Practical QoS Network System with Fault Tolerance

S. Das, M. Gerla, S. S. Lee, G. Pau, K. Yamada, and H. Yu
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095-1596
{shanky, gerla, sslee, gpau, kenshin, heeyeoly@cs.ucla.edu}

***Abstract*- In this paper, we present an "emulation environment" for the design and planning of intranets. Intranets must support Quality-of-Service (QoS) for real-time traffic and must be fault-tolerant for mission-critical applications such as tele-medicine. Our goal is to design the intranet as an overlay network within the IP network and to manage traffic in the intranet so that QoS requirements are satisfied. Routing is a key component of this architecture, in that it enables the efficient use of the provisioned bandwidth. Our approach is to make routers in the intranet capable of finding QoS paths in a distributed manner in response to changing traffic demands. As an extension of QoS routing, each router is able to compute multiple QoS paths to the same destination. The multiple QoS paths are utilized in parallel making the entire network system less prone to network failures. Our emulator permits us to design more reliable intranets and to test various applications for feasibility before deployment. This paper presents the emulation system architecture, the QoS algorithms used for fault tolerance and load balancing, and the relevant experiment results.**

Keywords—**QoS, intranet, multiple paths, fault tolerance**

I. Introduction

THE importance of QoS guarantees in IP networks for the currently emerging and future network applications has been discussed for many years, and substantial research contributions have been made accordingly. Following such a significant research stream, in this paper, we present an enhanced QoS-compliant network testbed that can serve as an intranet design and planning tool.

QoS design tools are of key importance to the Intranet. Intranet technology has been used for enterprise internal networks (i.e., private networks). Mostly, intranets are used to carry the core business information flows of the enterprise. Moreover, they carry a variety of internal multimedia traffic such as VoIP. In order to provide a seamless multimedia traffic support, the underlying network system must guarantee QoS. Network systems for providing QoS guarantees may consist of various building blocks such as *resource assurance*, *service differentiation*, and *QoS routing*. Among these aspects, the main research issue addressed in this paper is QoS routing: the capability of finding network paths satisfying given *multiple QoS constraints* in a given intranet boundary. This is fundamental groundwork for the other QoS components. Pioneering research about the aspect was started by [1], [2]. The latter showed a possible use of the Bellman-Ford algorithm in the link state (OSPF) routing environment for QoS support. These early contributions clearly pointed that a set of multiple QoS constraints (e.g., bandwidth, delay, reliability, etc.) must be simultaneously satisfied for QoS-sensitive applications, forming a body of meaningful specifications for practical approaches [3]. Along with these foundations, [4] showed the practicality of such QoS routing algorithms by applying them to a network environment for IP Telephony. QoS routing issues were further accelerated by the development of the Multiprotocol Label Switching Protocol (MPLS) [5] which deploys fast packet-forwarding mechanisms in IP core networks.

In this paper, we will review the OSPF based QoS architecture based on the above models, and will show how it can be applied to satisfy intranet QoS constraints. Moreover, we will address an additional issue that is often disregarded in conventional QoS routing architectures: robust QoS guarantee in an unreliable, unpredictable network environment [6], [7]. Network reliability is critical for several enterprise services. For instance, if an enterprise deploys an intranet for mission-critical applications (e.g., tele-medicine and distance surgical operations), the applications require not only reasonable QoS guarantees but also transparent reliability/robustness from failures or changes in the underlying network. For this purpose, a novel approach for reliable QoS support was recently introduced in [8], which effectively computes *multiple QoS paths* with minimally overlapped links. By provisioning multiple QoS paths, the network system can provide backup paths when one or more paths are detected as corrupted. Besides, the spreading of network traffic over the provisioned multiple QoS paths favors even network resource utilization. Benefits of provisioning multiple QoS paths are illustrated in [8], [9].

With such a fault-tolerant QoS-compliant architecture as a target, we have deployed a QoS testbed that will assist in the implementation, evaluation, and comparison of various architecture components. In this paper, we present the practical QoS system equipped with an effective QoS routing algorithm for multipath and fault tolerance implementation. In Section II, the core QoS routing algorithms are briefly reviewed. Section III depicts the entire network system architecture. Section IV presents experiment results obtained with the proposed network system and demonstrates its effectiveness in representative traffic scenarios.

II. QoS Algorithms

The network system presented in this paper has QoS routing mechanisms which are ready to serve QoS applications in both conventional and fault-tolerant ways. The mechanisms are associated with a certain call admission control (CAC). When a QoS application comes in and looks for its corresponding QoS services, it consults the underlying QoS routing algorithm (i.e., Q-OSPF with the enhanced routing algorithms in our case) for feasible paths. If no feasible path is found, the connection is rejected and the application exits. Thus, the QoS routing path computation algorithm not only provides the capability of finding QoS paths but also plays an important role in CAC. As previously mentioned and discussed in [8], we adopted the two different QoS routing algorithms in our system: the conventional single path algorithm and the newly introduced multiple path

algorithm.

