

# Implementation and Performance Measurements of QoS Routing Extensions to OSPF

George Apostolopoulos  
Computer Science Department  
University of Maryland  
College Park, MD 20742

Roch Guérin, Sanjay Kamat  
IBM T. J. Watson  
Research Center  
Yorktown Heights, NY 10598

## Abstract

In this paper, we report on an implementation of QoS routing extensions to the OSPF protocol, and on various performance measurements made on the basis of this implementation to assess the cost and feasibility of QoS routing in IP networks. The results provide insight into the respective weights of the two major components of QoS routing costs, processing cost and protocol overhead. More important, they establish strong empirical evidence that the cost of QoS routing is well within the limits of modern processors. Furthermore, we also find that although support for QoS routing does increase the amount of protocol traffic, it still remains a minute fraction of the bandwidth of most links. The paper also explores the sensitivity of our findings to variations in network size and topology, by combining the information obtained from the implementation to results generated by means of simulations. This provides a comprehensive investigation of the operational costs of QoS routing.

## 1 Introduction

Because of its potential benefits, Quality of Service (QoS) routing has recently received substantial attention in the context of its possible use in an integrated services IP network. QoS routing is the process of selecting the path to be used by the packets of a flow based on its QoS requirements, e.g., bandwidth or delay. Several recent research results [1, 2, 3, 4, 5] have pointed out the potential of QoS routing for improving network utilization and the service levels provided to requests with QoS requirements. The improvement to the service received by users is in the form of an increased likelihood of finding a path that meets their QoS requirements. Conversely, the improvement to network efficiency is usually in terms of increase in *revenue*, where revenue is typically a function of the number of flows or the amount of bandwidth carried by the network.

Despite these benefits, there remains much uncertainty regarding the value and feasibility of implementing QoS routing protocols in IP networks. This is primarily because of the additional costs that support for QoS routing entails. These added costs have two major components: *computational cost* and *protocol overhead*. The former is due to the more sophisticated and more frequent path selection computations, while the latter is caused by the need to distribute updates on the state of network resources that are of relevance to path selection, e.g., available link bandwidth. Such updates translate into additional network traffic and processing, in particular in the case of *link state* protocols, on which most of the QoS routing proposals currently being put forward are based, e.g., see [6] for an overview. In such a context, it is important to properly assess the magnitude of these costs, so that the weight of the associated benefits can be better evaluated. Such an assessment should strive to both understand fundamental cost components, and provide insight into actual implementations and the trade-offs they involve.

Several recent works have aimed at shedding some light into the costs inherent to QoS routing. In particular, different variations of path pre-computation [7, 8, 9, 10] and path caching [11, 12] have been investigated to explore the possibility of reducing the processing cost of QoS path computation. Similarly, a variety of link cost metrics and update triggering techniques [13, 14] or path selection techniques [14] have been proposed to lower the protocol overhead of QoS routing without significantly affecting its ability to compute efficient paths. However, these works rely primarily on simulations, and as a result are not able to fully capture some of the more implementation specific issues associated with QoS routing. Even the very detailed cost models used in [12, 14, 15], are not sufficient to precisely characterize the processing load that a real QoS capable router will have to support. Therefore, despite the greater insight into the cost and benefits trade-off of QoS routing, its feasibility in IP networks remains a question mark, and resolving it requires additional evidence that only a real implementation can provide.

As a result, the goal of this paper is to fill this gap by providing a detailed report and assessment of a complete implementation of a QoS routing protocol for IP networks. Specifically, we implement a QoS routing algorithm based on extensions to a popular implementation of the Open Shortest Path First (OSPF) [16] routing protocol under the *gated* [17] environment. In addition to describing the design of these extensions and their implementation, we also evaluate their performance under several scenarios. This provides valuable insight into the overall implementation complexity of QoS routing, and the behavior of several of its components. In particular, using this implementation we obtain realistic estimates for the cost of various QoS routing operations such as path computation, link state advertisements generation and reception, and compare the cost of our QoS enhanced version of OSPF to that of the standard OSPF protocol. Finally, the paper also attempts to combine simulation data and the findings based on our implementation, in order to emulate the operation of a router that is part of a large QoS enabled network and get some insight into the amount of load that an “off-the-shelf”, *gated* based QoS router can handle.

This paper is organized as follows: In Section 2, we present background information on OSPF, the QoS extensions we added, and the *gated* program, which is the base of our implementation. Section 3 gives an overview of our implementation and how we have added our QoS extensions to the *gated* base, while Section 4 reports on performance measurements for our implementation. Section 5 summarizes the findings of the paper and identifies a number of extensions and enhancements, which we are currently pursuing.

## 2 Background

### 2.1 Link State Routing and OSPF

The Open Shortest Path First (OSPF) [16] is a well established and widely deployed link state routing protocol that has been an Internet standard for some time. An important characteristic of link state routing protocols is that each router maintains the full topology of the network in a *link state database*, which can be relatively easily augmented to include QoS related link metrics. The database is constructed and updated by means of link state advertisements, that are generated by each router and propagated to all other routers using reliable flooding. This flooding is controlled by having neighboring routers form a logical link between them called an *adjacency*, so that information is flooded only between adjacent routers. Again, link state advertisements can themselves be extended to carry QoS information (see [18]).

The OSPF standard specifies that routers implementing the protocol run shortest path Dijkstra

computation on their local link state database, and determine the shortest path to all other vertices<sup>1</sup> in the network. Vertices in the OSPF topology can correspond to routers, transit networks, or stub networks. The shortest path computation performed by each router creates what is known as the *SPF* (shortest path first) tree. As part of the SPF tree computation, the next hop information for each destination is also derived and used to construct the corresponding routing table entry, i.e., identify where packets are to be sent next based on their destination. OSPF allows multiple equal cost paths to a destination to support load balancing.

In order to handle the scalability problems associated with both flooding and maintaining a complete network link state database, OSPF allows for a two level hierarchy within the routing domain. The routing domain (also called an Autonomous System or AS) can be split into multiple *areas* which are all interconnected (either physically or logically via virtual links) through a special *backbone area*. The use of areas reduces the routing protocol traffic and the size of the link state database maintained at each router, since each router needs to maintain a detailed topology database of only the area to which it belongs and the summarized list of reachable destinations within other areas.

Originally, OSPF specification allowed for Type of Service (TOS) based routing in order to support the five different Types of Service (viz., normal, minimum cost, maximum reliability, maximum throughput, and minimum delay) that were specified for IP datagrams. OSPF enabled TOS-based routing by allowing routers to advertise independent cost metrics for different service types and specifying that routers compute a different SPF tree for each type of service. In spite of this provision, there weren't multiple interoperable OSPF implementations supporting this feature. Neither were there host applications requesting specific TOS in IP datagrams. The semantics of the TOS byte in IP datagrams is being revised by the IETF and lack of deployment experience caused dropping of the TOS-based routing requirement from the OSPF specification.

However, in order to avoid potential backward compatibility problems, not all TOS features have been dropped from OSPF. Routers can indicate their TOS capability during adjacency formation and advertise TOS specific metrics in their link state advertisements although how such metrics are to be used for route computation is left unspecified. This has provided an opportunity to experiment with QoS routing as an extension of the TOS features provided by standard OSPF. There are several other aspects of the OSPF protocol, that are of importance to the QoS extensions we implemented and report on in this paper. In the rest of this section, we review the major ones and attempt to highlight aspects of relevance to QoS extensions.

One aspect of particular significance is the flooding mechanism as it has a direct bearing on the amount of routing protocol traffic, especially when additional QoS information needs to be distributed. The standard OSPF protocol traffic consists of several types of packets, and the interested reader is referred to [16] for details. Here, we briefly review those of most significance in the context of our implementation of QoS extensions to OSPF. *Link State Update* packets contain information about changes in the topology, and are used to carry multiple link state advertisements (LSA). As their name indicates, *Link State Acknowledgment* packets are used to acknowledge receipt of link state advertisements. Finally *Database Description* and *Link State Request* packets are used to synchronize the link state databases of neighboring routers when they form an adjacency.

There are also several types of link state advertisements (again, see [16] for details), with router and network link advertisements being the most relevant ones to our QoS extensions. Router LSAs contain information about a router and **all** its incident interfaces, while network LSAs describe the set of routers attached to a given network. As mentioned before, multiple LSAs can be included in a given link state update packet. Link state advertisements are either generated periodically

---

<sup>1</sup>Throughout this paper we use interchangeably the terms vertex and node.

or are triggered by topology changes such as link failures or recoveries. In the context of our QoS extensions, they will also be triggered by “significant” changes in the metrics, e.g., available bandwidth, of any of the links. Note that because a router LSA advertises the current state of all links on the router, a change in the state of a single link results in the state of all links being updated. We preserved this characteristic in our implementation of update extensions for QoS metrics. On receipt of an LSA, the receiving router acknowledges the LSA with a link state acknowledgment packet. Multiple LSAs can be acknowledged in a single link state acknowledgment packet.

