far$
          and $frag\_id$.
```

Consider the nodes $\{S_3, S_4, S_5, S_6, S_7, S_9\}$ in the topology illustrated in Figure 2. Suppose only S_6 detects the failure of S_9. S_6 sends a **Reconfig**($id(S_6), \{id(S_6)\}$) message to S_4 and S_5. the message arriving at S_4 is forwarded to S_3 and S_7 over edges, while that at S_5 is forwarded to S_3 over a link. When S_7 receives the **Reconfig** as forwarded from S_3 over a link after that from S_4, S_7 responds to S_3 with a **No_contention**. The latter prevents creation of an edge between S_3 and S_7, thereby avoiding a potential cycle between $\{S_3, S_4, S_7\}$. If S_3 received the **Reconfig** from S_4 and forwarded it to S_5, then S_3 and S_5 send a **No_contention** to each other. If S_3 received the **Reconfig** from S_5 and forwarded it to S_4, a contention occurs between S_3 and S_4 over an edge which, upon resolution, allows either S_3 to win generating a **No_contention** from S_4 to S_6 or S_4 to win generating a **No_contention** from S_3 to S_5. The case of S_6 receiving a **Reconfig** over the path $[S_5, S_3, S_4]$ is a message loop, detectable from the $node\_list = \{id(S_6), id(S_5), id(S_3), id(S_4)\}$.

Upon a contention resolution, $Port\_to\_coord$ is flipped in the direction towards the winning fragment and $Coord\_so\_far$ is updated to indicate this fragment id.

3.5 Fragment growth

Suppose a node S has sent out a **Reconfig**($\ldots, F_i$). Upon receipt of **No_contention** messages through all the ports where a response is awaited for, S can infer that there is no contention in the downstream direction. In this case, if there is a fragment rooted at S, then S should connect its vertex to a fragment in the upstream direction. For this purpose, S sends an **Accepted** message through $Port\_to\_coord$.

Thus a node S may receive either an **Accepted** message or a **No_contention** message through a port p, with the **Accepted** message indicating a successful acquisition of the fragments accessible through p. The termination condition for the algorithm is thus:

$$\textbf{Set\_of\_Ports} = \text{Port\_to\_coord} + \text{Q},$$

where Q is the set of ports through which either an **Accepted** or a **No_contention** message has been received. Upon evaluation of the termination condition to true, S sends:

- An **Accepted** message through the $Port\_to\_coord$ if: i) S has received an **Accepted** message through at least one port, or ii) a **No_contention** message has arrived through a port containing an edge and $Port\_to\_coord$ does not contain an edge;
- A **No_contention** message through the $Port\_to\_coord$ if S has received **No_contention** messages through all the ports.

In either case, S returns to the IDLE state completing its part of the reconfiguration. Both the messages inform the neighbor S' linked through $Port\_to\_coord$ that S has accepted F_i as the coordinator. Suppose S sends an **Accepted** message. If there is no edge over the link between S and S', an edge is created by these nodes by invoking **Assign_edge** ($Port\_to\_coord$) and **Assign_edge(Port_to**(sender_of(**Accepted**))) respectively. Such an edge interconnects the fragment with a vertex at S and that at S'. Thus the **Accepted** message allows the fragments acquired by S to be handed over to S'.
These actions are illustrated by the following code skeleton:

```
receive Accepted/No_contention
  if ∀p ∈ (Set_of_ports − Port_to_coord)
    if (at least one response received is Accepted)
      send Accepted through Port_to_coord;
      if Port_to_coord ∉ Get_Port
        Assign_Edge(Port_to_coord);
      else /* All responses received are No_contention */
      if ∃q ∈ (Set_of_ports − Port_to_coord)
        such that q ∈ Get_Port &&
                  Port_to_coord ∉ Get_Port
          send Accepted through Port_to_coord;
         Assign_Edge(Port_to_coord);
        else
          send No_contention through Port_to_coord.
```

For this purpose, the *recd_reply*[ ] also indicates whether an **Accepted** or a **No_contention** was received through a given port.

Before an **Accepted** message reaches S' it is possible that S' might have changed the *Coord_so_far* and consequently the *Port_to_coord* to a wave originated by a fragment F_j. Thus the fragment grown by the wave from F_i is inherited by the wave from F_j. This can be viewed as fragment acquisition by F_j from F_i.

As can be seen, various fragments keep combining with each other until the algorithm terminates at the elected coordinator node where $Port\_to\_coord = NULL$. Since the edge created by an **Accepted** message is never deleted, a newly created fragment is always incremental to the previous fragments. These aspects guarantee the steady growth of fragments. The propagation of information across various fragments is an instance of diffusing computations [7].