The single QoS path computation algorithm with multiple QoS constraints derives from the conventional Bellman-Ford algorithm as a breadth-first search algorithm minimizing the hop count and yet satisfying multiple QoS constraints. Each node in the network builds the link state database which contains all the recent link state advertisements from other nodes. With Q-OSPF, the topological database captures dynamically changing QoS information. The link state database accommodates all the QoS conditions, and we define each condition as a QoS metric and each link in the network is assumed to be associated with multiple QoS metrics which are properly measured and flooded by each node. Each of these QoS metrics has its own properties when operated upon in the path computation. The principal purpose of the path computation algorithm is to find the shortest (i.e., min-hop) path among those which have enough resources to satisfy given multiple QoS constraints, rather than the shortest path with respect to another cost metric (e.g., maximizing available bandwidth or minimizing end-to-end delay).

The proposed multiple QoS path algorithm is a heuristic solution. We do not limit ourselves to strictly "path disjoint" solutions. Rather, the algorithm searches for multiple, maximally disjoint paths (i.e., with the least overlap among each other) such that the failure of a link in any of the paths will still leave (with high probability) one or more of the other paths operational. The multiple path computation algorithm can then be derived from the single path computation algorithm with simple modifications. This multiple path computation algorithm produces incrementally a single path at each iteration rather than multiple paths at once. All the previously generated paths are kept into account in the next path computation.

The detailed descriptions of the single and the multiple path algorithms are in the appendix.

III. System Architecture

The testbed consists of PCs running Linux, and all the QoS-capable features are embedded in the Linux kernel. Each of the machines has a few modules running on it, namely the *link emulator*, *forwarding agent*, *metric collector*, *OSPF daemon* and the *application*, and the entire system architecture is depicted in Fig. 1. The following sections describe in more detail each of these modules individually, explaining their functions and their implementation issues.

A. Link Emulator

The testbed that we implement uses the wired Ethernet LAN in the lab, as the physical communication medium. Since we want to emulate several scenarios with widely varying link capacities and propagation delays, we require a network emulator to do the job for us. We have performed some of our experiments with the free tool developed by NASA called NIST Net [10] that emulates all the incoming links on the router that it is installed on.

NIST Net is implemented as a kernel module extension to the Linux operating system and a graphical user interface application. It allows a PC-based router to emulate numerous complex performance scenarios, including tunable packet delay distributions, congestion and background loss, bandwidth limitation, and packet reordering/duplication. The user interface allows users to select and monitor specific traffic streams passing through the router and to apply selected performance "effects" to the IP packets of the stream.

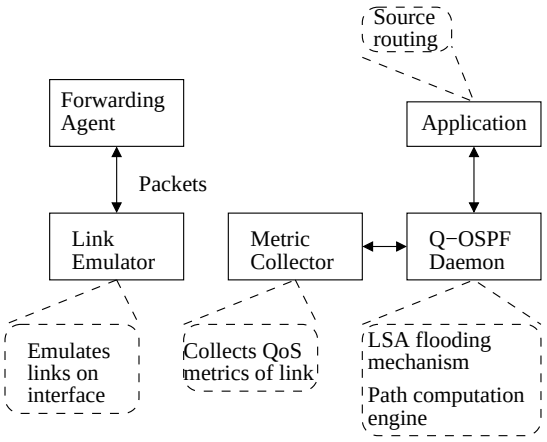

Fig. 1. The entire QoS system architecture for the experiment testbed.

Unfortunately, NIST Net does not have the capability of emulating outgoing links. Therefore, in the current implementation we have NIST Net working on every router and emulating the incoming links on the interfaces. This leads to modifications in the QoS metric manipulation by the Q-OSPF daemon.

After the use of NIST Net and its interaction with the Q-OSPF daemon, for better operational enhancements, we moved on in the latter half of our experiments to `tc`, the Linux kernel traffic shaper and controller. `tc` can emulate a wide variety of policies with hierarchies of filters and class based queues on the outgoing interfaces. We require simple bandwidth and delay emulation which can be done quite easily. This gives us the benefit of being able to emulate on the outgoing interface, and also of being much more stable than it originally was. A simple "`tc qdisc add dev eth0 root tbf rate 10 kbps latency 10 ms`" emulates a 10 Kbps link with a latency of 10 ms on the `eth0` interface.

B. Metric Collector

To provide QoS, reserving bandwidth and bounding on delay for a connection, we require the knowledge of link characteristics at all times. This information is needed by the QoS allocation module which looks at the current usage of the network and figures out if the new request can be satisfied or not. The link metric collection module, therefore, is an integral part of any routing scheme implementation that provides QoS routing.

The OSPFD (OSPF daemon) implementation does not have any link characteristics measurement module. The metrics we are interested in are the available bandwidth in the link and the delay. One could think of other metrics also, such as queue length, loss probability, etc., but bandwidth and delay are the two most important metrics used in QOS routing. Thus, our design goal was to write a module to measure these two metrics and integrate this code with the existing OSPFD implementation.

