The Cost of QoS Support in Edge Devices
An Experimental Study

R. Guérin[1]
guerin@ee.upenn.edu
U. Pennsylvania
200 S. 33rd Street
Philadelphia, PA 19104

L. Li, S. Nadas
lli,nadas@us.ibm.com
IBM Corp., Netw. Hdw. Div.
P.O. Box 12195
Research Triangle Park, NC 27709

P. Pan
pingpan@lucent.com
Lucent, Bell Labs,
101 Crawfords Corner Rd.
Holmdel, NJ 07733

V. Peris
vperis@watson.ibm.com
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Abstract—**This paper investigates the problem of making QoS guarantees available in access devices such as edge routers, that are commonly deployed in today's IP networks. In the paper, we propose a specific design which we evaluate by carrying out a complete implementation, whose performance we then measure in the context of an experimental testbed. Our results indicate that a reasonable level of service differentiation, i.e., rate and delay guarantees, can be provided with a minimal impact on the raw packet forwarding performance of edge devices.**

[1] This work was done while with the IBM T.J. Watson Research Center.

I. Introduction

Quality-of-Service (QoS) is one of the next challenges that the Internet faces, and there are numerous standard activities and new technologies that are being developed to help QoS become a reality. Most of the work so far has focused on developing mechanisms and algorithms that scale to the ever increasing speed of the Internet backbone, while enabling a wide range of QoS guarantees. These efforts have been successful at removing most of the technical hurdles to making the Internet backbone QoS capable. However, introducing the capabilities required to support QoS in the Internet infrastructure represents only half the problem. Another key component is to enable users and applications to access these new capabilities. That this is in itself a difficult task is by now well understood, and has often been quoted as one of the main reasons for the relatively slow deployment of QoS.

In particular, many users and applications lack the ability, or understanding, or both, to determine the exact level of QoS they need and should require from the network. Even assuming the ubiquity of a signalling protocol such as RSVP [1], which is by now becoming available as part of most operating systems, it is unlikely that many applications will be capable of leveraging this new capability, at least not initially. Instead, appropriately mapping user traffic onto available network QoS services is likely to be the responsibility of edge devices, which will be configured according to various administrative, policy, and user specific criteria. In addition, even as users and applications become more QoS aware and capable of specifying individual requirements, it is likely that for scalability purposes individual requests will be aggregated before being forwarded into the Internet backbone. This indeed is the model underlying the recent Diff-Serv standardization effort in the IETF [2], [3], [4], and explicitly outlined in [5], [6]. Edge devices are again the natural place for such a function as illustrated in Figure 1, that describes a likely scenario for deployment of QoS over IP networks. As a result, we expect edge devices to represent a key component in the deployment of QoS capabilities in IP networks. But they also have the potential for becoming a major obstacle, unless

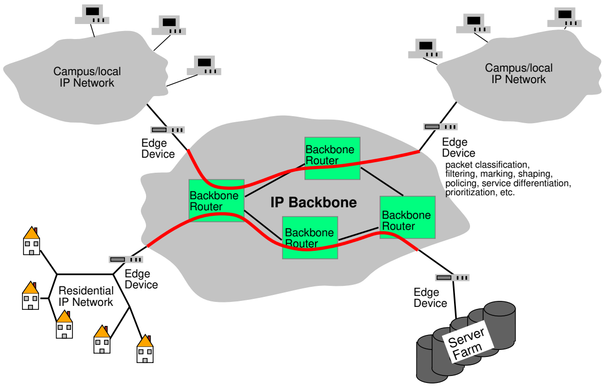


Fig. 1. Scenario for QoS Deployment Over IP Networks.

QoS enhancements can be introduced incrementally on the large installed base of edge devices.

The goal of this paper is, therefore, to investigate issues related to the role of edge devices in enabling service differentiation over the Internet, and in particular the feasibility of upgrading existing systems. Our focus is on mechanisms that can be easily introduced to support QoS in the relatively low-end edge devices that are deployed today. In particular, while performance requirements mandate the use of dedicated hardware to support QoS in backbone devices, this is typically not feasible for edge devices. Instead, QoS support in those devices is often software based, not only because of the low cost point of such devices, but also because of the need for flexibility and upgradeability. Indeed, given the evolving nature of the standards, e.g., Diff-Serv, such characteristics are desirable if not mandatory. An important issue in the context of a software based solution is that the greater path length associated with the additional instructions required by QoS, can affect the raw device throughput.