3.6 A sample scenario

Consider the fragmentation scenario illustrated in Figure 2. Suppose S_{10} is the only initiator of reconfiguration. S_{10} will send a **Reconfig** message to S_6 and S_{12}. The message through S_6 will eventually reach S_{13} through S_8, whereupon S_{13} generates a **No_contention** message to S_8. Since S_8 is a vertex in the graph and has only a link to S_7, S_8 transforms the **No_contention** message into an **Accepted** message and sends it to S_7, which creates an edge connecting $[S_7, S_8]$. Suppose the **Accepted** message passes through $[S_4, S_3, S_1]$. When the message traverses from S_1 to S_2, an edge is created connecting S_1 and S_2. The **Accepted** message then passes through $[S_2, S_5, S_6]$ and upon traversing from S_6 to S_{10} causes the latter nodes to be connected by an edge. The **Reconfig** message which reaches S_{11} causes an **Accepted** message to be generated since S_{11} is a vertex and has only a link to S_{12}. This **Accepted** message, upon receipt by S_{12}, creates an edge connecting S_{11} and S_{12} and then gets forwarded to S_{10}. When both the **Accepted** messages arrive at S_{10}, the algorithm terminates since S_{10} has its *Port_to_coord* set to $NULL$.

Thus, we may have an algorithm which, using a contention resolution mechanism, is able to reconfigure a graph. In the following sections 4 and 5, we discuss contention resolution using respectively: (i) ranking among fragments, and (ii) random arbitration among fragments.

4 Ranking based contention resolution

In this section, we will use unique ranking among node ids as a mechanism to resolve contention between two fragments. The network guarantees that the ranking remains stable during an execution of the algorithm.

4.1 Winning contention

We say a fragment F_i is ranked higher than a fragment F_j if $\text{id}(F_i) > \text{id}(F_j)$, and vice-versa. Consider the meeting of fragments F_i and F_j when they attempt to expand. If F_i is ranked higher than F_j, then F_i is the winner and F_j is the loser; otherwise F_j is the winner and F_i is the loser.

Suppose a node S is in the WAIT state, has $Coord\_so\_far = F_j$, and receives a **Reconfig**$(\ldots, F_i)$. The following code skeleton illustrates the activities of S:

```
if ( (Coord_so_far > F_i)||
   (Coord_so_far = F_i &&
        my_id > id(sender_of(Reconfig))) )
     send Stop(Coord_so_far) through
          Port_to(sender_of(Reconfig));
else
  Coord_so_far := F_i;
  send Stop (F_i) through Port_to_coord;
  Port_to_coord := Port_to(sender_of(Reconfig)).
```

As can be seen, when F_j wins, S sends a **Stop** message to the sender of **Reconfig** to inform all vertices in F_i that no other fragment ranked less than F_j will be contending on the path beyond the point at which the collision occurred. When F_j loses, S sends a **Stop** through *Port_to_coord* to indicate to other vertices in F_j about its losing the contention and then flips its *Port_to_coord* towards the winning fragment F_i. S then sets *coord_so_far* to F_i.

4.2 Fragment acquisition

Consider the arrival of **Stop**(F_i) at a node S with $coord\_so\_far = F_j$. Suppose F_i is of higher rank than F_j. Then S sends **Stop**(F_i) through *Port_to_coord* to notify the win of F_i over F_j to all vertices in F_j. S then flips the *Port_to_coord* towards the sender of **Stop**(F_i). If F_i is of lower rank than F_j the S sends a **Stop**(F_j) to the sender of **Stop**(F_i) to notify the win of F_j over F_i to all vertices in F_i. This is given by the following code skeleton:

```
receive Stop(F_i) through a port p
  if (F_i > coord_so_far)
    coord_so_far := F_i;
    send Stop(F_i) through Port_to_coord;
    Port_to_coord := p;
  if (F_i = coord_so_far)
    if (Port_to_coord ∉ Get_Port)
      send No_contention through Port_to_coord;
```

```
            mark $recd\_reply[Port\_to\_coord]$ as having
               received **No_contention**;
          else
             send **Stop**($F_i$) through $Port\_to\_coord$;
          $Port\_to\_coord := p$;
      if ($F_i < coord\_so\_far$)
          send **Stop**($F_j$) through $p$.