The OSPF standard mandates a variety of constants that control the frequency of the operations related to the flooding of LSAs. Several of them are of particular relevance to the QoS extensions we consider. In particular, the constant *MinLSInterval* specifies the minimum time between any two consecutive originations of a given LSA by a router. The default value of *MinLSInterval* is 5 seconds. This parameter is important in the context of QoS extensions, as it limits the frequency at which changes in link state can be advertised. QoS metrics being more variable than link status, LSAs aimed at updating QoS metrics are likely to be more frequent, and their processing could then be affected (see [14] for an extensive discussion on how to limit the frequency of QoS updates, and the associated impact on the performance of QoS routing). Another similar constant is *MinLSArrival*, which limits the frequency at which new instances of a given LSA can be accepted. If two consecutive instances of an LSA are less than *MinLSArrival* apart, the second is not processed and simply discarded. This constant was added in the last version of OSPF because of problems caused by too frequent retransmissions in some configurations. Its default value has been set to 1 second.

Besides LSAs triggered by state changes, a router also periodically generates LSAs with frequency *LSRefreshTime*, which has a default value of 30 minutes. Finally, recall that OSPF protocol messages are sent as raw IP datagrams and reliability is implemented within OSPF through acknowledgements and retransmissions. The time between LSA retransmissions is determined by *RxmtInterval*, which has a default value of 5 seconds, but needs to be set to well over the round-trip time between neighboring routers. *RxmtInterval* can be set on a per interface basis, while all other constants are common to all interfaces.

As mentioned earlier, of the above constants, *MinLSArrival* and *MinLSInterval* are the ones most likely to interfere with the more frequent updates associated with QoS metrics. Avoiding such problems without weakening the protection provided by these two constants, is one of the issues that needs to be addressed when implementing support for QoS extensions. In the next sub-section, we describe more precisely the nature of these extensions. In particular, we outline the modified path selection used to compute QoS routes, and identify the modifications made to LSAs to support advertisement of QoS metrics.

## 2.2 OSPF Extensions for QoS Support

Our implementation is based on the approach described in [18], which is similar to several other proposals [7, 10] for supporting QoS routing. It is limited to handling requests with bandwidth requirements, and as a result link bandwidth is the only metrics extension that has been implemented. [18] identifies several possible variations for QoS routing extensions, that include on-demand computation and pre-computation of QoS routes as well as both explicit and hop-by-hop routing modes. Our implementation performs path pre-computation and is based on a hop-by-hop routing mode. We chose to implement path pre-computation because of its potentially significant gains in terms of processing load, e.g., see [15]. Similarly, we opted for a hop-by-hop routing mode, simply because it can be accommodated without major changes to RSVP [19], the signaling protocol that we assume is used to request QoS guarantees, e.g., see [20].

Our implementation computes QoS paths using the *widest-shortest* path selection criterion described in [18]. At a router, the algorithm pre-computes paths from the router (the source vertex  $s$ ) to all destinations in the network. For each destination, the algorithm computes paths of all possible bandwidth values, and uses them to build a QoS routing table which is kept separate from the standard OSPF routing table (more on this later). Specifically, for a destination vertex  $d$ , the algorithm generates the set  $S_d = \langle S_d^1, S_d^2, \dots, S_d^n \rangle$ . This set further consists of an ordered list of sets, where the set  $S_d^h$  contains all paths from  $s$  to  $d$  with hop count  $h$  and the maximum bandwidth achievable for that hop count. In cases when an increase in hop count from  $h$  to  $h + 1$  does not improve the available bandwidth, the set  $S_d^{h+1}$  is currently<sup>2</sup> omitted. In other words, the set  $S_d$  maintains candidate paths in increasing order of hop count and available bandwidth.

The QoS routing table generated by the algorithm can then be conceptually viewed as a matrix, with each row associated with a particular destination (entry in the IP routing table), and each column giving the amount of bandwidth available for a given hop count. Each entry (row-column intersection) in the table contains, in addition to the available bandwidth and the hop count, information identifying the next hop on the path(s) to the destination<sup>3</sup>. Note that an entry in the QoS routing table can contain more than one next hop in cases when several paths exist for a given hop count and available bandwidth.

The information in the QoS routing table is used to identify paths capable of satisfying the bandwidth requirements of new requests. This is accomplished by comparing the amount of bandwidth requested by a new flow to the available bandwidth in successive entries in the row associated with the flow's destination. The search stops at the first entry with an available bandwidth larger than the requested value, at which point the corresponding next hop is returned and used to determine where to forward the request next (see [20] for details). If there is more than one next hop, one of them must be chosen. A possible selection criterion is to choose the path at random based on the available bandwidth on the associated local interface. That is, the probability  $p_i$  of forwarding the request to next hop  $n_i$  reachable through interface  $i$  with local available bandwidth  $B_i$ , is set to  $p_i = B_i / \sum_{j=1}^N B_j$ , where  $N$  is the number of possible next hops in the routing table entry. This is the approach used in our implementation as it favors local load balancing. However, it should be noted that this method ignores potential sharing of subsequent links between paths, and hence may have limited effectiveness in certain topologies. While other more sophisticated methods are possible, in the current implementation, we chose to focus on the above simple method.

In addition to the changes required to both the routing table and the path computation, the OSPF protocol and code also needs to be modified to support the propagation of appropriately extended link state advertisements. In particular, information about available bandwidth needs to be added to the link state database and updated through link state advertisements. The format of LSAs is, therefore, extended to carry available bandwidth information. This information is encoded using a new TOS field. The OSPF specification allows a variable number of TOS-metrics to be contained in an LSA. However, only 16 bits are available to advertise the value of the metric. While 16 bits are sufficient for advertising link costs for best-effort routing, advertising bandwidth values for links ranging from few kilobits per second to many terabits per second requires more careful encoding. One such encoding scheme is described in [18] and was used in our implementation.

The proposal of [18] does not specify a number of details such as the mechanisms for triggering updates or guidelines on how often to pre-compute paths. These issues were investigated through

---

<sup>2</sup>We have experimented with a number of variations on the path selection algorithm, that relax the hop count criterion, e.g., keeping paths with higher hop count but the same bandwidth, but they are not incorporated in the current implementation.

<sup>3</sup>As described in [18] this can easily be extended to also provide the information needed to construct a complete explicit route.

simulation in several other works [14, 15], and a number of design choices in the implementation we report on in this paper are based on the findings of those works. However, before we proceed with this discussion, we briefly review some of the basic aspects of `gated` on which our implementation is based.

### 2.3 Gate Daemon (`gated`)

`gated` [17] is a popular, public domain<sup>4</sup> 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 also facilitated implementation of some of the extensions that were required to support QoS routing. These `gated` services include:

- Support for creation and management of timers
- Memory management
- A simple scheduling mechanism
- Interfaces for manipulating the host's routing table and accessing the network
- 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 of the task. `gated`'s main loop consists, among other operations, of a *select()* call over all task sockets to determine if any read/write or error conditions occurred in any of them. Appropriate handlers for a variety of conditions are registered at task creation time and are invoked by `gated`'s main loop. In addition, the main loop also schedules the execution of expired timers and lower priority computations called *jobs*. It should be noted that the scheduling mechanisms of `gated` are low precision, and timers have a maximum resolution of 1 second. Furthermore, scheduling is non-preemptive and operations run to completion. The latter makes it particularly important to ensure that all tasks complete reasonably quickly. This needs to be taken into consideration when choosing a path selection algorithm for QoS routing.

As an example, OSPF is implemented as a single task, associated with a single raw Unix socket. Arriving OSPF packets are processed by dispatching the packet reception handler registered by the OSPF task. Periodic processing such as generation of link state updates is handled with timers while longer processing such as SPF computations are scheduled as jobs. Currently, most of our extensions to support QoS routing have been implemented using timers. These include both generation of LSAs to update QoS metrics, and re-computation of the QoS routing table. This is primarily because we are relying on periodic triggers for both these tasks (see [14] for motivations and evidence supporting these choices).

`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. Our QoS routing table is maintained as a separate table, primarily to avoid conflicts created by prioritization and because of our requirement to keep multiple entries for a given destination as a function of hop count and bandwidth.

