Network Working Group
Request for Comments: 2676
Category: Experimental

G. Apostolopoulos
D. Williams
IBM
S. Kamat
Lucent
R. Guerin
UPenn
A. Orda
Technion
T. Przygienda
Siara Systems
August 1999

QoS Routing Mechanisms and OSPF Extensions

Status of this Memo

Copyright Notice

Abstract

This memo describes extensions to the OSPF [Moy98] protocol to support QoS routes. The focus of this 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 briefly 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.

In addition, experience from an implementation of the proposed extensions in the GateD environment [Con], along with performance measurements is presented.

Table of Contents

1. Introduction

In this document, we describe a set of proposed additions to the OSPF routing protocol (these additions have been implemented on top of the GateD [Con] implementation of OSPF V2 [Moy98]) to support Quality-of-Service (QoS) routing in IP networks. Support for QoS routing can be viewed as consisting of three major components:

1. Obtain the information needed to compute QoS paths and select a path capable of meeting the QoS requirements of a given request,

2. Establish the path selected to accommodate a new request,

3. Maintain the path assigned for use by a given request.

Although we touch upon aspects related to the last two components, the focus of this document is on the first one. In particular, we discuss the metrics required to support QoS, the extension to the OSPF link state advertisement mechanism to propagate updates of QoS metrics, and the modifications to the path selection to accommodate QoS requests. The goal of the extensions described in this document is to improve performance for QoS flows (likelihood to be routed on a path capable of providing the requested QoS), with minimal impact on the existing OSPF protocol and its current implementation. 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.

In addition to describing the proposed extensions to the OSPF protocol, this document also reports experimental data based on performance measurements of an implementation done on the GateD platform (see Section 4).

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, 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 capable of identifying and advertising resources that remain available to new 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 a heterogeneous environment where some routers are QoS-capable and others 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 specifies them in some fashion that is accessible to the routing protocol. For example, this could correspond to the arrival of an RSVP [RZB+97] PATH message, whose TSpec is passed to routing together with the destination address. After processing such a request, the routing protocol returns the path that it deems the most suitable given the flow's requirements. Depending on the scope of the path selection process, this returned path could range from simply identifying the best next hop, i.e., a hop-by-hop path selection model, to specifying all intermediate nodes to the destination, i.e., an explicit route model. The nature of the path being returned impacts the operation of the path selection algorithm as it translates into different requirements for constructing and returning the appropriate path information. However, it does not affect the basic operation of the path selection algorithm (2).

For simplicity and also because it is the model currently supported in the implementation (see Section 4 for details), in the rest of this document we focus on the hop-by-hop path selection model. The additional modifications required to support an explicit routing model are discussed in appendix D, but are peripheral to the main focus of this document which concentrates on the specific extensions to the OPSF protocol to support computation of QoS routes.

In addition to the problem of selecting a QoS path and possibly reserving the corresponding resources, one should note that the successful delivery of QoS guarantees requires that the packets of the associated "QoS flow" be forwarded on the selected path. This typically requires the installation of corresponding forwarding state in the router. For example, with RSVP [RZB+97] flows a classifier entry is created based on the filter specs contained in the RESV message. In the case of a Differentiated Service [KNB98] setting, the classifier entry may be based on the destination address (or prefix) and the corresponding value of the DS byte. The mechanisms described in this document are at the control path level and are, therefore, independent of data path mechanisms such as the packet classification method used. Nevertheless, it is important to notice that consistent delivery of QoS guarantees implies stability of the data path. In particular, while it is possible that after a path is first selected, network conditions change and result in the appearance of "better" paths, such changes should be prevented from unnecessarily affecting existing paths. In particular, switching over to a new (and better) path should be limited to specific conditions, e.g., when the initial selection turns out to be inadequate or extremely "expensive". This aspect is beyond the scope

of QoS routing and belongs to the realm of path management, which is outside the main focus of this document. However, because of its potentially significant impact on the usefulness of QoS routing, we briefly outline a possible approach to path management.

Avoiding unnecessary changes to QoS paths requires that state information be maintained for each QoS path after it has been selected. This state information is used to track the validity of the path, i.e., is the current path adequate or should QoS routing be queried again to generate a new and potentially better path. We say that a path is "pinned" when its state specifies that QoS routing need not be queried anew, while a path is considered "un-pinned" otherwise. The main issue is then to define how, when, and where path pinning and un-pinning is to take place, and this will typically depend on the mechanism used to request QoS routes. For example, when the RSVP protocol is the mechanism being used, it is desirable that path management be kept as synergetic as possible with the existing RSVP state management. In other words, pinning and un-pinning of paths should be coordinated with RSVP soft states, and structured so as to require minimal changes to RSVP processing rules. A broad RSVP-routing interface that enables this is described in [GKR97]. Use of such an interface in the context of reserving resources along an explicit path with RSVP is discussed in [GLG+97]. Details of path management and a means for avoiding loops in case of hop-by-hop path setup can be found in [GKH97], and are not addressed further in this document.

1.2. Simplifying Assumptions

In order to achieve our goal of minimizing impact to the existing protocol and implementation, we impose certain restrictions on the range of extensions we initially consider to support QoS. The first restriction is on the type of additional (QoS) metrics that will be added to Link State Advertisements (LSAs) for the purpose of distributing metrics updates. Specifically, the extensions to LSAs that we initially consider, include only available bandwidth and delay. In addition, path selection is itself limited to considering only bandwidth requirements. In particular, the path selection algorithm selects paths capable of satisfying the bandwidth requirement of flows, while at the same time trying to minimize the amount of network resources that need to be allocated, i.e., minimize the number of hops used.

This focus on bandwidth is adequate in most instances, and meant to keep initial complexity at an acceptable level. However, it 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, multiple policies could be used to capture different requirements, each presenting to the path selection algorithm a correspondingly pruned network topology, on which the same algorithm would be used to generate an appropriate path. Alternatively, different algorithms could be used depending on the QoS requirements expressed by an incoming request. Such extensions are beyond the scope of this document, which limits itself to describing the case of a single metric, bandwidth. However, it is worth pointing out that a simple extension to the path selection algorithm proposed in this document allows us to directly account for delay, under certain conditions, when rate-based schedulers are employed, as in the Guaranteed Service proposal [SPG97]; details can be found in [GOW97].

Another important aspect to ensure that introducing support for QoS routing has the minimal possible impact, is to develop a solution that has the smallest possible computing overhead. Additional computations are unavoidable, but it is desirable to keep the computational cost of QoS routing at a level comparable to that of traditional routing algorithms. One possible approach to achieve this goal, is to allow pre-computation of QoS routes. This is the method that was chosen for the implementation of the QoS extensions to OSPF and is, therefore, the one described in detail in this document. Alternative approaches are briefly reviewed in appendices. However, it should be noted that although several alternative path selection algorithms are possible, the same algorithm should be used consistently within a given routing domain. This requirement may be relaxed when explicit routing is used, as the responsibility for selecting a QoS path lies with a single entity, the origin of the request, which then ensures consistency even if each router uses a different path selection algorithm. Nevertheless, the use of a common path selection algorithm within an AS is recommended, if not necessary, for proper operation.