B.1 Bandwidth Metric Collection

We are interested in measuring the available bandwidth in the outgoing link. A simple way to do that is to flood the link with dummy low priority packets and see how many packets we are able to push in the network. But this is not a practical solution because we have to collect the available bandwidth at low granularity of time intervals. Besides, frequent flooding causes undesirable network overhead. Another approach is to keep count of all the packets leaving the interface. The size of all the packets is the bandwidth consumed, and subtracting it from the maximum bandwidth yields available bandwidth. In fact, since we use application level forwarding, we examine each packet at the application level. Thus, we can calculate bandwidth by just keeping track of how much data we have pushed into each interface. However, finally we opted to simply use the log file maintained in `/proc/net/dev/` which contains information about the number of packets and bytes received and transmitted on each interface. By examining this file at regular intervals, we calculate the bandwidth used on the each outgoing interface.

B.2 Delay Metric Collection

The most common way of measuring delay in the link or a path is to send a packet with TTL equal to one (or number of hops if delay is desired for a path). When the packet reaches the hop where the TTL goes to zero, an ICMP error packet is issued to the destination. The time taken by the ICMP packet to arrive after we have sent the data packet can give the delay estimation of the link. For our implementation, we have used the "ping" utility to send ping probes to the other side of the link and collect the delay value.

The metric collection code sends ping message and collects bandwidth values from the `/proc/net/dev` log file every time interval (typically 10 ms). The values collected are exponentially averaged to smooth out the fluctuations.

C. Q-OSPF Daemon

To propagate QoS metrics among all routers in the domain, we need to use Interior Gateway Protocol (IGP). OSPF is one of the major IGPs and significant researches have been recently made on OSPF with traffic engineering extensions. We selected the open source OSPF daemon (OSPFD) [11] to implement our QoS routing scheme. The release 2.6 of this OSPFD includes the opaque LSA feature. This section explains how we extended the OSPFD code to incorporate our required functionalities. [12] defines Opaque LSA for OSPF nodes to distribute user-specific information. Likewise, we define our specific Opaque LSA entries by assigning new type value in the Opaque LSA format shown in Fig. 2.

When OSPFD runs at routers, it tries to find its neighbor nodes by sending "hello" messages. After establishing neighbor relationship, OSPFD asks the metric measurement module to calculate the QoS metrics of the established link. OSPFD gives as input the local interface address and the neighbor router interface address to the metric measurement module and generates the opaque LSA for each interface. The LSA contains the available bandwidth and queuing delay metric obtained from the metric measurement module. LSA update frequency is currently

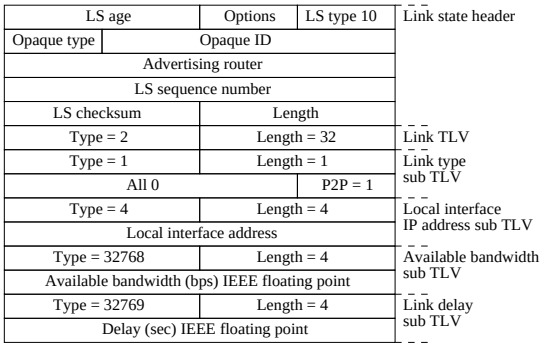

Fig. 2. Opaque LSA format.

restricted to MinLSInterval (5 seconds). In our implementation, we followed this restriction, but we know that the LSA update interval is an important parameter for the correctness of our algorithms because if there are too frequent changes to the generated LSA, there will be miscalculations during the QoS path computation due to the flooding delay.

In addition to LSA flooding, OSPFD exchanges router LSAs to build a full network topology. Router LSAs originate at each router and contain information about all router links such as interface addresses and neighbor addresses (e.g., designated router's router ID precisely because of point-to-point link). We bind the link metrics that we are interested in, viz. bandwidth and delay to the opaque LSA specified by the link interface address. Thus, we introduce the pointer from each link of a router LSA to the corresponding opaque LSA as shown in Fig. 3. Whenever OSPFD receives router LSAs or opaque LSAs, it just updates the link pointer to point to the new traffic metrics reported by that link.

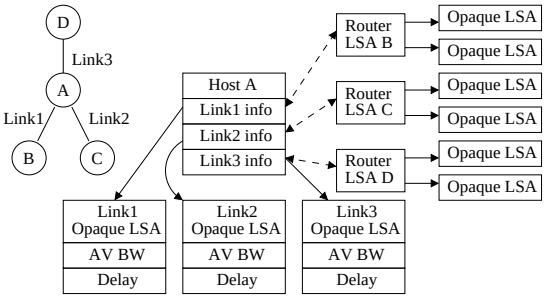

Fig. 3. Link state database.

D. Application

As an integral part to the whole process of making a testbed, implementing Q-OSPF on the routers and testing it was the development of applications running on top of this architecture. These applications request QoS-constrained routes, generate traffic, and in turn change the QoS metrics of the network. We developed two kinds of applications, one of which uses a single QoS path for its packets, and the other of which uses multiple QoS paths and scatters packets over them in a round robin fashion. A logical view of the application module is shown in Fig. 4.

