ATNP/WG 2

WP/

June 20, 1997

AERONAUTICAL TELECOMMUNICATION NETWORK PANEL

WORKING GROUP 2 (Internet)

Langen, GERMANY, 23 - 26 June 1997

QOS MANAGEMENT AND ROUTING

Prepared by: James Moulton

Presented by: James Moulton

Summary: The future work program for ATNP/3 includes an item on QoS provisioning and routing. As a starting point for this work, the latest IETF Internet Draft on QoS routing is attached.

InternetEngineering TaskForce   R.Guerin/S. Kamat/A. Orda
INTERNETDRAFT   IBM/IBM/Technion
T.Przygienda/D. Williams
Fore/IBM
25 March 1997

QoS Routing Mechanismsand OSPFExtensions
draft-guerin-qos-routing-ospf-01.txt

Status of This Memo

Abstract

This memo describes extensions to the OSPF [Moy94] protocol to support QoS routes. The focus of the document is on the algorithms used to compute QoS routes and on the necessary modifications to OSPF to support this function, e.g., the information needed, its format, how it is distributed, and how it is used by the QoS path selection process. Aspects related to how QoS routes are established and managed are also discussed. The goal of this document is to identify a framework and possible approaches to allow deployment of QoS routing capabilities with the minimum possible impact to the existing routing infrastructure.

Contents

1. Introduction

In this document we describe a set of proposed additions to the OSPF routing protocol (the additions are built on top of OSPF V2) to support Quality-of-Service (QoS) routing in IP. In particular, we discuss the metrics required to support QoS, the associated link advertisement mechanisms, the path selection algorithm, as well as aspects of route establishment (pinning and unpinning). Our goals are to define an approach which while achieving the goals of improving performance for QoS flows (likelihood to be routed on a path capable of providing the requested QoS), does so with the least possible impact on the existing OSPF protocol. Given the inherent complexity of QoS routing, achieving this goal obviously implies trading-off ‘‘optimality’’ for ‘‘simplicity’’, but we believe this to be required in order to facilitate deployment of QoS routing capabilities.

1.1. Overall Framework

We consider a network (1) that supports both best-effort packets and packets with QoS guarantees. The way in which the network resources are split between the two classes is irrelevant to our proposal, except for the assumption that each QoS capable router in the network is able to dedicate some of its resources to satisfy the requirements of QoS packets. QoS capable routers are also assumed to be able to identify and advertise the amount of their resources that remain available for additional QoS flows. In addition, we limit ourselves to the case where all the routers involved support the QoS extensions described in this document, i.e., we do not consider the problem of establishing a route in an heterogeneous environment with routers that are QoS-capable and others that are not. Furthermore, in this document we focus on the case of unicast flows, although many of the additions we define are applicable to multicast flows as well.

We assume that a flow with QoS requirements will specify them in some fashion that is accessible to the routing protocol. For example, this could correspond to the arrival of an RSVP [RZB+96] PATH message, whose TSpec is passed to routing together with the destination address. After processing such a request, the routing protocol returns a path that it deems the most suitable given the flow’s requirements. Depending on the scope of the path selection

----
1. In this document we commit the abuse of notation of calling a ‘‘network’’ the interconnection of routers and networks through which we attempt to compute a QoS path.

process, this returned path could range from simply identifying the best next hop, i.e., a traditional hop-by-hop routing, to specifying all intermediate nodes to the destination, i.e., a source route. Note that this decision impacts the operation of the path selection algorithm as it translates into different requirements in order to construct and return the appropriate path information. Note also that extension to multicast paths will impact differently a source routed and a hop-by-hop approach.

In this document, we will focus on hop-by-hop routing. The algorithms solutions for path computation and establishment can be easily modified for source routing and such extensions are discussed in appendix C.

Once a suitable path has been identified, the flow is assigned to it (pinning) and remains assigned to it until it either releases the path (unpinning) or deems that it has become unsuitable, e.g., because of link failure or unavailability of the necessary resources. Note that resource reservation and/or accounting should help limit the frequency of the latter.

In this document, we focus on the aspect of selecting an appropriate path based on information on link metrics and flow requirements. There are obviously many other aspects that need to be specified in order to define a complete proposal for QoS routing. Issues such as specifying the frequency of updates and the granularity of advertised changes to metrics, support for heterogenous areas with a mix of QoS capable and incapable routers, etc., require further study. The discussion of a complete solution to these problems is, however, deferred to subsequent versions of this draft.

1.2. Simplifying Assumptions

In order to achieve our goal of a minimum impact to the existing protocol, we impose certain restrictions on the range of requirements the QoS path selection algorithm needs to deal with directly. Specifically, a policy scheme is used to a priori prune from the network, those portions that would be unsuitable given the requirements of the flow. This limits the ‘‘optimization’’ performed by the path selection to a containable set of parameters, which helps keep complexity at an acceptable level. Specifically, the path selection algorithm will focus on selecting a path that is capable of satisfying the bandwidth requirement of the flow, while at the same time trying to minimize the amount of network resources that need to be allocated to support the flow, i.e., minimize the number of hops used.

This focus on bandwidth is adequate in most instances, but does not fully capture the complete range of potential QoS requirements. For example, a delay-sensitive flow of an interactive application could be put on a path using a satellite link, if that link provided a direct path and had plenty of unused bandwidth. This would clearly be an undesirable choice. Our approach to preventing such poor choices, is to assign delay-sensitive flows to a policy that would eliminate from the network all links with high propagation delay, e.g., satellite links, before invoking the path selection algorithm. In general, each existing policy would present to the path selection algorithm its correspondingly pruned network topology, and the same algorithm would then be used to generate an appropriate path.

Another important aspect in minimizing the impact of QoS routing is to develop a solution that has the smallest possible computing overhead. Additional computations are unavoidable, but it is desirable to keep the total cost of QoS routing at a level comparable to that of traditional routing algorithms. In this document, we describe several alternatives to the path selection algorithm, that represent different trade-offs between simplicity, accuracy, and computational cost. In particular, we specify algorithms that generate exact solutions based either on pre-computations or on-demand computations. We also describe algorithms that allow pre-computations at the cost of some loss in accuracy, but with possibly lower complexity or greater ease of implementation. It should be mentioned, that while several alternative algorithms are described in this document, the same algorithm needs to be used consistently within a given routing domain. This requirement can be relaxed when a source routed approach is used as the responsibility of selecting a QoS path lies with a single entity, the origin of the request, which ensures consistency. Hence, it may then be possible for each router to use a different path selection algorithm. However, in general, the use of a common path selection algorithm is recommended, if not necessary, for proper operation.

The rest of this document is structured as follows. In Section 2, we describe the path computation process and the information it relies on. In Section 3, we briefly review some issues associated with path management and their implications. In Section 4, we go over the extensions to OSPF that are needed in order to support the path selection process of Section 2. Finally, several appendices provide details on the different path selection algorithms described in Section 2, elaborate on path management mechanisms, and outline several additional work items.

2. Path Selection Information and Algorithms

This section describes several path selection algorithms that can be used to generate QoS capable paths based on different trade-offs between accuracy, computational complexity, and ease of implementation. In addition, the section also covers aspects related to the type of information, i.e., metrics, on which the algorithms operate, and how that information is made available, i.e., link state advertisements. The discussion on these topics is of a generic nature, and OSPF specific details are provided in Section 4.

2.1. Metrics

As stated earlier, the process of selecting a path that can satisfy the QoS requirements of a new flow relies on both the knowledge of the flow’s requirements and characteristics, and information about the availability of resources in the network. In addition, for purposes of efficiency, it is also important for the algorithm to account for the amount of resources the network has to allocate in order to support a new flow. In general, the network prefers to select the ‘‘cheapest’’ path among all paths suitable for a new flow. Furthermore, the network may also decide not to accept a new flow for which it identified a feasible path, if it deems the cost of the path to be too high. Accounting for these aspects involves several metrics on which the path selection process is based. They include:

- Link available bandwidth: As mentioned earlier, we assume that most QoS requirements are derivable from a rate-related quantity, termed ‘‘bandwidth’’. We further assume that associated with each link is a maximal bandwidth value, e.g., the link physical bandwidth or some fraction thereof that has been set aside for QoS flows. Since for a link to be capable of accepting a new flow with given bandwidth requirements, at least that much bandwidth must be still available on the link, the relevant link metric is, therefore, the (current) amount of available (i.e., unallocated) bandwidth.