A last aspect of concern regarding the introduction of QoS routing, is to control the overhead associated with the additional link state updates caused by more frequent changes to link metrics. The goal is to minimize the amount of additional update traffic without adversely affecting the performance of path selection. In Section 2.2, we present a brief discussion of various alternatives that trade accuracy of link state information for protocol overhead. Potential enhancements to the path selection algorithm, which seek to (directly) account for the inaccuracies in link metrics, are described in [GOW97], while a comprehensive treatment of the subject

can be found in [LO98, GO99]. In Section 4, we also describe the design choices made in a reference implementation, to allow future extensions and experimentation with different link state update mechanisms.

The rest of this document is structured as follows. In Section 2, we describe the general design choices and mechanisms we rely on to support QoS request. This includes details on the path selection metrics, link state update extensions, and the path selection algorithm itself. Section 3 focuses on the specific extensions that the OSPF protocol requires, while Section 4 describes their implementation in the GateD platform and also presents some experimental results. Section 5 briefly addresses security issues that the proposed schemes may raise. Finally, several appendices provide additional material of interest, e.g., alternative path selection algorithms and support for explicit routes, but somewhat outside the main focus of this document.

2. Path Selection Information and Algorithms

This section reviews the basic building blocks of QoS path selection, namely the metrics on the which the routing algorithm operates, the mechanisms used to propagate updates for these metrics, and finally the path selection algorithm itself.

2.1. Metrics

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 to support a new flow. In general, the network prefers to select the "cheapest" path among all paths suitable for a new flow, and it may even decide not to accept a new flow for which a feasible path exists, if the cost of the path is deemed 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 currently 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. Changes in this metric need to be advertised as part of extended LSAs, so that accurate information is available to the path selection algorithm.

- Link propagation delay: This quantity is meant to identify high latency links, e.g., satellite links, which may be unsuitable for real-time requests. This quantity also needs to be advertised as part of extended LSAs, although timely dissemination of this information is not critical as this parameter is unlikely to change (significantly) over time. As mentioned earlier, link propagation delay can be used to decide on the pruning of specific links, when selecting a path for a delay sensitive request; also, it can be used to support a related extension, as described in [GOW97].

- 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. As a result, the path selection algorithm will attempt to find the minimum hop path capable of satisfying the requirements of a given request. Note that contrary to bandwidth and propagation delay, hop count is a metric that does not affect LSAs, and it is only used implicitly as part of the path selection algorithm.

2.2. Advertisement of Link State Information

The new link metrics identified in the previous section need to be advertised across the network, so that each router can compute accurate and consistent QoS routes. It is assumed that each router maintains an updated database of the network topology, including the current state (available bandwidth and propagation delay) of each link. As mentioned before, the distribution of link state (metrics) information is based on extending OSPF mechanisms. The detailed format of those extensions is described in Section 3, but in addition to how link state information is distributed, another important aspect is when such distribution is to take place.

One option is to mandate 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. Ideally, routers should have the most current view of the bandwidth available on all links in the network, so that they can make the most accurate decision of which path to select. Unfortunately, this then calls for very frequent updates, e.g., each time the available bandwidth of a link changes, which is

neither scalable nor practical. In general, there is a trade-off between the protocol overhead of frequent updates and the accuracy of the network state information that the path selection algorithm depends on. We outline next a few possible link state update policies, which strike a practical compromise.

The basic idea is to trigger link state advertisements only when there is a significant change in the value of metrics since the last advertisement. The notion of significance of a change can be based on an "absolute" scale or a "relative" one. An absolute scale means partitioning the range of values that a metric can take into equivalence classes and triggering an update whenever the metric changes sufficiently to cross a class boundary (3). A relative scale, on the other hand, triggers updates when the percentage change in the metric value exceeds a predefined threshold. Independent of whether a relative or an absolute change trigger mechanism is used, a periodic trigger constraint can also be added. This constraint can be in the form of a hold-down timer, which is used to force a minimum spacing between consecutive updates. Alternatively, a transmit timer can also be used to ensure the transmission of an update after a certain time has expired. Such a feature can be useful if link state updates advertising bandwidth changes are sent unreliably. The current protocol extensions described in Section 3 as well as the implementation of Section 4 do not consider such an option as metric updates are sent using the standard, and reliable, OSPF flooding mechanism. However, this is clearly an extension worth considering as it can help lower substantially the protocol overhead associated with metrics updates.

In both the relative and absolute change approaches, the metric value advertised in an LSA can be either the actual or a quantized value. Advertising the actual metric value is more accurate and, therefore, preferable when metrics are frequently updated. On the other hand, when updates are less frequent, e.g., because of a low sensitivity trigger or the use of hold-down timers, advertising quantized values can be of benefit. This is because it can help increase the number of equal cost paths and, therefore, improve robustness to metrics inaccuracies. In general, there is a broad space of possible trade-offs between accuracy and overhead and selecting an appropriate design point is difficult and depends on many parameters (see [AGKT98] for a more detailed discussion of these issues). As a result, in order to help acquire a better understanding of these issues, the implementation described in Section 4 supports a range of options that allow exploration of the available design space. In addition, Section 4 also reports experimental data on the traffic load and processing overhead generated by links state updates for different configurations.

2.3. Path Selection

There are two major aspects to computing paths for QoS requests. The first is the actual path selection algorithm itself, i.e., which metrics and criteria it relies on. The second is when the algorithm is actually invoked.

The topology on which the algorithm is run is, as with the standard OSPF path selection, a directed graph where vertices (4) 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. 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, which corresponds to the amount of bandwidth that remains available on this interface. This metric is combined with hop count information to provide a cost value, whose goal is to pick 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 a minimal amount of network resources (as accounted by the hop-count metric); and the rule for selecting among these paths is meant to balance load as well as maximize the likelihood that the required bandwidth is indeed 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, because of the specific nature of the two objectives being optimized (bandwidth and hop count), the complexity of the above algorithm is competitive with even that of standard single-objective algorithms. For readers interested in a thorough treatment of the topic, with insights into the connection between the different algorithms, linear algebra and modification of metrics, [Car79] is recommended.

