Internet Engineering Task Force
INTERNET-DRAFT
draft-villamizar-mpls-omp-00
Curtis Villamizar
ANS
November 18, 1998

MPLS Optimized Multipath (MPLS–OMP)

Status of this Memo

This draft specifies a precedure for routing MPLS label switch paths and balancing load across these paths. A portion of the algorithm has been verified in simulations, however a portion has yet to be simulated. Some details of the algorithms are likely to change as simulations progress. This draft contains notes pointing out sections which are likely to change.

Abstract

MPLS ingress routers may establish one or more explicit paths to a given egress to the MPLS domain. Load can be balanced across a complex topology using MPLS and the technique referred to here as MPLS–OMP (Multiprotocol Label Switching – Optimized Multipath). The technique requires the use of an interior gateway protocol (IGP) such as OSPF or ISIS with extensions to flood loading information. It requires the ingress router be capable of computing a hash based on the IP source and destination and selecting a forwarding entry based on the outcome of the hash with a sufficiently fine level of granularity, then load can be balanced across a complex topology.

MPLS Optimized Mulitpath can be considered an extension to MPLS or a specific usage of MPLS. MPLS-OMP requires no additions or modification to the MPLS protocols. MPLS-OMP does require that the IGP be capable of flooding loading information as described in the OSPF-OMP and ISIS-OMP documents. At the MPLS ingress an algorithm is applied to select alternate paths where needed and adjust forwarding. Forwarding is adjusted gradually enough to insure stability yet fast enough to track long term changes in loading. The load adjustment algorithm itself is fully defined in

a related document describing OSPF–OMP. The application of this technique to MPLS is described here.

1 Overview

Networks are often heavily loaded. Topologies often evolve to include multiple paths. Multiple paths may be initially designed to provide redundancy but also result from incremental addition of circuits to accommodate traffic growth. The redundant paths provide a potential to distribute traffic loading and reduce congestion. Optimized Mulitpath (OMP) provides a means to make better use of this potential to distribute loading. Please refer to [11] for a more complete introduction to OMP.

MPLS involves a forwarding technique. The MPLS framework and architecture are defined by [1, 2]. The generic forwarding encapsulation is defined by [6]. MPLS–OMP involves the setup of MPLS Label Switched Paths (LSPs) with strictly routed Explicit Routes (ERLSPs). There are two methods of setting up LSPs, the Label Distribution Protocol (LDP) [10], and use of RSVP extensions [5, 4]. Both methods of setting up LSPs support strict (and loose) routed ERLSPs.

In an underutilized topology, criteria such as minimum delay dominate routing decisions. Often IGP link costs (costs in OSPF, metrics in IS–IS) are set roughly according to link propagation delay. If the entire topology is not underutilized, some consideration is given to lower capacity links or more heavily utilized links to reduce their utilization somewhat.

When using MPLS–OMP, link costs may be set roughly according to link propagation delay without regard to link capacity or expected utilization. An initial set of LSP will be set up according to these costs as described in Section 4. There will initially one or more LSP between any ingress and any egress and these LSP will take the path of least cost. There will be exactly one LSP between any ingress and egress except where multiple paths exist with equal cost. If metrics are set with sufficient precision, for example taking delay in tenths of a millisecond and using this as cost (such precision is possible with OSPF, but may not be in IS–IS where link metrics are limited to values of 0-63), then few if any equal cost paths will exist.

MPLS uses link loading and loss information flooded by OSPF–OMP or ISIS–OMP as described in Section 2. Each link along a given path is examined to see if loading persistently exceeds a specific set of criteria. An alternate path will be created according to criteria which depends on the severity of the loading, the amount of time the condition has persisted, and the ingress relative contribution to the loading. This is described in Section 5.

The distribution of load among a set of alternate paths is determined by the amount of number space from a IP source and destination hash computation that is allocated to each path. For example using a 16 bit hash, if there are two paths, one may be given 80% of the 16 bit number space or the values 0-52428 and the other given 52429-65535 to accomplish a split of approximately 4:1. The initial load applied to a newly created alternate LSP is small. As loading on the alternate path persists in being lower than loading on other paths in use, more load is moved to the path with lower loading. The increment of change is gradually increased. At some point, what was among the less loaded paths becomes more heavily loaded than all other paths. At this point, this path is designated the most heavily loaded, the load increment toward the previous most heavily loaded path is at least halved, and load is moved away from the new most heavily loaded path. At this point load will begin to converge such that loading on the most heavily loaded link on each paths tends toward equality. See

Section 8 and Section 9 for details.

When more than one path between an ingress and an egress exists, after load balancing, there may still be an excessive load on the entire set of paths. The same criteria in terms of severity of loading, persistence of the condition, and ingress contribution used when examining a single path to see if an alternate is needed is used to examine a set of paths. For each path currently in use, the loading on the most heavily loaded link of the path is considered to be the loading of the path. The lowest path loading is considered to be the loading on the set of paths in considering whether to add an additional path. The algorithm described in Section 5 covers this case as well.

If loading is reduced or failed links are restored, then sets of paths may be found to have excess capacity. Paths which are less desirable in terms of cost, and so therefore also in terms of delay, can be deleted. This is described in Section 6.

The impact of link state changes on the path layouts is discussed in Section 7. An algorithms is described that insures that path selection returns to an optimal state after arbitrary link state changes have occurred and the topology has then become quiescent.