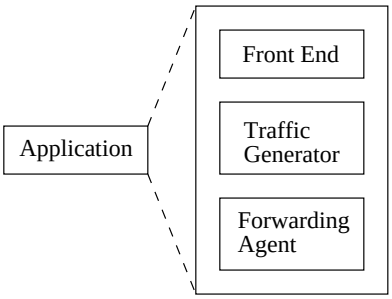

Fig. 4. Application and its internal modules.

D.1 Front End

The front end takes inputs from the user: the destination IP address and the QoS metrics desired for the path (i.e., bandwidth and delay values). With these inputs, the front end queries the Q-OSPF daemon for a route that satisfies the QoS constraints. On receiving no route, the application becomes refused. If a route is found successfully, a "source routed" connection is opened and the control is then passed on to the traffic generator which is given a socket descriptor for the network communication, a traffic generation rate, and the computed path for each outgoing packet.

D.2 Traffic Generator

The traffic generator is a relatively simple module that generates data at the given rate. Constant bit rate UDP traffic is generated at the moment. The generator creates packets of a predetermined size (i.e., 1000 bytes) and sends them out. The traffic generator for the case of multipath is basically the same. It sends packets at the same rate, but the packets are scattered in a round robin fashion over each of the multiple paths.

D.3 Forwarding Agent

To keep things generic, and have a greater degree of control over the testbed, we have moved things up to the application level. Thus, every router has a forwarding agent that looks at every packet and forwards it along the next hop on the source route. This is why the traffic generator stamps the source route, the index and the length of the packet on each packet. Due to fragmentation and aggregation, the forwarding agent has to make sure that it reads the exact length of the packet. The forwarding agent is connected to forwarding agents on its neighbors at the application level using datagram sockets. So depending on which neighbor it has to send the packet to, the forwarding agent merely pushes it out on the corresponding socket.

Of course, a lot of other alternative approaches to source routing exist, and we are in the process of developing a MPLS based forwarding engine, to speed up things further.

IV. Experiments

To evaluate the QoS network system implementation and also to validate the simulation results, we conducted two sets of experiments. The first set of experiments shows the QoS capability of the implemented system. Fig. 5 shows the network topology for the first set of experiments. 4 nodes (i.e., routers) are connected directly to each other through 1.5 Mbps links. The traffic injected into the testbed is CBR (Constant Bit Rate) traffic with 1000-byte UDP packets. The CBR traffic is intended to model real-time video distribution or video conference applications - both requiring QoS support. There are two extra connections creating interfering background traffic: a 1000 Kbps connection from OSPF1 to OSPF4 and a 500 Kbps connection from OSPF2 to OSPF3.

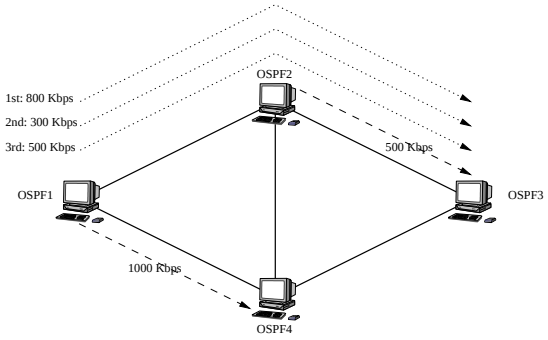

Fig. 5. The first experiment topology and corresponding network traffic.

We introduce three new connections (800, 300, 500 Kbps) from OSPF1 to OSPF3 at staggered starting times. Firstly, we examine the performance of the conventional IP routing which does not provide any QoS capability. In this case, packets of all the connections are routed over the shortest path (OSPF1, OSPF2, OSPF3). Thus, all three connections share the same path and cause network congestion on the link from OSPF2 to OSPF3, while other links are totally empty and unused. This is an inevitable condition due to the lack of the QoS capability in the current IP networks.

Fig. 6 shows the throughput of the connections as a function of time when the conventional IP routing (i.e., min-hop routing) is used. It can be seen from the graph that the throughput are proportionally decreased after the connections are inserted and their traffic exceeds the link capacity between OSPF2 and OSPF3. The throughput of each connection eventually drops to around 70 % of their desired bandwidth. This is an expected result and we can analyze this from the experiment parameters: link capacity over the incoming traffic rate (i.e., 1.5 Mbps / (500 Kbps + 800 Kbps + 300 Kbps + 500 Kbps)) is equal to 0.71.

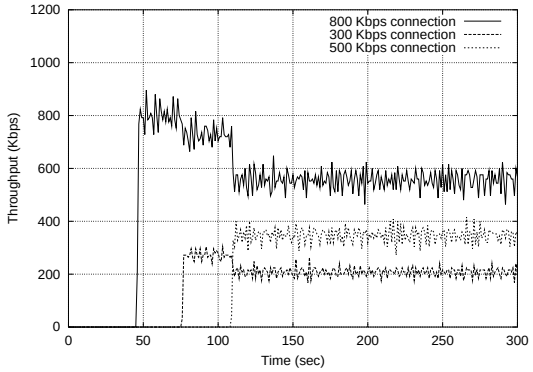

Fig. 6. Connection throughput measured from the system with the conventional IP routing (i.e., min-hop routing).