- Hop-count: This quantity is used as a measure of the path cost to the network. A path with a smaller number of hops (that can support a requested connection) is typically preferable, since it consumes fewer network resources. While as a general rule each edge in the graph on which the path is computed should be counted as one hop, some edges, specifically those that connect a transit network to a router, should not be taken into account. (See Appendix B for a detailed explanation.)

- Policy: As previously discussed, policies are used to identify the set of links in the network that need to be considered when running the path selection algorithm. In particular, policies are used to prune from the network links that are incompatible, performance or characteristics wise, with the requirements of a flow. A specific policy example of special importance, is the elimination of high latency links when considering path selection for delay sensitive flows. The use of policies to handle specific requirements allows considerable simplification in the optimization task to be performed by the path selection algorithm.

2.2. Advertisement of Link State Information

It is assumed that each router maintains an updated database of the network topology, including the current state (available bandwidth) of each link. As described in Section 4, the distribution of link state (metrics) information is based on extending OSPF mechanisms. However, in addition to how link state information is distributed, another important aspect is when such distribution is to take place.

Ideally, one would want routers to have the most current view of the bandwidth available on all links in the network, so that they can make the most accurate decision on which path to select. Unfortunately, this then calls for very frequent updates, e.g., close to every time the available bandwidth of a link changes, which is neither scalable nor practical. Alternatively, one may opt for a simple mechanism based on periodic updates, where the period of updates is determined based on a tolerable corresponding load on the network and the routers. The main disadvantage of such an approach is that major changes in the bandwidth available on a link could remain unknown for a full period and, therefore, result in many incorrect routing decisions.

As a result, we propose to use a simple hybrid update mechanism, that attempts to reconcile accuracy of link state information with the need for the smallest possible overhead. Periodic updates are used, say every T seconds, to notify nodes of any change of more than ffi in the available bandwidth of a link, and event-driven updates are generated immediately whenever the change in available link bandwidth since the last update exceeds . The values for T, ffi, and depend on network size, link speed, processing capabilities, and overall traffic patterns, but typical values would be: T 30sec, ffi   10%, 40%. Regardless of bandwidth changes, as in the current OSPF specifications, we also impose a minimum interval between consecutive updates, e.g., we do not allow any particular LSA to get updated more than once every MinLSInterval seconds, e.g., 5, in order to prevent the possibility of overload.

2.3. Path Selection Algorithms

There are several aspects to the path selection algorithms. The main ones include the optimization criteria it relies on, the exact topology on which it is run, and when it is invoked.

As mentioned before, invocation of the path selection algorithm can be either per flow, or when warranted by changes in link states when the algorithm used allows precomputation of paths (more on this below).

The topology on which the algorithm is run is, as with the standard OSPF path selection, a directed graph where vertices (2) consist of routers and networks (transit vertices) as well as stub networks (non-transit vertices). When computing a path, stub networks are added as a post-processing step, which is essentially similar to what is done with the current OSPF routing protocol. In addition, for each policy supported on a router, the topology used by the path selection algorithm is correspondingly pruned to reflect the constraints imposed by the policy, and in some instances the flow requirements.

The optimization criteria used by the path selection are reflected in the costs associated with each interface in the topology and how those costs are accounted for in the algorithm itself. As mentioned before, the cost of a path is a function of both its hop count and the amount of available bandwidth. As a result, each interface has associated with it a metric, that corresponds to the amount of bandwidth which remains available on this interface. This metric is combined with hop count information to provide a cost value, in a manner that depends on the exact form of the path selection algorithm. It will, therefore, be detailed in the corresponding sections below, but all the different alternatives that are described share a common goal. That is, they all aim at picking a path with the minimum possible number of hops among those that can support the requested bandwidth. When several such paths are available, the preference is for the path whose available bandwidth (i.e., the smallest value on any of the links in the path) is maximal. The rationale for the above rule is the following: we focus on feasible paths (as accounted by the available bandwidth metric) that consume

----
2. In this document, we use the terms node and vertex interchangeably.

a minimal amount of network resources (as accounted by the hop-count metric); and the rule for selecting among these paths aims at balancing load as well as maximizing the likelihood that the required bandwidth will indeed be available.

It should be noted that standard routing algorithms are typically single objective optimizations, i.e., they may minimize the hop-count, or maximize the path bandwidth, but not both. Double objective path optimization is a more complex task, and, in general, it is an intractable problem [GJ79]. Nevertheless, as we will see, because of the specific nature of the two objectives being optimized (bandwidth and hop count), the complexity of our proposed algorithm(s) is competitive with even that of standard single-objective algorithms.

2.3.1. Algorithm for exact pre-computed QoS paths

In this section, we describe a path selection algorithm, that for a given network topology and link metrics (available bandwidth) allows us to pre-compute all possible QoS paths, and also has a reasonably low computational complexity. Specifically, the algorithm allows us to pre-compute for any destination a minimum hop count path with maximum bandwidth, and has a computational complexity comparable to that of a standard shortest path algorithm (3).

The path selection algorithm is based on a Bellman-Ford (BF) shortest path algorithm, which is adapted to compute paths of maximum available bandwidth for all hop counts. It is a property of the BF algorithm that, at its h-th iteration, it identifies the optimal (in our context: maximal bandwidth) path between the source and each destination, among paths of at most h hops. In other words, the cost of a path is a function of its available bandwidth, i.e., the smallest available bandwidth on all links of the path, and finding a minimum cost path amounts to finding a maximum bandwidth path.

However, we also take advantage of the fact that the BF algorithm progresses by increasing hop count, to essentially get for free the hop count of a path as a second optimization criteria.

Specifically, at the kth (hop count) iteration of the algorithm, the maximum bandwidth available to all destinations on a path of no more than k hops is recorded (together with the corresponding routing information). After the algorithm terminates, this

----

3. See Appendix D for a more comprehensive discussion on the aspect of computational complexity.

information enables us to identify for all destinations and bandwidth requirements, the path with the smallest possible number of hops and sufficient bandwidth to accommodate the new request. Furthermore, this path is also the one with the maximal available bandwidth among all the feasible paths with this minimum number of hops. This is because for any hop count, the algorithm always selects the one with maximum available bandwidth.

We now proceed with a more detailed description of the algorithm and the data structure used to record routing information, i.e., the QoS routing table that gets built as the algorithm progresses (pseudo-code for the algorithm can be found in Appendix A). As mentioned before, the algorithm operates on a directed graph consisting only of transit vertices (routers and networks), with stub-networks subsequently added to the path(s) generated by the algorithm. The metric associated with each edge in the graph is the bandwidth available on the corresponding interface. Let us denote by bn;mthe available bandwidth on the edge between vertices n and m. The vertex corresponding to the router where the algorithm is being run, i.e., the computing router, is denoted as the ‘‘source node’’ for the purpose of path selection. The algorithm proceeds to pre-compute paths from this source node to all possible destination networks and for all possible bandwidth values. At each (hop count) iteration, intermediate results are recorded in a QoS routing table, which has the following structure:

The QoS routing table:

- a Kx H matrix, where K is the number of destinations (vertices in the graph) and H is the maximal allowed (or possible) number of hops for a path.

- The (n;h) entry is built during the hth iteration (hop count value) of the algorithm, and consists of two fields:

* bw: the maximum available bandwidth, on a path of at most h hops between the source node (router) and destination node n;

* neighbor: this is the routing information associated with the h (or less) hops path to destination node n, whose available bandwidth is bw. In the context of hop-by-hop path selection (4), the neighbor information is simply the identity of the node adjacent to the source node on that

----
4. Modifications to support source routing are discussed in Appendix C.

path. As a rule, the ‘‘neighbor’’ node must be a router and not a network (see Appendix B), the only exception being the case where the network is the destination node (and the selected path is the single edge interconnecting the source to it).

Next, we provide additional details on the operation of the algorithm and how the entries in the routing table are being updated as the algorithm proceeds. For simplicity, we first describe the simpler case where all edges count as ‘‘hops’’, and later explain how zero-hop edges (see Appendix B for further details) are handled.

When the algorithm is invoked, the routing table is first initialized with all bw fields set to 0 and neighbor fields cleared. Next, the entries in the first column (which corresponds to one-hop paths) of the neighbors of the computing router are modified in the following way: the bw field is set to the value of the available bandwidth on the direct edge from the source. The neighbor field is set to the identity of the neighbor of the computing router, i.e., the next router on the selected path.