---

<sup>4</sup>Access to some of the more recent versions of the `gated` is restricted to the GateD consortium members.

Finally, the OSPF link state database is implemented using a radix tree, for fast access to a particular link state record. In addition, link state records for neighboring network elements (such as adjacent routers) are linked together at the database level with pointers. This makes traversal of the link state database during the shortest path computation easy and efficient, avoiding the need for full database lookups for locating link state records of neighboring network elements. We take advantage of these structures during the computation of the QoS routing table.

## 3 Implementing the QoS Extensions of OSPF

### 3.1 Design Objectives and Limitations

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 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, but this is currently not supported.
- 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.
- Modularity 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, that 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 [19] is the mechanism used to request routes with specific QoS requirements. Such requests are communicated through an interface based on [21], and the RSVP code used was the one created by ISI, version 4.2a2 [22].

In addition, our implementation also relies on several of the simplifying assumptions made in [18], 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., interoperability with non-QoS aware versions of the OSPF protocol is not considered.
- All interfaces on a router are assumed to be QoS capable.

### 3.2 Architecture

The above design decisions and assumptions resulted in the architecture shown in Figure 1. 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 paper we concentrate on the structure and operation of the QoS routing component. As can be seen in Figure 1, the QoS routing extensions are further divided into the following modules:

- **Update trigger module** determines when to advertise local link state updates.
- **Pre-computation trigger module** determines when to perform QoS path pre-computation.
- **Path pre-computation module** computes the QoS routing table based on the QoS specific link state information.
- **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.
- **QoS routing table module** implements the QoS specific routing table, which is maintained independently of the other gated routing table.
- **Tspec mapping module** maps request requirements expressed in the form of RSVP Tspecs and Rspecs into the bandwidth requirements that QoS routing uses.

In the rest of this section, we outline the main functions of each of these modules.

### 3.3 QoS Routing Table and Path Pre-Computation Modules

QoS paths are pre-computed and stored in the QoS routing table as outlined in Section 2.2. However, instead of the “conceptual” matrix format described in that section, each row of the QoS routing table consists of a *path structure* in the form of a linked list. Each entry in the list corresponds to a different hop count and bandwidth value, arranged in increasing order, i.e., an entry for a given hop count is created only if it has a larger bandwidth than previous entries with a smaller hop count. This avoids having to allocate memory for entries associated with hop count values that do not correspond to an increase in bandwidth. In addition to the hop count and bandwidth information, each entry in the list also contains a linked list of next hops associated with the different paths available with the same hop count and bandwidth value. As mentioned earlier, this is used for load balancing between equal cost paths.

Besides the linked list structure of each row of the QoS routing table, another important aspect is the means for accessing the row associated with a path request to a given destination. Specifically, a path request specifies a destination address and a bandwidth requirement, and the first step in identifying a suitable path is to retrieve the path structure associated with the vertex through which the destination is reachable. Such a vertex can be a transit network, a stub network, or the destination host itself. Identifying this vertex is accomplished by reusing the same radix tree

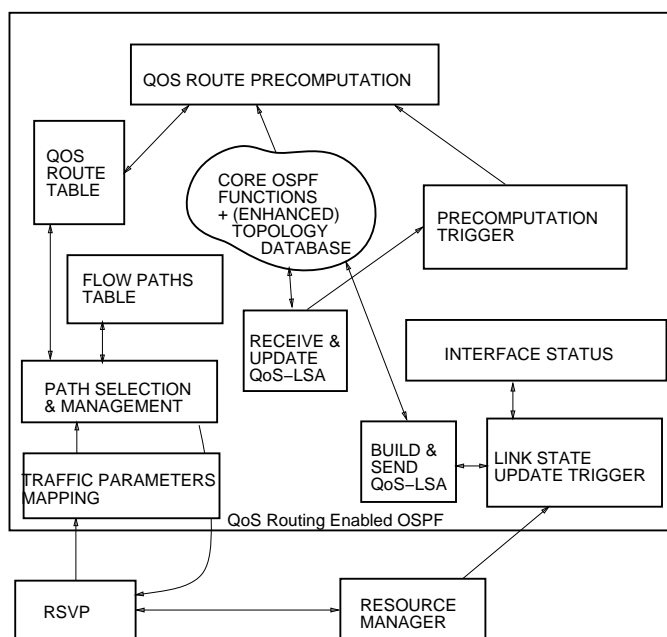


Figure 1: The software architecture

structure used to implement the *gated* routing table itself. The *gated* radix tree structure carries some overhead because it needs to allow coexistence of multiple routing protocols, and as a result a simpler and more efficient data structure could have been devised if we had focused only on the requirements of the QoS routing table. However, we chose to reuse the existing structure to simplify the implementation (no new component) and reduce the development time. As a result, the existing *gated* facility is used to construct a separate radix tree based on the address and reachability information contained in the link state database. Each leaf of the radix tree corresponds to a destination vertex, and a pointer is provided to the QoS path structure associated with this vertex. This pointer is implemented using the *gated tsi* mechanism, that allows the installation of user defined data in a leaf of the radix tree.

Note that the bulk of the QoS routing table is in the path structures describing the different available paths to destination vertices, while the radix tree is used to facilitate access to these path structures. Since the radix tree structure is the same as the one used by the main *gated* routing table, a possible option would have been to share this structure between the two routing tables. We decided to keep the QoS routing table independent of the *gated* routing table to conform to our design goals of localizing changes and minimizing impact on the existing OSPF code. In particular, while it would be possible to attach the QoS path structures directly to the leaves of the radix tree implementing the main *gated* routing table, this could create potential consistency problems as other routing protocols continuously manipulate and modify the *gated* routing table. In addition, the QoS routing table gets re-built independently of the *gated* routing table; a further reason for keeping the two separate. Finally, the *gated* routing table contains routes discovered by all active routing protocols, while the QoS routing table is only associated with our QoS extended version of OSPF.

Another important aspect of the QoS routing table is the overhead involved in building it during the Bellman-Ford computation. During this construction phase, it is necessary to associate the link

state database entities (vertices) that are being expanded by the Bellman-Ford algorithm with the corresponding path structures that contain the paths discovered so far for this vertex. This can be accomplished by using the radix tree of the QoS routing table to search for the address contained in the LSA associated with the vertex being expanded (router id's in the case of a router LSA, and network id's in the case of a network LSA). This provides a general, albeit inefficient solution, as it requires a full lookup in the radix tree each time a vertex is expanded in the Bellman-Ford computation. In order to avoid such a penalty in our implementation, we added a pointer directly to the path structure inside the vertex structure in the link state database. This required a small additional modification to the existing vertex structure, which had to be modified anyhow to support the QoS extensions.

When the Bellman-Ford computation is instantiated, it proceeds from the source vertex and expands vertices as the hop count increases. When a vertex is expanded, the pointer in the vertex structure that points to the vertex path structure is examined. A null value for this pointer is an indication that this is the first time the vertex is being visited. As a result, this triggers both the insertion of the vertex in the radix tree of the QoS routing table and the creation of an initial path structure for this vertex. At this time, the pointer in the vertex structure is updated to point to the newly created path structure for the vertex. In addition, the *ttsi* field in the new (leaf) node inserted in the radix tree with this vertex as destination, is also updated to point to the same path structure. Both pointers are then available for the rest of the path pre-computation phase, and provide immediate access to the existing path structure of any vertex being expanded. Another pointer back to the vertex is also added in the path structure itself, in order to allow decoupling the path structure from the vertex structure, and facilitate porting of the QoS specific extensions to other OSPF implementations. Its main benefit is that it decouples the path structures and the path computation from the exact format used in the link state database. An overview of the resulting QoS routing table structure is shown in Figure 2.

The last issue of significance in the construction of the QoS routing table, is allocation and de-allocation of memory. Currently, when a new QoS routing table is to be computed, both the radix tree and the path structures are freed. This is accomplished by traversing the radix tree in-order, and de-allocating each node in the tree along with any path structure that may be pointed to by the *ttsi* field of the leaf node. 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 because of a different number of paths with distinct hop counts and bandwidth values. However, despite the potential gains, these optimizations have not been included in our initial implementation. The main reason is that they involve subtle and numerous complexities 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.

### 3.4 Update and Pre-computation Trigger Modules

The update trigger module determines when a router floods a new LSA to advertise changes in its link metrics. The pre-computation module is responsible for initiating the computation of a new QoS routing table. In order to allow for experimentation, these two modules support a number of options that can be configured in the implementation.