Before proceeding with a more detailed description of the path selection algorithm itself, we briefly review the available options when it comes to deciding when to invoke the algorithm. The two main options are: 1) to perform on-demand computations, that is, trigger

a computation for each new request, and 2) to use some form of pre-computation. The on-demand case involves no additional issues in terms of when computations should be triggered, but running the path selection algorithm for each new request can be computationally expensive (see [AT98] for a discussion on this issue). On the other hand, pre-computing paths amortizes the computational cost over multiple requests, but each computation instance is usually more expensive than in the on-demand case (paths are computed to all destinations and for all possible bandwidth requests rather than for a single destination and a given bandwidth request). Furthermore, depending on how often paths are recomputed, the accuracy of the selected paths may be lower. In this document, we primarily focus on the case of pre-computed paths, which is also the only method currently supported in the reference implementation described in Section 4. In this case, clearly, an important issue is when such pre-computation should take place. The two main options we consider are periodic pre-computations and pre-computations after a given (N) number of updates have been received. The former has the benefit of ensuring a strict bound on the computational load associated with pre-computations, while the latter can provide for a more responsive solution (5). Section 4 provides some experimental results comparing the performance and cost of periodic pre-computations for different period values.

2.3.1. Path Computation Algorithm

This section describes a path selection algorithm, which for a given network topology and link metrics (available bandwidth), pre-computes all possible QoS paths, while maintaining a reasonably low computational complexity. Specifically, the algorithm pre-computes for any destination a minimum hop count path with maximum bandwidth, and has a computational complexity comparable to that of a standard Bellman-Ford shortest path algorithm. The Bellman-Ford (BF) shortest path algorithm 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, because the BF algorithm progresses by increasing hop count, it essentially provides 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 information provides 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 at most these many 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 (the 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 b(n;m) the available bandwidth on the link from node n to 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 KxH 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 (6), the neighbor information is simply the identity of the node adjacent to the source node on that path. As a rule, the "neighbor" node must be a router and not a network, 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 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 are handled. Zero-hop edges arise in the case of transit networks vertices, where only one of the two incoming and outgoing edges should be counted in the hop count computation, as they both correspond to the same physical hop. Accounting for this aspect requires distinguishing between network and router nodes, and the steps involved are detailed later in this section as well as in the pseudo-code of Appendix A.

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). The value of H could be implicit, i.e., the diameter of the network or, in order to better control the worst case complexity, it can be set explicitly thereby limiting path lengths to at most H hops. In the latter case, H must be assigned a value larger than the length of the minimum hop-count path to any node in the graph.

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)-th iteration. The algorithm then looks at each link (n;m) where n is a node whose bw value changed in the previous iteration, and checks the maximal available bandwidth on an (at most) 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 b(n;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 at most 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.

As mentioned earlier, extending the above algorithm to handle zero-hop edges is needed due to the possible use of multi-access networks, e.g., T/R, E/N, etc., to interconnect routers. Such entities are also represented by means of a vertex in the OSPF topology, but a network connecting two routers should clearly 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 Figure 1.

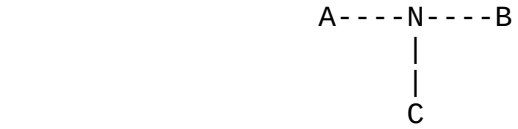


Figure 1: Zero-Hop Edges

In the example of Figure 1, although there are directed edges in both directions, an edge from the network to any of the three routers 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, e.g., a subnet bandwidth manager. The specification and operation of such an entity is beyond the scope of this document.

Accommodating zero-hop edges in the context of the path selection algorithm described above is done as follows: 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. This is the case when m is a transit network. In that case, we attempt to further improve the entry of node k within the current iteration, 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 b(m;k) kept in the topology database (7). 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 can be easily supported. 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 [Moy98], 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 links appear in the router's link advertisements 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 bw(s) field initialized to zero and its neighbor set to null. When a 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:

$$bw(S,h) = \max (bw(S,h) ; \min (bw(V,h) , b(V,S))),$$

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 path 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.

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 simpler than that of explicit routing. This is because, only the next hop needs to be returned instead of an explicit 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. Assuming that the QoS routing table was constructed using the Bellman-Ford algorithm presented later in this section, the search then 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 equal to or larger than B. This entry points to the initial information identifying the selected path.

If the path computation algorithm stores multiple equal cost paths, then some degree of load balancing can be achieved at the time of path selection. A next hop from the list of equivalent next hops can be chosen in a round robin manner, or randomly with a probability that is weighted by the actual available bandwidth on the local interface. The latter is the method used in the implementation described in Section 4.

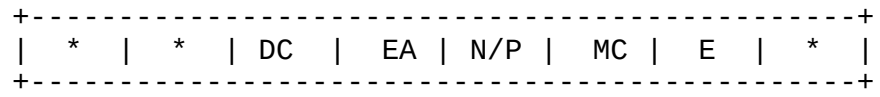
The case of explicit routing is discussed in Appendix D.

3. OSPF Protocol Extensions

As stated earlier, one of our goals 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 describe the extensions to the OSPF protocol needed to support QoS as outlined in the previous sections.

3.1. 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 within an OSPF routing domain. Currently, the OSPF standard [Moy98] specifies the following 5 bits in the options octet:



Note that the least significant bit (‘T’ bit) that was used to indicate TOS routing capability in the older OSPF specification [Moy94] has been removed. However, for backward compatibility with previous versions of the OSPF specification, TOS-specific information can be included in router-LSAs, summary-LSAs and AS-external-LSAs.

We propose to reclaim the ‘T’ bit as an indicator of router’s QoS routing capability and refer to it as the ‘Q’ bit. In fact, QoS capability can be viewed as an extension of the TOS-capabilities and QoS routing as a form of TOS-based routing. A router sets this bit in its hello packets to indicate that it is capable of supporting such routing. When this 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 this bit is set in a network link state advertisement it means that the network described in the advertisement is QoS capable.

We need to be careful in this approach so as to avoid confusing any old style (i.e., RFC 1583 based) TOS routing implementations. The TOS metric encoding rules of QoS fields introduced further in this section will show how this is achieved. Additionally, unlike the RFC 1583 specification that unadvertised TOS metrics be treated to have same cost as TOS 0, for the purpose of computing QOS routes, unadvertised TOS metrics (on a hop) indicate lack of connectivity for the specific TOS metrics (for that hop).

3.2. 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 routers (if any) when misinterpreting the QoS extensions.

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 most significant 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.

3.2.1. 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].

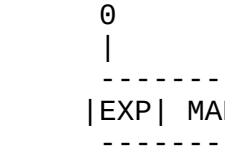
Given a base of 8, the 3 most significant bits should be reserved for the exponent part and the remaining 13 for the mantissa. This allows a simple comparison for two numbers encoded in this form, which is often useful during implementation.

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: "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 and advertise 2's complement of the above representation."



Thus, the above encoding advertises a numeric value that is

2^16 -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 = 1024^3 Bytes/s, its encoding would consist of an exponent value of 6 since 1024^3= 4,096*8^6, which would then have a granularity of 8^6 or approx. 260 kBytes/s. The associated binary representation would then be %(110) 0 1000 0000 0000% or 53,248 (8). 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 (2^16 - 1) - 53,248 = 12,287. Assuming now a current reservation level of 6;400 Mbits/s = 200 * 1024^2, 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 * 8^5, which corresponds to a granularity of 8^5 or approx. 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 (2^16-1) -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.