In contrast, the enabled QoS routing feature in the system

shows the capability of routing the given connections over the paths with sufficient resources. Fig. 7 depicts the path selections for the given connections with the QoS routing.

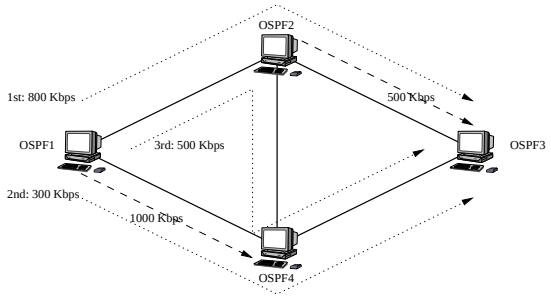

Fig. 7. Path selections for the given network connections with QoS routing.

The path computation process in the extended OSPF daemon of the system computes three different paths as expected for the incoming connections. Fig. 8 shows the throughput of the connections when the QoS routing feature becomes effective. All the connections completely meet their requirements by avoiding congestion. This experiment proves that appropriate paths are selected by the QoS routing capability with given QoS constraints (e.g., bandwidth constraints).

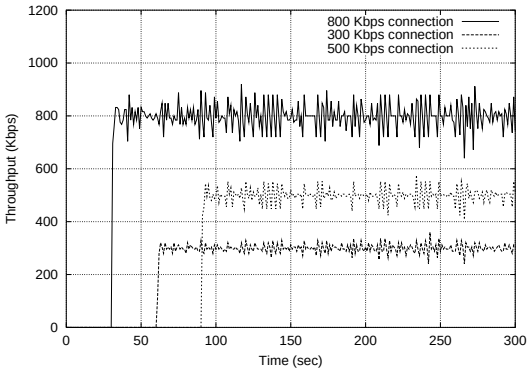

Fig. 8. Connection throughput measured from the system with QoS routing.

The second set of experiments examines the fault tolerance capability by provisioning multiple QoS paths between source and destination and spreading packets over the multiple paths. Fig. 9 and Fig. 10 show the network topology for the second experiments. In the first instance, 6 network nodes (emulated by PCs) were connected to each other through 1.5 Mbps links. We inserted 30 connections generating 100 Kbps CBR traffic each. The source and destination pairs are randomly selected. Random link failures are "forced" during the execution of the experiments. For the link failure model, the state transition of each link from the link-down state to link-up is geometrically distributed with 5 seconds. Likewise, the transition from link-up to link-down is also geometrically distributed. The mean down time is easily derived from the link-down rate. In the second instance, we have the same characteristics, except that we have 9 nodes and 100 connections. We also control the maximum number of multiple paths that a connection can use since that is expected to be an important implementation factor. In the 6 node case, we did not have any choice, since only 2 path options existed. In this case, we have up to 4 multiple paths, so we experimented for MAX-MULTIPLE-PATH for 1 (single path case), 2, and 4.

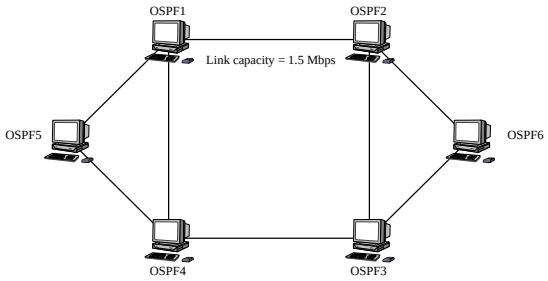

Fig. 9. The network topology for multiple QoS path fault tolerance: 6 nodes.

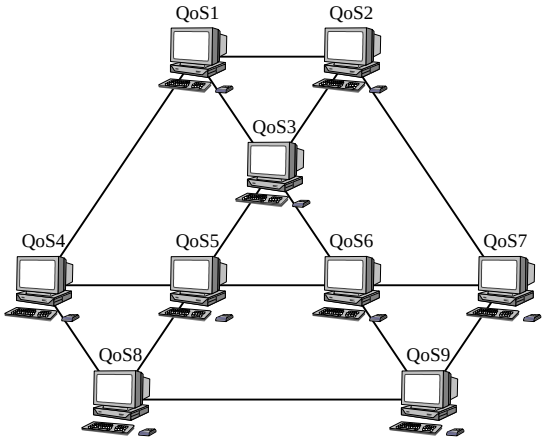

Fig. 10. The network topology for multiple QoS path fault tolerance: 9 nodes.