The **update trigger module** implements:

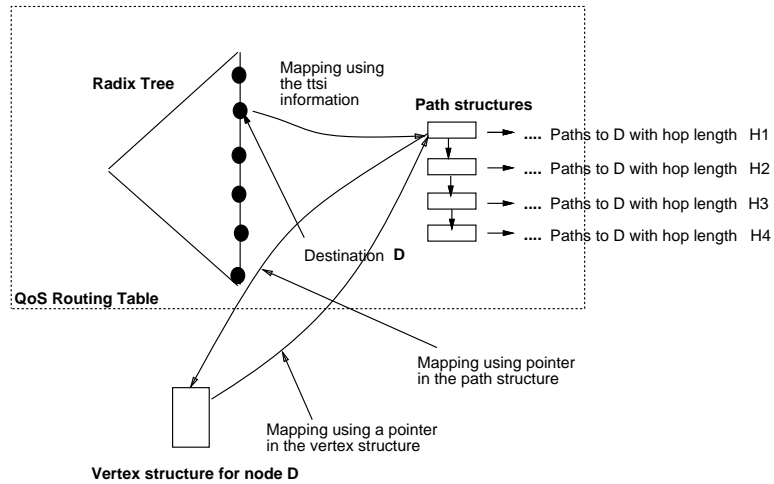


Figure 2: Mapping between different data structures

- A variety of triggering policies. Examples of such policies are: a) Threshold based policy: an update is triggered when the difference between the previously flooded and the current value for available link bandwidth is larger than a configurable threshold. b) Class based policy: split the capacity of a link in a number of classes and advertise when a class boundary is crossed. Classes may have equal or exponentially increasing sizes for larger values of available bandwidth.

- Periodic update generation.

The **pre-computation trigger module** implements:

- Periodic pre-computation.
- Triggered pre-computation each time  $N$  distinct link state advertisement have been received.

To implement this functionality both modules need to:

- Be able to receive notification of events of interest. In particular, the pre-computation module needs to be notified of the arrival of a link state advertisement or a timer expiration event. This can be accomplished by inserting hooks into the OSPF code responsible for receiving and processing LSAs. Similarly, the update triggering module needs to be informed of changes in the available bandwidth on local links. This is needed to ensure that the correct value is sent in the next LSA, and in some instances to determine if an update is needed. This is accomplished through the use of a simple messaging interface, that allows the resource manager to notify the update triggering module of such changes.
- Maintain and post their own timers. Both modules need two types of timers: a) Clamp down timers, that are used to limit the frequency of metrics updates, i.e., the update can only be sent if the timer has expired. Clamp-down timers can be implemented through a combination of a standard timer and a flag. The flag is set when the timer is started and reset when the timer expires. By inspecting the flag the module can then determine if the clamp down timer has expired or not. b) Timers for periodic operation. These are simple timers coupled to the appropriate timer expiration handler.

- Be notified when a local link changes status (up/down). This is accomplished through a small addition to the the OSPF code that handles interface status changes.

Ideally, the operation of the above two modules should be transparent to the existing OSPF mechanisms for managing both path computation and update triggering. While this is easy to accomplish for the path computation component as it operates independently of the regular OSPF path computation, it is harder for update triggering. As was alluded to in the previous section, this is because the regular OSPF update triggering rules can interfere with the triggering policy implemented by the update triggering module. Specifically, the periodic (*RxmtInterval*) and own clamp down timers (*MinLSInterval* and *MinLSArrival*) of OSPF may need to be disabled or bypassed in order to avoid interfering with the generation of QoS LSAs. One option is to disable the existing OSPF clamp-down mechanisms in the case of QoS related LSAs. However, it remains necessary to implement a similar mechanism to ensure stability of the protocol during periods of overload. As a result, we opted for the simple approach of decreasing the value of *MinLSInterval* to allow more frequent QoS updates. Note that since the current resolution of gated timers is in seconds, one second is the smallest possible value we can specify, and it is the one we currently use in the implementation. If in some situations, a smaller value is required, modifications to the Gated timer code will be needed to allow for a finer resolution.

In addition to the above issue, it is also important to inform the QoS update triggering module of any standard LSA generated by OSPF as this affects its book-keeping regarding the last available bandwidth values advertised. This is achieved by implementing the update triggering module so that all LSAs sent by the OSPF protocol are trapped and routed through it. As a result, the update triggering now “sees” all LSAs and can, therefore, control which ones to block, if any, e.g., the OSPF originated periodic updates. For simplicity and given that periodic OSPF updates are originated infrequently (the lifetime of a database entry is 1,800 seconds), the current implementation allows all updates originated by OSPF. Changes in the status of a local interface could be inferred from the associated LSA generated by OSPF, but again for simplicity we opted for an explicit notification of the update triggering module. Note that the pre-computation trigger module does not rely on such an explicit notification mechanism, as it learns about changes in link status from the component responsible for processing incoming LSAs (an LSA is also used to reflect any change of status of a local link in the link state database). This decoupling from local state information also extends to link metrics. In particular, computation of the QoS routing table is done using the metrics information available in the link state database, even for local links for which more accurate metrics information could be obtained. This is important to minimize inconsistencies in the QoS routing tables computed at different routers.

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. Because updates of QoS metrics are likely to increase the amount update traffic, this could cause overflows in the retransmission queues of the sender. Unfortunately, it is difficult to accurately characterize the transmission patterns of LSAs corresponding to QoS updates, so that determining an appropriate value for delaying acknowledgment is also difficult. In addition, the current default value is fairly large (about 20 seconds), which would interfere with QoS updates except maybe when using a periodic trigger mechanism with a large trigger (greater than 20 seconds). As a result, since achieving a meaningful level of aggregation for acknowledgments appears to require a delay value that will most certainly interfere with QoS updates, we chose in our current implementation to bypass this mechanism altogether and immediately acknowledge LSAs received from neighboring routers.

Another approach which we considered but did not implement at this point, 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 more frequent transmission, but more important because the loss of a QoS LSA will not interfere with the base operation of OSPF, and only reduce the quality of paths discovered by QoS routing. We plan on implementing and experimenting with such an option in the future.

Finally, it should be noted that the above discussion has focused on existing OSPF mechanisms that have the potential to interfere with the operation of the extensions needed for QoS routing. An equally and even more important aspect, is to ensure that QoS extensions do not interfere with the proper operation of the base OSPF protocol itself. One such instance is the impact of QoS LSAs on the standard SPF computation. Because such LSAs do not correspond to changes in link status, a new SPF computation is typically not required and should, therefore be avoided. Our implementation addresses this issue through the use of a flag that identifies QoS LSAs and is used to bypass the SPF computation.

### **3.5 Path Selection and Management Module**

This module has two parts, of which only the path selection part has been implemented. Path selection is responsible for handling incoming requests for QoS routes, e.g., triggered by the receipt of an RSVP PATH message. This is done by first using the destination information provided in the request, to search through the radix tree of the QoS routing table. This search identifies the path structure associated with the destination. Recall that the path structure for a destination consists of a list of next hops corresponding to longer, but higher bandwidth paths to the destination. Once the appropriate path structure has been identified, it is scanned until the minimum hop path with an available bandwidth larger than or equal to the amount requested is found. The next hop information, i.e., the associated interface, or one of the next hops if more than one is present, is then returned and used to forward the request onward. When more than one next hop is present, one is selected at random with probability proportional to the amount of available bandwidth on the interface as discussed in Section 2.2. If the search of the path structure terminates without finding a path with sufficient bandwidth, we currently return (one of) the widest available paths (i.e., the corresponding next hop) to the destination, i.e., we do not perform call admission at the time of path selection.

Once a path has been identified, a corresponding entry is created in a flow table, and is associated with the request. The management of this entry in the presence of path or reservation failures (see [20] for details) is the responsibility of the path management part of the module. However, this last feature is not yet supported in the current implementation.

### **3.6 Tspec Mapping Module**

This module simply extracts information from the RSVP Tspec that describes the QoS requirements of a request, and maps it to the QoS model supported by the system, i.e., bandwidth. Currently, we support only a simple mapping where the token rate of the request is used as the bandwidth requirement of the request. Other more sophisticated mappings can be added later without affecting the rest of the system. The integrated services data structures are the same ones used by the ISI RSVP, version 4.2a2.

## 4 Performance Evaluation

### 4.1 Methodology

In this section, we attempt to evaluate the cost of QoS routing, when using our implementation. As discussed in the introduction, we assume that compared to regular Best-Effort routing, QoS routing has benefits for the network and the user, and the primary issue is the potentially greater cost of QoS routing. Consequently, we do not discuss routing performance issues or compare routing performance of QoS and non-QoS routing (see [14] for such a discussion), and concentrate instead on implementation cost comparisons. We explore three different dimensions in our comparisons: a) processing cost, b) message generation and reception cost, and c) memory requirements. For processing cost, we further subdivide it into path pre-computation and path selection costs.