3.2.2. 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 (2^13-1) *4^7 i.e., approx. 134 seconds.

3.3. Packet Formats

Given the extended TOS notation to account for QoS metrics, no changes in packet formats are necessary except for the (re)introduction of T-bit as the 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.

To support QoS, there are additions to two Link State Advertisements, the Router Links Advertisement and the Summary Links Advertisement. As stated above, a router identifies itself as supporting QoS by setting the Q-bit in the options field of the Link State Header. When a router that supports QoS receives either the Router Links or Summary Links Advertisement, it should parse the QoS metrics encoded in the received Advertisement.

3.4. 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.

3.5. 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 these 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.

4. A Reference Implementation based on GateD

In this section we report on the experience gained from implementing the pre-computation based approach of Section 2.3.1 in the GateD [Con] environment. First, we briefly introduce the GateD environment, and then present some details on how the QoS extensions were implemented in this environment. Finally, we discuss issues that arose during the implementation effort and present some measurement based results on the overhead that the QoS extensions impose on a QoS capable router and a network of QoS routers. For further details on the implementation study, the reader is referred to [AGK99]. Additional performance evaluation based on simulations can be found in [AGKT98].

4.1. The Gate Daemon (GateD) Program

GateD [Con] is a popular, public domain (9) program that provides a platform for implementing routing protocols on hosts running the Unix operating system. The distribution of the GateD software also includes implementations of many popular routing protocols, including the OSPF protocol. The GateD environment offers a variety of services useful for implementing a routing protocol. These services

include a) support for creation and management of timers, b) memory management, c) a simple scheduling mechanism, d) interfaces for manipulating the host's routing table and accessing the network, and e) route management (e.g., route prioritization and route exchange between protocols).

All GateD processing is done within a single Unix process, and routing protocols are implemented as one or several tasks. A GateD task is a collection of code associated with a Unix socket. The socket is used for the input and output requirements of the task. The main loop of GateD contains, among other operations, a select() call over all task sockets to determine if any read/write or error conditions occurred in any of them. GateD implements the OSPF link state database using a radix tree for fast access to individual link state records. In addition, link state records for neighboring network elements (such as adjacent routers) are linked together at the database level with pointers. GateD maintains a single routing table that contains routes discovered by all the active routing protocols. Multiple routes to the same destination are prioritized according to a set of rules and administrative preferences and only a single route is active per destination. These routes are periodically downloaded in the host's kernel forwarding table.

4.2. Implementing the QoS Extensions of OSPF

4.2.1. Design Objectives and Scope

One of our major design objectives was to gain substantial experience with a functionally complete QoS routing implementation while containing the overall implementation complexity. Thus, our architecture was modular and aimed at reusing the existing OSPF code with only minimal changes. QoS extensions were localized to specific modules and their interaction with existing OSPF code was kept to a minimum. Besides reducing the development and testing effort, this approach also facilitated experimentation with different alternatives for implementing the QoS specific features such as triggering policies for link state updates and QoS route table computation. Several of the design choices were also influenced by our assumptions regarding the core functionalities that an early prototype implementation of QoS routing must demonstrate. Some of the important assumptions/requirements are:

- Support for only hop-by-hop routing. This affected the path structure in the QoS routing table as it only needs to store next hop information. As mentioned earlier, the structure can be easily extended to allow construction of explicit routes.

- Support for path pre-computation. This required the creation of a separate QoS routing table and its associated path structure, and was motivated by the need to minimize processing overhead.

- Full integration of the QoS extensions into the GateD framework, including configuration support, error logging, etc. This was required to ensure a fully functional implementation that could be used by others.

- Ability to allow experimentation with different approaches, e.g., use of different update and pre-computation triggering policies with support for selection and parameterization of these policies from the GateD configuration file.

- Decoupling from local traffic and resource management components, i.e., packet classifiers and schedulers and local call admission. This is supported by providing an API between QoS routing and the local traffic management module, which hides all internal details or mechanisms. Future implementations will be able to specify their own mechanisms for this module.

- Interface to RSVP. The implementation assumes that RSVP [RZB+97] is the mechanism used to request routes with specific QoS requirements. Such requests are communicated through an interface based on [GKR97], and used the RSVP code developed at ISI, version 4.2a2 [RZB+97].

In addition, our implementation also relies on several of the simplifying assumptions made earlier in this document, namely:

- The scope of QoS route computation is currently limited to a single area.

- All routers within the area are assumed to run a QoS enabled version of OSPF, i.e., inter-operability with non-QoS aware versions of the OSPF protocol is not considered.

- All interfaces on a router are assumed to be QoS capable.

4.2.2. Architecture

The above design decisions and assumptions resulted in the architecture shown in Figure 2. It consists of three major components: the signaling component (RSVP in our case); the QoS routing component; and the traffic manager. In the rest of this section we concentrate on the structure and operation of the QoS routing component. As can be seen in Figure 2, the QoS routing extensions are further divided into the following modules:

- Update trigger module determines when to advertise local link state updates. This module implements a variety of triggering policies: periodic, threshold based triggering, and class based triggering. This module also implements a hold-down timer that enforces minimum spacing between two consecutive update triggerings from the same node.

- Pre-computation trigger module determines when to perform QoS path pre-computation. So far, this module implements only periodic pre-computation triggering.

- Path pre-computation module computes the QoS routing table based on the QoS specific link state information as described in Section 2.3.1.

- Path selection and management module selects a path for a request with particular QoS requirements, and manages it once selected, i.e., reacts to link or reservation failures. Path selection is performed as described in Section 2.3.1. Path management functionality is not currently supported.

- QoS routing table module implements the QoS specific routing table, which is maintained independently of the other GateD routing tables.

- Tspec mapping module maps request requirements expressed in the form of RSVP Tspecs and Rspecs into the bandwidth requirements that QoS routing uses.

4.3. Major Implementation Issues

Mapping the above design to the framework of the GateD implementation of OSPF led to a number of issues and design decisions. These issues mainly fell under two categories: a) interoperation of the QoS extensions with pre-existing similar OSPF mechanisms, and b) structure, placement, and organization of the QoS routing table. Next, we briefly discuss these issues and justify the resulting design decisions.