In the paper, we report on an investigation of a software implementation for QoS support in a typical edge device. To properly assess the cost and capabilities of such a software based solution, the implementation is carried out on a fully operational access router platform. The modifications to the router code are made with flexibility in mind so that as the standards progress, the implementation can be evolved to accommodate different models for QoS guarantees. A major emphasis of the implementation is to minimize the number of additional instructions required by QoS extensions, so as to introduce the smallest possible performance penalty. In order to assess the magnitude of the overhead due to QoS support, we compare the raw through-

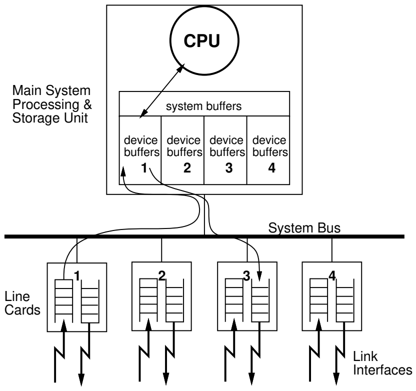


Fig. 2. System Architecture Overview.

put of our router platform with and without QoS extensions, and as a result verify that the approach chosen avoids any significant penalty. In addition, we also establish that basic performance guarantees, e.g., rate guarantees, are met even in the presence of badly misbehaving users.

The results provide some insight and initial evidence of the feasibility of delivering relatively comprehensive QoS guarantees in simple edge devices, with minimal impact on their raw performance. This should help accelerate the deployment of QoS capabilities in IP networks as it implies that they can be made available incrementally, i.e., via a software upgrade, to users of such networks, and with minimal impact on their basic performance. There are clearly many possible enhancements to the implementation we describe, and in the paper we point several of them out. Unfortunately, we expect some of these enhancements to come at a cost, i.e., a degradation in throughput, but, as yet, we have not been able to assess their magnitude.

The rest of this paper is structured as follows: In Section II we outline the overall structure of the system on which our implementation is built, and identify its basic characteristics. Section III describes the different services supported in our implementation as well as the components responsible for ensuring them. It also highlights our design goals. In Section IV, we briefly review our test setup and the methodology we use to obtain our performance estimate. Section V reports on the results of our tests and measurements and discusses their implications. Finally, Section VI summarizes our findings.

II. System Structure Overview

In this section, we briefly describe the overall structure of the router platform on which our implementation is based, and also point to some of the constraints it introduces. Some of these constraints are specific to the platform, but several of them are generic and likely to be present in many edge devices.

Our edge device has an architecture often seen in access routers and relatively common among first generation routers. It consists of a central processing unit responsible for all packet forwarding and classification functions, to which a number of link adapters are connected. An overview of the system structure is shown in Figure 2. Incoming packets are temporarily stored in buffers on the link adapters, before being transferred across the system bus into the main system memory. Once in system memory, packets are processed and this processing includes both forwarding and classification decisions. The processes of interest in the context of this paper are those associated with classification as this is where service differentiation is enforced. The central processor is also responsible for queueing packets for transmission on the output devices.

The data path followed by a packet is, therefore, as follows: An incoming packet is first received in a buffer in the device associated with the link on which the packet is arriving. Fully received packets are then DMA'ed into system memory, where memory space has been allocated for each input device. It should be noted that for a number of implementation specific reasons that have mostly to do with minimizing the overhead associated with buffer manipulations, packet buffers all have the same size. As a result, memory consumption is sensitive to the number of packets rather than to packet sizes, i.e., many small packets use more memory than a few large ones. This is an important design constraint when it comes to providing service guarantees as it affects how we need to perform memory allocation. In particular, consumption of CPU cycles and memory needs to be accounted for in packets/sec, while bytes/sec is the relevant unit when it comes to controlling bandwidth usage.

Once in system memory, the packet is ready to be processed. The processing performed on a packet consists first of an IP route lookup, that returns the next hop on which the packet needs to be forwarded as well as information needed to construct the appropriate link header for the outgoing packet. In addition to identifying the appropriate next hop for the packet, the processing also includes classification of the packet in order to determine the level of service to which it is entitled. Classification is in itself a potentially complex function that deserves an extensive discussion. It is, however, beyond the scope of this paper whose focus is primarily on the mechanisms used to enforce service differentiation. As a result, we only briefly review the issue of packet classification, and outline the general mechanisms available in our router platform to support it.