This overview provides a qualitative description of the MPLS–OMP algorithms. The details of these algorithms are provided in the sections that follow.

2 Use of ISIS–OMP and OSPF–OMP Flooding

Both OSPF and IS–IS are link state protocols commonly used as interior gateway protocols (IGPs) [9, 7, 3]. MPLS relies on an IGP to determine the feasibility of paths and the relative preference of paths. MPLS–OMP also relies upon the IGP to provide link loading information. The flooding rate and encapsulation of link loading information provided by OSPF or IS–IS is defined in [11, 8].

The OSPF or IS–IS flooding of link loading information must be enabled. The OSPF or IS–IS load adjustment should be disabled since the amount of traffic forwarded directly by IP (without MPLS encapsulation) will be minimal and have little or no impact on loading.

3 Link, Path, and Path Set Equivalent Load

Link loading and link loss is determined from information flooded by OSPF or IS–IS as described in Section 2. “Link equivalent load” on each individual link is computed based on link utilization and link loss as described in [11]. A path is a series of links. “Path equivalent load” is the maximum link equivalent load along the path. “Path set equivalent load” is the minimum path equivalent load among a set of paths. Path and path set equivalent load are defined precisely in Appendix B.2.

4 Initial Path and Initial Loading

From the standpoint of a given ingress a given egress at some point either at some epoch, or after an outage, makes a transition from completely unreachable to reachable. When this transition occurs, a single initial LSP is created from the ingress to the egress without regard to traffic loading. The shortest path based on IGP metrics is used. If multiple paths have identical shortest metrics, then all of these paths may be used as an initial path set. The best path criteria may be further relaxed as described and recommended in [11] to create a larger number of initial paths. All traffic to the given egress is put on this path or set of paths. Further details are found in Appendix B.3

5 Creating Alternate Paths

A path set may consist of one or more paths. Many path sets will contain only the initial path created when the egress first became reachable as described in Section 4. The path set loading is determined as described in Section 3. The path set loading is compared to a set of thresholds to determine if an alternate path should be created.

The set of thresholds should be configurable as described in Appendix A. The symbolic names MIN_NEW_PATH, MAX_NEW_PATH, NEW_PATH_INCR, and OVERLOAD_PERSIST will be used here. MIN_NEW_PATH, MAX_NEW_PATH, and NEW_PATH_INCR are in units of equivalent load, where 1.0 is full link utilization. OVERLOAD_PERSIST is in units of time, generally seconds.

A array of values will be maintained for each path set. The array will be indexed such that there is an array entry for each value from MIN_NEW_PATH to MAX_NEW_PATH inclusive of MAX_NEW_PATH in increments of NEW_PATH_INCR. For example, if MIN_NEW_PATH is 0.80, MAX_NEW_PATH is 1.15, and NEW_PATH_INCR is 0.05, there would be array entries corresponding to 0.85, 0.90, 0.95, 1.00, 1.05, 1.10, 1.15. Each array entry can contain a timestamp or a value illegal as a timestamp referred to here as NEVER. Each array element is initialized to NEVER.

If the path set loading is less than MIN_NEW_PATH then all array entries are set to NEVER. If the path set loading is greater than MIN_NEW_PATH, then each array element that corresponds to a loading below or equal to the path set loading that contains the value NEVER is set to the current time. Each array entry that corresponds to a value above the current path load is set to NEVER.

When some array values are not all set to NEVER, a conditional defined in Appendix B.4 is periodically checked for each array entry. If the conditional is true for any array entry, the decision process is terminated and the search for an alternate path is undertaken.

To determine if an alternate path is available, and if so determine the best alternate path, an SPF calculation is performed after temporarily removing all links from the OSPF or IS–IS link state database whose link equivalent load equals or exceed the path set equivalent load. This SPF may result in no path between the ingress and egress. If so, no additional paths are available which can remove load from the path set under consideration, and no action can be taken. Otherwise, the SPF will yield one or more best paths between ingress and egress. LSPs for these paths are set up and added to the path set.

The algorithms described above are provided in a more concise and precise form in Appendix B.4.

When a path is added to a path set, an initial loading and initial rate at which this loading is increased is set. These initial values and the loading adjustment algorithm is provided in Section 8.

The arrays are updated when new loading information arrives or every PATH_CHECK seconds and the arrays checked every PATH_CHECK seconds. This parameters is configurable as described in Appendix A.

6 Deleting from Underutilized Path Sets

Please note that simulations of this portion of the algorithm have not been completed. Some details may change.

Normal time of day load fluctuations or restoral of link failures can result in path sets that are significantly underutilized. When this occurs, one path at a time can be dropped from the path set. Parameters are defined in Appendix A which determine the conditions under which a path can be dropped. The loading on the most heavily loaded path in the path set must be below PATH_DEL_SHARE. When this condition becomes true a timestamp must be retained. If the condition is no longer true, the timestamp must be invalidated. If this condition remains true for a period of time, referred to here as "time to deletion", then a path can be selected for deletion.

The time to deletion must be dependent on loading contribution in order to avoid many ingress routers performing deletions at once. The computation must favor deletions from path sets with small traffic contributions. This favors eliminating paths with marginal benefit. The time to deletion is computed as follows. The constant PATH_DEL_PERSIST seconds is multiplied by the larger of zero or the difference between one and the ratio of total absolute (bytes/second) traffic contributed by the path set to the sum of absolute excess bandwidth on the paths.