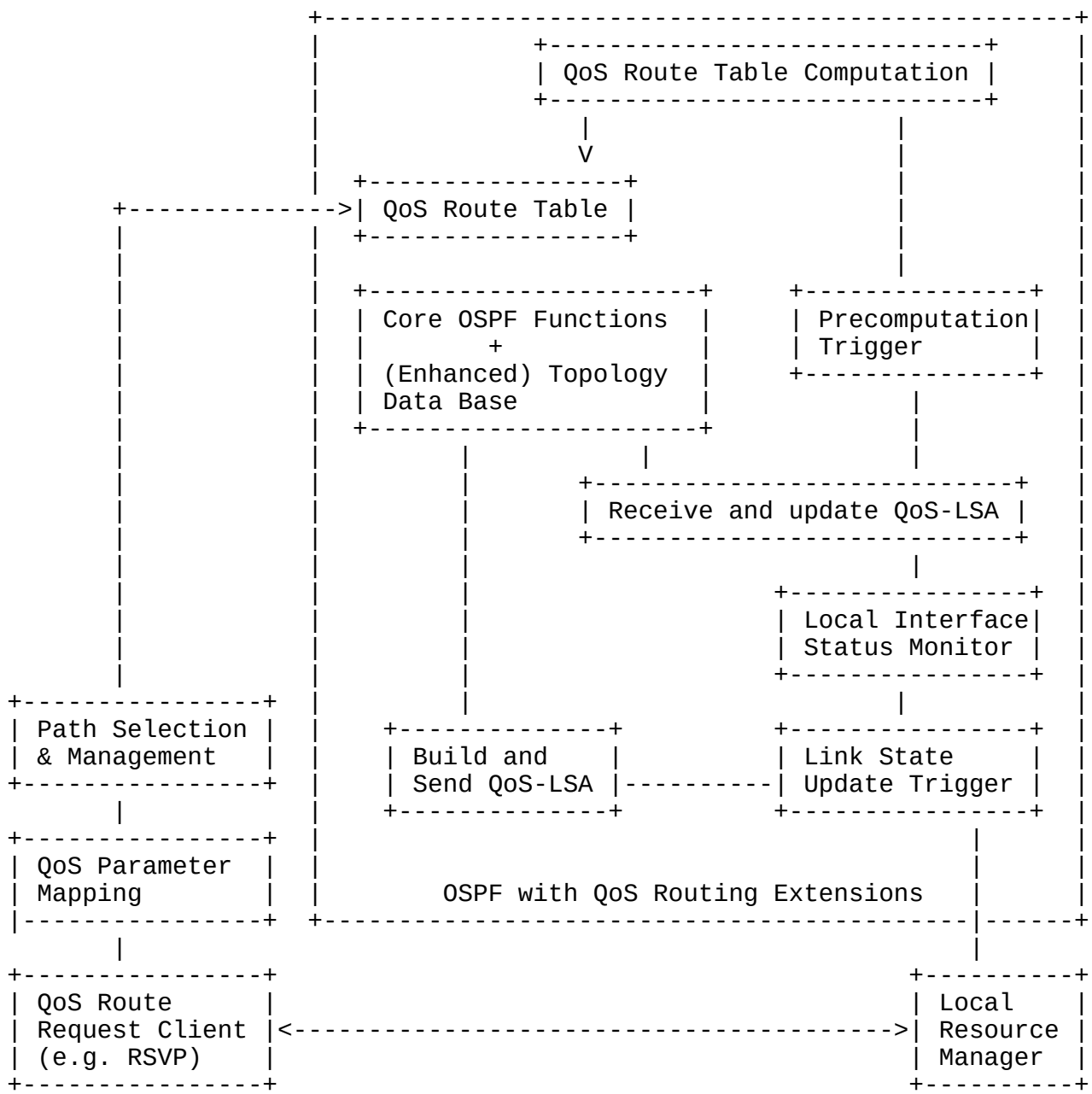


Figure 2: The software architecture

The ability to trigger link state updates in response to changes in bandwidth availability on interfaces is an essential component of the QoS extensions. Mechanisms for triggering these updates and controlling their rate have been mentioned in Section 2.2. In addition, OSPF implements its own mechanism for triggering link state updates as well as its own hold down timer, which may be incompatible with what is used for the QoS link state updates. We handle such potential conflicts as follows. First, since OSPF triggers updates on a periodic basis with low frequency, we expect these updates to be only a small part of the total volume of updates generated. As a result, we chose to maintain the periodic update triggering of OSPF. Resolving conflicts in the settings of the different hold down timer settings requires more care. In particular, it is important to ensure that the existing OSPF hold down timer does not interfere with QoS updates. One option is to disable the existing OSPF timer, but protection against transient overloads calls for some hold down timer, albeit with a small value. As a result, the existing OSPF hold down timer was kept, but reduced its value to 1 second. This value is low enough (actually is the lowest possible, since GateD timers have a maximum resolution of 1 second) so that it does not interfere with the generation of the QoS link state updates, which will actually often have hold down timers of their own with higher values. An additional complexity is that the triggering of QoS link state updates needs to be made aware of updates performed by OSPF itself. This is necessary, as regular OSPF updates also carry bandwidth information, and this needs to be considered by QoS updates to properly determine when to trigger a new link state update.

Another existing OSPF mechanism that has the potential to interfere with the extensions needed for QoS routing, is the support for delayed acknowledgments that allows aggregation of acknowledgments for multiple LSAs. Since link state updates are maintained in retransmission queues until acknowledged, excessive delay in the generation of the acknowledgement combined with the increased rates of QoS updates may result in overflows of the retransmission queues. To avoid these potential overflows, this mechanism was bypassed altogether and LSAs received from neighboring routers were immediately acknowledged. Another approach which was considered but not implemented, was to make QoS LSAs unreliable, i.e., eliminate their acknowledgments, so as to avoid any potential interference. Making QoS LSAs unreliable would be a reasonable design choice because of their higher frequency compared to the regular LSAs and the reduced impact that the loss of a QoS LSA has on the protocol operation. Note that the loss of a QoS LSA does not interfere with the base operation of OSPF, and only transiently reduces the quality of paths discovered by QoS routing.

The structure and placement of the QoS routing table also raises some interesting implementation issues. Pre-computed paths are placed into a QoS routing table. This table is implemented as a set of path structures, one for each destination, which contain all the available paths to this destination. In order to be able to efficiently locate individual path structures, an access structure is needed. In order to minimize the develpement effort, the radix tree structure used for the regular GateD routing tables was reused. In addition, the QoS routing table was kept independent of the GateD routing tables to conform to the design goal of localizing changes and minimizing the impact on the existing OSPF code. An additional reason for maintaining the QoS routing separate and self-contained is that it is re-computed under conditions that are different from those used for the regular routing tables.

Furthermore, since the QoS routing table is re-built frequently, it must be organized so that its computation is efficient. A common operation during the computation of the QoS routing table is mapping a link state database entry to the corresponding path structure. In order to make this operation efficient, the link state database entries were extended to contain a pointer to the corresponding path structure. In addition, when a new QoS routing table is to be computed, the previous one must be de-allocated. This is accomplished by traversing the radix tree in-order, and de-allocating each node in the tree. This full de-allocation of the QoS routing table is potentially wasteful, especially since memory allocation and de-allocation is an expensive operation. Furthermore, because path pre-computations are typically not triggered by changes in topology, the set of destinations will usually remain the same and correspond to an unchanged radix tree. A natural optimization would then be to de-allocate only the path structures and maintain the radix tree. A further enhancement would be to maintain the path structures as well, and attempt to incrementally update them only when required. However, despite the potential gains, these optimizations have not been included in the initial implementation. The main reason is that they involve subtle and numerous checks to ensure the integrity of the overall data structure at all times, e.g., correctly remove failed destinations from the radix tree and update the tree accordingly.

