Abstract
Real time applications place very stringent demands on current networks. A breakdown in a routing path in a network is unacceptable for these applications. It is desired that the effect of the failure caused by a link or host failure in the path be reduced by restoring it locally. Local restorability means that on failure the first node upstream from the failure is able to forward the traffic on another path to the destination. This can be done if protection is provided in the form of a bypass backup path so that in case of a failure traffic in transit can be switched on that path rather than informing the source and than following a new path from the source to the destination. This paper discusses an algorithm “Routing Traffic Reliably”. It provides one to one and facility backup for fast reroute which are discussed later in the paper. RTR follows the greedy approach by computing the shortest path first and than finding backups for that path. It is expected to perform reliably as validated by simulation.

What is fast Reroute?
Simply put ‘reroute’ means that in case of any problem with our routing path we are able to route traffic on another path. It means that in case of failure data can be routed on another redundant path especially computed for this purpose. The advantage of fast reroute is that data in transit can be routed on another path rather than being lost due to the original path failure. The term Fast Reroute is becoming much popular with regard of MPLS (Multi protocol Label switching) because of the QOS guarantees provided by MPLS. We now give a little insight into MPLS.

MPLS
MPLS is a technique that was basically employed to reduce path lookup time for internet traffic. It originated from tag switching as the base of MPLS is a label (tag) attached to every traffic packet. This label is than used for routing. Normally this assignment happens when a packet enters an MPLS cloud (An MPLS enabled network). The Ingress router to this cloud assigns each packet to an FEC (Forward Equivalence Class). Several factors may influence the assignment of a packet to a FEC. The assignment of a packet to an FEC by the MPLS ingress router has the benefit that once this assignment is done it remains consistent throughout the current MPLS cloud. However in case of IP switching each router again itself assigns a packet to a particular FEC which has clearly more overhead as compared to MPLS. Now once a packet is assigned to an FEC a label is attached to the packet. That label is than used in the MPLS cloud for routing purposes. This reduces the lookup time considerably as the label lookup table is really small as compared to routing tables maintained at the router. With the advent of modern table lookup algorithms the use of MPLS was not limited to lookups only and it is mostly employed for ensuring QOS (Quality of Service) for a particular traffic flow.

LSP (Label Switch Path)
A LSP is a sequence of hops in which a packet travels on the basis of its label. A path (R1…Rn) is a LSP in which a packet travels from R1 to Rn through label switching [1]

Fast Reroute and MPLS
As mentioned MPLS can provide guarantees to QOS. One problem that is often faced in networks is the breakdown of the active path. An active path is the path on which the traffic is routed. In case of the failure of the active path there should exist another backup path that could be used for supporting traffic of the active path. So Fast Reroute for MPLS employs that traffic is switched sufficiently fast on a backup path in case of a failure. For fast reroute however the definition of a backup path is more subtle. Basically we require a backup bypass path at every
node so that in case of its upstream neighbor it can use that path to bypass the failed link or node.

Consider the topology in the figure. Suppose the traffic reaches node B and then the link from B to C breaks down. In this case there should be another path leading from B to C otherwise we won’t be able to do a fast reroute. We may have another backup path from A to D but in that case out traffic currently at B would get lost. So our problem is that to do a fast reroute we need certain guarantees much more than just another backup path. In the above case if there exists another path from B to C than B is called a protected LSP (Label Switched Path) where an LSP is an MPLS label switched path.

**Fast Reroute Schemes**

Currently two type of fast reroute schemes are used

- One to One Backup
- Facility Backup

**One to One Backup:** In One to one backup technique an LSP is established which intersect the original LSP somewhere downstream to the point of link or node failure [2].

In the figure A is protected by A->C->B. Note that the Backup LSP intersects the original LSP at B

The Path A->C->B in the above figure is termed as the detour. In case of failure of link A->B the traffic is routed on the detour link A->C. In one to one backup a detour is created for every protected LSP

**Facility Backup:** In facility backup instead of creating a separate path for every LSP a single LSP is created which serves as a backup for a set of LSP’s.
In the figure C has established a tunnel E→F called a bypass Tunnel and A use it for protection from failure of B. Notice how it also serves B in case B→D fails.

Hence in case we want to fully protect an LSP that has n nodes than it would require n-1 such tunnels.

**Special Case of Disjoint Path Finding**

Our problem is hence a special case of finding disjoint paths in a network since a backup for the active path would not include the link or node which is being protected. Disjoint paths are classified as:

- **Link Disjoint**: Two paths having no common Link
- **Node Disjoint**: Two paths having no common node. A node disjoint path would also serve as a link disjoint path

The problem of finding disjoint paths in the network becomes more complicated when the disjoint path have to be computed on the basis of certain constraints. These constraints can be classified into two types:

- **Additive**: An additive constraint is calculated on the basis of the whole path; like the total bandwidth available on the path which is calculated by summing the bandwidth available on all the links in the path
- **Boolean**: A Boolean constraint is defined on a per link basis on the path; like a link in the path should at least have 10mbps

Furthermore these constraints are defined in term of links constraints and path constraints. A constraint on the whole path can be the max number of hops allowed in the path while a constraint on a particular link may be the bandwidth available on the link

Classifications of these constraints are:

**Link Constraints**: The links constraints are defined as:

**Interface switching capability**: In the realm of GMPLS[A], optical links are considered to be Logical interfaces each interface interconnected by two network elements or nodes. Each Interface may have different switching capabilities, such as packet switch capable or TDM switch capable. This constraint requires that all the links on the path have the same Interface switching capability. Each path must initiate and terminate at the same level of
GMPLS LSP hierarchy [3].