The performance statistic collect during this experiment is the down time for each connection. When a link on the path goes down, the packets on the connection stop going through. They may, however, flow on a back up path if one were available. When the link comes up, the connection resumes transmission on that link. A connection is defined to be "down" when none of its packets reach destination (i.e., all paths have failed). The duration of the experiments was 10 minutes. Fig. 11 and Fig. 12 show the connection down-time percentage when single path or multiple paths are provisioned for the connections. The result shows that multiple path provisioning has lower connection down rate as expected. Note that these experiments are slightly different from the simulation experiments reported in [8]. In [8], the authors assume that a connection gets aborted when all its paths have one or more faults, and does not come up automatically when the faults get repaired. Note that the difference between the multi-path fault tolerance and the single-path fault tolerance is greater in the 9 node scenario compared to the 6 node scenario. This is to be expected, since in the 9 node topology, there is much greater redundancy, resulting in more multi-paths to be used (around 4) compared to the 6 node case (around 2). In fact, as Fig. 12 shows, the results are similar for 6 nodes and 9 nodes if we constrain the 9 node topology to use just 2 multiple paths. Thus the more multiple paths are available, the better the fault tolerance, so these benefits are expected to even greater for larger networks.

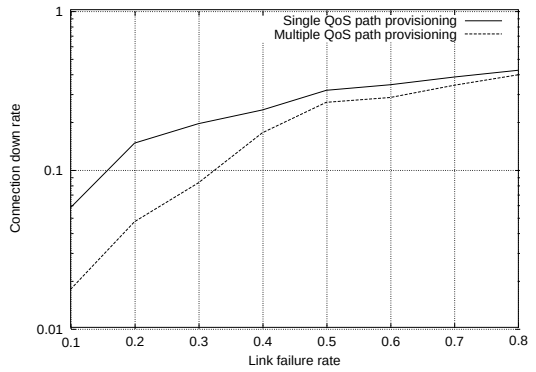

Fig. 11. Connection down rates as a function of link failure rates: 6 nodes

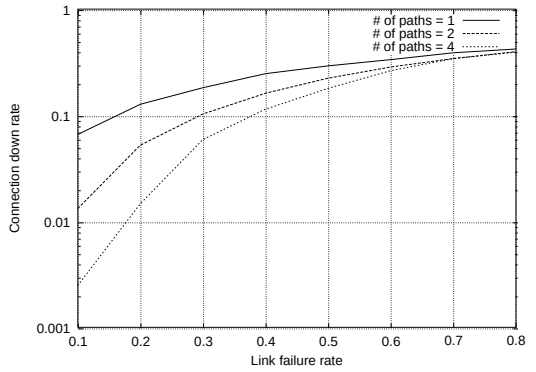

Fig. 12. Connection down rates as a function of link failure rates: 9 nodes

V. Conclusion

This paper described a testbed that has been implemented on Linux based PCs to provide an emulation environment for design and evaluation of intranet protocols and algorithms. We have evaluated various aspects of the Q-OSPF algorithm including throughput satisfaction and fault tolerance support as described in [8]. We have experimented with various topologies ranging from four nodes to nine nodes. The testbed results closely match simulation. They prove that these algorithms can actually be implemented and deployed in an intranet, yielding attractive benefits following the use of techniques like multipath packet scattering.

VI. Future Work

The emulation environment described in the paper is in a very active stage of evolution, with new features tested and incorporated every week. The current goals include the MPLS layer embedding into the kernel and the setting up mechanisms for label assignment, distribution, and signaling, so that applications can transparently use the network. We have completed the first phase of active network monitoring using application layer forwarding. This way, we could verify the accuracy of our design. Next, we will implement MPLS layer forwarding and other techniques that will enhance performance. Another area of future research is the extension of Q-OSPF from the intranet domain to the Global Internet. That is to make it work across domains. This will involve hierarchical routing and we are currently looking into that aspect. As the first step, we plan to deploy hierarchical algorithm on our testbed and check its feasibility and performance enhancements.

References

[1] R. Guerin, A. Orda, and D. Williams, "QoS Routing Mechanisms and OSPF Extensions," in *Proc. of Global Internet (Globecom)*, Phoenix, Arizona, Nov. 1997.

[2] Dirceu Cavendish and Mario Gerla, "Internet QoS Routing using the Bellman-Ford Algorithm," in *IFIP Conference on High Performance Networking*, 1998.

[3] G. Apostolopoulos, S. Kama, D. Williams, R. Guerin, A. Orda, and T. Przygienda, "QoS Routing Mechanisms and OSPF Extensions," Request for Comments 2676, Internet Engineering Task Force, Aug. 1999.

[4] Alex Dubrovsky, Mario Gerla, Scott Seongwook Lee, and Dirceu Cavendish, "Internet QoS Routing with IP Telephony and TCP Traffic," in *Proc. of ICC*, June 2000.

[5] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," Request for Comments 3031, Internet Engineering Task Force, Jan. 2001.

[6] Shigang Chen and Klara Nahrstedt, "An Overview of Quality of Service Routing for Next-Generation High-Speed Networks: Problems and Solutions," vol. 12, no. 6, pp. 64–79, Nov. 1998.

