A Distributed Multi-commodity Flow Approximation Algorithm for Path Restoration

S.Venkatesan, Maulin Patel, and Neeraj Mittal
Department of Computer Science and Telecommunication Engineering Program
University of Texas at Dallas, Richardson, Texas 75083–0688
venky@utdallas.edu

***Abstract*— The distributed multicommodity network flow problem has many applications in the areas of routing and telecommunications. Compared to decades of research in centralized multicommodity flow algorithms, distributed algorithms have received very little attention. This paper presents an online distributed multicommodity flow approximation algorithm specifically tailored for *path restoration*, which is an important problem in building survivable telecommunication backbone networks. Path-based restoration schemes are known their for high restoration efficiencies and their ability to protect against single link, multiple link and node failures. Our algorithm uses $O(|E|diam^2)$ messages and $O(diam^2)$ time in the worst case, and substantially less messages and time in practical networks. When simulated on a sample real-life backbone network similar to those used by the telecommunication service providers, the algorithm finds a solution significantly faster than many published algorithms.**

I. Introduction

The multicommodity network flow (MCNF) [1] problem consists of several different commodities (each one has an associated demand) which are to be sent from their respective sources (origins) to their respective destinations through a common network such that the total flow on each link does not exceed its capacity. The MCNF problem is a well-studied problem. It has a wide variety of applications, typically in the areas of production planning, warehousing, transportation and telecommunication networks [2]. Several centralized algorithms have been proposed by many researchers to solve MCNF problems [1], [3], [4]. Many problems in the areas of routing and telecommunication systems can be expressed as MCNF problems. However, some of those problems require distributed algorithms for MCNF problems and traffic restoration in backbone networks is one of them.

A. Online Restoration Problem

Online restoration of disrupted traffic in the event of a failure is a prime concern in self healing (fault-tolerant) mesh telecommunication networks. When a link and/or node fails, the traffic going through that link or node is disrupted. To ensure that the degree of disruption caused by outages is minimized, efficient algorithms are needed for real-time restoration of the disrupted traffic. The online restoration scheme attempts to restore the disrupted traffic by diverting it to alternate routes bypassing the failed component(s). The traffic must be restored within a critical time. Many different restoration techniques have been proposed in the literature [5]–[15]. We focus on path restoration for mesh-type self-healing networks with shared spare capacity.

B. Path Restoration

Path Restoration [7]–[11], [14] reroutes each disrupted path over a single alternate path (or a set of alternate paths) between the source and the destination of the primary path, using the spare capacity of the network. Path Restoration involves finding alternate paths for each disrupted source-destination pair separately. Thus, finding restoration paths using path restoration scheme is a multi-commodity flow problem, which is $\mathcal{NP}$-complete if the flow values are integers [16]. Path restoration is known for its high restoration efficiencies and its ability to protect against single link, multiple link and node failures.

The backbone network has a set of say n paths between n sources and n destinations. (Note that some of the paths may share the same source node and some of the paths may share the same destination node.) Each path P_i also has a certain traffic flow requirement or capacities, say, c_i. For a path P_i to go over a link l, link l dedicates c_i units of link capacity to path P_i. Thus, link l dedicates capacities equal to the sum of the capacities of all of the paths using l. This capacity is called working capacity of l and denotes the amount of live traffic carried by l. In addition to the working capacity, a link has spare capacity. The spare capacity of a link does not carry any live traffic.

When a link fails, all the paths using that link are disrupted. In order to survive this link failure, the disrupted paths will have to be rerouted so that they use other links. A path can be rerouted on a link, only when that link has sufficient spares to carry the rerouted traffic (in addition to its live traffic). Spare capacity allocation is done off-line (before failures) during network planning time. The on line path restoration algorithm is run at the time of failures to find alternate paths and switch traffic.

For example, consider the small network consisting of six nodes shown in Figure 1(a). For each link, two numbers are shown; the first number denotes the total capacity (working capacity plus spare capacity) and the second number denotes the working capacity. Link (3,4) is used by two paths: path P_1 carrying 1 unit of flow from node 1 to node 6 via nodes 3 and 4; path P_2 carrying 1 unit of flow from node 2 to node 5 via

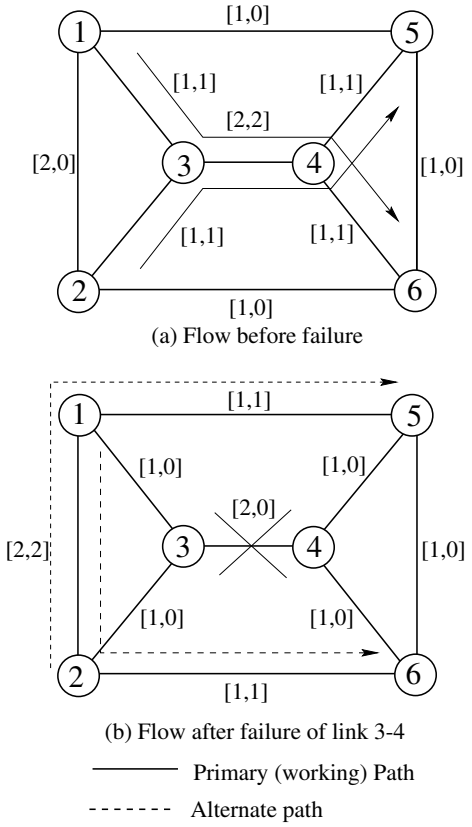


Fig. 1. Path Restoration Example

nodes 3 and 4. Only links (1,3), (2,3), (3,4), (4,5), (4,6) carry live traffic. If link (3,4) fails, both P_1 and P_2 are disrupted. The rerouted path for P_1 is (1,2,6) and for P_2 is (2,1,5). Note that both rerouted paths go over links without exceeding spare capacities available in any of the links.

C. Classification of Restoration Schemes

The restoration schemes for mesh networks can be classified by their computation timing, by their execution mechanisms and by the type of rerouting used [8]. The real time approach [5]–[7] finds alternate paths after the failure has been detected, while the preplanned approach [8]–[10] precomputes the alternate paths for the given failure scenarios. Preplanned approach requires that all the nodes maintain upto-date databases of current network state which requires elaborate scheme of database updating and constant recalculation of routing tables. Real-time approach is suitable for the networks where traffic patterns change very frequently. Preplanned approach is faster than real-time approach but suffers from poorer capacity utilization than real-time approach [17]. There are two types of execution mechanisms: Distributed [5]–[8] and centralized [9]. In contrast to the distributed schemes, the centralized schemes perform the computation needed to find alternate paths at a central cite, where accurate information about the current network state is assumed to be available. Centralized schemes are capacity efficient but have single point of failure, communication overhead and scalability issues.