The best candidate for deletion is the path which has the highest IGP cost. If a subset of paths containing more than one path have exactly equal IGP cost and are the highest, then the path among this subset with the lowest traffic share is chosen. Only one path should be deleted at a time.

The total contribution of traffic to the path set, as measured by the absolute LSP traffic levels, must not exceed PATH_DEL_EXCESS times the excess bandwidth on the remaining paths. When a path is selected as the best path to delete, this criteria must be checked. If this path cannot be deleted and there is no other path of equal IGP cost, then no deletion should take place. Deleting a path of low IGP cost and lower bandwidth might still be possible but such a path should not be deleted, since the lower IGP cost path that would be deleted is considered better than some of the remaining paths.

Once a path is selected for deletion, its traffic share should be divided among the remaining paths in proportion to the absolute bandwidth available on each remaining path. The LSP can then be deleted, after traffic is moved and the path is known to be idle. After a path is deleted, the time stamp should be artificially advanced by PATH_DEL_WAIT. This will insure that path deletions on this path set are spaced out in time by at least this amount of time.

The algorithms for deleting paths are desribed in Appendix B.5.

7 Handling Link State Changes

Please note that simulations of this portion of the algorithm have not been completed. Some details may change. In particular, optimizations done in the simulation code which dramatically reduce the number of SPF calculations have not been described here.

Some initial paths may be placed prior to complete convergence of the link state protocol. In addition, when a failed link is restored, there may not be an overload condition to force traffic back onto the restored link. If this were not corrected the network could settle into a state where path layout and possibly also loading was suboptimal.

The adjustments that occur due to a link failure will occur immediately or quickly. In some cases, all paths from a given ingress to a given egress will be lost. In this case, if connectivity is still feasible, the best remaining path in terms of cost is used, as described in Section 4. If paths remain, load is shifted to these paths as if the links had been deleted as described in Section 6.

The result of a link loss on a well utilized topology will generally be a period of suboptimal loading and congestion. As this condition persists, alternate paths will be added as described in Section 5. The most heavily congested links will be offloaded first, with the ingress contributing the largest share of traffic reacting first. The less loaded links and lesser contributors will react if congestion continues. In some cases, paths will be added faster than the balancing makes use of some of the new paths. As the balancing progresses, path sets which have created excess links will begin to prune the less beneficial links as described in Section 6.

When a link is restored to service, the SPF will yield multiple better paths to some egress than paths currently in use, which may also be less loaded than paths in use. If the topology is not overloaded, these paths may not be immediately considered.

Two conditions have been described which may cause better path to not be selected and may not be forced into consideration later by loading conditions. The first is where an initial path is selected prior to link state database convergence. The second is when a link is restored to service. This condition should not be allowed to persist indefinitely.

When a SPF is required due to a link transition to a full adjacency, those path sets for which the new best path is better than some paths in use and is not overloaded are marked. Where the existing set of paths is heavily loaded the new best paths will be used very quickly. For the remainder, the better paths will be added to path sets based on an event timer. This algorithm is described in detail in Appendix B.7.

8 Load Balancing

There is no load balancing done on a path set with only one path. When there is more than one path, fine adjustments can be made in the amount of traffic on each path by adjusting boundary values in a source/destination hash number space associated with each path. This technique is described in detail in [11].

The terms "traffic share" and "load increment" are defined in [11]. The loading on a particular path is determined by its traffic share. The rate of increase in loading, when offloading traffic from other paths, is determined by the move increment. Both of these variables are in units of hash table space. For example, if a 16 bit hash is used, these variables are in the range 0—65535.

A path set is constructed according to the rules described in Section 4 and Section 5. When a new path is added to an existing path set, the initial traffic share is set to a small value as described in Section 5.

9 MPLS–OMP Forwarding

The load balance itself utilizes the "next hop structures" described in [11]. The next hop structures themselves will differ slightly in that an ingress to an MPLS domain will be adding a label to the label stack in addition to making a forwarding decision. The algorithm for determining when to adjust load and for adjusting load is exactly as described in [11]. MPLS–OMP forwarding is described in detail in Appendix B.8.

Acknowledgments

Simulations were performed using hypotheitc cases and using UUNET's topology and loading. Simulations involving UUNET's network would not have been possible without data gathered by Dave Cleary, Yong Xue, and Yong Zhue of UUNET.

@@ To be completed. Get your valuable comments in please. @@

References

[1] Ross Callon, George Swallow, N. Feldman, A. Viswanathan, P. Doolan, and A. Fredette. A framework for multiprotocol label switching. Internet Draft (Work in Progress) draft-ietf-mpls-framework-02, Internet Engineering Task Force, 11 1997. ftp://ftp.isi.edu/internet-drafts/draft-ietf-mpls-framework-02.txt.

[2] Ross Callon, A. Viswanathan, and E. Rosen. Multiprotocol label switching architecture. Internet Draft (Work in Progress) draft-ietf-mpls-arch-02, Internet Engineering Task Force, 7 1998. ftp://ftp.isi.edu/internet-drafts/draft-ietf-mpls-arch-02.txt.