4.4. Bandwidth and Processing Overhead of QoS Routing

After completing the implementation outlined in the previous sections, it was possible to perform an experimental study of the cost and nature of the overhead of the QoS routing extensions proposed in this document. In particular, using a simple setup consisting of two interconnected routers, it is possible to measure the cost of individual QoS routing related operations. These operations are: a) computation of the QoS routing table, b) selection of a path from the QoS routing table, c) generation of a link state update, and d) reception of a link state update. Note that the last two operations are not really specific to QoS routing since regular OSPF also performs them. Nevertheless, we expect the more sensitive update triggering mechanisms required for effective QoS routing to result in increased number of updates, making the cost of processing updates an important component of the QoS routing overhead. An additional cost dimension is the memory required for storing the QoS routing table. Scaling of the above costs with increasing sizes of the topology database was investigated by artificially populating the topology databases of the routers under measurement.

Table 1 shows how the measured costs depend on the size of the topology. The topology used in the measurements was built by replicating a basic building block consisting of four routers connected with transit networks in a rectangular arrangement. The details of the topology and the measurements can be found in [AGK99]. The system running the GateD software was an IBM IntelliStation Z Pro with a Pentium Pro processor at 200 MHz, 64 MBytes or real memory, running FreeBSD 2.2.5-RELEASE and GateD 4. From the results of Table 1, one can observe that the cost of path pre-computation is not much higher than that of the regular SPF computation. However, path pre-computation may need to be performed much more often than the SPF computation, and this can potentially lead to higher processing costs. This issue was investigated in a set of subsequent experiments, that are described later in this section. The other cost components reported in Table 1 include memory, and it can be seen that the QoS routing table requires roughly 80% more memory than the regular routing table. Finally, the cost of selecting a path is found to be very small compared to the path pre-computation times. As expected, all the measured quantities increase as the size of the topology increases. In particular, the storage requirements and the processing costs for both SPF computation and QoS path pre-computation scale almost linearly with the network size.

Link state database size	25	49	81	121	169	225
Regular SPF time (microsec)	215	440	747	1158	1621	2187
Pre-computation time (microsec)	736	1622	2883	4602	6617	9265
SPF routing table size (bytes)	2608	4984	8152	12112	16864	22408
QoS routing table size (bytes)	3924	7952	13148	19736	27676	36796
Path Selection time (microsec)	.7	1.6	2.8	4.6	6.6	9.2

Table 1: Stand alone QoS routing costs

In addition to the stand alone costs reported in Table 1, it is important to assess the actual operational load induced by QoS routing in the context of a large network. Since it is not practical to reproduce a large scale network in a lab setting, the approach used was to combine simulation and measurements. Specifically, a simulation was used to obtain a time stamped trace of QoS routing related events that occur in a given router in a large scale network. The trace was then used to artificially induce similar load conditions on a real router and its adjacent links. In particular, it was used to measure the processing load at the router and bandwidth usage that could be attributed to QoS updates. A more complete discussion of the measurement method and related considerations can be found in [AGK99].

The use of a simulation further allows the use of different configurations, where network topology is varied together with other QoS parameters such as a) period of pre-computation, and b) threshold for triggering link state updates. The results reported here were derived using two types of topologies. One based on a regular but artificial 8x8 mesh network, and another (isp) which has been used in several previous studies [AGKT98, AT98] and that approximates the network of a nation-wide ISP. As far as pre-computation periods are concerned, three values of 1, 5 and 50 seconds were chosen, and for the triggering of link state update thresholds of 10% and 80% were used. These values were selected as they cover a wide range in terms of precision of pre-computed paths and accuracy of the link state information available at the routers. Also note that 1 second is the smallest pre-computation period allowed by GateD.

Table 2 provides results on the processing load at the router driven by the simulation trace, for the two topologies and different combinations of QoS parameters, i.e., pre-computation period and threshold for triggering link state updates. Table 3 gives the bandwidth consumption of QoS updates on the links adjacent to the router.

	Pre-computation Period		
Link state threshold	1 sec	5 sec	50 sec
10%	.45% (1.6%)	.29% (2%)	.17% (3%)
80%	.16% (2.4%)	.04% (3%)	.02% (3.8%)

isp

10%	3.37% (2.1%)	2.23% (3.3%)	1.78% (7.7%)
80%	1.54% (5.4%)	.42% (6.6%)	.14% (10.4%)

8x8 mesh

Table 2: Router processing load and (bandwidth blocking).

In Table 2, processing load is expressed as the percentage of the total CPU resources that are consumed by GateD processing. The same table also shows the routing performance that is achieved for each combination of QoS parameters, so that comparison of the different processing cost/routing performance trade-offs can be made. Routing performance is measured using the bandwidth blocking ratio, defined as the sum of requested bandwidth of the requests that were rejected over the total offered bandwidth. As can be seen from Table 2, processing load is low even when the QoS routing table is recomputed every second, and LSAs are generated every time the available bandwidth on a link changes by more than 10% of the last advertised value. This seems to indicate that given today's processor technology, QoS routing should not be viewed as a costly enhancement, at least not in terms of its processing requirements. Another general observation is that while network size has obviously an impact, it does not seem to drastically affect the relative influence of the different parameters. In particular, despite the differences that exist between the isp and mesh topologies, changing the pre-computation period or the update threshold translates into essentially similar relative changes.

Similar conclusions can be drawn for the update traffic shown in Table 3. In all cases, this traffic is only a small fraction of the link's capacity. Clearly, both the router load and the link bandwidth consumption depend on the router and link that was the target of the measurements and will vary for different choices. The results shown here are meant to be indicative, and a more complete discussion can be found in [AGK99].

Link state threshold	
10%	3112 bytes/sec
80%	177 bytes/sec

isp

10%	15438 bytes/sec
80%	1053 bytes/sec

8x8 mesh

Table 3: Link state update traffic

Summarizing, by carrying out the implementation of the proposed QoS routing extensions to OSPF we demonstrated that such extensions are fairly straightforward to implement. Furthermore, by measuring the performance of the real system we were able to demonstrate that the overheads associated with QoS routing are not excessive, and are definitely within the capabilities of modern processor and workstation technology.

5. Security Considerations

The QoS extensions proposed in this document do not raise any security considerations that are in addition to the ones associated with regular OSPF. The security considerations of OSPF are presented in [Moy98]. However, it should be noted that this document assumes the availability of some entity responsible for assessing the legitimacy of QoS requests. For example, when the protocol used for initiating QoS requests is the RSVP protocol, this capability can be provided through the use of RSVP Policy Decision Points and Policy Enforcement Points as described in [YPG97]. Similarly, a policy server enforcing the acceptability of QoS requests by implementing decisions based on the rules and languages of [RMK+98], would also be capable of providing the desired functionality.