D. Performance criteria

The selection of an appropriate restoration scheme depends on many factors in addition to spare capacity requirement. Some of the important performance criteria are: computational efficiency, restoration speed, complexity, and scalability [8], [9], [11]. The computational efficiency refers to the processing power and the memory required for the computation. The restoration speed is the time required to restore the disrupted traffic after a failure occurs. Ease of online restoration operation defines the complexity. A good scheme should offer acceptable performance as the network size and the number of demands grows (scalability).

To minimize data loss and call dropping probability, the restoration should be completed within 2 seconds of failure occurrence and preferably within 1 second as per Bellcore's advisory [18].

E. Our Contribution

We propose a new real-time distributed control restoration scheme. The algorithm is an online distributed multicommodity flow approximation algorithm for path restoration. Simulation results on realistic backbone topologies show that our scheme has the potential to achieve target restoration speed.

The rest of the paper is organized as follows:Section II describes the system model. Section III explains the details of the newly developed restoration algorithm. The details of a preprocessing stage are described in section IV. Complexity analysis is done in Section V. Section VI presents the results of our simulations. A brif summary of related work is given in Section VII and Section VIII points out some directions for future research in related areas.

II. System Model

The backbone network consists of nodes and bidirectional communication links. In the network, a path starts from a source node, ends at a destination node and consists of a series of neighboring nodes[1] along with the connecting links. For each link, we are given the amount of live traffic and spare capacity available.

A. Knowledge of the Nodes Before Failures

Each node has local topological information, in addition to the information obtained in the preprocessing step. A node has the following knowledge before the restoration algorithm is started:

1) Number of links incident on it.
2) For each link incident on that node, the following information is stored. (Some of this information is obtained from the operator of the network.)
   a) The id of the node at the other end.
   b) Working and spare capacities of that link.
   c) Status of the link—down or up.

[1]Two nodes are neighbors if they are connected by a single bidirectional link.

d) Number of paths going through that link.
e) Path information for the link. For each path going through that link, the identity of the path (id of the source of the path, id of the destination of the path and the capacity of the path).

This knowledge is available before any link fails. Also, no node knows the entire topology of the network.

Traditionally, only single link failures have been considered since the probability of multiple simultaneous failures is very small. Thus, when the telecommunication companies design the spare capacity network, only single link failures are assumed to occur. Our algorithm can cope with (and recover from) multiple link and node failures. A link *fails totally* if it cannot be used for transmission in both directions. Partial failures occur when the link loses its ability to transmit messages in one direction. An underlying failure detection mechanism exists to detect the failure of a link.

III. A New Algorithm

A. Failure Detection and Failure Notification

Failure notification occurs after failure detection. Immediately after detecting the link failure, the two end nodes of the faulty link assemble a *failed* message and send it on all of the outgoing links. The *failed message* contains the number of paths going through the faulty link and the path id of the paths going through the faulty link. For each path, path id consists of the source node id, destination node id, and the amount of traffic going on that path.

When an arbitrary node u receives the *failed* message, it checks if it has already received this message. If not, it records the *failed* message locally and forwards the message (once) on all links except on the link on which it received the *failed* message.

It is possible to release the capacities allocated to paths that get disrupted. These relased capacities can be added to the spare capacity on the links.

If the node had already received the *failed* message (which can be easily determined by checking if the *failed* message has been locally recorded), then the *failed* message is discarded.

The two end nodes of the faulty link start the algorithm immediately after detecting the link failures and sending out the *failed* message.

The other nodes begin the algorithm immediately after receiving the first *failed* message.

B. Maintaining the Nodes in Rhythm

Our algorithm requires that the nodes be kept in "rhythm." Each node maintains a step number. The step numbers have the following property:

A message m sent by node u to neighboring node v when u's step number is i will be received by v when v's step number is $i + 1$. (Note that u and v are directly connected by a link.) One of the three synchronizers α, β, γ proposed by Awerbuch [19] can be used. We use a synchronizer that is simpler and faster than the three synchronizers. We ensure that each node sends exactly one message on each link. Thus, it is easy for a node to check if it can proceed to the next step: as soon as it receives one message on each link for the current step, it can proceed to the next step.

The following two rules are used to keep the nodes in rhythm:

1) During step i of the restoration algorithm, if u has a message to send to its neighbor v, then u sends it. Otherwise (if u has no message to send during step i), u creates a *step_completed* message (which is similar to a null message) and sends the *step_completed* message to v. This step is performed for each neighbor of u.
2) After u receives one message from each neighbor and processes them, u proceeds to the next step.

The cumulative step numbers are maintained locally only. The step numbers are not sent in any of the messages.

C. Message Set

All of the messages described here contain the id of the sender and the id of the recipient of the messages. Also, we have assumed that the number of nodes, links, and path capacities do not exceed 256. Thus, one byte is sufficient to represent each of these.

1) Failed Message: This message notifies all nodes that a particular link has failed. This message has two fields: (1) the number of paths of traffic going through the faulty link (let call this number n) and (2) the identities of the n disrupted paths (in the form of Source id, Destination id, and the amount of traffic for each path). The *failed* message can be represented using $2 + 3 \times n$ bytes, one byte to store the message type, one byte to store the number of disrupted paths and three bytes to store path id (the source id of the path, the destination id of the path, and the amount of traffic on that path).

2) Explore Message: The *explore* messages are generated at the source nodes of the disrupted paths and they are propagated by the other nodes.

Each *explore* message contains the following fields:

1) The source id of one disrupted path for which this *explore* message is sent to find alternate paths.
2) Destination id of the path.
3) Amount of traffic to be restored for the path.
4) Hop count.

The total number of bytes needed for *explore* message is seven—one byte to store the message type and one byte each for the four fields and two bytes for the sender and recipient ids.

3) Return Message: The *return* messages are generated by the destination nodes of the disrupted paths and are propagated by the other nodes.

The *return* message contains the following information:

1) Source id.
2) Destination id.
3) Amount of traffic to be restored.
4) Information about the subnetwork traversed by the *return* messages.