[3] R.W. Callon. Use of osi is-is for routing in tcp/ip and dual environments. Technical Report RFC 1195, Internet Engineering Task Force, 1990. ftp://ftp.isi.edu/in-notes/rfc1195.txt.

[4] Bruce Davie, Tony Li, Y Rekhter, and E. Rosen. Explicit route support in mpls. Internet Draft (Work in Progress) draft-davie-mpls-explicit-routes-00, Internet Engineering Task Force, 11 1997. ftp://ftp.isi.edu/internet-drafts/draft-davie-mpls-explicit-routes-00.txt.

[5] Bruce Davie, Y Rekhter, A. Viswanathan, S. Blake, Vijay Srinivasan, and E. Rosen. Use of label switching with rsvp. Internet Draft (Work in Progress) draft-ietf-mpls-rsvp-00, Internet Engineering Task Force, 3 1998. ftp://ftp.isi.edu/internet-drafts/draft-ietf-mpls-rsvp-00.txt.

[6] D. Farinacci, Tony Li, A. Conta, Y Rekhter, Dan Tappan, E. Rosen, and G. Fedorkow. Mpls label stack encoding. Internet Draft (Work in Progress) draft-ietf-mpls-label-encaps-03, Internet Engineering Task Force, 9 1998. ftp://ftp.isi.edu/internet-drafts/draft-ietf-mpls-label-encaps-03.txt.

[7] ISO/IEC. Iso/iec 10589 - information processing systems - telecommunications and information exchange between systems - intermediate system to intermediate system intra-domain routing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (iso 8473). Technical report, International Organization for Standardization, 1992. ftp://merit.edu/pub/iso/iso10589.ps.gz.

[8] Tony Li and Curtis Villamizar. Is-is optimized multipath (isis-omp). Internet Draft (Work in Progress) draft-villamizar-isis-omp-00, Internet Engineering Task Force, 10 1998. ftp://ftp.isi.edu/internet-drafts/draft-villamizar-isis-omp-00.txt.

[9] J. Moy. Ospf version 2. Technical Report RFC 2328, Internet Engineering Task Force, 1998. ftp://ftp.isi.edu/in-notes/rfc2328.txt.

[10] Bob Thomas, N. Feldman, P. Doolan, Loa Andersson, and A. Fredette. Ldp specification. Internet Draft (Work in Progress) draft-ietf-mpls-ldp-02, Internet Engineering Task Force, 11 1998. ftp://ftp.isi.edu/internet-drafts/draft-ietf-mpls-ldp-02.txt.

[11] Curtis Villamizar. Ospf optimized multipath (ospf-omp). Internet Draft (Work in Progress) draft-ietf-ospf-omp-01, Internet Engineering Task Force, 10 1998. ftp://ftp.isi.edu/internet-drafts/draft-ietf-ospf-omp-01.txt.

Security Considerations

MPLS–OMP is a method of setting up MPLS LSPs and balancing load across MPLS LSPs. MPLS–OMP, like MPLS itself, depends on the integrity of information obtains from the IGP. MPLS–OMP requires a link state protocol IGP relies upon loading information provided by the IGP in addition to the underlying routing functionality that is minimally required for any use of MPLS. If the IGP is compromised such that integrity of the information provided by the IGP is no longer maintained, substantial denial of service is possible. Use of MPLS–OMP does not add exposure to the IGP, nor does it provide any means to leverage attacks on the IGP beyond the existing potential for denial of service.

MPLS–OMP relies upon the ability of LDP or MPLS RSVP extensions to set up explicit LSPs. As a technique which uses these capabilities with no signaling mechanisms of its own, MPLS–OMP adds no further exposure to these signaling mechanisms. If sufficiently misconfigured, MPLS–OMP could add significant load to MPLS LSRs if misconfiguration is sufficient that paths are repeatedly created and deleted.

Author's Addresses

Curtis Villamizar
ANS
<curtis@ans.net>

A Configuration Options

MPLS–OMP configurable parameters do not control the rate of traffic adjustment or the rate of change in traffic adjustment. These rates should be kept as defined in OSPF–OMP [11] in order to insure that stability is not impacted.

The MPLS–OMP configurable parameters control the conditions for creating new paths, the initial state of new paths, and the conditions for deleting paths.

Parameters: Conditions for Creating New Paths

Parameter Name	Units	Range	Default
MIN_NEW_PATH	unitless	0.50-1.00	0.45
MAX_NEW_PATH	unitless	see below	1.10
NEW_PATH_INCR	unitless	0.01-0.25	0.05
OVERLOAD_PERSIST	seconds	60-3600	60

```
MAX_NEW_PATH >= MIN_NEW_PATH + NEW_PATH_INCR
MAX_NEW_PATH <= 10.0
```

Parameters: Initial State of New Paths

Parameter Name	Units	Range	Default
NEW_LOAD_FRACTION	unitless	0.01-1.00	0.10
NEW_LOAD_EPSILON	unitless	0.00-1.00	0.00
NEW_MOVE_FRACTION	unitless	0.01-1.00	0.25
NEW_MOVE_EPSILON	unitless	0.00-1.00	0.01

Parameters: Conditions for Deleting Paths

Parameter Name	Units	Range	Default
PATH_DEL_SHARE	hash space	0.00-1.00	0.35
PATH_DEL_PERSIST	seconds	60-86400	1200
PATH_DEL_EXCESS	hash space	0.00-1.00	0.75

PATH_DEL_SHARE must be significantly less than MIN_NEW_PATH to insure sufficient hysteresis.