Afterwards, the algorithm iterates for at most H iterations (considering the above initial iteration as the first). H can be either the maximum possible hop count of any path, i.e., an implicit value, or it can be set explicitly in order to limit path lengths to some maximum value (5) to better control worst case complexity.

At iteration h, we first copy column h - 1 into column h. In addition, the algorithm keeps a list of nodes that changed their bw value in the previous iteration, i.e., during the h- 1-st iteration. The algorithm then looks at each link (n;m) and checks the maximal available bandwidth on an h-hop path to node m whose final hop is that link. This amounts to taking the minimum between the bw field in entry (n;h -1) and the link metric value bn;m kept in the topology database. If this value is higher than the present value of the bw field in entry (m;h), then a better (larger bw value) path has been found for destination m and with h hops. The bw field of entry (m;h) is then updated to reflect this new value. In the case of hop-by-hop routing, the neighbor field of entry (m;h) is set to the same value as in entry (n;h - 1). This records the identity of the first hop (next hop from the source) on the best path identified thus far for destination m and with h (or less) hops.

----

5. This maximum value should be larger than the length of the minimum hop-count path to any node in the graph.

We conclude by outlining how zero-hop edges are handled. At each iteration h (starting with the first), whenever an entry (m;h) is modified, it is checked whether there are zero-cost edges (m;k) emerging from node m, which is the case when m is a transit network (see Appendix B). In that case, we attempt to improve the entry of node k that corresponds to the h-th hop, i.e., entry (k;h) (rather than entry (k;h + 1)), since the edge (m;k) should not count as an additional hop. As with the regular operation of the algorithm, this amounts to taking the minimum between the bw field in entry (m;h) and the link metric value bm;kkept in the topology database. If this value is higher than the present value of the bw field in entry (k;h), then the bw field of entry (k;h) is updated to this new value. In the case of hop-by-hop routing, the neighbor field of entry (k;h) is set, as usual, to the same value as in entry (m;h) (which is also the value in entry (n;h- 1)).

Note that while for simplicity of the exposition, the issue of equal cost, i.e., same hop count and available bandwidth, is not detailed in the above description, it is straightforward to add this support. It only requires that the neighbor field be expanded to record the list of next (previous) hops, when multiple equal cost paths are present.

Addition of Stub Networks

As was mentioned earlier, the path selection algorithm is run on a graph whose vertices consist only of routers and transit networks and not stub networks. This is intended to keep the computational complexity as low as possible as stub networks can be added relatively easily through a post-processing step. This second processing step is similar to the one used in the current OSPF routing table calculation [Moy94][Section 16, p. 148], with some differences to account for the QoS nature of routes.

Specifically, after the QoS routing table has been constructed, all the router vertices are again considered. For each router, stub networks whose link appears in the router's links advertisement will be processed to determine QoS routes available to them. The QoS routing information for a stub network is similar to that of routers and transit networks and consists of an extension to the QoS routing table in the form of an additional row. The columns in that new row again correspond to paths of different hop counts, and contain both bandwidth and next hop information. We also assume that an available bandwidth value has been advertised for the stub network. As before, how this value is determined is beyond the scope of this document. The QoS routes for a stub network S are constructed as follows:

Each entry in the row corresponding to stub network S has its bws field initialized to zero and its neighbor set to null. When stub network S is found in the link advertisement of router V, the value bw(S,h) in the hth column of the row corresponding to stub network S is updated as follows:

bw(S,h) = min (bw(S,h) ; min (bw(V,h) , b(V,S))),

where bw(V,h) is the bandwidth value of the corresponding column for the QoS routing table row associated with router V, i.e., the bandwidth available on an h hop path to V, and b(V,S) is the advertised available bandwidth on the link from V to S. The above expression essentially states that the bandwidth of a h hop paths to stub network S is updated using a path through router V, only if the minimum of the bandwidth of the h hop path to V and the bandwidth on the link between V and S is larger than the current value.

Update of the neighbor field proceeds similarly whenever the bandwidth of a path through V is found to be larger than or equal to the current value. If it is larger, then the neighbor field of V in the corresponding column replaces the current neighbor field of S. If it is equal, then the neighbor field of V in the corresponding column is concatenated with the existing field for S, i.e., the current set of neighbors for V is added to the current set of neighbors for S.

2.3.2. Algorithm for on-demand computation of QoS paths

In the previous section, we described an algorithm that allows pre-computation of QoS routes. However, it may be feasible in some instances, e.g., limited number of requests for QoS routes, to instead perform such computations on-demand, i.e., upon receipt of a request for a QoS route. The benefit of such an approach is that depending on how often recomputation of pre-computed routes is triggered, on-demand route computation can yield better routes by using the most recent link metrics available. Another benefit of on-demand path computation is the associated storage saving, i.e., there is no need for a QoS routing table. This is essentially the standard trade-off between memory and processing cycles.

In this section, we briefly describe how a standard Dijkstra algorithm can, for a given destination and bandwidth requirement, generate a minimum hop path that can accommodate the required bandwidth and also has maximum bandwidth. Because the Dijkstra algorithm is already used in the current OSPF route computation, only differences from the standard algorithm are described. Also, while for simplicity we do not consider here zero-hop edges (see Appendix B), the modification required for supporting them is straightforward.

The algorithm essentially performs a minimum hop path computation, on a graph from which all edges, whose available bandwidth is less than that requested by the flow triggering the computation, have been removed. This can be performed either through a pre-processing step, or while running the algorithm by checking the available bandwidth value for any edge that is being considered. Another modification to a standard Dijkstra based minimum hop count path computation, is that the list of equal cost next (previous) hops which is maintained as the algorithm proceeds, needs to be sorted according to available bandwidth. This is to allow selection of the minimum hop path with maximum available bandwidth. Alternatively, the algorithm could also be modified to, at each step, only keep among equal hop count paths the one with maximum available bandwidth. This would essentially amount to considering a cost that is function of both hop count and available bandwidth.

2.3.3. Algorithm for approximate pre-computed QoS paths

This section outlines a Dijkstra-based algorithm that allows pre-computation of QoS routes for all destinations and bandwidth values. The benefit of using a Dijkstra-based algorithm is a greater synergy with existing OSPF implementations. The cost is, however, a loss in the ‘‘accuracy’’ of the pre-computed paths, i.e., the paths being generated may be of a larger hop count than needed. This loss in accuracy comes from the need to rely on quantized bandwidth values, that are used when computing a minimum hop count path. In other words, the range of possible bandwidth values that can be requested by a new flow is mapped into a fixed number of quantized values, and minimum hop count paths are generated for each quantized value. For example, one could assume that bandwidth values are quantized as low, medium, and high, and minimum hop count paths are computed for each of these three values. A new flow is then assigned to the minimum hop path that can carry the smallest quantized value, i.e., low, medium, or high, larger than or equal to what it requested.

Here too, we discuss the elementary case where all edges count as ‘‘hops’’, and note that the modification required for supporting zero-hop edges is straightforward.

As with the BF algorithm, the algorithm relies on a routing table that gets built as the algorithm progresses. The structure of the routing table is as follows:

The QoS routing table:

- a K x Q matrix, where K is the number of vertices and Q is the number of quantized bandwidth values.

- The (n;q) entry contains information that identifies the minimum hop count path to destination n, that is capable of accommodating a bandwidth request of at least bw[q] (the qth quantized bandwidth value). It consists of two fields:

  * hops: the minimal number of hops on a path between the source node and destination n, that can accommodate a request of at least bw[q] units of bandwidth.

  * neighbor: this is the routing information associated with the minimum hop count path to destination node n, whose available bandwidth is at least bw[q]. With a hop-by-hop routing approach, the neighbor information is simply the identity of the node adjacent to the source node on that path.

The algorithm operates again on a directed graph where vertices correspond to routers and transit networks. The metric associated with each edge in the graph is as before the bandwidth available on the corresponding interface, where bn;mis the available bandwidth on the edge between vertices n and m. The vertex corresponding to the router where the algorithm is being run is selected as the source node for the purpose of path selection, and the algorithm proceeds to compute paths to all other nodes (destinations).

Starting with the highest quantization index, Q, the algorithm considers the indices consecutively, in decreasing order. For each index q, the algorithm deletes from the original network topology all links (n;m) for which bn;m< bw[q], and then runs on the remaining topology a Dijkstra-based minimum hop count algorithm (6) between the source node and all other nodes (vertices) in the graph. Note that as with the Dijkstra used for on-demand path computation, the elimination of links such that $b_{n;m} < bw[q]$ could also be performed while running the algorithm.