The subnetwork consists of all the links that have been traversed by the *return* message so far. The *return* messages vary in size. The total number of bytes needed for the *return* message is $6 + 3 \times n_l$. Of this, one byte is for representing the type of message, one byte each for the source id, the destination id, the amount of traffic to be restored, sender id and recipient id, and n_l is the number of links representing the subnetwork. (The subnetwork is simply a list of links.) For each link, we have three bytes, one byte each for the id of the two end nodes of the network, and one byte for the number of spares allocated for this path.

4) Step_Completed Message: This message is sent to indicate that the sender has completed one step. When a node u has no *explore* or *return* message to send to a neighbor v during a step, node u sends a *step_completed* message to v.

5) Path_Restored Message: This message is sent once by the source node of each path when the source node restores the disrupted path completely. This message contains the path id (source id and destination id). The source and destination ids are one byte each and the message type is one byte long. Thus, this message is three bytes long.

D. A High Level Description of the Algorithm

The restoration algorithm proceeds in *iterations*. Each iteration consists of two phases—an explore phase and a return phase. The explore phase is started by the source nodes of the disrupted paths. The return phase is started by the destination nodes of the disrupted paths.

Consider the i^{th} iteration. (Initially, $i = 1$.) Each phase (explore phase and the return phase) of the i^{th} iteration consists of exactly $i + 1$ **steps**. Thus, the i^{th} iteration consists of $2 \times (i + 1)$ steps.

a) A Step:: During each step (except the first step of the explore phase and the first step of the return phase), a node waits for a message (*explore* or a *step_completed* message in the case of *explore* phase and *return* or *step_completed* message in the case of the return phase) from each neighbor, processes the received messages and sends one message (*explore* or a *step_completed* message in the case of *explore* phase and *return* or *step_completed* message in the case of the return phase) to each neighbor. Each source S_j finds the subnetwork that consists of all the nodes (and the links) that are reachable from S_j within $i+1$ hops. Using this subnetwork only, S_j locally finds the maximum amount of traffic that can be restored between S_j and D_j and restores the maximum possible traffic. If all of the nodes are in rhythm, then it is easy to see that by the end of the i^{th} iteration, all paths of hopcount less than or equal to $i + 1$ from S_1 to D_1, from S_2 to D_2, ..., S_n to D_n are considered. We next describe how the subnetwork is obtained at each source. Determining the maximum amount of traffic on the subnetwork is described next.

b) Explore Phase:: The i^{th} iteration starts with the explore phase. The source nodes send *explore* messages. The *explore* messages traverse links looking for the correct destination nodes. To control the flow of *explore* messages, a hop count is given. For the i^{th} iteration, the hop count is set to $i + 1$ at the source nodes. When an *explore* message traverses a link, the hop count field is decremented. When hop count reaches zero, the *explore* message is not propagated further. When an *explore* message reaches the intended destination node, the *explore* messages are not propagated further. Instead, we wait till the end of the explore phase and then start the return phase.

c) Return Phase:: During the return phase, the destination nodes of the disrupted paths assemble a *return* message if an *explore* message sent by the corresponding source node has reached the destination node during the explore phase of the present iteration. (All other nodes send *step_completed* messages.) The *return* messages are sent by the destination nodes to their neighbors and are propagated. The *return* messages also acquire spare capacities on each link traversed by them. Contention for spares of the links is resolved in this phase. (Details in § III-E.4.) When a *return* message reaches the source node, the received information is locally stored. Recall that the *return* message has both the source and destination ids in its field.

At the end of the return phase (at the end of the $i + 1^{st}$ step of the return phase), each source node locally has the subnetwork that consists of the nodes and the links (along with the spares acquired for restoring the disrupted traffic) that have been explored by the *explore* messages. Using the subnetwork, each source node locally runs the max flow algorithm and determines the maximum amount of traffic possible between the source and the destination using the acquired spare capacities as edge capacities. Any of the mex flow algorithms (sequential) algorithms can be used. If the computed maximum traffic is less than the original amount of traffic carried by the path, then more iterations are needed and the source node proceeds to the next iteration. Otherwise, the source node knows that one path has been restored and the source node sends a *path_restored* message. It also releases the capacities on links that have been allocated for restoring this path but are not needed.

When a *path_restored* message is sent by all of the n source nodes (where n is the number of disrupted paths), the nodes receiving these n distinct *path_restored* messages terminate the restoration algorithm.

E. Algorithm Details

1) Algorithm Initiation Phase: The algorithm starts with the algorithm initiation phase. This phase is started when a link failure is detected. The *failed* message is assembled and broadcast to all of the nodes.

2) Explore Phase:

Actions of node v at the beginning of an iteration

As noted earlier, the i^{th} iteration consists of exactly $2 \times (i + 1)$ steps. Also, the i^{th} iteration begins at a node only when the previous iteration ends. Thus, a node can locally determine when an iteration begins and when an iteration ends.

Consider an arbitrary node v. If v is a source node of some path P_j and if P_j has not been fully restored, then it begins

the explore phase of the i^{th} iteration (after it ends the $i-1^{st}$ iteration). At this time, node v assembles an *explore* message. The hop count field of the *explore* message is equal to $i+1$. The field "amount of traffic to be restored" is initialized to the difference between the path's original traffic and the amount of traffic that has been restored for that path in all of the iterations so far.

If v is the source of more than one disrupted path, then an *explore* message is assembled for each path for whom the source is v.

If node v is not a source node of any of the disrupted paths, then node v assembles a *step_completed* message locally.

After assembling either *explore* message(s) or a *step_completed* message, node v sends the assembled message(s) on all of its outgoing links. Node v then waits for a message (*explore* or a *step_completed* message) from each neighbor. (Note that the message (that v is waiting for) may be sent by the neighbor either autonomously or in response to the message v had sent earlier.) If node v has assembled several *explore* messages (because it is the source of several disrupted paths), then all of those messages are bundled and sent as a single (long) message.

Actions of node v at other times

If it is the not the beginning of an iteration, then node v waits for a message from each neighbor and processes these messages. (*step_completed* messages are discarded after receiving them.)