**Bandwidth:** The bandwidth available on each link.

**Link Protection type:** It defines the protection capability available for the link. The Protection types include: Extra Traffic, Unprotected, Shared, Dedicated 1:1, Dedicated 1+1 and Enhanced.

**Traffic engineering (TE) metric:** This constraint is related to the cost of the link if it is used for a path. The cost of the path is the sum of TE metrics on all the links along the path. Strictly speaking, this is not a constraint but an object function that CSPF aims to minimize.

**Shared Risk Link Group (SRLG):** A SRLG group contains such links all of which are affected if a particular resource fails. An example of this can be if a particular node has more than a single links than all of its outgoing or incoming links are an SRLG with respect to that node. This also means that a link can be a part of more than one SRLG.

**Link resource class:** We can associate links with some resource class attributes by which the CSPF can either choose to include or exclude them from a path. This can be useful to achieve policy enforcement.

**Path Level Constraints:** These include:

**Hop Count Limit:** It defines an upper bound on the total number of hops in the path.

**Path Priority:** Paths can be prioritized on the basis of various attributes and then their priority can be used to provide service to different traffic flows by allocation different paths to different service classes.

**Path Restoration:** Paths can be classified on the basis of their restoration. A completely disjoint path can be restored completely by making the traffic flow follow the backup path.

Hence keeping up with all of these constraints and trying to find a shortest path with respect to delay, number of hops etc. forms a problem in the CSPF (constraint based shortest path first). CSPF is a very hard problem due to the introduction of multiple constraints. With only one additive constraint it is known to be NP-complete [3].

**Theorem 1.** NP complete proof for CSPF with multiple constraints [4]

\[
\begin{align*}
S & \\
0 & \\
\text{i} & \text{i+1} \\
S-a_i & a_i \\
\end{align*}
\]

Figure 4: The assignment of link weights to the links in chain topology between nodes i and i+1.

**Proof:** Given a chain topology with n+1 node and 2n links. Each link has a two component weight vector as shown in figure and a set of numbers \(a_i \in A\), \(0 \leq a_i \leq S\), for \(i=1, \ldots, n\), where \(S=\sum a_i\) where \(i=1\) to \(n\). The Constraints are chosen as follows: \(L_1 = nS-(S/2)\) and \(L_2 = (S/2)\). To solve MCLPP we require two paths P and Q from node 1 to node n+1 such that both of them fulfill the constraints. For all link weight vectors the sum of the components equals S than \(w_1 (P)\) and \(w_2 (P) = nS\). A solution fulfilling the constraints is only found if \(w_1 (P \text{ and Q}) = nS-(S/2)\) and \(w_2 (P \text{ and Q}) = (S/2)\). The problem now becomes an instance of the well known NP-Complete partition problem [5] and can now only be solved by finding the set \(A'\) belonging to \(A\) for which \(\sum a_i = (S/2)\). A feasible path P exists if the set \(A'\) exists. That P consists of the lower link if \(a_i \in A'\) and the upper link if \(a_i \notin A'\). The path Q then follows the remaining links.
System Model for RTR

The system model in which RTR is to function can be modeled as a graph $G(V, E)$ where $V$ is a set of vertices and $E$ a set of edges. Note that the edges are bidirectional. The weights are assigned to the edges. The cost function for assigning weights to a function can be defined as $W(V_n) = P_1C_1+...+P_nC_n$ where $V_n$ belongs to $V$, $C_1...C_n$ belongs to the constraints set which contain the constraints imposed and $P_1...P_n$ is the priorities assigned to each constraint. The priority of each constraint can be tweaked according to its importance by the traffic engineer.