After the algorithm terminates, the q-th column in the routing table is updated. This amounts to recording in the hops field the hop

----

6. Note that a Breadth-First-Search (BFS) algorithm [CLR90] could also be used. It has a lower complexity, but would not allow reuse of existing code in an OSPF implementation.

count value of the path that was generated by the algorithm, and by updating the neighbor field. As before, the update of the neighbor field depends on the scope of the path computation. In the case of a hop-by-hop routing decision, the neighbor field is set to the identity of the node adjacent to the source node (next hop) on the path returned by the algorithm. However, note that in order to ensure that the path with the maximal available bandwidth is always chosen among all minimum hop paths that can accommodate a given quantized bandwidth, a slightly different update mechanism of the neighbor field needs to be used in some instances. Specifically, when for a given row, i.e., destination node n, the value of the hops field in column q is found equal to the value in column q + 1 (here we assume q < Q), i.e., paths that can accommodate bw[q] and bw[q+ 1] have the same hop count, then the algorithm copies the value of the neighbor field from entry (n;q+ 1) into that of entry (n;q).

Addition of Stub Networks

This proceeds in a manner very similar to that of Section 2.3.1, except for some minor variations reflecting differences in the structure of the QoS routing table. Specifically, the columns of the QoS routing table now correspond to quantized bandwidth values, and the bw field of a column entry has been replaced by a hops field. Hence, the QoS routes for a stub network S are constructed as follows:

Each entry in the row corresponding to stub network S has its hops field initialized to zero and its neighbor set to null. When stub network S is found in the link advertisement of router V, the value hops(S,q) in the qth column of the row corresponding to stub network S is updated as follows:

hops(S,q) = hops(V,q) IF (hops(V,q) <= hops(S,q) AND b(V ,S) >= bw[q]), where bw[q] is the qth quantized bandwidth value, and b(V,S) is the advertised available bandwidth on the link from V to S. The above expression essentially states that the hop count of the path to stub network S capable of supporting a bandwidth allocation of bw[q], is updated using a path through router V, only if the corresponding path through V has fewer hops than the current one, and the bandwidth on the link between V and S is larger than bw[q].

Update of the neighbor field proceeds similarly whenever the path through router V capable of supporting a bandwidth allocation of bw[q], is found to yield a hop count smaller than or equal to the current value. If it is smaller, then the neighbor field of V in the corresponding column replaces the current neighbor field of S.

If it is equal, then the neighbor field of V in the corresponding column is concatenated with the existing field for S, i.e., the current set of neighbors for V is added to the current set of neighbors for S.

2.4. Extracting Forwarding Information from Routing Table

When the QoS paths are precomputed, the forwarding information for a flow with given destination and bandwidth requirement needs to be extracted from the routing table. The case of hop-by-hop routing is much simpler compared to source routing. This is because, only the next hop needs to be returned instead of a complete source route.

Specifically, assume a new request to destination, say, d, and with bandwidth requirements B. The index of the destination vertex identifies the row in the QoS routing table that needs to be checked to generate a path. How the row is searched to identify a suitable path depends on which algorithm was used to construct the QoS routing table. If the Bellman-Ford algorithm of Section 2.3.1 is used, the search proceeds by increasing index (hop) count until an entry is found, say at hop count or column index of h, with a value of the bw field which is greater than or larger than B. This entry points to the initial information identifying the selected path. If the Dijkstra algorithm of Section 2.3.3 is used, the first quantized value bBsuch that Bb   B is first identified, and the associated column then determines the first entry in the QoS routing table that identifies the selected path.

The next hop information is then directly retrieved from the neighbor information of the first entry pointed to in the QoS routing table.

The case of source routing is discussed in Appendix C.

3. Establishment and Maintenance of QoS Routes

In this section, we briefly review issues related to how QoS paths are established and maintained. For both, there are functional and protocol aspects that need to be covered.

The goal of QoS routing is to select paths for flows with QoS requirements, in such a manner as to increase the likelihood that the network will indeed be capable of satisfying them. The use of QoS routing algorithms such as the ones described in this document have a number of implications above and beyond what is required when using standard routing algorithms.

First, a specific mechanism needs to be used to identify flows with QoS requirements, so that they can be assigned to the corresponding QoS routing algorithm. In this section, we assume that the RSVP protocol [RZB+96] is used for that purpose. Specifically, RSVP PATH messages serve as the trigger to query QoS routing. Second, because of variations in the availability of resources in the network, routes between the same source and destination and for the same QoS, may often differ depending on when the request is made. However, it is important to ensure that such changes are not always reflected on existing paths. This is to avoid potential oscillations between paths and limit changes to cases where the initial selection turns out to be inadequate.

As a result, some state information needs to be associated with a QoS route to determines its current validity, i.e., should the QoS routing algorithm be queried to generate a new and potentially better route, or does the current one remain adequate. We say that a path is ‘‘pinned’’ when its state specifies that QoS routing need not be queried anew, while a path is considered ‘‘unpinned’’ otherwise. The main issue is then to define how, when, and where route pinning and unpinning is to take place. In our context, where the RSVP protocol is used as the vehicle to request QoS routes, we also want this process to be as synergetic as possible with the existing RSVP state management. In particular, our goal is to support pinning and unpinning of routes in a manner consistent with RSVP soft states while requiring minimal changes to the RSVP processing rules.

It should be noted that some changes are unavoidable, especially to the interface between RSVP and routing. Specifically, QoS routing requires, in addition to the current source and destination addresses, at a minimum, knowledge of the flow’s traffic characteristics (TSpec), and possibly also service types (as per the information in the Adspec), PHOP, IP TTL value, etc. In this document, we assume that the information provided by RSVP to QoS routing includes at least the sender TSpec in addition to the source and destination addresses. While such changes seem unavoidable, our goal is again to keep them as small as possible and also to avoid any change to the existing RSVP message format.

Specifically, we assume interactions between RSVP and routing that are very similar to what is currently defined. During the processing of RSVP PATH messages, RSVP queries (QoS) routing to obtain the next hop(s) for forwarding the PATH message. The PATH message is then forwarded on the interface(s) returned by (QoS) routing. As mentioned before, the sender TSpec is part of the information made available to routing, and a QoS routing algorithm selects the next hop along a path to the destination that is most likely to support the flow specified by the sender TSpec. Thus, forwarding the PATH message along this next hop should improve the likelihood of the reservation request succeeding during RESV message processing. In particular, RESV messages, if any, propagate as before in the reverse direction of the PATH messages and attempt to reserve the required resources along the path delineated by PATH messages.

In the context of the above hop-by-hop routing, there are two main issues associated with the pinning and unpinning of QoS paths.

1. Detection of loops that may be caused by inconsistencies in the QoS routes returned by QoS routing at different nodes. Such inconsistencies are typically transient, but it is important that the pinning of a path does not result in the formation of permanent loops .

2. Query of a new QoS route in case of failures. An example of such failures is a reservation failure because the RESV message arrived substantially later after the QoS route was initially selected. Other failures include the usual link failures, and in general it is important to allow QoS routing to become aware of the failure and select a better route if one is available.

The QoS path management scheme proposed here addresses the above two issues based on the following two design rules:

1. What is pinned is a ‘‘path’’ taken by a specific RSVP flow and not a ‘‘route’’ as computed by the routing algorithm. Hence pinning and unpinning are RSVP domain operations, and as a result completely independent of the specific QoS routing algorithm being used.

2. Path pinning and unpinning is kept ‘‘soft’’ by tying it to the existing RSVP soft state mechanism. In other words, we rely on existing RSVP refreshes and time-out mechanisms to detect the state changes that trigger pinning and unpinning of paths. In

addition, such changes are triggered only on the basis of current RSVP state information.

Appendix F specifies how the above two rules translate into the conditions and processing rules for pinning and unpinning paths, that address the problem of loops while also enabling reactions to failures.

4. OSPF Protocol Enhancements

As stated above, a goal of this work is to limit the additions to the existing OSPF V2 protocol, while still providing the required level of support for QoS based routing. To this end, all of the existing OSPF mechanisms, data structures, advertisements, and data formats remain in place. The purpose of this section of the document is to list the enhancements to the OSPF protocol to support QoS as outlined in the previous sections.

4.1. QoS -- Optional Capabilities

The OSPF Options field is present in OSPF Hello packets, Database Description packets and all LSAs. The Options field enables OSPF routers to support (or not support) optional capabilities, and to communicate their capability level to other OSPF routers. Through this mechanism, routers of differing capabilities can be mixed with an OSPF routing domain.