Let $\{m_1, m_2, \ldots, m_x\}$ be the *explore* messages received by v. First node v "remembers" on which link it received the *explore* messages. Consider message m_j $(1 \leq j \leq x)$ received by node v sent by node u on the link $u \rightarrow v$. Let m_j = *explore*$(S_{j_1}, D_{j_1}, c_{j_1}, h_{j_1})$. It first decrements h_{j_1}. Node u locally stores (in its main memory) the entry {sender=u, source=S_{j_1}, destination=D_{j_1}, traffic to be restored = c_{j_1}, and decremented hop count = h_{j_1}}. This entry signifies that v has received an *explore* message from u with parameters $S_{j_1}, D_{j_1}, c_{j_1}$, and h_{j_1} (h_{j_1} is the decremented hop count). If v has already received an *explore* message with the same source and destination ids in the same iteration (but at an earlier step), then m_j is not propagated further. If m_j and m_k have the same source and destination identities (for some $1 \leq k \leq x$), then only one of them will be forwarded. Also, if the destination D_{j_1} is equal to v then the *explore* message is not forwarded.

To summarize, node v processes message m_j as follows: (Note that h_{j_1} is the decremented hop count in the following description.)

- If $h_{j_1} > 0$, $v \neq D_{j_1}$ and v has not received an *explore* message with the same source and destination id earlier in the same iteration, then v forwards the explore message after decrementing hopcount field and storing the *explore* message locally.
- If m_j and m_k have the same source and destination identities, then at most one of them will be forwarded.
- If $v = D_{j_1}$, then the *explore* message is not forwarded and v assembles a *return* message. If (decremented) hop count $h_{j_1} = 0$ then a *return* message is generated and sent back. If $h_{j_1} > 0$, then we do not start the return phase immediately. Instead, we wait for $2 \times h_{j_1}$ "steps." (Waiting is necessary to make sure that if some other path is contending for a link that will be traversed by this *explore* message, then the contention must be known before assigning spares to the path $S_{j_1} \rightarrow D_{j_1}$.) This is equivalent to the *explore* message traversing an imaginary path of length h_{j_1} links and returning back. Let c be the value of the local counter of the synchronizer. Node v records the information {pending,*explore*$(S_{j_1}, D_{j_1} = v, c_{j_1}, h_{j_1}), c + 2 \times h_{j_1}, u\}$.

D_i	h_i	Action
$\neq v$	$= 0$	Ignore m_i
$\neq v$	> 0	Store m_i locally and propagate m_i if necessary
$= v$	$= 0$	Assemble a *return* message and send it to u.
$= v$	> 0	wait for $2 \times h_i$ "steps"; assemble a *return* message and sent it to u.

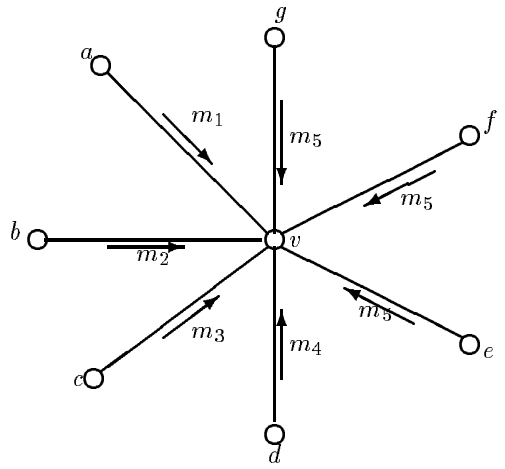
Fig. 2. Summary of actions needed to process an *explore* message m_i

Forwarding an explore message

Node v, in response to the *explore*$(S_{j_1}, D_{j_1}, c_{j_1}, h_{j_1})$ message, checks if $h_{j_1} > 0$ and this *explore* messages is to be forwarded. If so, it sends an *explore*$(S_{j_1}, D_{j_1}, c_{j_1}, h_{j_1})$ message to all neighbors from which it has not received an *explore* message with the same source and destination id. All of the messages are processed in this manner.

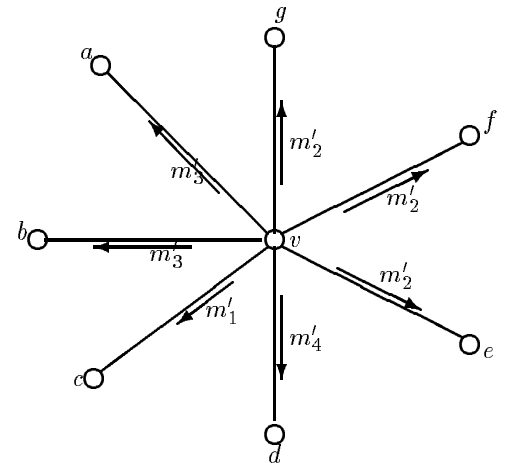
If v does not send an *explore* message on a link, it sends a *step_completed* message on that link. The actions are summarized in Figure 2.

a) An Example:: Consider an arbitrary node v of a network as shown in Figure 3. Assume that it is the second iteration now and one step has elapsed. Since this is the second iteration, we are trying to find paths of hop count 3 or less. Note that all paths of hop count 3 or less include paths of hop count 2. Those paths are included again since some spare capacities that were allocated to the other paths may have been released if those spares were not used by the other paths. After one step, the hop count field of the *explore* message is 2. Let nodes a, b, c, d, e, f and g be the 7 neighbors of node v. Assume that nodes e, f and g are not neighbors of any of the sources and the other nodes (a, b, c and d) are neighbors of one of the sources. Let m_1 = *explore*$(w, x, 5, 2)$ be the message sent by nodes a and b to v. Assume that c sends *explore*$(p, q, 2, 2)$ to v and d sends the message {*explore*$(w, x, 5, 2)$, *explore*$(p, q, 2, 2)$} to v. The other nodes e, f and g send a *step_completed* message to v. Node v stores {*explore*$(w, x, 5, 2)$, *explore*$(p, q, 2, 2)$} locally. Node v decrements the hop count of all of the *explore* messages. The *explore* messages sent by a and b (m_1 and m_2) will be sent to all neighbors except a, b and d. (Recall that d had sent m_4 to v and $m_4 = \{m_1, m_3\}$.) The other *explore* messages are propagated similarly and the results are are shown in Figure 4.