The most comprehensive and accurate method for measuring the cost of QoS routing is to construct a real network with routers running our implementation, and observe and measure their operational performance. Unfortunately, this approach is not practical for any non-trivial size network due to the high cost of building a real large scale network. As a result, we investigate measurement methods that are accurate without requiring a large equipment investment. The first step in that direction is to notice that most of the above costs can be measured individually or as stand-alone operations. For example, the time needed for a single path pre-computation, or the size of the QoS routing table can be estimated based on a single router, whose link state topology database has been populated using some external mechanism. The same holds for measuring the time it takes to select a path. Even the cost of receiving or originating LSAs can be measured reasonably accurately by using only two routers after they form an adjacency and start exchanging LSAs. Thus, it is possible, with a minimum amount of equipment, to obtain good atomic estimates of the cost of all individual operations of interest. We refer to this type of performance measurements as “stand-alone” evaluation mode.

Stand-alone performance results alone are, however, not sufficient to provide a complete assessment of the impact of QoS routing on a router’s operation. This is because, while this accurately estimates the intrinsic cost of QoS related operations, it does not fully capture the many dependencies and interactions that take place in a real operational environment. These affect performance, if only because they determine the frequency and timing of many of those operations, and these parameters are difficult to estimate without a full scale network environment. In order to address this shortcoming of the stand-alone measurement mode, we propose to combine its results with simulations that we use to create the appearance of a large network.

Specifically, we define a simulation environment that allows us to specify an operational network with the following parameters: a) network topology, b) traffic characteristics such as size of requests, arrival rates and distribution of request sources and destinations, and c) choice of path pre-computation and link state update generation trigger policies in the routers. Each of the above parameters is “tuned” based on our previous experience with this simulation environment, so as to correspond to representative and realistic operational conditions. Once a given set of parameters has been fixed, we select one of the simulated routers as a “test-node” and use it to obtain an estimate of the operational behavior of a QoS routing enabled router in the simulated network. This is accomplished as follows:

While performing a simulation run, we generate a log that contains the time at which each of the following operations occurred at the test node: a) generation of an LSA, b) reception of an LSA, c) initiation of path pre-computation and, d) initiation of path selection. The information gathered in the simulation logs is then used to derive operational costs of the test-node router using either one of the following two methods. The first method uses the individual operations

costs derived from the stand-alone experiment to compute a cumulative cost at the test-node. This cumulative cost is obtained by adding individual costs based on the simulation log that identifies all the different operations performed by the router. The cumulative cost can then be divided by the total simulation time to yield an estimate of the router load. The second method uses the timing and operation information contained in the simulation log to drive a real router and observe its behavior. This is actually accomplished using two routers. One used to generate LSAs according to information in the simulation log on reception of LSAs. The second is our test router, and is used to perform all the other operations specified in the test log. The load on this test router is then measured and used to provide another estimate of the impact of QoS routing. Each method provides a different estimate of performance costs, and as a result allows us to cross-validate results. In particular, the first method allows us to estimate the number of operations performed as various parameters such as network size, traffic load, etc., vary, but it assumes an additive relation between them and the growth of operational costs. On the other hand, the second method focuses on capturing possible interactions that can take place within a router as the processing load triggered by external events varies.

Before we proceed with our performance evaluation, it is important to point out that the simulation based method we just described remains an approximation, albeit a reasonable one. First, the simulator (a modified version of MaRS [23] built for previous works) used to derive the operations and timing logs, does not exactly mimic the behavior of the OSPF protocol. The main difference is that OSPF implements two types of LSAs, router and network LSAs, and the simulator assumes that only router LSAs are sent. However, while handling of router and network LSAs do differ, the impact of ignoring network LSAs should be minimal as updates of link bandwidth information are provided only through router LSAs. Such LSAs represent, therefore, the bulk of LSAs when operating a QoS routing enabled domain, with network LSAs being originated only in case of topology changes which we anyhow do not consider in our simulation.

Another discrepancy between the simulator and the real implementation, is the minimum spacing of 1 second we impose between the transmission of two consecutive LSAs. This constraint forces the queuing of updates generated at a faster rate, so that they are properly spaced before being sent to a neighboring router. This affects the arrival patterns of LSAs and is also likely to cause the combination of multiple LSAs into a single OSPF network packet, which lowers their transmission overhead. These effects are not captured in the simulator, which does not impose any minimum spacing between consecutive LSAs, and further assumes that each LSA is transmitted in its own OSPF packet. This can lead to slightly larger estimates for operational costs since the reception of multiple individual LSAs is likely to be more expensive than the reception of a single OSPF packet containing multiple LSAs.

Hardware limitations introduce another source of discrepancy, which is more relevant to the second evaluation method. Specifically, only a single network interface was available for the test machine. As a result, when measuring the utilization of the test router, all the traffic, which in the simulated node came and left through multiple interfaces, is now passing through the same interface. However, we don't expect this to have a major impact. Finally, the machine used as test router is an ordinary Unix based host running gated without any of the software and hardware optimizations, that most commercial routers are equipped with. Nevertheless, despite these shortcomings, we believe that this study provides realistic insight into the processing requirements of "typical" QoS routing loads.

In all experiments, the test systems used are IBM Intellistations with a Pentium Pro 200 MHz processor, 64 Mbytes of real memory, 3.4 Gbytes of disk, running FreeBSD 2.2.5-RELEASE and gated 3.6a.2 software. The Ethernet adapters used in the tests are 10 Mbit/second SMC PCI.

## 4.2 Stand-Alone Cost

We appropriately instrument the implementation in order to measure the cost of each of the QoS specific operations we identified earlier. In particular, we measure the time needed to compute the QoS routing table, the amount of memory this table needs, and the time it takes to select a path from this table. After obtaining these measurements, we contrast them with corresponding ones in a non-QoS routing enabled implementation. In particular, we compare the cost of pre-computing the QoS routing table to that of the OSPF shortest path computation, and the memory requirements of the QoS routing table to those of the standard OSPF routing table. Path selection cost in QoS routing has two components: the radix tree search based on destination address and the linear scanning of the path structure list for a feasible path based on bandwidth requirement. The first of these two components is common to best-effort routing while the second is unique to QoS routing.

It is important to note that QoS routing specific costs depend not only on the network topology, as is the case for the standard OSPF protocol, but also on the distribution of available bandwidth on links. This is because path pre-computation maintains alternate paths with bottleneck bandwidth larger than that of minimum hop paths. As a result, the number of distinct paths to a given destination varies according to the distribution of available bandwidth on network links. A larger number of paths implies a higher path computation time as well as a bigger QoS routing table. In order to report representative figures for the overhead of QoS routing, we therefore need to consider the impact of link bandwidth distribution, and possibly report results for different cases.

One possible approach is to attempt to identify, for a particular network topology, an assignment of link bandwidths, that results in the maximum distinct number of paths being generated when computing the QoS routing table. Unfortunately, such a direct approach appears too complex as identifying such a worst case bandwidth distribution seems to require the enumeration of an exponential number of paths. This makes the problem difficult even in small topologies. As an intermediate solution, we propose to compute both best and average cases for those costs that we know to depend on link bandwidth distribution. Specifically, we obtain the best case for a given topology by defaulting the path pre-computation algorithm to a minimum hop count algorithm. This yields the minimum possible (one) number of paths to each destination. Next, we obtain an estimate of the average case by executing the path pre-computation algorithm for a sequence of random link bandwidth distribution, and averaging the resulting costs.

Comparison between QoS routing enabled and standard versions of the OSPF protocol, require that we also measure the cost of operations that are common to both. In particular, generation and reception of LSAs. The cost of these operations can be measured by considering each of them in isolation. However, there are also dependencies on external factors such as network size and topology. In particular, generation and handling of LSAs typically involves accessing the link state database, and the cost of such an operation can depend on the size and organization of the database. This in turn is likely to depend on network size. We investigate the extent of any such dependency by considering multiple network sizes and topologies.

Finally, in order to perform any of the above measurements, we need to first initialize the internal state of our test routers<sup>5</sup> to the one corresponding to the network topology we are assuming. In particular, this means artificially populating the link state topology database of the routers with the corresponding set of entries. We consider two types of topologies.

The first one is a topology that has been used in a number of previous studies, and is representative of the network topology of a typical Internet service provider in the US. This is the `isp` topology shown in Figure 3(a), where all nodes correspond to router nodes interconnected by point-to-point

---

<sup>5</sup>Recall that we use two.