This document describes one of the Option bits: the Q-bit (for QoS capability). The Q-bit is set in router, network, and summary links advertisements, and is used to identify routers and networks that support (or do not support) QoS routing as defined in this document. When the Q-bit is set in a router or summary links link state advertisement, it means that there are QoS fields to process in the packet. When the Q-bit is set in a network link state advertisement it means that the network described in the advertisement is QoS capable.

```
          -------------------------------
         | * | Q | DC| EA|N/P| MC| E | T |
          -------------------------------
```

The Q-bit is in its semantics very similar to the T-bit and QoS capability can be viewed just as an extension of TOS-capabilities. This similarity will be reinforced by the encoding introduced further in this section. However, one important difference remains between TOS and QOS capabilities. A router that does not have TOS capabilities forwards packets along TOS 0 paths. Additionally, TOS metrics that are not advertised are assumed to have the same cost as TOS 0. In case of QoS resources such as bandwidth, this does not make sense and could possibly introduce routing loops due to different criteria for which the routes are optimized. Therefore, if no route exists through routers all supporting QoS or any of those does not provide the necessary metric for the resource considered, communication with the requested quality of service is not possible. Since each of the router link directions is described in an independent LSA, even a uni-directional failure to communicate is possible since the availability of a resource metric in one direction does not guarantee its accessibility for the other one.

4.2. Encoding Resources as Extended TOS

Introduction of QoS should ideally not influence the compatibility with existing OSPFv2 routers. To achieve this goal, necessary extensions in packet formats must be defined in a way that either is understood by OSPFv2 routers, ignored or in the worst case ‘‘gracefully’’ misinterpreted. Encoding of QoS metrics in the TOS field which fortunately enough is longer in OSPF packets than officially defined in [Alm92], allows us to mimic the new facility as extended TOS capability. OSPFv2 routers will either disregard these definitions or consider those unspecified. Specific precautions are taken to prevent careless OSPF implementations from influencing traditional TOS routing when misinterpreting the extension introduced.

For QoS resources, 32 combinations are available through the use of the fifth bit in TOS fields contained in different LSAs. Since [Alm92] defines TOS as being four bits long, this definition never conflicts with existing values. Additionally, to prevent naive implementations that do not take all bits of the TOS field in OSPF packets into considerations, the definitions of the ‘QoS encodings’ is aligned in their semantics with the TOS encoding. Only bandwidth and delay are specified as of today and their values map onto ‘maximize throughput’ and ‘minimize delay’ if the uppermost bit is not taken into account. Accordingly, link reliability and jitter could be defined later if necessary.

OSPF encoding	RFC 1349 TOS values
0	0000 normal service

2	0001	minimize monetary cost
4	0010	maximize reliability
6	0011	
8	0100	maximize throughput
10	0101	
12	0110	
14	0111	
16	1000	minimize delay
18	1001	
20	1010	
22	1011	
24	1100	
26	1101	

28	1110	
30	1111	

OSPF encoding	‘QoS encoding values’	
32	10000	
34	10001	
36	10010	
38	10011	
40	10100	bandwidth
42	10101	
44	10110	
46	10111	
48	11000	delay
50	11001	
52	11010	
54	11011	
56	11100	
58	11101	
60	11110	
62	11111	

Representing TOS and QoS in OSPF.

4.2.1. Encoding bandwidth resource

Given the fact that the actual metric field in OSPF packets only provides 16 bits to encode the value used and that links supporting bandwidth ranging into Gbits/s are becoming reality, linear representation of the available resource metric is not feasible. The solution is exponential encoding using appropriately chosen implicit base value and number bits for encoding mantissa and the exponent.

Detailed considerations leading to the solution described are not presented here but can be found in [Prz95].

Given a base of 8, the 3 most significant bits should be reserved for the exponent part and the remaining 13 for the mantissa. This allows a simple comparison for two numbers encoded in this form, which is often useful during implementation.

The following table shows bandwidth ranges covered when using different exponents and the granularity of possible reservations.

```
     exponent
```

value x	range (2^13-1)*8^x	step 8^x
0	8,191	1
1	65,528	8
2	524,224	64
3	4,193,792	512
4	33,550,336	4,096
5	268,402,688	32,768
6	2,147,221,504	262,144
7	17,177,772,032	2,097,152

Ranges of Exponent Values for 13 bits,
base 8 Encoding, in Bytes/s

The bandwidth encoding rule may be summarized as:

1. represent available bandwidth in 16 bit field as a 3 bit exponent (with assumed base of 8) followed by a 13 bit mantissa as shown below

```
         0       8       16
         |       |       |
         -----------------
        |EXP| MANT        |
         -----------------
```

2. advertise 2’s complement of the above representation.

Thus, the above encoding advertises a numeric value that is

216- 1-(exponential encoding of the available bandwidth):

This has the property of advertising a higher numeric value for lower available bandwidth, a notion that is consistent with that of cost.

Although it may seem slightly pedantic to insist on the property that less bandwidth is expressed higher values, it has, besides consistency, a robustness aspect in it. A router with a poor OSPF implementation could misuse or misunderstand bandwidth metric as normal administrative cost provided to it and compute spanning trees with a ‘‘normal’’ Dijkstra. The effect of a heavily congested link advertising numerically very low cost could be disastrous in such a scenario. It would raise the link’s attractiveness for future traffic instead of lowering it. Evidence that such considerations are not speculative, but similar scenarios have been encountered, can be found in [Tan89].

Concluding with an example, assume a link with bandwidth of 8 Gbits/s = 10243Bytes/s, its encoding would consist of an exponent value of 6 since 10243=4; 096*86, which would then have a granularity of 86 260 kBytes/s. The associated binary representation would then be %(110) 0 1000 0000 0000% or 53,248 (7). The bandwidth cost (advertised value) of this link when it is idle, is then the 2's complement of the above binary representation, i.e., %(001) 1 0111 1111 1111% which corresponds to a decimal value of (216- 1)- 53;248 = 12;287. Assuming now a current reservation level of of 6;400 Mbits/s = 200   *   10242, there remains 1;600 Mbits/s of available bandwidth on the link. The encoding of this available bandwidth of 1'600 Mbits/s is 6;400 * 85, which corresponds to a granularity of 85   30 kBytes/s, and has a binary representation of %(101) 1 1001 0000 0000% or decimal value of 47,360. The advertised cost of the link with this load level, is then %(010) 0 0110 1111 1111%, or (216- 1)- 47;360 =18;175.

Note that the cost function behaves as it should, i.e., the less bandwidth is available on a link, the higher the cost and the less attractive the link becomes. Furthermore, the targeted property of better granularity for links with less bandwidth available is also achieved. It should, however, be pointed out that the numbers given in the above examples match exactly the resolution of the proposed encoding, which is of course not always the case in practice. This leaves open the question of how to encode available bandwidth values when they do not exactly match the encoding. The standard practice is to round it to the closest number. Because we are ultimately interested in the cost value for which it may be better to be pessimistic than optimistic, we choose to round costs up and, therefore, bandwidth down.

4.2.2. Encoding Delay

Delay is encoded in microseconds using the same exponential method as described for bandwidth except that the base is defined to be 4 instead of 8. Therefore the maximum delay that can be expressed is (213 -1) *47 134 seconds.

----
7. exponent in parenthesis

4.3. Packet Formats

Given the extended TOS notation to account for QoS metrics, no changes in packet formats are necessary except for the introduction of Q-bit in the options field. Routers not understanding the Q-bit should either not consider the QoS metrics distributed or consider those as ‘unknown’ TOS.

4.4. Calculating the Inter-area Routes

This document proposes a very limited use of OSPF areas, that is, it is assumed that summary links advertisements exist for all networks in the area. This document does not discuss the problem of providing support for area address ranges and QoS metric aggregation. This is left for further studies.

4.5. Open Issues

Support for AS External Links, Virtual Links, and incremental updates for summary link advertisements are not addressed in this document and are left for further study. For Virtual Links that do exist, it is assumed for path selection that this links are non-QoS capable even if the router advertises QoS capability. Also, as stated earlier, this document does not address the issue of non-QoS routers within a QoS domain.

Acknowledgments

We would like to thank the many people who have helped shape various aspects of this document and the approaches it describes, either through discussions or explicit suggestions. In particular, we would like to acknowledge the help and inputs of John Moy, Sanjay Kamat, Dilip Kandlur, and Dimitrios Pendarakis.