$m_1 = m_2 =$ *explore*$(w, x, 5, 2)$
$m_3 =$ *explore*$(p, q, 2, 2)$
$m_4 = \{$*explore*$(w, x, 5, 2)$, *explore*$(p, q, 2, 2)\}$
$m_5 =$ *step_completed*

Fig. 3. Node v at step 5 (iteration 2)



$m'_1 =$ *explore*$(w, x, 5, 1)$
$m'_2 = \{$*explore*$(w, x, 5, 1)$, *explore*$(p, q, 2, 1)\}$
$m'_3 =$ *explore*$(p, q, 2, 1)$
$m'_4 =$ *step_completed*

Fig. 4. Node v at step 6 (iteration 2)

3) Return Phase: The return phase starts after the explore phase ends. In the return phase, the paths traversed by the *explore* messages are retraced by the *return* messages[2]. The return phase ends when the *return* messages reach the sources. In the return phase, the contention for spare capacity is resolved.

a) Origination of the return *messages::* When node v receives an *explore*(S_i, D_i, c_i, h_i) message from node u, it first decrements h_i and then checks if $D_i = v$. If $D_i = v$ (the *explore* message has reached the destination), then it is not forwarded, but a *return* message is generated.

[2]using the information stored by the nodes when they received the *explore* messages

Case (i): $D_i = v$ and $h_i = 0$:

The *explore* message has reached the destination node from the source S_i. Thus, node v immediately generates a *return* message and sends it to node u. The *return* message contains the source id, the destination id, amount of traffic to be restored, and the subnetwork field. Initially, the subnetwork field is empty.

Case (ii): $D_i = v$ and $h_i > 0$.

Node v records the information {pending, *explore*$(S_i, D_i, c_i, h_i), 2 \times h_i + c, u\}$ locally.

Let count be the value of the local counter of node v. Node v checks if it has an entry {pending, *explore*$(S_i, D_i, c_i, h_i), count, y\}$. If so, then a *return* message is generated and sent to node y. The *return* message sent by node v to node u is *return*(S_i, v, c_i, ϕ). This is similar to case (i) discussed above.

Processing *return* messages:

Let $m =$ *return*(S_i, D_i, c_i, L) be the message received by node v from node u on the link (u, v). L is the set of links and their spare capacities. L will be used in restoring traffic between S_i and D_i. Spare capacity is given to the links using the contention resolution steps described in § III-E.4.

Node v receives a *return* message from u only if v had sent an *explore* message to u earlier in the same iteration. Clearly, for node v to send an *explore* message, (a) it must have either received an *explore* message from one of its other neighbors earlier or (b) v is the source of the disrupted path $P_i = (S_i, D_i)$ for some P_i. If case (a) applies, then node id S_i found in the *return* message is not equal to v. In such a case, node v forwards the *return* message. For the *return*(S_i, D_i, c_i, L) message, node v checks its locally recorded information. It must have received an *explore* message with the same source and destination identities. Consider only those *explore* messages received and recorded by node v such that the source and destination ids are equal to the source and destination ids of the *return* message received. Among these *explore* messages, let $m' =$ *explore*(S_i, D_i, c_i, h_i) be the *explore* message, recorded locally at v, with the maximum hop count h_i. Assume that the sender of m' is w. This message (m') is the earliest *explore* message received by v with source S_i and destination D_i. Now, c', the spare capacity of the link (u, v) that can be assigned to the path $P_i = (S_i, D_i)$, is found by the contention resolution rule and assigned. After this step, node v sends the message *return*$(S_i, D_i, c_i, L \cup \{l = (u, v), c'\})$ to its neighbor w. (Recall that w is the node that sent message m' to v earlier.) For each neighbor $x \neq w$ from which v had received an *explore* message with the same source and destination ids, node v sends a *return*(S_i, D_i, c_i, ϕ) message to neighbor x. The subnetwork information is sent in one *return* message only and is not duplicated on every *return* message. If case (b) is applicable, then v is the source of the path $P_i = (S_i, D_i)$. (That is, $v = S_i$.) Now, node v locally saves the list of links and spares on those links that are part of the *return* message received. During the last "step" of the current iteration, the max flow algorithm is run by the source node.

4) Resolving Contention: Often, there will be contention by several disrupted paths for the available spare capacity of a link. Resolving contention is important. Contention resolution is performed during the return phase.

Consider an arbitrary node v and its neighbor u. Assume that v receives a *return* message on the link (u, v) sent by node u. Let sp be the amount of spares currently available in the link (u, v).

Assigning spare capacity of link (u, v) to path $P_i = (S_i, D_i)$: When a *return*(S_i, D_i, c_i, L) message is received by v on the link (u, v), node v determines how much of the spare capacity sp of link (u, v) is to be assigned for restoring P_i. Let $m_1, m_2, \ldots, m_a$ be the *explore* messages, which correspond to the paths $P_1, P_2, \ldots, P_a$ with unique source-destination ids, received by v during the current iteration. Clearly, for every pair m_j, m_k, $1 \leq j, k \leq a$ and $j \neq k$, m_j and m_k are *explore* messages with sources S_j and S_k and destinations D_j and D_k such that (i) $S_j \neq S_k$ or (ii) $D_j \neq D_k$ or (iii) $S_j \neq S_k$ and $D_j \neq D_k$. Now, spare capacity sp of link (u, v) is uniformly distributed among the contending paths in proportional to their demands.

5) Max Flow Algorithm: The source node stores the subnetwork in the form of a series of links along with their spares. Next we use a Max Flow Algorithm. Any of the algorithms from the literature for the sequential model (centralized) is sufficient.

If the total flow obtained from the max flow algorithm in the current and previous iterations is sufficient to restore the traffic in that disrupted path, then the source sends a *path_restored* to all of the nodes by a broadcast. In that case, the source does not participate in the exploration phase of any higher hop counts for that path id. If the traffic needs of the disrupted path are not completely met at the end of this iteration, the source proceeds to the next iteration (with next higher hop count).

6) Terminating the Algorithm: When one path is fully restored by the source node, the source node broadcasts a *path_restored* message. This message is sent to all the nodes (including itself). When a node receives this *path_restored* message for the first time, it decrements its variable NUM_PATHS. (By recording the *path_restored* messages, a node can identify if a *path_restored* message has already been received in a manner similar to *failed* messages.) When NUM_PATHS reaches the value 0, the node terminates the restoration algorithm. Also, note that when the number iterations is equal to $O(diam)$ where $diam$ is the diameter of the network, each source would have explored the complete topology and all nodes terminate the algorithm.

F. Algorithm Novelties

This algorithm has some novelties.