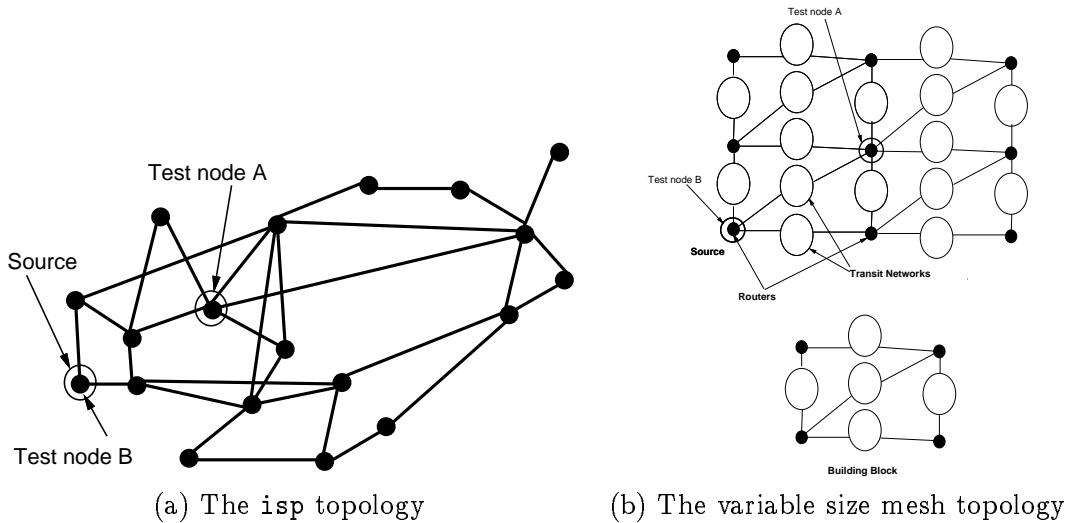


Figure 3: Topologies used in cost measurements

links. In the `isp` topology, the maximum path length for the Bellman-Ford computation was set to 16 hops. The topology is dimensioned for uniform traffic, assuming minimum hop routing, so that link capacities range between 20 and 70 Mbits/second. The second type of topology we consider, is an artificial, mesh like topology that is constructed by repeating a basic building block. The basic building block which consists of 4 routers and 5 transit networks is depicted in Figure 3(b). The small black dots correspond to router nodes, while the white circles indicate transit network nodes in the OSPF topology database. An  $N \times N$  mesh topology is constructed by repeating the building block along two dimensions. Figure 3(b), also illustrates  $2 \times 2$  mesh topology. In this topology, all links are assumed to have a capacity of 45 Mbits/second. In our experiments we use instances of this topology ranging from  $1 \times 1$  to  $10 \times 10$ . For a topology of size  $N \times N$ , the maximum hop limit for the Bellman-Ford computation was set to  $N + 2$ .

#### 4.2.1 QoS Routing Table Computation

Varying the size of the mesh topology allows us to observe how the computation cost varies with network size. A recent survey [24] of vendors who have deployed OSPF in real networks, reports that the figure for the number of routers in one area ranges from 20 to 350 with 100 being the median and 160 being the mean. We show results for networks of up to 400 nodes. The time needed for computing the standard SPF tree and the QoS routing table for the mesh topology are shown in Figure 4. The performance estimate for both the SPF computation and the QoS path pre-computation are based on artificially configuring the link state database of our test router for the different network sizes we assume. In the case of QoS routing, the pre-computation times include the cost of de-allocating the previous QoS routing table, and results are shown for the best and average case. In both cases, the processing cost of QoS path pre-computation is not significantly larger than that of the SPF computation. However, one should remember that the frequency of QoS path pre-computation may be significantly larger than that of the SPF computation, and this is an aspect which we investigate in Section 4.3.

It is interesting to break down the cost of path pre-computation into individual components, as it can help identify items that may be worthy of further optimization, either in the implementation

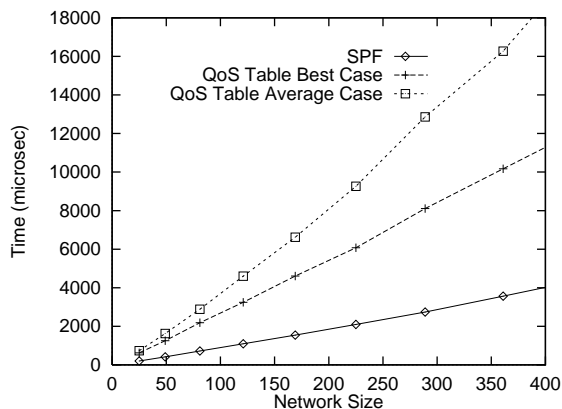


Figure 4: Processing time for path computation

or in the operation of the protocol itself. For that purpose, we use the profiling tool `gprof` to estimate individual operational costs. The tool provides us with information about the total time spent inside a function and its descendants. Based on this, we find that of the overall path pre-computation cost, 37% of the total time is spent in de-allocating the previous QoS routing table. Another 23% is used for accessing the link state database to retrieve information about nodes and their links, for searching through link state records, and for following pointers between link state database entries. The remaining 40% includes the various computations performed by the algorithm as well as management and updates of path structures, and allocation of new path entries. One conclusion that can be drawn from the above numbers, is that given the cost of de-allocating the previous QoS routing table, the memory management optimizations that were previously discussed, appear to have the potential to yield significant performance improvements.

#### 4.2.2 Memory Requirements of the QoS Routing Table

Figure 5 illustrates the differences in memory requirements between the QoS routing table and the standard gated routing table for different network sizes. As before, the mesh topology is used to generate networks of varying size. It should also be noted that although the gated routing table normally contains routes learned from all the active routing protocols, in our experiments only the OSPF protocol was active. As a result, the comparison between routing table sizes is meaningful. Conclusions similar to the ones that could be drawn from the previous comparison of processing costs, apply to memory requirements. Specifically, while the memory requirements for QoS routing are clearly higher, given the cost and availability of memory, the difference is again not extremely significant, e.g., about a factor of 2 for the average case. This difference was to be expected since both tables contain a radix tree of all destinations, and the QoS routing table requires additional storage for the path structures. It is this additional storage that corresponds to the differences shown in Figure 5.

#### 4.2.3 Cost of Path Selection

Path selection consists of accessing the QoS routing table for a given destination and bandwidth value and returning a suitable path (next hop). Accessing the routing table requires first a lookup in the radix tree based on the destination, and second a search of the path structure associated with

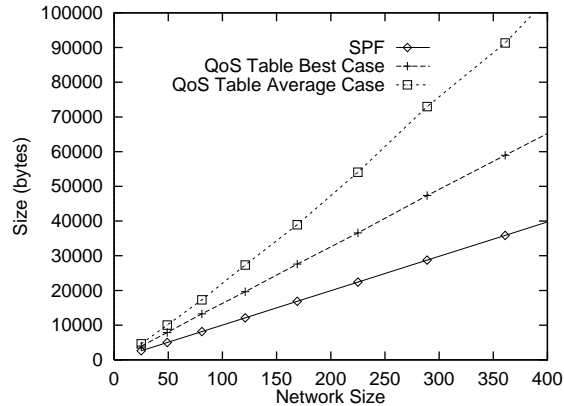


Figure 5: Comparison of memory requirements

the destination until a path capable of satisfying the requested bandwidth is found. The cost of these operations is shown in Figure 6 for both the best and average cases. As mentioned earlier, the best case is obtained by forcing all path structures to only contain one entry, the one corresponding to the minimum hop count. As far as the average case is concerned, there are two dimensions along which averaging can take place. For a given topology and distribution of link bandwidth, averaging can be done based on the destination node and the amount of requested bandwidth. The destination affects not only the cost of the lookup in the radix tree, but also the potential depth of the search in the path structure associated with the destination, as the number of entries in the path structure is likely to differ from destination to destination. The search in the path structure is also affected by the requested bandwidth as large values will typically require stepping through more entries in the path structure. In our measurements, we average based only on destinations, and the average is computed across all possible destinations in the network. Averaging based on bandwidth is avoided by forcing all requests to be for an amount of bandwidth larger than the capacity of the network links. This results in the maximum search time through the path structure, so that we are in effect measuring an average worst case.