APPENDICES

A. Pseudocode for BF Algorithm

Note: The pseudocode below assumes a hop-by-hop forwarding approach in updating the neighbor field. The modifications needed to support a source routed approach are straightforward. The pseudocode also does not handle equal cost multi-paths for simplicity, but the modification needed to add this support are straightforward.

```
Input:
  V = set of vertices, labeled by integers 1 to N.
  L = set of edges, labeled by ordered pairs (n,m) of vertex labels.
  s = source vertex (at which the algorithm is executed).
  For all edges (n,m) in L:
    * b(n,m) = available bandwidth (according to last received update)
    on interface associated with the edge between vertices n and m.
    * If(n,m) outgoing interface corresponding to edge (n,m) when n is
      a router.
  H = maximum hop-count (at most the graph diameter).

Type:
  tab_entry: record
                 bw = integer,
                 neighbor = integer 1..N.

Variables:
  TT[1..N, 1..H]: topology table, whose (n,h) entry is a tab_entry record, such
                  that:
                    TT[n,h].bw is the maximum available bandwidth (as known
                      thus far) on a path of at most h hops between
                      vertices s and n,
                    TT[n,h].neighbor is the first hop on that path (a neighbor
                      of s). It is either a router or the destination n.
  S_prev: list of vertices that changed a bw value in the TT table
          in the previous iteration.
  S_new: list of vertices that changed a bw value (in the TT table etc.\ ) in
t
he
         current iteration.

The Algorithm:

begin;

  for n:=1 to N do  /* initialization */
  begin;
```

```
    TT[n,0].bw := 0;
    TT[n,0].neighbor := null
    TT[n,1].bw := 0;
    TT[n,1].neighbor := null
  end;
  TT[s,0].bw := infinity;

  reset S_prev;
  for all neighbors n of s do
  begin;
    TT[n,1].bw := max( TT[n,1].bw, b[s,n]);
    if (TT[n,1].bw = b[s,n]) then TT[n,1].neighbor := If(s,n);
             /* need to make sure we are picking the maximum */
             /* bandwidth path for routers that can be reached */
             /* through both networks and point-to-point links */
    if ( (n is a router) and ({n} not in S_prev) )
       then  S_prev :=  S_prev union {n}
             /* only routers are added to S_prev, but we need to */
             /* check they are not already included in S_prev */
    else     /* n is a network: */
             /* proceed with network--router edges, without */
             /* counting another hop */
    for all (n,k) in L, k <> s, do
             /* i.e., for all other neighboring routers of n */
    begin;
      TT[k,1].bw := max( min( TT[n,1].bw, b[n,k]), TT[k,1].bw );
             /* In case k could be reached through another path */
             /* (a point-to-point link or another network) with */
             /* more bandwidth, we do not want to update TT[k,1].bw */
      if (min( TT[n,1].bw, b[n,k]) = TT[k,1].bw )
             /* If we have updated TT[k,1].bw by going through */
             /* network n  */
         then TT[k,1].neighbor := If(s,n);
             /* neighbor is interface to network n */
      if ( {k} not in S_prev) then S_prev :=  S_prev union {k}
```

```
                /* only routers are added to S_prev, but we again need */
                /* to check they are not already included in S_prev */
      end
    end;


    for h:=2 to H do   /* consider all possible number of hops */
    begin;
      reset S_new;
      for all vertices m in V do
      begin;
        TT[m,h].bw := TT[m,h-1].bw;
        TT[m,h].neighbor := TT[m,h-1].neighbor
      end;
      for all vertices n in S_prev do
               /* as shall become evident, S_prev contains only routers */
      begin;
        for all edges (n,m) in L do
        if min( TT[n,h-1].bw, b[n,m]) > TT[m,h].bw then
        begin;
          TT[m,h].bw := min( TT[n,h-1].bw, b[n,m]);
          TT[m,h].neighbor := TT[n,h-1].neighbor;
          if m is a router then S_new :=  S_new union {m}
               /* only routers are added to S_new */
          else /* m is a network: */
               /* proceed with network--router edges, without counting them as
  */
               /* another hop */
          for all (m,k) in L, k <> n,
               /* i.e., for all other neighboring routers of m */
          if min( TT[m,h].bw, b[m,k]) > TT[k,h].bw then
          begin;
               /* Note: still counting it as the h-th hop, as (m,k) is a */
               /* network--router edge */
            TT[k,h].bw := min( TT[m,h].bw, b[m,k]);
            TT[k,h].neighbor := TT[m,h].neighbor;
            S_new :=  S_new union {k}
               /* only routers are added to S_new */
          end
        end
      end;
      S_prev := S_new;
              /* the two lists can be handled by a toggle bit */
      if S_prev=null then h=H+1   /* if no changes then exit */
    end;

  end.
```

B. Zero-Hop Edges

The need to handle zero-hop edges is due to the potential presence of multiple access networks, e.g., T/R, E/N, or ATM, to interconnect routers. Such entities are also represented by means of a vertex in the current OSPF operation. Clearly, in such cases a network connecting two routers should be considered as a single hop path rather than a two hop path. For example, consider three routers A, B, and C connected over an Ethernet network N, which the OSPF topology represents as:

In the above example, although there are directed edges in both directions, an edge from the network to any of the three routers

```
A----N----B
     |
     |
     C
```

must have zero ‘‘cost’’, so that it is not counted twice. It should be noted that when considering such environments in the context of QoS routing, it is assumed that some entity is responsible for determining the ‘‘available bandwidth’’ on the network. The specification of the operation of such an entity is beyond the scope of this document.

C. Source Routing Support

As mentioned before, the scope of the path selection process can range from simply returning the next hop on the QoS path selected for the flow, to specifying the complete path that was computed, i.e., a source route. Obviously, the information being returned by the path selection algorithm differs in these two cases, and constructing it imposes different requirements on the path computation algorithm and the data structures it relies on. While the presentation of the path computation algorithms focused on the hop-by-hop routing approach, the same algorithms can be applied to generate source routes with minor modifications. These modifications and how they facilitate constructing source routes are discussed next.

The general approach to facilitate construction of source routes is to update the neighbor field differently from the way it is done for hop-by-hop routing as described in Section 2. Recall that in the path computation algorithms the neighbor field is updated to reflect the identity of the node adjacent to the source node on the partial path computed. This facilitates returning the next hop at the source for the specific path. In the context of source routing, the neighbor information is updated to reflect the identity of the previous router on the path.

With this change, the basic approach used to extract the complete list of verti ces on a path from the neighbor information in the QoS routing table is to proceed recursively from the destination to the origin vertex. The path is extracted by stepping through the precomputed QoS routing table from vertex to vertex, and identifying at each step the corresponding neighbor (precursor) information. Once the source router is reached, the concatenation of all the neighbor fields that have been extracted forms the desired source route. This applies to the source-routed versions of both algorithms of Sections 2.3.1 and 2.3.3.

Specifically, assume a new request to destination, say, d, and with bandwidth requirements B. The index of the destination vertex identifies the row in the QoS routing table that needs to be checked to generate a path. How the row is searched to identify a suitable path depends on which algorithm was used to construct the QoS routing table. If the Bellman-Ford algorithm of Section 2.3.1 is used, the search proceeds by increasing index (hop) count until an entry is found, say at hop count or column index of h, with a value of the bw field which is greater than or larger than B. This entry points to the initial information identifying the selected path. If the Dijkstra algorithm of Section 2.3.3 is used, the first quantized value bBsuch that Bb   B is first identified, and the associated column then determines the first entry in the QoS routing table that identifies the selected path.

Once this first entry has been identified, reconstruction of the complete list of vertices on the path proceeds similarly, whether the table was built using the algorithm of Sections 2.3.1 or 2.3.3. Specifically, in both cases, the neighbor field in each entry points to the previous node on the path from the source node and with the same bandwidth capabilities as those associated with the current entry. The complete path is, therefore, reconstructed by following the pointers provided by the neighbor field of successive entries.

In the case of the Bellman-Ford algorithm of Section 2.3.1, this means moving backwards in the table from column to column, using at each step the row index pointed to by the neighbor field of the entry in the previous column. Each time, the corresponding vertex index specified in the neighbor field is pre-pended to the list of vertices constructed so far. Since we start at column h, the process ends when first column is reached, i.e., after h steps, at which point the list of vertices making up the path has been reconstructed.