Parameters: Timer based processing

Parameter Name	Units	Range	Default
PATH_CHECK	seconds	15-300	30
PATH_ADD_WAIT	seconds	15-3600	240
PATH_DEL_WAIT	seconds	15-3600	240
SPF_PERSIST	seconds	60-3600	300
SPF_RECHECK	seconds	60-86400	900

B Concise Statement of Algorithms

MPLS–OMP uses links state information and link loading information flooded via a link state IGP. MPLS–OMP uses this information to determine where to place MPLS LSPs and to adjust load where multiple LSPs are established per traffic flow. The traffic flows are assumed to be aggregates of very large numbers of end-to-end flows such that dividing traffic using a hash over the IP source and IP destination provides a means of distributing load.

The MPLS–OMP algorithms are stated here in pseudocode with limited accompanying text. All upper case variables are generally constants or parameters defined in Appendix A. The pseudocode syntax is not formally defined but is sufficiently similar to the "C" or perl programming languages that it should be understandable.

B.1 Flooding Loading Information

The flooding of loading information is the responsibility of the link state IGP. The data formats and flooding rate is provided in the OSPF–OMP and ISIS–OMP documents [11, 8]. The MPLS routers are all assumed to participate in the IGP routing. The ingress router MPLS implementation and IGP implementation must be sufficiently coupled such that links state changes and loading information is passed as received from the IGP to MPLS.

B.2 Link, Path, and Path Set Equivalent Load

Link loading and link loss is determined from information flooded by OSPF or IS–IS. The variables "Utilization" and "Loss" below are the utilization and loss values flooded by the IGP. The utilization and loss directly yield a "link equivalent load". The maximum utilization of any link on a path yields the "path equivalent load". The minimum utilization of a set of paths associated with reaching a specific egress are a "path set equivalent load".

```
EventHandler NewLoadingInfo(Link): {
    if (Loss{Link} < 0.005) {
        LinkEquivLoad{Link} = Utilization{Link};
    } else {
        if (Loss{Link} <= 0.09) {
            LossComp = 10 * sqrt(Loss{Link});
        } else {
            LossComp = 3;
        }
        LinkEquivLoad{Link} = Utilization{Link} * LossComp;
    }
}
```

Path equivalent load is the maximum link equivalent load on a given path.

```
Function ComputePathLoad(Path) {
```

```
        PathEquivLoad = 0;
        foreach Link ( Links{Path} ) {
            Load = LinkEquivLoad{Link}
            if (Load > PathEquivLoad)
                PathEquivLoad = Load;
        }
        PathEquivLoad{Path} = PathEquivLoad;
    }
```

The path set equivalent load is the minimum path equivalent load of a set of LSPs between a given ingress and a given egress. Some later computations also require the maximum loading of any path on a path set. Both can be computed at the same time.

```
    Function ComputePathSetLoad(PathSet) {
        undefine PathSetEquivLoad;
        PathSetMaxLoad = 0;
        foreach Path ( Paths{PathSet} ) {
            ComputePathLoad(Path);
            Load = PathEquivLoad{Path};
            if (!defined(PathSetEquivLoad)) {
                PathSetEquivLoad = Load;
            } else if (Load < PathSetEquivLoad) {
                PathSetEquivLoad = Load;
            }
            if (Load > PathSetMaxLoad)
                PathSetMaxLoad = Load;
        }
        PathSetEquivLoad{PathSet} = PathSetEquivLoad;
        PathSetMaxLoad{PathSet} = PathSetMaxLoad;
    }
```

The pseudocode fragments above and elsewhere in this section do not implement any form of data structure threading to allow all parameters dependent on a given link to be updated when the link utilization changes. This sort of data structure threading to reflect dependencies among computed parameters can save considerable time when processing the periodic timer events. This type of implementation optimization is not expressed in the pseudocode but may be nessecary to produce an implementation that scales well with respect to the number of path set, paths, and links.

B.3 Initial Path and Initial Loading

At some point a path that had been unreachable becomes reachable. When this occurs, an initial path is set according to the results of the SPF calculation. The following pseudocode describes the initialization.

```
    EventHandler NewlyReachable(Egress): {
        new Paths = SPF(Egress);
```

```
        new PathSet;
        Egress{PathSet} = Egress;
        undefined Paths{PathSet};
        insert PathSets PathSet;
        foreach Path ( Paths ) {
            insert Paths{PathSet} Path;
            PathSetBelowMax{PathSet} = NEVER;
            for (Util = MIN_NEW_PATH + NEW_PATH_INCR;
                 Util += NEW_PATH_INCR;
                 Util <= MAX_NEW_PATH) {
                PathSetArray{PathSet,Util} = NEVER;
            }
        }
        ComputePathSetLoad(PathSet);
    }
```

This initialization may also occur when a link failure makes all of the paths to a given egress currently in use unfeasible. In this case, the old path set corresponding to this egress is deleted. A new path must be selected and the egress is treated as if it had just become reachable.

B.4 Creating Alternate Paths

Deciding whether to create an alternate path and then creating one can be broken down into a number of smaller procedures.