```

Since $coord\_so\_far$ is updated to F_i only when its current value (i.e., id(F_j)) is less than id(F_i) and it is accompanied by a **Stop**(F_i) to all vertices in F_j, each of these vertices will eventually agree at the winning of F_i by setting their $Coord\_so\_far$ to id(F_i). Thus the fragment F_j is acquired by F_i and the winning of F_i is propagated across all vertices in F_j. As can be seen, this guarantees monotonically increasing values of $Coord\_so\_far$. It means that a tree is rooted currently at some vertex acting as the coordinator, and can change to any other vertex with a higher rank.

4.3 Deterministic Termination

Use of ranking to arbitrate between colliding fragments terminates the algorithm deterministically. Suppose the fragments F_1, F_2, and F_4 start the reconfiguration and F_3 is too slow to start the reconfiguration. By the time F_3 starts, some other fragment subdues it. Now it is up to F_1, F_2, and F_4 to complete the reconfiguration. Some of the edges could be accumulated by F_1, some by F_2 and some by F_4. Let us say $id(F_4) > id(F_2) > id(F_1)$. Since this information is stable, the protocol will eventually terminate with F_4 as the coordinator. This is because whenever there is contention, the higher ranked wave wins. Thus F_2 can win over F_1, but F_4 will win over both F_2 and F_1. This happens in finite time since message transmission delay over a link is bounded and contentions will be resolved within the maximum diameter of the network $\mathcal{D}$.

5 Random Arbitration to resolve contention

When the ranking information is not available to the algorithm, it is not possible to declare a fragment F_i as a winner over a fragment F_j or vice-versa, when the two fragments meet resulting in a collision. Random arbitration can declare any of the fragments as a winner in a probabilistic manner. This however leads to non-determinism, because F_i could win over a fragment F_j at one meeting point but could loose at another point. So this non-determinism has to be tackled in the algorithm.

5.1 Random backoff

When two fragments F_i and F_j meet, resulting in a collision, the algorithm requires both the fragments to backoff for a random period of time before attempting reconfiguration in the direction where the collision occurred. This may allow one fragment to win over the other after one or more attempts.

In the algorithm, the two nodes detecting the collision move from the WAIT state into a BACKOFF state. In this state, each node will retain its $Coord\_so\_far$ and $Port\_to\_coord$ values, but remains inactive (or frozen), i.e., does not send any messages for a certain interval of time. After this interval, a node say F_i re-sends the **Reconfig** message and moves into WAIT state. If the message reaches F_j while the latter is still in the BACKOFF state, then F_i is declared a winner over F_j. But if F_j had become active in the meanwhile, there will be contention again, and the same backoff procedure will be repeated. A winner then finally continues the reconfiguration.

Consider the arrival of **Reconfig**($\ldots, F_i$) at a node S in a WAIT state and $Coord\_so\_far = F_j$. Contention is said to have occurred if $F_i \neq F_j$. This causes the wave F_j to backoff. Since the message **Reconfig**($\ldots, F_j$) sent by S will also suffer collision at the neighbor that sent the **Reconfig**($\ldots, F_i$), S simply assumes that it has got the latter fragment back and that it is going to restart the reconfiguration after the backoff interval. Also, when S moves to the BACKOFF state, it sends an **Abort** message through all ports over which neither an **Accepted** nor a **No_contention** has been received. The **Abort** message informs all vertices in the fragment about the contention. This is illustrated by the following code skeleton:

```
receive Reconfig($\ldots, F_i$)
  if ($status$ = WAIT)
     if ( ($Coord\_so\_far \neq F_i$) || ($Coord\_so\_far = F_i$ &&
          Port_To(sender_of(Reconfig)) $\in$ Get_Port))
          $status$ := BACKOFF;
          $\forall p \in$ (Set_of_ports − sender_of(Reconfig))
            if $recd\_reply[p]$ indicates no response has
               been received
                send Abort through $p$;
     start random backoff timer.
```

The **Abort** message arriving at a node S in a WAIT state freezes its reconfiguration activity by placing S in BACKOFF state. The reconfiguration at S is re-activated from where it was frozen upon receiving a **Reconfig** message again. However, if S is in IDLE or BACKOFF state, the arrival of an **Abort** does not affect S. This is illustrated by the following code skeleton:

```
receive Abort
  if ($status \neq$ BACKOFF && $status \neq$ IDLE)
     $status$ := BACKOFF;
     if ($coord\_so\_far \neq my\_id$)
         $\forall p \in$ (Set_of_ports − sender_of(Abort))
            if $recd\_reply[p]$ indicates no response has
               been received
            send Abort through $p$.
```

As mentioned earlier, a node S in the BACKOFF state does not send any messages. However, if an **Accepted** or a **No_contention** is received on a port while in the BACKOFF state, the $recd\_reply[p]$ is updated accordingly.