In the case of the Dijkstra algorithm of Section 2.3.3, the backtracking process is similar although slightly different because of the different relation between paths and columns in the routing table, i.e., a column now corresponds to a quantized bandwidth value instead of a hop count. The backtracking now proceeds along the column corresponding to the quantized bandwidth value needed to satisfy the bandwidth requirements of the flow. At each step, the vertex index specified in the neighbor field is pre-pended to the list of vertices constructed so far, and is used to identify the next row index to move to. The process ends when an entry is reached whose neighbor field specifies the origin vertex of the flow. Note that since there are as many rows in the table as there are vertices in the graph, i.e., N, it could take up to N steps before the process terminates.

Note that the identification of the first entry in the routing table is identical to what was described for the hop-by-hop routing case. However, as described in this section, the update of the neighbor fields while constructing the QoS routing tables, is being performed differently in the source and hop-by-hop routing cases. Clearly, two different neighbor fields can be kept in each entry and updates to both could certainly be performed jointly, if support for both source routing and hop-by-hop routing is needed.

D. Computational Complexity

One generic aspect of the algorithmic complexity of computing QoS paths is the efficiency of the shortest path algorithm used. Specifically, in this document, we have described approaches based on both Bellman-Ford and Dijkstra shortest paths algorithms. Dijkstra's algorithm has traditionally been considered more efficient for standard shortest path computations because of its lower worst case complexity. However, the answer is not as simple as may appear, and in this section we briefly review a number of considerations, in particular in the context of multi-criteria QoS paths, which indicate that a BF approach may often provide a lower complexity solution.

The asymptotic worst-case complexity of the Dijkstra algorithm is O(NlogN   +   M), where N is the number of vertices in the graph, and M the number of edges. However, this bound is obtained under the assumption of a Fibonnaci heap implementation of the Dijkstra algorithm, which is impractical due to the large constants involved [CLR90]. In practice, the Dijkstra algorithm is typically implemented using binary heaps, for which the asymptotic worst-case bound is O(MlogN).

The asymptotic worst-case bound for the BF algorithm is $O(H \cdot M)$, where M is again the number of edges in the graph, and H, which is the maximum number of iterations of the algorithm, is an upper-bound on the number of hops in a shortest path. Although, theoretically, H can be as large as $N - 1$, in practice it is usually considerably smaller than N. Moreover, in some network scenarios an upper-bound U of small size (i.e., $U \ll N$) is imposed on the allowed number of hops; for example, it might be decided to exclude paths that have more than, say, 16 hops, as part of a call admission scheme. In such cases, the number of iterations of the BF algorithm can be limited to U, thus bounding the number of operations to $O(U \cdot M)$, i.e., effectively to O(M). As a consequence, as noted in [BG92], in practical networking scenarios, the BF algorithm can offer an efficient solution to the shortest path problem, one that often outperforms the Dijkstra algorithm. (8)

In the context of QoS path selection, the potential benefits of the BF algorithm are even stronger. As mentioned before, efficient selection of a suitable path for flows with QoS requirements cannot usually be handled using a single-objective optimization criterion. While multi-objective path selection is known to be an intractable problem [GJ79], the BF algorithm allows us to handle a second objective, namely the hop-count, which is reflective of network resources, at no additional cost in terms of complexity. The Dijkstra algorithm requires some modifications (or approximations, e.g., bandwidth quantization) in order to be able to deal with hop count as a second objective.

Therefore, in the context of a QoS path selection algorithm, where one objective is some QoS-oriented metric, such as available bandwidth, whereas the second is a hop-count metric, a BF-based algorithm provides an efficient scheme for pre-computing paths, i.e., one with a worst case asymptotic complexity of O(H   .   M). Alternatively, if QoS paths are pre-computed using a Dijkstra algorithm with Q quantized bandwidth values, the corresponding worst case asymptotic complexity is O(Q   .   (M logN)). Both approaches provide solutions of comparable orders of complexity, whose exact merits depend on the respective values of H, Q and N. If on-demand computations of QoS paths are practical, then a standard Dijkstra algorithm provides a solution of complexity O(MlogN).

E. Extension: Handling Propagation Delays

In general, the framework proposed for path selection does not allow us to explicitly account for link propagation delays. As mentioned, this aspect is dealt with through a policy mechanism, which for delay-sensitive connections deletes from the topology database links with high propagation delays, such as satellite links. However, it is worth pointing out that a simple extension to the proposed path selection algorithm allows us to directly account for delay in a

----

8. For example, in the experimental comparison reported in [CGR94], the BF algorithm outperformed the Dijkstra algorithm in about one third of the studied types of topology, and in several of the other topologies it outperformed the Dijkstra algorithm for networks of up to about 16,000 nodes. It should be noted that in those experiments no upper bound on the number of hops in a shortest path was set.

number of special cases. We proceed to describe next this extension and the case where it applies.

A common way to map delay guarantees into bandwidth guarantees (e.g., consistent with the schedulers and corresponding delay bounds presented in [GGPS96, PG94]) is according to the following expression:

$$D(p) = A(h(p))=b + sum(l\ in\ p)\ d(l) \qquad (1)$$

where p is the path traversed, D(p) is the guaranteed (upper-bound) end to end delay, h(p) is the number of hops, b is the reserved bandwidth, d(l) is the (fixed) propagation delay of a link l, and A(h) is a parameter that grows with h (a typical value is A(h)= B +h . c, where B is the burst size and c is the maximum packet size).

Since we deal with intra-domain routing, and since links with prohibitively high propagation delays are assumed to be filtered out by means of policy, it can be assumed that typically there is some value d which is a reasonable upper bound on the propagation delays d(l) of all links. Expression (1) then implies that an end to end delay requirement D can be translated into a bandwidth requirement b(h) by the following expression:

$$b(h) = A(h)=(D - h.\ d) \qquad (2)$$

where h is the number of hops on the path established for the connection.

F. QoS Path Establishment and Management with RSVP

In this section, we briefly illustrate the use of the QoS path selection approach described in this document, for unicast RSVP flows. The objective is to path set up QoS paths for RSVP flows and keep them pinned a s long as it is desirable to do so, while requiring minimal changes to RSVP. Clearly, some changes are needed, particularly to RSVP’s interface to routing and its message processing rules. These will be detailed next. In addition, the impact of this path management approach data path is considered and alternative approaches and extensions are discussed.

F.1. RSVP/Routing interface

Currently, RSVP acquires routing entries using its asynchronous query-response interface to routing [Zap96]. Route query is of the form

```
Route_Query( [SrcAddress], DestAddress, Notify_flag )
```

and Routing responds with OutInterface (or OutInterface_list in case of a multicast connection)

In order for RSVP to interact with a QoS routing algorithm, QoS_Route_Query needs to also include (at a minimum) the sender_TSpec, so that it is now of the form

```
Route_Query( [SrcAddress], DestAddress, TSpec, Notify_flag )
```

and again responds with OutInterface (or OutInterface_list in case of a multicast connection).

Another small difference with the current interface is that the Notify_flag should always be set to True. This is because there will be no Route_Query to QoS routing in the case of pinned paths. Hence, it is important that a trigger be provided to unpin the path in case of failure. However, note that QoS routing will only generate an asynchronous Route_Change callback to RSVP in the case of the failure of a local (to the router) link currently used by the QoS path.

F.2. Path Management Rules

The state of a QoS path as maintained by RSVP consists of a flag that is used to indicate whether the path is currently pinned or not. Specifically, a pinned path means that QoS routing need not be queried for a new path (next hop) for forwarding a PATH refresh. The rules for pinning and unpinning routes are as follows:

1. Routes get pinned during processing of PATH messages.

2. Routes get unpinned when

   (a) corresponding path states are removed (time-out or PATH _TEAR),

   (b) some of the parameters received in PATH messages change,

   (c) a local admission control failure error is detected after receiving a RESV message,

(d) a PATH _ERR with a specific error code is received, or

(e) failure notification of a local link belonging to the path is received.

Minor changes to RSVP message processing rules are adequate to handle pinning and unpinning of paths as needed. These specific changes are described below.

PATH message processing:

When receiving the first PATH message, RSVP determines that no PATH state exists for the flow. It then queries QoS routing to obtain the next hop along the best available path. This next hop is stored as part of the PATH state with its pinned flag set.

Upon receiving a PATH refresh, RSVP checks for changes in PATH state that are of relevance to QoS routing. In particular, it checks for changes in PHOP and the IP TTL value. If there are no changes and the current next hop is indicated as pinned, it will be used to forward the next PATH refresh. If the PATH state has changed or the current next hop is marked as unpinned, RSVP queries QoS routing again to obtain (and pin) a new next hop that is to be used when forwarding the next PATH refresh. Similarly, at the time when a PATH refresh is to be sent, RSVP checks if the current next hop is pinned or not. If it is, it is used to forward the PATH refresh. Otherwise, QoS routing is again queried to obtain (and pin) a new next hop.