Every PATH_CHECK seconds the set of arrays used to determine whether to add or delete a path are checked.

```
    Function AdjustPathArrays(PathSet) {
        Load = PathSetEquivLoad{PathSet};
        for (Util = MIN_NEW_PATH + NEW_PATH_INCR;
             Util += NEW_PATH_INCR;
             Util <= MAX_NEW_PATH) {
            if (Util > Load) {
                if (PathSetArray{PathSet,Util} == NEVER)
                    break;
                else
                    PathSetArray{PathSet,Util} = NEVER;
            } else if (PathSetArray{PathSet,Util} == NEVER) {
                PathSetArray{PathSet,Util} = NOW;
            }
        }
    }
```

The function below is used in pseudocode later in this section. When loading is sufficient to require checking the path's contributions to the loadings on the available paths, the following computation is made.

```
Function GetPathContribution(PathSet): {
    TotalTraffic = 0;
    TotalCapacity = 0;
    foreach Path ( Paths{PathSet} ) {
        TotalTraffic += LoadOnLSP{Path};
        undefine Capacity;
        foreach Link ( Links{Path} ) {
            if (!defined(Capacity))
                Capacity = LinkCapacity{Link};
            else if (LinkCapacity{Link} < Capacity)
                Capacity = LinkCapacity{Link};
        }
        TotalCapacity += Capacity;
    }
    return TotalTraffic / TotalCapacity;
}
```

Every PATH_CHECK seconds a check is made to determine whether to try to create a new path. An elapsed time is computed. The elapsed time is the difference between the current time and the time stored in the array entry. A load compensation factor is computed. It is the ratio of two differences, the difference between the loading corresponding to the array entry and MIN_NEW_PATH and the difference between MAX_NEW_PATH and MIN_NEW_PATH. A traffic contribution factor is computed. This is ratio of the current total path set traffic to the sum of the capacities of each path. The capacity of an individual path is taken to be the capacity of the slowest link on that path. The OVERLOAD_PERSIST constant is compared to the product of the elapsed time, the load compensation factor, and the traffic contribution factor. If the product is the greater of the two, the condition is true for this array entry and an atempt is made to create a new path.

```
Function CheckForNewPath(PathSet) {
    undefine PathContribution;
    for (Util = MIN_NEW_PATH + NEW_PATH_INCR;
         Util += NEW_PATH_INCR;
         Util <= MAX_NEW_PATH) {
        Then = PathSetArray{PathSet,Util};
        if (Then == NEVER)
            return 1;
        Elapsed = NOW - Then;
        LoadComp = (Util - MIN_NEW_PATH)
            / (MAX_NEW_PATH - MIN_NEW_PATH);
        if (!defined(PathContribution))
            PathContribution
                = GetPathContribution(PathSet);
        if (Elapsed * LoadComp * PathContribution
            > OVERLOAD_PERSIST) {
            CreateNewPath(PathSet);
            break;
        }
    }
}
```

When the decision has been made to attempt to create a new path, the first step is to search for a candidate path. If a new path is available it is initialized. The set of arrays is incremented by PATH_ADD_WAIT whether a new path is found or not in order to defer repeating the check for this amount of time.

```
Function CreateNewPath(PathSet) {
    NewPaths = FindAlternatePath(PathSet);
    if (defined(NewPaths)) {
        MostCapacity = 0;
        undefine BestNewPath
        foreach NewPath ( NewPaths )
            ComputePathLoad(NewPath);
            undefine Capacity;
            foreach Link ( Links{Path} ) {
                ExcessCapacity = LinkCapacity{Link}
                    * (1 - PathEquivLoad{NewPath});
                if (!defined(Capacity))
                    Capacity = ExcessCapacity;
                else if (ExcessCapacity < Capacity)
                    Capacity = ExcessCapacity;
            }
            if (MostCapacity < Capacity) {
                MostCapacity = Capacity;
                BestNewPath = NewPath;
            }
        }
        if (defined(BestNewPath))
            InitilizeNewPath(PathSet,BestNewPath);
    }
    for (Util = MIN_NEW_PATH + NEW_PATH_INCR;
         Util += NEW_PATH_INCR;
        Util <= MAX_NEW_PATH) {
        if (PathSetArray{PathSet,Util} == NEVER)
            break;
        else
            PathSetArray{PathSet,Util}
                += PATH_ADD_WAIT;
    }
}
```

Finding an alternate path involves running an SPF calculation with some of the links disabled. Note that the SPF can return more than one equal cost path. There can also be enough links disabled that the SPF returns no path at all.

```
Function FindAlternatePath(PathSet) {
    Load = PathSetEquivLoad{PathSet};
    foreach Link ( LinkStateDataBase ) {
        if (LinkEquivLoad{Link} >= Load)
            TemporarilyDisableLink(Link);
```

```
        }
        NewPaths = SPF(Egress{PathSet});
        EnableAllLinks();
        return NewPaths;
    }
```

Once an alternate path has been found, the traffic share and the move increment are initialized. The new path is then added to the path set.