[7] Henning Schulzrinne, "Keynote: Quality of Service - 20 Years Old and Ready to Get a Job?," *Lecture Notes in Computer Science*, vol. 2092, pp. 1, June 2001, International Workshop on Quality of Service (IWQoS).

[8] Scott Seongwook Lee and Mario Gerla, "Fault Tolerance and Load Balancing in QoS Provisioning with Multiple MPLS Paths," *Lecture Notes in Computer Science*, vol. 2092, pp. 155–, 2001, International Workshop on Quality of Service (IWQoS).

[9] Scott Seongwook Lee and Giovanni Pau, "Hierarchical Approach for Low Cost and Fast QoS Provisioning," in *Proc. of IEEE Global Communications Conference (GLOBECOM)*, Nov. 2001.

[10] National Institute of Standards and Technology, "NIST Net," http://www.antd.nist.gov/itg/nistnet.

[11] John T. Moy, "OSPFD Routing Software Resources," http://www.ospf.org.

[12] R. Coltun, "The OSPF Opaque LSA Option," Tech. Rep.

Appendix

I. Single QoS Path Computation

Definition 1: QoS metrics. Consider a network represented by a graph $G = (V, E)$ where V is the set of nodes and E is the set of links. Each link $(i, j) \in E$ is assumed to be associated with R multiple QoS metrics. These QoS metrics are categorized mainly into *additive*, *transitive*, and *multiplicative* ones. Each QoS metric is manipulated by their corresponding concatenation functions, and regardless of the specific properties of the functions, we generalize and compound the functions such that we have $F = \{F_1, \ldots, F_R\}$ since R multiple QoS metrics are assumed to be associated with each link.

Definition 2: QoS descriptor. Along with the individual QoS metrics, QoS descriptor $D(i, j)$ is defined as a set of multiple QoS metrics associated with link (i, j): $D(i, j) = \{q_1(i, j), \ldots, q_R(i, j)\}$. With $D_{q_l}(i, j)$ defined as l^{th} QoS metric in $D(i, j)$, $D(1, j)$ becomes $\{F_l(D_{q_l}(1, i), D_{q_l}(i, j))|1 \leq l \leq R\}$.

The nodes of G are numbered from 1 to n, so $N = \{1, 2, \ldots, n\}$. We suppose without loss of generality that node 1 is the source. $D(i)$ implies the QoS descriptor from the source to node i. Neighbors of node i are expanded with the QoS metrics associated with link (i, j) and $D(j)$ becomes $F(D(i), D(i, j))$ where $j \in N(i)$ and $N(i)$ is the set of neighbors of i. The algorithm iteratively searches for the QoS metrics of all reachable nodes from the source as the hop count increases. In this procedure, the QoS descriptors of the nodes must be checked if they satisfy given QoS constraints. If not satisfying, the nodes

of the non-satisfying descriptors are *pruned* to search only the constraint-satisfying paths.

Definition 3: Constraint verification. A set of multiple QoS constraints is defined as Q and the set is in the same format of D: $Q = \{c_1, \ldots, c_R\}$ such that each QoS metric in D is verified with corresponding constraints. A Boolean function $f_Q(D)$ is also defined to verify if D *satisfies* Q. The verification is to find paths of sufficient resources satisfying all the multiple constraints without considering how sufficiently the individual metrics satisfy the constraints. $f_Q(D) = 1$ if $c_l \in Q, q_l \in D, c_l$ is satisfied by q_l for all $1 \leq l \leq R$. Otherwise, $f_Q(D) = 0$.

Leaving the specific characteristics of individual metrics abstract, we focus only on verifying if the QoS descriptors of each node satisfy the QoS constraints, rather than verifying whether or not the individual QoS metrics are qualified since the QoS metrics all together must lie in the feasible region of the given QoS constraints. When $f_Q(D(i)) = 0$, node i gets *pruned* by the algorithm. Otherwise, it is *projected* and its neighbors are further expanded.

We now define more concretely the single path computation algorithm which iteratively projects reachable nodes with qualified QoS metrics. To determine where the projected nodes pass, we add an extra field in the QoS descriptor to keep track of the preceding node from which the node of the descriptor is expanded, and $D(i)$ becomes $\{q_1(i), \ldots, q_R(i), p\}$. To find the complete path, we can follow the pointers p in the descriptors backward from the destination to the source after the algorithm successfully finds the desired destination.

Definition 4: The single path computation algorithm.
The algorithm starts with the initial state: $T = \{D(1)\}$ and $P = \emptyset$ where T is a temporary set of QoS descriptors and initialized with the QoS descriptor of source node 1, and P is the set which collects all the qualified QoS descriptors of projected nodes. After this initialization, the algorithm runs a loop of steps like the following searching for the destination d.