1) We do not choose paths that may decrease the amount of traffic that can be restored. This is achieved by keeping the nodes in "rhythm." It may appear that keeping the nodes in "rhythm" slows the algorithm. For example, as per § III-B, a node waits till it receives a message from each neighbor. Thus, if one neighbor is slow to start the algorithm or one of the links is slow, then we may be slowing down the node. However, keeping nodes in "rhythm" ensures that nodes carefully select paths for restoration. For example, consider the sample network shown in Figure 5.

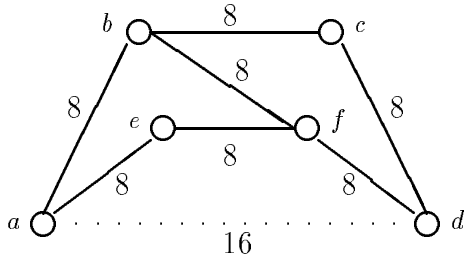


Fig. 5. A Sample Subnetwork

Assume that link (a, d), carrying 16 units of a single path $P = (a, d)$ of traffic, fails. If the restoration path chosen is (a, b, f, d), then at most 8 units of traffic can be restored. This situation may occur during the execution of many of the published algorithms. In our algorithm, during the second iteration, all paths of hop count 3 between a and d are found. For this example, all the three paths, path (a, b, c, d), (a, e, f, d), and (a, b, f, d) are examined by the node a at the same time. By using the max flow algorithm, we will choose the paths (a, b, c, d) and (a, e, f, d) instead of the path (a, b, f, d).

2) Contention resolution is handled in a uniform way in our algorithm. In many of the published algorithms, the first explore message traversing a link may preemptively use all of the spare capacities leaving nothing for the explore messages of the other paths. At a later time, if the first explore message does not use all of the spare capacities obtained, it will release them so that the released spares may be used by others for restoring some other path. However, this process of reserving and then releasing later on consumes time. Also, this process may happen in a cascaded manner. Our algorithm avoids this problem to a large extent by resolving contention in a uniform way.

IV. PREPROCESSING STEPS

The algorithm can be improved by some one-time preprocessing. We suggest the following preprocessing:

Each node locally keeps a table (one dimension) showing the minimum number of hops needed to visit each node of the network starting from itself. Consider node with id, say u. It has an array, called minhops. minhops[i] is the number of hops (number of links) of the shortest path from itself (node u) to the node with id i. This array is maintained within node u. Clearly, this array can be constructed at each node before the restoration algorithm starts. For example, the algorithm of [20] can be used. This is done before failures.

This array is used in deciding when to propagate *explore* messages. Let the node with id u receive an *explore* message with source S and destination D. Node u, on receiving this *explore* message checks if minhops[D] is greater than

the hop count allowed (this number is part of the *explore* message). If minhops[D] is less than or equal to the hop count allowed, then the *explore* message is propagated on other links (provided node u has not already propagated an *explore* message with the same source and destination ids in the current iteration). On the other hand, if minhops[D] is greater than the hop count allowed, then it is clear that the *explore* message cannot reach the destination within the hop count allowed. In such a case, the *explore* message is not propagated but is discarded.

Although the minhops array of a node is determined before the link failure, the link failure does not reduce the minimum number of hops needed between any two nodes; instead, the link failure may increase the minimum number of hops. Thus, when a node decided to drop an *explore* message, the node will not be making a mistake. Some of the *explore* messages may be sent unnecessarily since the minimum number of hops may have increased because of the link failure, but this does not affect the correctness of the algorithm.

An alternate method is to compute the minimum number of hops from each node to all of the other nodes at run time (immediately after the link failure). Note that in the first iteration, knowledge about 2-hop neighbors is needed (and knowledge about nodes that are more than 2 hops away is not needed in the first iteration). To gain this knowledge, each node sends its list of 1-hop neighbors on all links in the beginning. Subsequently, k-hop neighborhood information ($k \geq 2$) can be sent to all neighbors (by each node) by piggybacking this information on the *explore*/*return*/*step_completed* messages. Thus, there is a delay of one step, but the minhops array will have the accurate value.

V. Complexity Analysis

. Clearly, when the current iteration count is equal to $diam$, where $diam$ is the diameter of the network, an *explore* message sent by a source node reaches all the nodes by the end of the explore phase. Therefore, $O(diam)$ iterations are sufficient to explore the entire network. Iteration i consists of $2 * (i + 1)$ steps. During each phase, exactly one message is sent on each link (in each direction). The number of message sent per step is $O(|E|)$ where $|E|$ is the number of links in the network. The total number of messages sent is $O(diam^2|E|)$. Note that if the minhops array is computed on the fly (and not as a preprocessing step), one additional step is needed. The time complexity is $O(diam^2)$ since each step takes one time unit.

VI. Simulations

The performance of the algorithm was tested through simulation. Simulations were done on the network, specified in Bellcore advisory [18], which represents a reasonable metropolitan size network of 15 nodes and 28 links. The advisory also reports the simulated performance of three on-line restoration algorithms and we compare the performance of our algorithm with the performance of three algorithms reported in [18]. The working and spare capacities were designed in [18] to guarantee 100% restoration for any single link failure. The performance figures are obtained by simulating the failure of each of the 28 links, one failure at a time.

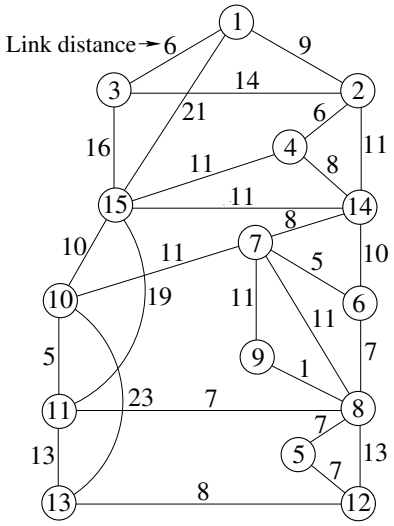


Fig. 6. The test network

The percentage of traffic restored (PTR) relative to the traffic disrupted for each link failure event is used as a performance metric. Another performance metric measured is the restoration time (RT) which is the maximum time required to restore allf the disrupted paths completely from the time the algorithm is initiated.