The unpinning of the path upon detecting changes in either the PHOP or the IP TTL value of an incoming PATH message is used to ensure that transient loops caused by inconsistent routing information are eventually cleared [GKH97].

PATH_TEAR message processing:

Processing is similar to what is currently done. PATH and RESV states are removed.

RESV message processing:

The only change needed is for the case when the resource reservation attempt fails. As currently specified, a RESV_ERR message with "admission control failure" error code is still sent downstream in such instances. However, some additional processing is needed in order to enable selection of a better path in case one exists. This starts with the unpinning of the current next hop, and then proceeds in either one of two ways: attempt *local* repair of the QoS path or not.

In case local repair is attempted, RSVP queries again its local QoS routing table. If a different next hop is returned, i.e., the reservation may now succeed, then local repair is attempted by pinning the new hop and sending a PATH message along the new route. If the same next hop is returned, then local repair has failed. In this case or when local repair is not attempted, the current next hop is then unpinned in the PATH state (but kept). Furthermore, a PATH _ERR message is sent upstream with a new QoS_Path_Failure Error Code (the exact code point is tbd) and an associated Error Value specifying that the type of error was "Requested QoS unavailable" (the specific format of the Error Value field is tbd). As described below, the receipt of a PATH _ERR message with the QoS_Path_Failure Error Code triggers unpinning of the next hop information at upstream router. This ensures that QoS routing will be queried at the time of the next PATH refresh, so that a better path, if one exists, can be identified.

Route_Change notification processing:

A Route_Change notification is triggered when QoS routing detects that a local link currently used by a QoS path failed. Upon receiving such a notification, RSVP immediately unpins the current next hop. As in the case of reservation failure, RSVP can then first attempt local repair, i.e., query QoS routing for a new next hop. If a new next hop is returned by QoS routing, RSVP uses it to replace the previous next hop, marks it as pinned, and forwards a PATH message towards the new next hop. If QoS routing responds that no path to the destination is available or if local repair is not attempted, RSVP sends upstream a PATH _ERR message with the

QoS_Path_Failure Error Code and an Error Value specifying "Link failure".

PATH_ERR message processing:

The only modification is, as mentioned above, to recognize the new QoS_Path_failure Error Code and unpin the associated next hop. This forces a fresh QoS route query during the processing of the next PATH refresh.

RESV_ERR message processing:

There are no changes to RESV_ERR processing.

F.3. Impact of QoS Routing on the Data Path

The use of QoS routing only affects the choice of a data path and not how the actual forwarding of data packet takes place. Nevertheless, there is an important aspect that needs to be noted. Specifically, while PATH messages are immediately forwarded onto the next hop returned by QoS routing, the same need not apply to data packets. This is because of the potential for transient loops in QoS paths. Forwarding PATH messages on a QoS path that may contain loops has minimal impact on the routers and is actually useful to detect and eliminate loops (more on this below). However, depending on how fast loops can be resolved, forwarding data packets on a QoS path may be best deferred until the absence of a loop has been verified.

As a result, it is proposed that modification of the packet classifier in the forwarding loop that will result in data packets being sent towards the next hop specified by QoS routing, be deferred until the time a RESV message is received. As discussed below, the receipt of a RESV message also implies that loops are not present in the QoS path. Note that the update of classifiers at the time of receipt of a RESV message is consistent with when this is done using the current default routing. The main difference is that the actual flow of data packets may not start following the QoS path until after the classifier has been updated in the first node where the default and the QoS paths start differing.

There are some drawbacks with the above approach, e.g., inability to take advantage of partial reservations in some instances, and they can be addressed in a number of ways. One possibility, that may be acceptable if transient loops are detected and removed quickly, is to actually update classifiers upon receipt of a PATH message (or a certain number of PATH messages, when it appears that the QoS path is stable and loops are not present). Another more comprehensive alternative is to couple this process with the handling of policy information. Such a coupling is a natural step as the ability for users to specify how much of a partial reservation is acceptable to them, i.e., does one need to look for another path, is really a policy issue. In order to support such a coupling, policy data objects would have to be included in PATH, RESV, RESV_ERR, and PATH_ERR messages, in order to enable the local policy control module to assess the suitability of a QoS path. The discussion and description of such an approach is the subject of future work.

For a detailed discussion of how these QoS path management rules within RSVP prevent loops and handle race conditions, the reader is referred to [GKH97].

F.4. Alternatives and Extensions

In the path management approach described here, bulk of the responsibility for QoS path management, i.e., pinning and unpinning of next hop information, lies with RSVP. This was motivated in part by the need to couple path management with the RSVP soft state management, and by the close relation to existing RSVP processing rules. However, it is also possible to defer this responsibility to routing itself. The cost of such an approach would be the need for QoS routing to replicate some of the RSVP state information, e.g. , store PHOP, NHOP, TSpec, etc. , for each flow, and also by requiring that this information be passed across the interface between RSVP and routing.

Another different design approach is to rely on the inclusion of source (explicit) route objects, that would be carried in RSVP PATH messages as opaque objects and passed to QoS routing at each node. Such a design affords some simplification as it avoids the problem of loops altogether, but issues related to pinning and unpinning of paths (at the source) remain for the cases of reservation and link failures. The discussion of such a design is clearly of interest, but it is beyond the scope of this document.

References

[Alm92] P. Almquist. Type of Service in the Internet Protocol Suite. INTERNET-RFC, Internet Engineering Task Force, July 1992.

[BG92] D. Bertsekas and R. G. Gallager. Data Networks. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1992.

[CGR94] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. In Proceedings of the 5th Annual ACM SIAM Symposium on Discrete Algorithms, pages 516--525, Arlington, VA, January 1994.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. Introduction to Algorithms. MIT Press, Cambridge, MA, 1990.

[GGPS96] L. Georgiadis, R. Guerin, V. Peris, and K. N. Sivarajan. Efficient Network QoS Provisioning Based on per Node Traffic Shaping. IEEE/ACM Transactions on Networking, 4(4):482--501, August 1996.

[GJ79] M.R. Garey and D.S. Johnson. Computers and Intractability. Freeman, San Francisco, 1979.

[GKH97] R. Guerin, S. Kamat, and S. Herzog. QoS Path Management with RSVP (draft-ietf-guerin-qos-pathmgt-rsvp-00.txt). INTERNET-DRAFT, Internet Engineering Task Force, March 1997.

[Moy94] J. Moy. OSPF Version 2 - RFC No. 1583 (rfc1583.ps,txt). INTERNET-RFC, Internet Engineering Task Force, March 1994.

[PG94] A. K. Parekh and R. G. Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: the Multiple Node Case. IEEE/ACM Transactions on Networking, 2:137--150, 1994.

[Prz95] A. Przygienda. Link State Routing with QoS in ATM LANs. Ph.D. Thesis Nr. 11051, Swiss Federal Institute of Technology, April 1995.

[RZB+96] R. Braden (Ed.), L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reSerVation Protocol (RSVP) version 1, functional specification (draft-ietf-rsvp-spec-13.ps). INTERNET-DRAFT, Internet Engineering Task Force - RSVP WG, July 1996.

[Tan89] A. Tannenbaum. Computer Networks. Addisson Wesley, 1989.

[Zap96] D. Zappala. RSRR: A Routing Interface for RSVP (draft-ietf-rsvp-routing-01.ps). INTERNET-DRAFT, Internet

Engineering Task Force - RSVP WG, November 1996.

Authors' Address

Roch Guerin
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
guerin@watson.ibm.com
VOICE   +1 914 784-7038

FAX +1 914 784-6205

Sanjay Kamat
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
sanjay@watson.ibm.com
VOICE +1 914 784-7402
FAX +1 914 784-6205

Ariel Orda
Dept. Electrical Engineering
Technion - I.I.T
Haifa, 32000 - ISRAEL
ariel@ee.technion.ac.il
VOICE +011 972-4-8294646
FAX +011 972-4-8323041

Tony Przygienda
FORE Systems Inc.
174 Thorn Hill Rd
Warrendale PA, 15086
prz@fore.com

Doug Williams
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
dougw@watson.ibm.com
VOICE +1 914 784-5047
FAX +1 914 784-6318