$h = 0$
while $D(d) \notin P$ **and** $T \neq \emptyset$ **do**
Step 0: $T' = \emptyset$
for each $D(i) \in T$ **do**
if $f_Q(D(i)) = 1$ **then** $T' \leftarrow D(i)$
Step 1: **for each** $D(i) \in T'$ **do**
if $D(i) \notin P$ **then** $P \leftarrow D(i)$
else discard $D(i)$ **from** T'
Step 2: $T = \emptyset$
for each $D(i) \in T'$ **do**
for each $j \in N(i)$ **do**
if $j \neq D_p(i)$ **then** $T \leftarrow F'(D(i), D(i,j))$
else skip j
$h = h + 1$

where h is the hop count. F' is a new generic concatenation function which performs the same operations as F and an additional operation on D_p. When the qualified nodes in T' are projected (i.e., $T \leftarrow F'(D(i), D(i,j))$), $D_p(j)$ becomes i and this is to record the preceding node of each projected node so that the final path can be tracked in reverse from destination node d to source node 1.

Eventually, when the loop ends, P becomes $\{D(i)|f_Q(D(i)) = 1\}$, which means that P contains all the qualified reachable nodes from the source. If the desired destination d has not been included in P, then there is no such a path satisfying the multiple QoS constraints. Otherwise, the final path to the destination can be generated by selecting the precedences $D_p(i)$ of the nodes in P from the destination d. Thus, the shortest path to the destination is found with the multiple QoS constraints all satisfied as long as the destination has been successfully projected.

II. Multiple Path Computation

Definition 5: New or old paths. A new field p is added to the QoS descriptor to determine where the projected nodes pass. Besides, the QoS descriptor is augmented with two new variables, n for "new" and o for "old," to keep track of the degree of being disjoint. These two variables are updated by checking if the node of the QoS descriptor has been already included in any previously computed paths. n increases when the node is not included in any previously computed paths, and o increases when it is detected in those paths. These two variables play the most important role in the multiple path computation algorithm such that a path *maximally disjoint* from previously computed paths can be found. $D(i)$ becomes $\{q_1(i), \ldots, q_R(i), p, n, o\}$.

Definition 6: The multiple path computation algorithm.

$T = \{D(1)\}$
$P = \emptyset$
$h = 0$
while $T \neq \emptyset$ **do**
Step 0: $T' = \emptyset$
for each $D(i) \in T$ **do**
if $f_Q(D(i)) = 1$ **then** $T' \leftarrow D(i)$
Step 1: **for each** $D(i) \in T'$ **do**
if $D(i) \notin P$ **then** $P \leftarrow D(i)$
else if $D'_o(i) > D_o(i)$
or $(D'_o(i) = D_o(i)$ **and** $D'_n(i) > D_n(i))$ **then** $P \leftarrow D(i)$
else discard $D(i)$ **from** T'
Step 2: $T = \emptyset$
for each $D(i) \in T'$ **do**
if $i = d$ **then skip** i
for each $j \in N(i)$ **do**
if $j \neq D_p(i)$ **then** $T \leftarrow F'(D(i), D(i,j))$
else skip j
$h = h + 1$

where T is a temporary set of QoS descriptors and initialized with the QoS descriptor of source node 1, and P is the set which collects all the qualified QoS descriptors of projected nodes. $D'(i)$ in Step 1 means the QoS descriptor of node i which has been already projected and included in P. The algorithm does not stop just after finding the destination but instead keeps investigating all qualified nodes. Since the algorithm does not end on finding d and instead it ends after finding all qualified nodes (i.e., $T = \emptyset$), the neighbors of d are kept from being reached through d such that unnecessary routes going through d are not constructed. The iteration runs over the entire nodes until no further expansion can occur and P becomes to include all qualified nodes. As this comprehensive iteration runs, newer routes to each node are constructed by Step 1. F' is a new function based on F, and it performs the concatenation and records the preceding node in D_p. In addition, F' updates the new two variables in the QoS descriptor, n and o, as discussed in Definition 5. F' checks if the link between i and its neighbor j has been already included in any previously computed paths. If the link is found in the previously computed paths, $D(j)$ expanded from $D(i)$ through the link (i, j) becomes to have $D_o(i) + 1$ for its $D_o(j)$. Otherwise, $D_n(j)$ becomes $D_n(i) + 1$.

Now that o and n are updated in the described way whenever neighbors of nodes are discovered, the sum of their values represents the hop counts to the nodes from 1. The values of o and n are used in Step 1 which determines which

route is newer and more preferable in terms of hop counts. Although the algorithm prefers newer paths compared to previously computed paths, it does not mean that the algorithm finds longer paths. Instead, it looks for a path more disjoint from previously computed paths, but if several new paths exist and their degree of being disjoint are the same, the shortest one among them is selected. This is achieved by the condition in Step 1, **if** $D'_o(i) > D_o(i)$ **or** $(D'_o(i) = D_o(i)$ **and** $D'_n(i) > D_n(i))$ **then** $P \leftarrow D(i)$.

This multiple QoS path computation algorithm searches for maximally disjoint paths through the nodes satisfying given QoS constraints yet minimizing hop counts. This satisfies our requirement for fault tolerance with multiple QoS path provisioning.