APPENDICES

A. Pseudocode for the BF Based Pre-Computation Algorithm

Note: The pseudocode below assumes a hop-by-hop forwarding approach in updating the neighbor field. The modifications needed to support explicit route construction 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 the 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) then
           S_prev :=  S_prev union {n}
             /* only a router is added to S_prev, */
             /* if it is 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. On-Demand Dijkstra Algorithm for QoS Path Computation

In the main text, 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, 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 (see the pseudocode that follows). 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.

Note: The pseudocode below assumes a hop-by-hop forwarding approach in updating the neighbor field. Addition of routes to stub networks is done in a second phase as usual. The modifications needed to support explicit route construction are straightforward. The pseudocode also does not handle equal cost multi-paths for simplicity, but the modifications needed to add this support are also easily done.

```
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.
  d = destination vertex.
  B = requested bandwidth for the flow served.

Type:
  tab_entry: record
                 hops = integer,
                 neighbor = integer 1..N,
                 ontree = boolean.

Variables:
  TT[1..N]: topology table, whose (n) entry is a tab_entry
                  record, such that:
                    TT[n].bw is the available bandwidth (as known
                        thus far) on a shortest-path between
                        vertices s and n,
                    TT[n].neighbor is the first hop on that path (a
                        neighbor of s). It is either a router or the
                        destination n.
  S: list of candidate vertices;
  v: vertex under consideration;

The Algorithm:

begin;
  for n:=1 to N do  /* initialization */
  begin;
    TT[n].hops := infinity;
    TT[n].neighbor := null;
    TT[n].ontree := FALSE;
  end;
  TT[s].hops := 0;

  reset S;
  v:= s;
  while v <> d do
  begin;
    TT[v].ontree := TRUE;
    for all edges (v,m) in L and b(v,m) >= B do
    begin;
```

```
      if m is a router
      begin;
        if not TT[m].ontree then
        begin;
          /* bandwidth must be fulfilled since all links >= B */
          if TT[m].hops > TT[v].hops + 1 then
          begin
            S := S union { m };
            TT[m].hops := TT[v].hops + 1;
            TT[m].neighbor := v;
          end;
        end;
      end;
      else /* must be a network, iterate over all attached routers */
      begin; /* each network -- router edge treated as zero hop edge */
        for all (m,k) in L, k <> v,
             /* i.e., for all other neighboring routers of m */
        if not TT[k].ontree and b(m,k) >= B then
        begin;
          if TT[k].hops > TT[v].hops  then
          begin;
            S := S union { k };
            TT[k].hops := TT[v].hops;
            TT[k].neighbor := v;
          end;
        end;
      end;
    end; /* of all edges from the vertex under consideration */

    if S is empty then
    begin;
      v=d; /* which will end the algorithm */
    end;
    else
    begin;
      v := first element of S;
      S := S - {v}; /* remove and store the candidate to consider */
    end;
  end; /* from processing of the candidate list */
end.
```

C. Precomputation Using Dijkstra Algorithm

This appendix 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 solution to compute all "best" paths is to consecutively compute shortest path spanning trees starting from a complete graph and removing links with less bandwidth than the threshold used in the previous computation. This yields paths with possibly better bandwidth but of course more hops. Despite the large number of Dijkstra computations involved, several optimizations such as incremental spanning tree computation can be used and allow for efficient implementations in terms of complexity as well as storage since the structure of computed paths leans itself towards path compression [ST83]. Details including measurements and applicability studies can be found in [Prz95] and [BP95].

A variation of this theme is to trade the "accuracy" of the pre-computed paths, (i.e., the paths being generated may be of a larger hop count than needed) for the benefit of using a modified version of Dijkstra shortest path algorithm and also saving on some computations. This loss in accuracy comes from the need to rely on quantized bandwidth values, which 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. We restrict our discussion here to this "quantized" version of the algorithm.

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, which 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, which 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 b(n;m) is 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 b(n;m) < bw[q], and then runs on the remaining topology a Dijkstra-based minimum hop count algorithm (10) 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 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).

Note: The pseudocode below assumes a hop-by-hop forwarding approach in updating the neighbor field. The modifications needed to support explicit route construction are straightforward. The pseudocode also does not handle equal cost multi-paths for simplicity, but the modification needed to add this support have been described above. Details of the post-processing step of adding stub networks are omitted.

```
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).
  bw[1..Q] = array of bandwidth values to "quantize" flow requests to.
  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.

Type:
  tab_entry: record
                hops = integer,
                neighbor = integer 1..N,
                ontree = boolean.

Variables:
  TT[1..N, 1..Q]: topology table, whose (n,q) entry is a tab_entry
                  record, such that:
                    TT[n,q].bw is the available bandwidth (as known
                        thus far) on a shortest-path between
                        vertices s and n accommodating bandwidth b(q),
                    TT[n,q].neighbor is the first hop on that path (a
                        neighbor of s). It is either a router or the
                        destination n.
  S: list of candidate vertices;
  v: vertex under consideration;
  q: "quantize" step

The Algorithm:

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