The Algorithm

There are three phases of RTR.

i. Find a shortest path from source to destination. The nodes in that path are known as the key nodes
ii. Find the distance of non key nodes from the key nodes
iii. Find the backup for the key nodes

A. Finding the Shortest Path

The shortest path is found by using a Dijkstra like algorithm that finds the shortest distance of each node from the source

i. Assign node $s$ (source) $tcost$ 0. $tcost$ is the distance of each node from the source. Assign each other node inf and set there status as ‘red’
ii. Set the status of source ‘green’ and the status of all the other nodes ‘red’. A node with the status ‘green’ is used for putting the next link and node in the path
iii. For each node with status ‘green’ pick its shortest link that goes to a node whose status is ‘red’. Sum that link’s cost with the tcost of the node and put that in tcost
iv. Pick the node with the smallest tcost and choose the node reached. The node thus reached is changed by setting its status to ‘green’ and set its tcost to tcost of the node whose link is choosen
v. The algorithm converges when all the nodes have their status set to ‘green’
vi. Start from the destination node back to the source node and pick nodes by the above strategy i.e. the node whose sum of tcost and the link cost that leads to it are the minimum. Set the status of that key node to ‘yellow’

Hence this gives us the shortest path among the source and the destination. The worst case complexity for this phase is $O(n^2)$ with $n$ nodes and $m$ vertices as it allows for directed cycles. For sparse graphs however its complexity would improve

B. Updating non-key nodes

The next phase of the algorithm is to calculate the distance of the non key nodes from the key nodes. The main point that reduces the complexity here is that only the distance from the key nodes is calculated for a non-key node and not its distance from every other node.

i. Pick a key node and set the distance of all the non-key nodes(NKN’s) from it to inf and set their status to ‘red’
ii. The NKN neighbors of that key node calculate their distance from it and store it in tcost and set their status to ‘green’. Store the node that sent the
distance from that node in gnode (an array that stores the node used for reaching a particular key node)

iii. For each green node send its neighbor the sum of its tecost entry for the particular key node and its distance from that neighbor if and only if that neighbor is not listed in the gnode entry for that key node

iv. Set the neighbor’s status to ‘green’ and if and only if its tecost entry for that particular key node is empty or is greater than the entry send to it

i. Set its keynodes (array that holds the key node) to the key node and keycost (array that holds cost to reach it) to the tecost entry received and gnode (array to hold node to follow to reach keynode) to the node that send it the entry for that particular key node. Repeat step 3 on this node

Otherwise

ii. Do nothing

v. Repeat the above procedure for each key node

In this way a NKN will know its distance from each key node. As for the convergence of this phase the neighbors of a green node can be put in a queue and then processed. The algorithm converges when the queue becomes empty. If a NKN has its tecost set to infinity with respect to a key node than it cannot reach that key node. Also note that a green node can be visited again if it can reach the destination (a particular key node) from another node at a lower cost. This operation is like in the Bellman-Ford algorithm and the running time of Bellman-Ford is $O(VE)$

C. Finding Backups

The next comes the phase in which backup among key nodes have to be established.

i. Pick the two key nodes among which backups are desired

ii. Set ‘dnode’ to the key node closer to the source node in the original shortest path

iii. Start from the other key node

iv. Pick its neighbor whose key node equal to dnode has the shortest tecost entry

v. From than on pick the node given in gnode for the key node given in dnode

vi. Algorithm converges when node given in dnode is reached

Since in this case a node will only be visited once so this phase would have $O(n)$ running time where $n$ is the number of nodes

Simulation

RTR employs the greedy strategy which does not always yield optimal results [6]. However the algorithm was simulated to check whether it converges by using a simulator specially coded for RTR named RTR-Sim. The simulator was coded in C#. It uses a structure as the basic data structure to store information regarding a particular node

```csharp
public struct vertex
{
    public String [] neighbors; // Keep track of neighbors
    public int [] cost; // Cost to reach each neighbor
    public String label; // label of the node
    public int tcost; //
    public int tneigh; // Total number of neighbors
}
public int tkeyn;                    // Total key nodes accessible
public String status;              // status of node
public int tecost;                     //temporary used in calculations
public String[] keynodes;       //holds key node labels
public int[] keycost;                //holds keynode distance
public String[] gnodes;         // node to follow to reach key node
}

The simulator produces result on the basis of each phase. It displays optimal path first. Then it go through phase 2 and than finds backup paths.

Conclusion

This paper discusses RTR an algorithm designed to accomplish fast rerouting. It finds bypass backup paths employing a greedy strategy so at times it may not produce the most optimal solution. However, simulations on RTR using RTR –Sim show that the algorithm does converge. However for future work we may want to accomplish fast reroutes by exploring this problem using a heuristic approach to algorithm design
Appendix:

[A] GMPLS: Simply put GMPLS is an enhancement of MPLS to support optical core networks. Optical core networks incorporates factors like wavelength, TDM and fiber switching into traffic engineering and MPLS cannot cater for them. GMPLS provides the necessary architectural enhancements MPLS to support them.
Acknowledgments

I would like to specially thank Dr Sarmad Abassi for listening to my whimsical ideas. Also I would like to thank Dr Zartash Afzal Uzmi for being understanding at times. Last but not least Saqib Raza for pushing me through the project at times when I was completely stuck.

References:


[2] Fast Reroute Extensions to RSVP-TE for LSP Tunnels   Internet Draft

[3] Mahi Networks Implementing a Constraint-based Shortest Path First Algorithm in Intelligent Optical Networks

[4] Link-Disjoint Paths for Reliable QOS Routing Yuchun Guo, Fernando Kuipers and Piet Van Meighem