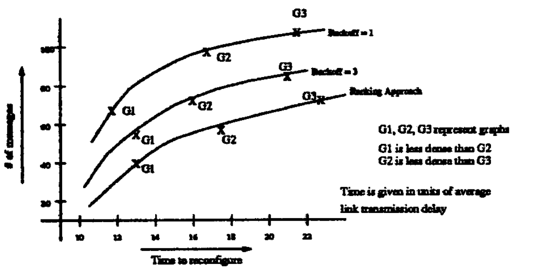

Figure 3: Message Complexity Vs Time Complexity

5.2 Restart of reconfiguration

The BACKOFF state can be seen as a frozen state because the node, having generated **Abort** messages, wakes up upon arrival of a **Reconfig** message. In addition, a node in BACKOFF state may also wake up upon expiry of the backoff timer. If a node in BACKOFF state is woken up by the arrival of a **Reconfig** message, the node loses the contention in favor of the fragment that sent the **Reconfig**, and hence cancels the backoff timer. Upon waking up, a node re-sends a reconfiguration message on all ports over which neither an **Accepted** nor a **No_contention** was received. This is illustrated by the following code skeleton:

```
receive Reconfig(..., $F_i$)
    if ($status$ = BACKOFF)
        cancel backoff timer;
        $status$ := WAIT;
        if ($Coord\_so\_far = F_i$)
            if $\forall p \in$ (Set_of_ports − sender_of(Reconfig))
              $recd\_reply[p]$ indicates no response has
               been received
              send Reconfig(..., $Coord\_so\_far$) through $p$.
```

The **Accepted** or **No_contention** messages received by a node while in backoff state cause the edges created along the direction of these messages to be frozen for acquisition by a subsequently winning wave.

Consider, for instance, the fragments F_i and F_j contending at two different points. F_i may win over F_j at one point and lose at the other point. In this case, both F_i and F_j will regenerate the reconfiguration activities at the winning points. F_i and F_j may collide again in these regenerated activities, possibly repeating the above scenario. This continues until one of the fragments wins at all the points. Since the number of times a fragment will backoff is unknown, we cannot place an upper bound on the execution time of the protocol. However, since every reconfiguration activity potentially causes addition of new edges and the already acquired edges are never deleted, the graph grows steadily and finally terminates at the fragment which emerged as the eventual winner.

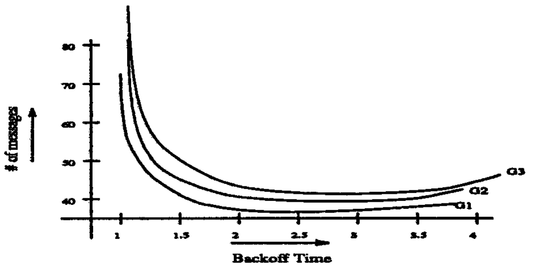

Figure 4: Effect of varying Backoff time on the Message Complexity

6 Evaluation of the algorithms

We discuss in this section the performance of ranking based and random backoff based reconfiguration algorithm by studying their simulated behavior.

6.1 Simulation Studies

The reconfiguration algorithms are studied by employing discrete event simulation. The message transfer delay over a link is modelled as a random variable. For fiber optic networks, the link delay can range from 5-10 microseconds for a 100 byte control message. Various arbitrary network topologies were generated by specifying the nodes and the links between them and the initial graph.

6.2 Performance evaluation

As the density of the graph increased, i.e., as the average degree of connectivity increased, the time and message complexity increased. For sparse graphs, fewer messages were generated as compared to graphs which were more dense, as shown in figure 3. Here, for the ranking approach, as we increased the average degree of connectivity $\bar{e}$, the number of messages increased within the range 20 - 70. Correspondingly, the time taken to termination of the protocol also increased within a range of 12 - 22 time units. For the Random backoff approach, the messages required increased on increasing the average degree of connectivity $\bar{e}$ as well as by decreasing the random backoff interval.

The time unit used in the reconfiguration latency represents the average transfer delay over a link. Thus a time delay of 100 units to reconfigure means a reconfiguration latency of 1 millisecond (approx) on a high speed network. We believe that such a latency will have a negligible effect on the network applications. In a video conferencing for instance, a glitch in the workstation window may not last for more than a few milliseconds.