A. Simulation Parameters

The simulation methodology accurately measures the local time at each node and hence it permits the calculation of the total running time of the algorithm. To simulate the real performance of the network, we need to add time delays at various points. To compute these delays, the following parameters are used:

1) The line speed constant is set to half the speed of light. This is used to calculate the signal propagation delay.
2) Data Transmission rate for the links is set to 64 Kbits/sec. This is used to compute the delay for physical transmission of a message between nodes.
3) Data transfer rate between the port and CPU is 128 Kbits/sec. The internal port delay, which is the time taken by the a packet to go from the input port to the CPU (or from the CPU to the output port), can be computed using packet size and port-CPU data transfer rate.
4) The CPU processing rate is used to estimate how long the CPU at each node takes to execute the instructions of the algorithm. The CPU is assumed to have a performance of 1 Million Instructions Per Second.

The test network of [18] is shown in Figure 6.

B. Results

Figure 7 shows four plots of PTR versus RT on the test network. Plot 1 represents the simulation results of our

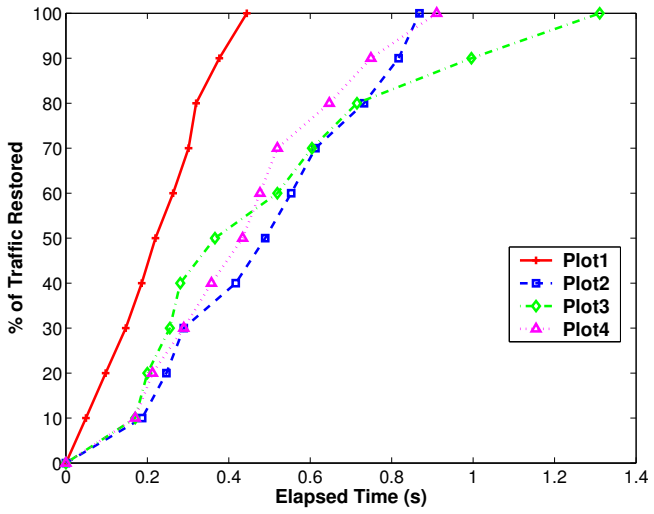


Fig. 7. Relative performance of the algorithm

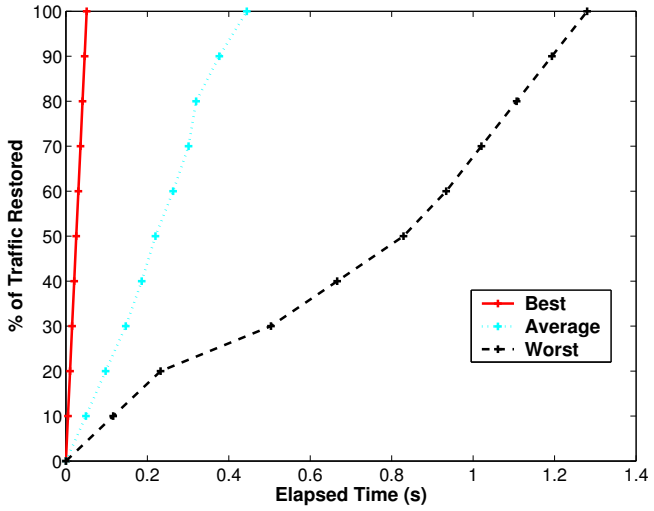


Fig. 8. The best, average and worst case restoration time

algorithm. The result shown is the average value assuming that each link failure is equally likely. Plots 2, 3 and 4 are from [18] which represent the simulation results of the distributed restoration algorithms of [6], [21], [22]. The input parameters used in our simulations are identical to the input parameters used by Bellcore in their simulations [18].

The best, average and worst case time taken for restoration by our algorithm is shown in the Fig. 8.

Based on simulation, our restoration algorithm is fast and achieves 100% restoration. Fig. 7 shows that improvement in the restoration time is significant.

Although only single link failures were simulated and results are shown in the paper, multiple link failures and node failures can also be handled with minimum amount of changes. For double link failures, for example, two failed messages will be broadcast. Node failures can be treated as the failure of all of the links incident on that node. For node failures, the paths for which either the source or the destination is the faulty node will not be restored. The other paths will be restored. No other changes are needed.

VII. Comparison with Related Work

Our work is most closely related to the work of [6], [7], [21]. A main characteristic of all of these algorithms is the following: Each affected source node u, on learning about the disruption of the path for which u is the source node, constructs a packet containing the source id, destination id, amount of traffic to be restored, and a hop count, and floods the network with this packet. When an arbitrary node receives such a packet, it sets the amount traffic to be restored (on the packet) to the smaller of the value of the spare capacity of the link on which this message was received and the amount of flow to be restored (available in the packet itself) assuming that all the spare capacity is available for restoring this path. The recipient decrements the hop count and adds its own id to the packet (so that the packet contains the path traversed so far). When a packet reaches the intended destination, a path from the source to the corresponding destination has been found. Details vary from one algorithm to the other in terms of when the spare capacities are allocated to a path; some algorithms allocate in the forward phase (source to destination) and other allocate in the return phase. The methods create situations where one or more paths reserve (allocate) needed spare capacities too soon, only to find that these paths do not need them or cannot use them (because a path is unable to reserve spares on all the links of the path; only on some) because reserving is done in an uncoordinated way. This can lead to a cascaded allocation and deallocation scenarios.

Awerbuch and Leighton [23] present approximation algorithms for the multicommodity flow problem. Their approach is based on assigning "potentials" to the nodes and balance the potentials using links. For example, at the source add a potential equal to the amount of flow that is disrupted. This potential is divided uniformly among all the incident links and moved to the 1-hop neighbors of the source, if possible, subject to a local optimization rule and a rule to ensure that high potential nodes do not overtake link capacities too soon. The number of steps (rounds) needed is $O(|E|^2 K^{\frac{5}{2}} L \epsilon^{-3} log K)$ where K is the total number of commodities, L is the length of the longest path and $0 < \epsilon$ is a parameter chosen. The advantage enjoyed by their algorithm is that if there exists a feasible solution in the current network when the demand is increased by a factor of $(1+\epsilon)$, their algorithm finds a solution for the needed demand. (The network is "over designed" by a factor of $(1+\epsilon)$.) The algorithm of Kamath et al [24] is an improvement over the algorithm of [23]. The algorithm of [24] runs in $O(\frac{|E|^2 logn}{\epsilon^2})$ steps (rounds) and produces flows that are $(1-\epsilon)$ fraction of each demand, provided feasible flows exists with the current spare capacities for the required flows. In comparison, our algorithm uses $O(diam^2)$ steps (rounds), but may need more spares.