Figure 6 shows similar performance for the average and best cases for networks of 50 nodes and for networks of size over 300 nodes. There are a number of reasons for the observed behavior. The first is that the lookup in the radix tree is the dominant component, and since it is identical for both the average and the best case, the differences between the two should be minimal. Another cause is the artificial nature of the mesh topology, which has a large number of minimum hop count paths and few link disjoint paths of higher hop count. This again minimizes any difference between average and best cases. Finally, the limited precision of the profiling tool might also have affected the result to some extent. The main message from Figure 6 is that while the cost of path selection may slightly increase with network size, this sensitivity is small and so is the actual cost of selecting a path, at least in the context of a pre-computed QoS routing table.

We should also note that in a real operational environment, requests will be originated by RSVP and passed to gated through the RSVP-gated interface. Since in this study we want to isolate the costs of QoS routing from other costs of the routing system such as RSVP and interface costs, we simulated the arrival of a request by directly invoking the path selection function.

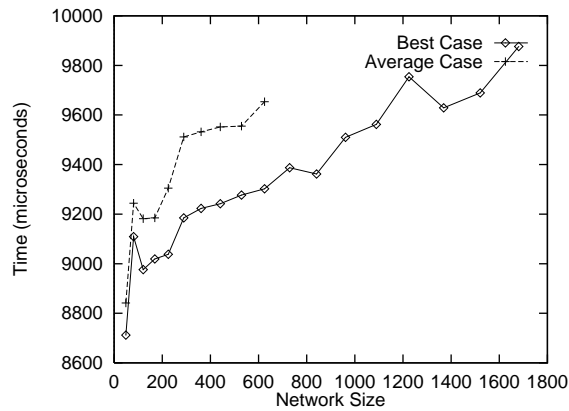


Figure 6: Cost of path selection

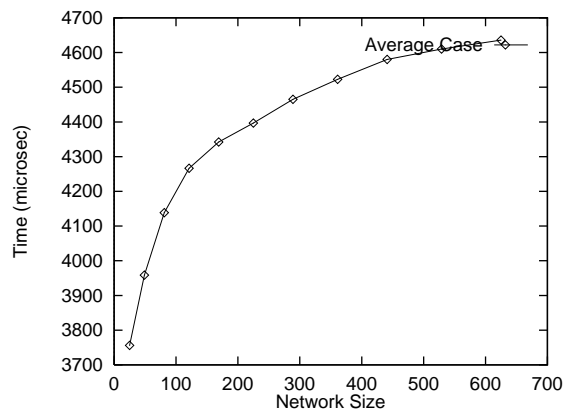


Figure 7: Cost of accessing the link state database

#### 4.2.4 Link State Advertisements Generation and Reception

The last set of parameters, whose costs we want to estimate, are common to the standard and QoS routing versions of OSPF. They consist of the cost of generating and receiving LSAs, that are incurred in both cases. Generating an LSA involves scanning all the active interfaces of the router, constructing the link state advertisement packet, and flooding it over all interfaces. We assumed in our measurement that each LSA was sent in its own packet. Conversely, when an LSA is received, the information about the remote router needs to be entered in the link state database, and therefore a database access needs to be performed.

The time required for generating and receiving LSAs was measured using two similar machines connected to each other, and running `gated` with our QoS routing enabled version of OSPF. The machines were configured so as to form an OSPF adjacency, and then exchange LSAs between them. The OSPF clamp down mechanism based on *MinLSInterval* and *MinLSArrival* was disabled, in order to get accurate measurements of the LSA generation and reception costs.

According to our measurements, either generating or receiving an LSA takes about 200 microseconds in our test system. This is however not a marginal cost when compared to some of

the other costs we measured. The size of link state database does not appear to have a significant impact on the cost of link state update generation and reception. This is likely due in part to the use of an internal radix tree within the database, that provides for efficient access even for large database sizes. This insensitivity of database access to network size is illustrated in Figure 7.

### 4.3 Operational Cost

In this section, we apply the two methods described in Section 4.1 to estimate the actual operational behavior of a router running our QoS routing enabled version of OSPF. Our first task consists, therefore, of simulating a complete network and recording operation and timing logs at a test router in the simulation. These logs are then used to derive estimates of the test router’s performance. In order to obtain sample points representative of different network sizes and topologies, we run two simulations, one for the `isp` topology and the other for an `8 × 8 mesh`. The `isp` topology is characteristic of a typical medium sized ISP network and the `8 × 8 mesh` topology with its 284 nodes provides an example of a fairly large network. In both cases, logs are kept at multiple test nodes to investigate how location might affect our measurements and the router’s performance. The locations of the test nodes for each topology are shown in Figure 3. For both topologies, the main difference between the two test nodes is in terms of number of neighbors. As a result, we expect a slightly higher load at the test node with the greater number of neighbors (test node A).

In all simulations, we generate a workload assuming that requests arrive according to a Poisson distribution, and are independent and uniformly distributed across source-destination pairs. The mean request inter-arrival time at a node is set to 15 seconds, and bandwidth requirements are uniformly distributed between a minimum of 64 Kbits/sec and a maximum of 5 Mbits/sec. The duration of requests is assumed to be exponentially distributed with a mean of 3 minutes. The resulting workload corresponds to reasonably realistic operating conditions of moderate overload, with a small but non-zero blocking probability (less than 5%). In addition to this workload configuration, the simulations also assume specific values for several operational parameters of the protocol. In particular, path pre-computations are performed periodically, for period values of 1 second (the minimum allowed by our implementation, given the available timer precision), 5 seconds, and 50 seconds. Similarly, a threshold based mechanism was chosen to trigger the generation of LSAs, and two different threshold values of 10% and 80% were used. A value of 10% provides very precise link state information but corresponds to a relatively large level of LSA traffic. A value of 80% substantially reduces the amount of LSA traffic, at the cost of greater inaccuracy in link state information. We believe that these combinations of parameter settings provide a reasonably comprehensive coverage of different operational environments.

#### 4.3.1 Router Load

The measured processing load of the test router is reported in Table 1 for test node A in both the `isp` and `mesh` topologies, and for the different combinations of path pre-computation period and link state update threshold mentioned above. In addition to router load, the table also gives, in parentheses, the bandwidth blocking ratio for each combination. This provides some indication on the trade-off between higher processing cost and routing performance. The results were generated using the cumulative cost method described earlier. Similar results were obtained, and are therefore not reported, when the information in the log files was used to drive a real router, although the relatively low load levels made it difficult, in some instances, to achieve precise measurements. Nevertheless, we feel that the strong correlation between the results generated by each method, validates the values reported in Table 1.

	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%)

8 × 8 mesh

Table 1: Router utilization and bandwidth blocking

There are several conclusions that we can draw from Table 1. First and foremost, the 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.

Besides these general conclusions, Table 1 also provides some insight into the specific impact of different protocol parameters. For example, in both topologies, increasing the update threshold from 10% to 80% has the expected effect of reducing the processing load, but this reduction is larger for large pre-computation periods. This is because in these cases the cost of updates is the major cost component. In addition, for large pre-computation periods, increasing the link state update threshold from 10% to 80% reduces the processing load by a factor between 5 and 10, while the corresponding loss in performance, i.e., the increase in blocking probability, is by a factor of less than 2. This means that when operating with large pre-computation periods, increasing the update threshold is a cost effective trade-off. This is in line with the findings of [14]. On the other hand, for small pre-computation periods, processing cost savings are smaller and have the same magnitude as routing performance loss. A similar trade-off can be observed when varying the pre-computation period. Specifically, when the update threshold is at 10%, increasing the pre-computation period yields savings in processing cost of similar magnitude as the routing performance loss. Conversely, with an 80% threshold, an increase in the pre-computation period affords a much larger relative saving in processing cost than the corresponding increase in blocking probability.

### 4.3.2 Bandwidth Requirements of LSAs

The last important cost component is the amount of link bandwidth that is consumed by LSAs. Not only do LSAs contribute to the processing load of the router, but the bandwidth needed to transmit them cannot be used by regular data traffic. This bandwidth consumption can be computed for each link using the timing information contained in the simulation logs and our knowledge of the LSA format. A router LSA in OSPF has a size of  $88 + 16 \times l$  bytes, where  $l$  is the number of links of the originating router, and is acknowledged with a link state acknowledgment packet that has a size of 28 bytes. Both the LSA and the acknowledgment packet need an IP header of 20 bytes since we assume a single OSPF packet per network packet. As we already discussed, we ignore

network LSAs as they are not used to transmit QoS updates, which we expect to be the dominant component of LSA traffic.

As expected, a large update threshold results in lower LSA traffic. For the `isp` topology the average data rate due to LSAs over one of the incident links of test node A, which has a capacity of 70 Mbits/second, is 3112 bytes/second for a 10% threshold and only 177 bytes/second for an 80% threshold. For test node B, which has fewer interfaces and an incident link capacity of 32 Mbits/sec, the corresponding numbers are 2230 bytes/second and 97 bytes/second. Again, the variation due to the location of the test node is relatively small, when considering the potential impact due to the different number of interfaces. More importantly, the bandwidth consumed by LSAs represents only a minute fraction of the link bandwidth in the `isp` topology.