Using the random arbitration technique, we observed that there was a significant change in the number of messages required, as shown in figure 4. Here, for a graph G1, with a lower average degree of connectivity $\bar{e}$, the number of messages decreased as the backoff time was increased to 3 time units. Beyond that, the number of messages again

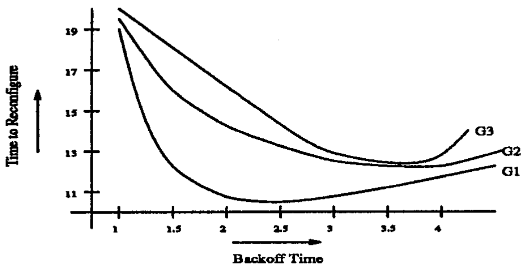

Figure 5: Effect of varying Backoff time on the Time Complexity

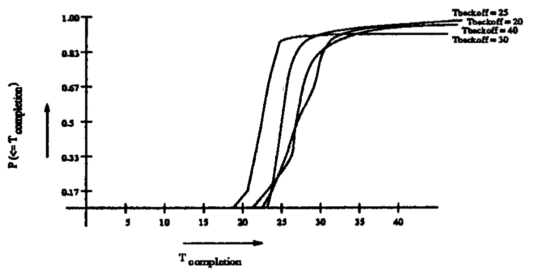

Figure 6: Probability distribution of reconfiguration delay for random backoff approach

started increasing. This shows that a certain backoff value is optimum for a certain graph. For graphs G1 and G2, where $\bar{e}(G1) < \bar{e}(G2) < \bar{e}(G3)$, as $\bar{e}$ was increased, the number of messages increased.

Similarly, there was a significant change in the time required to termination, as shown in figure 5. Here, as the backoff time was varied between 2 and 4 time units, the total time taken varied between 10 and 20 time units. As the reconfiguration protocol terminates, we observed that the reconfigured path was incremental.

The probabilistic termination of the random backoff approach can be seen in figure 6. Here we simulated the algorithm for various random number sequences and upon varying the backoff time, the total time varied within an acceptable range.

7 Summary

The paper described reconfiguration of graph-connected paths in a network in the presence of node failures. We modeled the reconfiguration problem as one of connecting two or more subgraphs together which hitherto were not connected with one another. The fragment interconnection require coordination among the concerned fragments so that the new graph composed from the fragments does not contain cycles. The coordination problem manifests itself as resolving contention among the fragments to determine the edges to be created. To deal with environments with potentially large number of fragments, the paper described two classes of protocols:

- Based on a pre-established ranking among the fragments;
- Based on a random arbitration among the fragments.

The paper designed the reconfiguration algorithms and evaluated them. We are currently extending the algorithms to a general case of dealing with multiple failures and joining/leaving of nodes [6]. For instance, dealing with multiple failures requires composing execution of multiple instances of the reconfiguration algorithms discussed above.

The reconfiguration algorithms proposed in this paper are different from that in Autonet [5] in that this work requires networkwide update of the entire physical topology information whenever a change occurs. [8] also proposed a reconfiguration algorithm but it requires some centralized information ('replacement sets') to reconfigure the tree. [9] gives an algorithm for constructing a spanning tree but does not deal with reconfigurations.

References

[1] B. M. Waxman. **Routing of Multipoint Connections.** In *IEEE Journal on Selected Areas in Communications*, Vol.SAC-6, No.9,pp.1617-1622, Dec 1988.

[2] K. Ravindran, M. Sankhla and P. Gupta. **Multicast Models and Routing Algorithms for High Speed Multi-service Networks** *Technical Report*, Kansas State University, Feb 1992.

[3] R. Perlman. **An Algorithm for Distributed Computation of a Spanning Tree in an Extended LAN.** In Proc. *Communication Architectures and Protocols*, ACM SIGCOMM, pp. 44-53, Sept 1985.

[4] D. Bertsekas and R. Gallager. **Routing in Data Networks** Chapter 5 in *Routing in Data Networks.*, Prentice Hall Publ. Co., 1992.

[5] M. D. Schroeder and T. L. Rodeheffer. **Automatic Reconfiguration in Autonet** In *Technical Report*, Digital Equipment Corporation (Systems Research Center), Palo Alto (CA), 1991.

[6] K. Ravindran and G. Singh. **Algorithmic Frameworks for Reconfiguration of Spanning Trees in Distributed networks.** *Work in Progress*, Kansas State University, Oct. 1992.

[7] E. Dijkstra and C. Scholteh. **Termination Detection for Diffusing Computations** *Information Processing Letters*, 11(1), Aug 1980.

[8] I. A. Cimet, C. Cheng and S. P.R. Kumar. **On the Design of Resilient Protocols for Spanning Tree Problems.** *In IEEE transactions in Communications* 1989.

[9] R. G. Gallagher, P. Humblet and P. Spira. **A distributed algorithm for minimal spanning tree,** *ACM Transactions of Programming languages and Systems*, 30(12), Dec 1983.