```
    begin;
      TT[n,r].hops     := infinity;
      TT[n,r].neighbor := null;
      TT[n,r].ontree   := FALSE;
    end;
    TT[s,r].hops := 0;
  end;
  for r:=1 to Q do
  begin;
    S = {s};
    while S not empty do
    begin;
      v := first element of S;
      S := S - {v}; /* remove and store the candidate to consider */
      TT[v,r].ontree := TRUE;
      for all edges (v,m) in L and b(v,m) >= bw[r] do
      begin;
        if m is a router
        begin;
          if not TT[m,r].ontree then
          begin;
            /* bandwidth must be fulfilled since all links >= bw[r] */
            if TT[m,r].hops > TT[v,r].hops + 1 then
            begin
              S := S union { m };
              TT[m,r].hops := TT[v,r].hops + 1;
              TT[m,r].neighbor := v;
            end;
          end;
        end;
        else /* must be a network, iterate over all attached
                routers */
        begin;
          for all (m,k) in L, k <> v,
               /* i.e., for all other neighboring routers of m */
          if not TT[k,r].ontree and b(m,k) >= bw[r] then
          begin;
            if TT[k,r].hops > TT[v,r].hops + 2 then
            begin;
              S := S union { k };
              TT[k,r].hops := TT[v,r].hops + 2;
              TT[k,r].neighbor := v;
            end;
          end;
        end;
      end; /* of all edges from the vertex under consideration */
    end; /* from processing of the candidate list */
  end; /* of "quantize" steps */
```

end.

D. Explicit 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., an explicit 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 explicit routes with minor modifications. These modifications and how they facilitate constructing explicit routes are discussed next.

The general approach to facilitate construction of explicit routes is to update the neighbor field differently from the way it is done for hop-by-hop routing as described in Section 2.3. Recall that in the path computation algorithms the neighbor field is updated to reflect the identity of the router 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 explicit routing, the neighbor information is updated to reflect the identity of the previous router on the path.

In general, there can be multiple equivalent paths for a given hop count. Thus, the neighbor information is stored as a list rather than single value. Associated with each neighbor, additional information is stored to facilitate load balancing among these multiple paths at the time of path selection. Specifically, we store the advertised available bandwidth on the link from the neighbor to the destination in the entry.

With this change, the basic approach used to extract the complete list of vertices 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. The process is described as recursive since the neighbor node identified in one step becomes the destination node for table look up in the next step. Once the source router is reached, the concatenation of all the neighbor fields that have been extracted forms the desired explicit route. This applies to algorithms of Section 2.3.1 and Appendix C. If at a particular stage there are multiple neighbor choices (due to equal cost multi-paths), one of them can be chosen at random with a probability that is weighted, for example, by the

associated bandwidth on the link from the neighbor to the (current) destination.

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. The row is then searched to identify a suitable path. If the Bellman-Ford algorithm of Section 2.3.1 was 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 that is equal to or greater than B. This entry points to the initial information identifying the selected path. If the Dijkstra algorithm of Appendix C is used, the first quantized value qB such that qB >= 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 Section 2.3.1 or Appendix C. 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 the 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 Appendix C, 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 explicit 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 xplicit routing and hop-by-hop routing is needed.

Endnotes

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.

2. This is true for uni-cast flows, but in the case of multi-cast flows, hop-by-hop and an explicit routing clearly have different implications.

3. Some hysteresis mechanism should be added to suppress updates when the metric value oscillates around a class boundary.

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

5. Various hybrid methods can also be envisioned, e.g., periodic computations except if more than a given number of updates are received within a shorter interval, or periodic updates except if the change in metrics corresponding to a given update exceeds a certain threshold. Such variations are, however, not considered in this document.

6. Modifications to support explicit routing are discussed in Appendix D.

7. Note, that this does not say anything on whether to differentiate between outgoing and incoming bandwidth on a shared media network. As a matter of fact, a reasonable option is to set the incoming bandwidth (from network to router) to infinity, and only use the outgoing bandwidth value to characterize bandwidth availability on the shared network.

8. exponent in parenthesis

9. Access to some of the more recent versions of the GateD software is restricted to the GateD consortium members.

10. 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.

References

[AGK99] G. Apostolopoulos, R. Guerin, and S. Kamat. Implementation and performance meassurements of QoS routing extensions to OSPF. In Proceedings of INFOCOM'99, pages 680--688, New York, NY, March 1999.

[AGKT98] G. Apostolopoulos, R. Guerin, S. Kamat, and S. K. Tripathi. QoS routing: A performance perspective. In Proceedings of ACM SIGCOMM'98, pages 17--28, Vancouver, Canada, October

[Alm92] Almquist, P., "Type of Service in the Internet Protocol Suite", RFC 1349, July 1992.

[AT98] G. Apostolopoulos and S. K. Tripathi. On reducing the processing cost of on-demand QoS path computation. In Proceedings of ICNP'98, pages 80--89, Austin, TX, October 1998.

[BP95] J.-Y. Le Boudec and T. Przygienda. A Route Pre-Computation Algorithm for Integrated Services Networks. Journal of Network and Systems Management, 3(4), 1995.

[Car79] B. Carre. Graphs and Networks. Oxford University Press, ISBN 0-19-859622-7, Oxford, UK, 1979.

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

[Con] Merit GateD Consortium. The Gate Daemon (GateD) project.

[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. In Proceedings of the 2nd IEEE Global Internet Mini-Conference, pages 1914-1918, Phoenix, AZ, November

[GKR97] Guerin, R., Kamat, S. and E. Rosen, "An Extended RSVP Routing Interface, Work in Progress.

[GLG+97] Der-Hwa G., Li, T., Guerin, R., Rosen, E. and S. Kamat, "Setting Up Reservations on Explicit Paths using RSVP", Work in Progress.

[GO99] R. Guerin and A. Orda. QoS-Based Routing in Networks with Inaccurate Information: Theory and Algorithms. IEEE/ACM Transactions on Networking, 7(3):350--364, June 1999.

[GOW97] R. Guerin, A. Orda, and D. Williams. QoS Routing Mechanisms and OSPF Extensions. In Proceedings of the 2nd IEEE Global Internet Mini-Conference, pages 1903-1908, Phoenix, AZ, November 1997.

[KNB98] Nichols, K., Blake, S., Baker F. and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, December 1998.

[LO98] D. H. Lorenz and A. Orda. QoS Routing in Networks with Uncertain Parameters. IEEE/ACM Transactions on Networking, 6(6):768--778, December 1998.

[Moy94] Moy, J., "OSPF Version 2", RFC 1583, March 1994.

[Moy98] Moy, J., "OSPF Version 2", STD 54, RFC 2328, April 1998.

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

[RMK+98] R. Rajan, J. C. Martin, S. Kamat, M. See, R. Chaudhury, D. Verma, G. Powers, and R. Yavatkar. Schema for differentiated services and integrated services in networks. INTERNET-DRAFT, October 1998. work in progress.

[RZB+97] Braden, R., Editor, Zhang, L., Berson, S., Herzog, S. and S. Jamin, "Resource reSerVation Protocol (RSVP) Version 1, Functional Specification", RFC 2205, September 1997.

[SPG97] Shenker, S., Partridge, C. and R. Guerin, "Specification of Guaranteed Quality of Service", RFC 2212, November 1997.

[ST83] D.D. Sleator and R.E. Tarjan. A Data Structure for Dynamic Trees. Journal of Computer Systems, 26, 1983.

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

[YPG97] Yavatkar, R., Pendarakis, D. and R. Guerin, "A Framework for Policy-based Admission Control", INTERNET-DRAFT, April 1999. Work in Progress.

Authors' Addresses

George Apostolopoulos
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Phone: +1 914 784-6204
Fax: +1 914 784-6205
EMail: georgeap@watson.ibm.com

Roch Guerin
University Of Pennsylvania
Department of Electrical Engineering, Rm 367 GRW
200 South 33rd Street
Philadelphia, PA 19104--6390

Phone: +1 215-898-9351
EMail: guerin@ee.upenn.edu

Sanjay Kamat
Bell Laboratories
Lucent Technologies
Room 4C-510
101 Crawfords Corner Road
Holmdel, NJ 07733

Phone: (732) 949-5936
email: sanjayk@dnrc.bell-labs.com

Ariel Orda
Dept. Electrical Engineering
Technion - I.I.T
Haifa, 32000 - ISRAEL

Phone: +011 972-4-8294646
Fax: +011 972-4-8323041
EMail: ariel@ee.technion.ac.il

Tony Przygienda
Siara Systems
300 Ferguson Drive
Moutain View
California 94043

Phone: +1 732 949-5936
Email: prz@siara.com

Doug Williams
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Phone: +1 914 784-5047
Fax: +1 914 784-6318
EMail: dougw@watson.ibm.com

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.