VIII. Guidelines for Future Work

The on line restoration can find alternate paths well below the 2 seconds time limit. However, significant amount of cross connections will have to be made at all of the nodes that are in the restored paths. At present, as the Bellcore study [18] indicates, the cross connect times are very significant resulting in total restoration times exceeding the 2 second time frame. One of the problems for future study is to find the earliest times, during the time that the restoration algorithm is running, when some part of the cross connections can be performed. This will ensure that nodes do not wait for the restoration algorithm to terminate before starting cross connection. Also, new architectures that use parallel processing are needed to speed up the cross connect operations at all of the nodes. With such changes and enhancements, it is hoped that restoration can be achieved in under 2 seconds.

Alternate approaches for restoration may be attempted for speeding up the path finding phase. More preprocessing (without preplans) may be done for quick restoration.

Acknowledgement

The problem formulation was done in cooperation with researchers at Alcatel Network Systems and parts of the simulation study was done with their support. Their help is gratefully acknowledged.

References

[1] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows*. New Jersey: Prentice Hall, 1993.

[2] I. Ali, D. Barnett, K. Farhangian, J.Kennington, B. Patty, B. Shetty, B. McCarl, and P.Wong, "Multicommodity network problems: Applications and computations," *IIE Transactions*, vol. 16, no. 2, pp. 127–134, 1984.

[3] A. Assad, "Multicommodity network flows: A survey," *Networks*, vol. 8, pp. 37–91, 1978.

[4] J. Kennington, "A survey of linear cost multicommodity network flows," *Operations Research*, vol. 26, pp. 209–236, 1978.

[5] E. C. Chow, J. Bicknell, S. McCaughey, and S. Syed, "A fast distributed network restoration algorithm," in *Proc. of Computers and Communications*, Tempe, AZ, Mar. 1993, pp. 261–267.

[6] H. Komine, T. Chujo, T. Ogura, K. Miyazaki, and T. Soejima, "A distributed restoration algorithm for multiple-link and node failures of transport networks," in *Proc. of IEEE GLOBECOM*, vol. 1, San Diego, CA, Dec. 1990, pp. 459–463.

[7] R. R. Iraschko, W. Grover, and M. MacGregor, "A distributed real time path restoration protocol with performance close to centralized multicommodity max flow," in *Proc. of 1st International Conference on Design of Reliable Communication Networks*, Brugge, Belgium, May 1998, p. paper O9.

[8] B. T. Doshi, S. Dravida, P. Harshavardhana, O. Hauser, and Y. Wang, "Optical network design and restoration," *Bell Labs Technical Journal*, vol. 4, no. 1, pp. 58–84, Jan./Mar. 1999.

[9] J. Anderson, B. Doshi, S. Dravida, and P. Harshavardhana, "Fast restoration of ATM networks," *IEEE Journal on Selected Areas in Communications*, vol. 12, no. 1, pp. 128–138, Jan. 1994.

[10] R. Kawamura, K.-I. Sato, and I. Tokizawa, "Self-healing atm networks based on virtual path concept," *IEEE Journal on Selected Areas in Communications*, vol. 12, no. 1, pp. 120–127, Jan. 1994.

[11] Y. Xiong and L. G. Mason, "Restoration strategies and spare capacity requirements in self-healing ATM networks," *IEEE/ACM Transactions on Networking*, vol. 7, no. 1, pp. 98–110, Feb. 1999.

[12] P. Demeester, M. Gryseels, A. Autenrieth, C. Brianza, L. Castagna, G. Signorelli, R. Clemente, M. Ravera, A. Jajszczyk, D. Janukowicz, K. V. Doorselaere, and Y. Harada, "Resilience in multilayer networks," *IEEE Communications Magazine*, vol. 37, no. 8, pp. 70–76, Aug. 1999.

[13] W. Lai and D. McDysan, "Network hierarchy and multilayer survivability," RFC 3386, Nov. 2002.

[14] W. D. Grover, R. R.Iraschko, and Y. Zheng, *Comparative methods and issues in design of mesh-restorable STM and ATM networks*, P. Soriano and B.Sanso, Eds. Kluwer Academic Publishers, 1999, ch. 10 in Telecommunication Network Planning, pp. 169–200.

[15] M. Patel, R. Chandrasekaran, and S. Venkatesan, "A comparative study of restoration schemes and spare capacity assignments in mesh networks," in *Proc. of IEEE 12th International Conference on Computer Communications and Networks*, Dallas, 2003, pp. 399–404.

[16] S. Even, A. Itai, and A. Shamir, "On the complexity of timetable and multi-commodity flow problems," *SIAM Journal of Computing*, vol. 5, no. 4, pp. 691–703, 1976.

[17] S. S. Lumetta, M. Médard, and Y.-C. Tseng, "Capacity versus robustness: A tradeoff for link restoration in mesh networks," *Journal of Lightwave Technology*, vol. 18, no. 12, pp. 1765–1775, 2000.

[18] "Digital cross-connect systems in transport network survivability," Bellcore," Special Report SR-NWT-002514, Issue 1, January 1993.

[19] B. Awerbuch, "Complexity of network synchronization," *Journal of the ACM*, vol. 32, no. 4, pp. 804 – 823, Oct. 1985.

[20] K. Ramarao and S. Venkatesan, "On finding and updating shortest paths distributively," *Journal of Algorithms*, vol. 13, no. 2, pp. 235–257, 1992.

[21] H. Sakauchi, Y. Nishimura, and S. Hasegawa, "A self-healing network with an economical spare-channel assignment," in *in Proceedings of IEEE GLOBECOM*, Dec. 1990, pp. 438–443.

[22] K. Miyazaki, T. Chujo, H. Komine, and T. Ogura, "Spare capacity assignment for multiple-link failures," in *Proc. of International Workshop on Advanced Communications and Applications for High Speed Networks*, Mar. 1992, pp. 191–197.

[23] B. Awerbuch and T. Leighton, "Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks," in *Proc. of ACM Symposium on the Theory of Computing*, Montreal, Quebec, Canada, May 1994, pp. 487–496.

[24] A. Kamath, O. Palmon, and S. Plotkin, "Simple and fast distributed multicommodity flow algorithm," unpublished Manuscript.