Note that in simulations, the initial share was set to 0 and the initial increment was set to about 0.01. Either the simulation will be changed to reflect the algorithm below or the algorithm will be simplified. The initial conditions do not seem to have much of an effect if set very conservatively as recommended or if set conservatively and fixed as in the current simulations.

```
    Function InitilizeNewPath(PathSet,NewPath): {
        TrafficShare{PathSet,NewPath} = 0;
        PathExcessCapacity = ExcessCapacity(NewPath);
        TotalExcessCapacity = PathExcessCapacity;
        foreach Path ( Paths{PathSet} ) {
            Excess = ExcessCapacity(Path);
            if (Excess > 0)
                TotalExcessCapacity += Excess;
        }
        TargetShare = NEW_LOAD_EPSILON;
        TargetShare2 = HASH_SPACE_SIZE * NEW_LOAD_FRACTION
              (PathExcessCapacity / TotalExcessCapacity);
        if (TargetShare2 < TargetShare)
            TargetShare = TargetShare2;
        TargetIncr = NEW_MOVE_EPSILON;
        TargetIncr2 = HASH_SPACE_SIZE * NEW_MOVE_FRACTION
            (PathExcessCapacity / TotalExcessCapacity);
        if (TargetIncr2 < TargetIncr)
            TargetIncr = TargetIncr2;
        MoveIncrement{PathSet,NewPath} = TargetIncr;
        TotalCapacity = 0;
        foreach Path ( Paths{PathSet} ) {
            TotalCapacity += Capacity{Path};
        }
        foreach Path ( Paths{PathSet} ) {
            Shares = TargetShare
                * Capacity{Path} / TotalCapacity;
            if (TrafficShare{PathSet,Path} < Shares)
                Shares = TrafficShare{PathSet,Path};
            TrafficShare{PathSet,NewPath} += Shares;
            TrafficShare{PathSet,Path} -= Shares;
        }
        insert Paths{PathSet} NewPath
    }
```

B.5 Deleting Alternate Paths

Please note that simulations of this portion of the algorithm have not been completed. Some details may change.

Every PATH_CHECK seconds each path set is checked to see if the set of paths contain sufficient excess bandwidth to drop one of the paths. A simple single threshold is used but the ratio of contributed traffic to excess capacity is used to determine the time delay before deletion. This should avoid synchronized actions taken by multiple ingress routers.

```
Function CheckPathExcess(PathSet) {
    MaxLoad = PathSetMaxLoad{PathSet};
    if (MaxLoad >= PATH_DEL_SHARE) {
        PathSetBelowMax{PathSet} = NEVER;
        return;
    }
    if (PathSetBelowMax{PathSet} == NEVER) {
        PathSetBelowMax{PathSet} = NOW;
        return;
    }
    Elapsed = NOW - PathSetBelowMax{PathSet};
    Traffic = 0;
    ExcessCapacity = 0;
    foreach Path ( Paths{PathSet} ) {
        Traffic += LoadOnLSP{Path};
        Excess = ExcessCapacity(Path);
        if (Excess > 0)
            ExcessCapacity += Excess;
    }
    TimeToDeletion = PATH_DEL_PERSIST
        * (1 + (Traffic / ExcessCapacity));
    if (Elapsed >= TimeToDeletion) {
        DeletePath(PathSet);
        PathSetBelowMax{PathSet} += PATH_DEL_WAIT;
    }
}
```

Once the decision is made to drop a path, a path must be selected to drop. The load must be moved to the remaining paths and storage and and other resources recovered.

```
Function DeletePath(PathSet): {
    HighCost = 0;
    foreach Path ( Paths{PathSet} ) {
        Cost = PathLinkCosts(Path);
        if ((Cost > HighCost)
            || (Cost == HighCost
                && TrafficShare{PathSet,Path} < LowShare) {
            PathToDelete = Path;
```

```
                HighCost = Cost;
                LowShare = TrafficShare{PathSet,Path};
            }
        }
        foreach Path ( Paths{PathSet} ) {
            if (Path != PathToDelete)
                TotalCapacity += Capacity{Path};
        }
        while (TrafficShare{PathSet,PathToDelete} != 0) {
            TargetShare = TrafficShare{PathSet,PathToDelete};
            foreach Path ( Paths{PathSet} ) {
                if (Path == PathToDelete)
                    continue;
                Shares = TargetShare
                    * Capacity{Path} / TotalCapacity;
                if (Shares < 1)
                    Shares = 1;
                if (TrafficShare{PathSet,PathToDelete} < Shares)
                    Shares = TrafficShare{PathSet,PathToDelete};
                TrafficShare{PathSet,PathToDelete} -= Shares;
                TrafficShare{PathSet,Path} += Shares;
                if (TrafficShare{PathSet,PathToDelete} == 0)
                    break;
            }
        }
        delete Paths{PathSet} PathToDelete
        RecoverStorage(PathToDelete);
    }
```

B.6 Load Balancing

There are no differences in the algorithm to balance a set of paths in MPLS–OMP from the methods in OSPF–OMP. The path set in MPLS–OMP maps directly into a next hop structure as described in the OSPF–OMP document [11]. Behavior differs slightly in that in MPLS–OMP the decision made at the ingress determines the entire path of the traffic while in OSPF–OMP the decision at the ingress is affected by decisions at every downstream router. Regardless the algorithm to adjust the traffic share is identical.

B.7 Handling Link State Change

Please note that simulations of this portion of the algorithm have not been completed. Some details may change. In particular, optimizations done in the simulation code which dramatically reduce the number of SPF calculations have not been described here.

Link state changes may make some paths infeasible and make make new paths available. An SPF calculation may involve one or many links state changes.