For test node A in the larger mesh topology, we observe a bandwidth consumption of 15438 bytes/second with a 10% threshold, and 1053 bytes/second with an 80% threshold. The numbers are again similar for test node B, with 11677 bytes/second with a 10% threshold and 809 bytes/second with an 80% threshold. This shows that while there is an expected increase in the amount bandwidth consumed by LSA traffic as the network size grows, this increase is moderate and the absolute values remain well within what can be tolerated given the 45 Mbits/second link speeds in the mesh topology.

The above numbers indicate that even with the more frequent updates that QoS routing requires, the bandwidth consumption of protocol traffic should not be of significant concern, at least not for the kind of link bandwidth we have assumed. However, these results represent only a single sample point, and a natural question is whether and how this traffic varies with link speed. For example, if it remains fairly constant as link bandwidth is reduced, it may become a significant overhead over low speed links. Similarly, if it increases much faster than link bandwidth, a similar problem would be present at link speeds higher than the ones we considered. Such scenarios appear unlikely as lower speed links will correspond to lower arrival rates for a given level of blocking, and this will in turn decrease the update rate of QoS LSAs. Similarly, while the benefits of greater statistical multiplexing gains may result in arrival rates increasing slightly faster than link speed for a given blocking probability, we also don't expect this to have a substantial impact on the relative volume of LSA traffic. In order to confirm our expectations, we performed an additional test by doubling the bandwidth of all links in the `isp` topology and increasing the request arrival rate so as to achieve an essentially similar blocking probability. In this set-up, the volume of LSA traffic for the same link of test node A as above was 6844 bytes/second with a 10% threshold and 576 bytes/second with an 80% threshold. This roughly corresponds to a factor of two increase, that is similar to the increase in link bandwidth. This confirms our expectation that the ratio of update traffic to link capacity should remain approximately constant.

## 5 Conclusion

In this paper, we reported on a detailed evaluation of the overhead incurred by extensions needed to support QoS routing in the OSPF protocol. Our investigation was based on a real implementation of those extensions, which showed that the increased processing cost of QoS routing is not excessive, and remains well within the capabilities of medium-range modern processors. In the worst case which we tested, path pre-computation took only 13 milliseconds. In general, we found processor utilization in our test system to be quite low, leaving ample margin and room to operate at even higher frequencies of path pre-computation than the ones we used. In addition, we verified that while LSA generation and reception costs are indeed a major cost component of QoS routing, they remain tolerable even for large networks. More important, bandwidth consumption associated with

LSA traffic was also found to represent only a small fraction of link bandwidth.

In addition to these general conclusions which contribute to establish the feasibility of QoS routing, we also identified several implementation specific issues. In particular, based on our profiling results, we recognized that the area of most potential improvement for our implementation was memory management of the QoS routing table, i.e., memory allocation and de-allocation for the radix tree and the path structures. Such improvements should make the cost of QoS routing even more tolerable. Beyond those improvements, there are also a number of architectural and functional enhancements that are possible, and can further improve the standing of QoS routing. One of them is support for explicit routes as it provides a better control of the paths used by flows with QoS requirements. This would require extensions to the RSVP protocol, and would also impact the path management component. Another, probably more important, enhancement is to extend support for QoS routing in OSPF to more than a single area, i.e., through the backbone network and to remote areas. This could then be further extended to support inter-domain routing as discussed in [6].

## References

- [1] H. Ahmadi, J. S.-C. Chen, and R. Guérin, “Dynamic routing and call control in high speed networks,” in *Proceedings of Workshop Sys. Eng. Traf. Eng., ITC'13*, Copenhagen, Denmark, June 1991.
- [2] Z. Wang and J. Crowcroft, “Quality-of-service routing for supporting multimedia applications,” *IEEE J. Select. Areas Commun.*, vol. 14, no. 7, pp. 1228–1234, September 1996.
- [3] W. C. Lee, M. Hluchyj, and P. Humblet, “Routing subject to quality-of-service constraints in integrated communication networks,” *IEEE Networks*, pp. 46–55, July/August 1995.
- [4] R. Widyonon, “The design and evaluation of routing algorithms for real-time channels,” Technical Report TR-94-024, University of California at Berkeley, June 1994.
- [5] V. P. Kompella, J. C. Pasquale, and G. C. Polyzos, “Two distributed algorithms for the constrained Steiner tree problem,” in *Proceedings of 2nd International Conference on Computer Communication and Networking*, 1993, pp. 343–349.
- [6] E. Crawley, R. Nair, B. Rajagopalan, and H. Sandick, “A framework for QoS-based routing in the Internet,” Internet Draft, QoS Working Group, `draft-ietf-qosr-framework-06.txt`, July 1998, (Work in Progress).
- [7] Q. Ma and P. Steenkiste, “On path selection for traffic with bandwidth guarantees,” in *Proceedings of IEEE International Conference on Network Protocols*, Atlanta, GA, October 1997.
- [8] R. Guérin, A. Orda, and D. Williams, “QoS routing mechanisms and OSPF extensions,” in *Proceedings of GLOBECOM*, Phoenix, AZ, November 1997.
- [9] J.-Y. Le Boudec and T. Przygienda, “A router pre-computation algorithm for integrated services networks,” *Journal of Networks and Systems Managements*, vol. 3, no. 4, pp. 427–449, 1995.
- [10] A. Shaikh, J. Rexford, and K. Shin, “Efficient pre-computation of quality-of-service routes,” in *Proceedings of Workshop on Network and Operating Systems Support for Digital Audio and Video*, Cambridge, United Kingdom, July 1998.

- [11] M. Peyravian and A. D. Kshemkalyani, "Network path caching: Issues, algorithms, and a simulation study," *Computer Communications*, vol. 20, pp. 605–614, 1997.
- [12] G. Apostolopoulos, R. Guérin, S. Kamat, and S. K. Tripathi, "On reducing the processing cost of on-demand QoS path computation," *Journal of High Speed Networks*, 1998, (To appear).
- [13] A. Shaikh, J. Rexford, and K. Shin, "Dynamics of quality-of-service routing with inaccurate link-state information," Technical Report CSE-TR-350-97, University of Michigan, November 1997, (A more recent version -May 1998- is in submission).
- [14] G. Apostolopoulos, R. Guérin, S. Kamat, and S. K. Tripathi, "Quality of service based routing: A performance perspective," in *Proceedings of SIGCOMM*, Vancouver, Ontario, Canada, August 1998, (To appear).
- [15] G. Apostolopoulos and S. K. Tripathi, "On the effectiveness of path pre-computation in reducing the processing cost of on-demand QoS path computation," in *Proceedings of IEEE Symp. Comput. Commun.*, Athens, Greece, June 1998.
- [16] J. Moy, "OSPF Version 2," Request For Comments (Standard) RFC 2178, Internet Engineering Task Force, July 1997.
- [17] "The GateDaemon (GateD) project," Merit GateD Consortium, URL: <http://www.gated.org>.
- [18] R. Guérin, S. Kamat, A. Orda, T. Przygienda, and D. Williams, "QoS routing mechanisms and OSPF extensions," Internet Draft, `draft-guerin-qos-routing-ospf-03.txt`, January 1998, (Work in Progress).
- [19] R. Braden (Ed.), L. Zhang, S. Berson, S. Herzog, and S. Jamin, "Resource reSerVation Protocol (RSVP) version 1, functional specification," Request For Comments (Proposed Standard) RFC 2205, Internet Engineering Task Force, September 1997.
- [20] R. Guérin, S. Kamat, and S. Herzog, "QoS path management with RSVP," in *Proceedings of GLOBECOM*, Phoenix, AZ, November 1997.
- [21] R. Guérin, S. Kamat, and E. Rosen, "Extended RSVP-routing interface," Internet Draft, `draft-guerin-ext-rsvp-routing-intf-00.txt`, July 1997, (Work in Progress).
- [22] "The RSVP project," URL: <http://www.isi.edu/div7/rsvp/>.
- [23] C. Alettinglu, A. U. Shankar, K. Dussa-Zieger, and I. Matta, "Design and implementation of MaRS: A routing testbed," *Journal of Internetworking Research and Experience*, vol. 5, no. 1, pp. 17–41, 1994.
- [24] J. Moy, "OSPF standardization report," Request For Comments (Informational) RFC 2329, Internet Engineering Task Force, April 1998.