When an SPF calculation is required, first the set of paths is checked to see if any of those paths have become infeasible. These paths are deleted. This may also render some egress temporarily unreachable. If so, the path set structures are deleted as well.

```
@@ pseudo-code to be converted from simulation perl code @@
```

After processing the set of paths that must be deleted, the set of egress that are now unreachable are checked. If there are feasible paths, these paths are created. The remaining egress are checked to see if paths now exist that are less loaded than the path set and have a lower cost than some of the paths in use. If so, these path sets are marked with a time stamp.

```
@@ pseudo-code to be converted from simulation perl code @@
```

The loading on some paths may be reduced by balancing done for other egress or balancing done by other routers. For this reason, the SPF must be rechecked periodically to see if better paths than the ones being used exist. This check is performed every SPF_RECHECK seconds. Path sets are marked as they would be if an SPF was just completed.

```
@@ pseudo-code to be converted from simulation perl code @@
```

Every PATH_CHECK seconds those path sets that are timestamped to indicate that a better path needs to be considered are checked to see if it is time to add the new path. The constant SPF_PERSIST is used with a compensation factor for load and traffic contribution similar to the compensations made in Appendix B.4. If the time criteria is met, then the path is added. The SPF must be rechecked to see if other paths are available which are better than some of the paths in use. If a few such paths are added the path set can end up with some excess capacity and the worst paths can be dropped as described in Appendix B.5.

```
@@ pseudo-code to be converted from simulation perl code @@
```

B.8 MPLS–OMP Forwarding

MPLS–OMP requires that the ingress router perform a hash on the IP source and IP destination in order to to spread traffic over a set of paths. This is a simple operation, but access to the IP header is not something that can always be taken for granted.

In the case where an MPLS domain receives only IP encapsulated traffic from neighboring domains, the IP header is accessible. LSRs interior to the MPLS domain have no need to look at the IP header.

The case where the IP header is not immediately accessible is where traffic is exchanged across MPLS domains that is already MPLS encapsulated. The purpose of supporting a label stack is to allow the LSP for multiple MPLS domains to be indicated at the ingress of the first MPLS domain. Supplying an MPLS label to an upstream MPLS domain that only applies to a continuously changing specific set of source and destination hash values is not likely to be feasible. If so, the top label must be assigned at the MPLS domain ingress. Labels further down the stack might be present and relevant to downstream MPLS domains. In such a scenario it may make sense to use the label provided to the ingress to indicate traffic classification only, with the egress and hash determined by finding the IP header and looking at both IP source and IP destination. This isolates the upstream MPLS domain from any change to internal routing.

The use of MPLS across MPLS domains as a means of passing traffic classification is beyond the scope of this document. The inability to quickly find the IP header in an MPLS stack of unknown depth is a cause for concern. This may be an issue that has to be addressed in the MPLS encapsulation. Alternately label numbers provided to the upstream MPLS domain may be assigned such that the label number can reliably indicate the label stack depth of incoming traffic.

Given that the IP header is at the top of the stack or can be located quickly enough, a hash is performed on the IP source and destination. The hash number space is subdivided on arbitrary boundaries among the set of possible LSPs. This is indicated in the figure below.

```
+-------+-------+------+-------+
|   IP Source Address          |
+-------+------+-------+-------+
|   IP Destination Address     |
+-------+------+------+-------+
                |
                V
         hash function
                |
                V
        +------+-------+
        |  hash value   |
        +------+-------+
                |
                V
+-------+------+------+-------+
| hash boundary |  MPLS LSP #1  |
+------+------+------+-------+
| hash boundary |  MPLS LSP #2  |
+-------+------+------+-------+
     ...
+-------+------+------+-------+
| hash boundary |  MPLS LSP #N  |
+-------+------+------+-------+
                |
                V
            selection
                |
                V
           MPLS label
```

Conceptually the selection process involves walking through the set of hash boundary LSP pairs. If the hash value exceeds the boundary value, the next LSP is tried. If the last LSP in the data structure is reached, it is used regardless of the boundary. The selection would yield a MPLS label to use which would in turn yield all of the nessecary forwarding information.

The actual algorithm used is unlikely to be this one, but would instead be an algorithm with the same net effect but more efficient with a large number of possible LSPs.

C Comparison of MPLS–OMP and OSPF-OMP Dynamics

ISIS–OMP and OSPF–OMP differ in fairly insignificant ways. The behavior of MPLS–OMP and OSPF–OMP differ in more significant ways.

In OSPF–OMP the paths are a direct result of the SPF calculation. As a result, in OSPF–OMP there are limitations to the set of paths on which traffic can be split for a given ingress and egress that can result in an imperfect load balance. This limitation can be overcome by carefully selecting the link costs. The most serious difficulty arises when links fail and the OSPF costs are suboptimal for the remaining topology.

MPLS–OMP differs from OSPF–OMP in the way in which the candidate set of paths are created. MPLS–OMP is capable of optimal load balancing in situations in which OSPF–OMP does not create a perfect balance across critical cutsets in the network topology. The immediate benefit is relief from the tedious task (one of N–P complete time complexity) of selecting link costs. A second benefit is an improvement in the behavior in the presence of link failure.

D Simulations of MPLS–OMP Behavior

@@ To be completed. This work will be made available at http://engr.ans.net/mpls-omp/. @@