Packet classification requires matching a number of attributes of the incoming packet, against values whose combination is associated with rules that determine how to handle the packet. The complexity of this matching operation depends on the number of attributes to be matched. For example, classification based on only the DS byte [4] is straightforward, and one of the motivations behind the Diff-Serv effort. However, edge devices are usually required to identify the level of service to which a packet is entitled on the basis of more extensive attributes such as source and destination addresses as well as port numbers, protocol type, ingress and egress interfaces, and even possibly additional attributes such as time of day. Once this information has been retrieved, it can be used to select the proper DS byte value for the packet, which can then be used to classify the packet by subsequent routers.

It is possible to devise algorithms capable of efficiently performing the full lookups required in edge devices, e.g., see [7], [8], but they usually require dedicated support (processor or ASIC) and are unlikely to be feasible in the kind of low end edge devices we consider. As a result, the approach we rely on for classification in our router follows the traditional cache-based

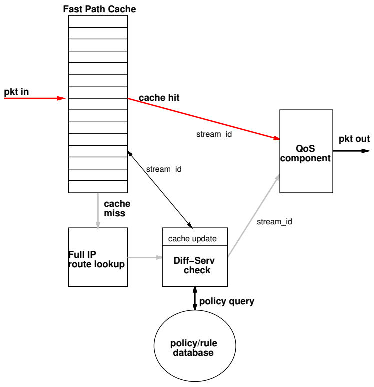


Fig. 3. System Structure

solution used to improve forwarding performance in many first generation routers. In other words, the data path is split between a fast path and a slow path. For every packet sent on the slow path, a full lookup is performed into a complete rule database and used to create an entry specific to this packet (flow[1]) in the fast path cache. Entries in the fast path are accessed based on an exact match on packet attributes, so that the lookup can subsequently be performed (for packets from the same flow) using a simple hash function.

There are many design aspects related to ensuring the efficiency of such an approach, but for the purpose of this paper it suffices to know that whether obtained from the fast path cache or the complete rule database searched in the slow path, information is retrieved that identifies the appropriate service level of each packet. This information is in the form of a stream_id[2] that is passed as a handle to the QoS component responsible for enforcing service differentiation. The overall structure of the system is shown in Figure 3, and in the rest of this paper we concentrate on the operation and performance of the QoS component.

III. QoS Component and Service Characteristics

The main design criterion behind our QoS components is to minimize any increase in the path length of the main forwarding loop, while at the same time allowing basic service guarantees. The service guarantees we target in our implementation are relatively primitive, and inspired from the base service models currently being defined in the IETF Differentiated Service group [2]. These service guarantees are along the two main dimensions of delay and rate guarantees.

Specifically, we first consider a service that aims at emulating a virtual leased line. Its characteristics are of bounded incoming traffic that needs to be guaranteed a given rate as well as small latency. This is essentially a service that can be built using the expedited forwarding (EF) per-hop-behavior (PHB) of [9], which is akin to the ATM CBR service [10] and shares with it strict performance guarantees together with an inflexible service definition, i.e., no excess traffic. A second service which we consider is based on the Assured Forwarding (AF) PHB [11], for which we provide rate guarantees but with looser delay bounds. The main feature of this service is that the rate guarantee corresponds to a floor guarantee, and a stream is allowed to send at a higher rate and access idle resources to the extent they are available.

In our implementation, both services are supported with the same set of basic mechanisms, but are kept isolated from each other. As mentioned earlier, our main goal is to enable service differentiation with the minimal possible impact on raw forwarding performance. As a result, complex per packet processing operations should be avoided, and this limits our ability to use sophisticated scheduling algorithms. This is not so much because of the complexity of the scheduling algorithm itself, although this certainly needs to be considered, but mostly because of the cost of the sorting operation required each time a packet is transmitted. This constraint combined with the need to provide tighter delay guarantees as part of the EF PHB, led us to a solution where we rely on only two queues. EF and AF packets are assigned to separate queues, and the two queues are served using a simple variation of self-clocked fair queueing (SCFQ) [12]. Because we only have two queues, there is no sorting cost associated with identifying the next packet to transmit as this can be achieved through a simple comparison of the transmission times associated with each queue.

The remaining cost of scheduling is also minimal as the computations performed to update the scheduler are of low complexity. Specifically, the operation of the scheduler arbitrating between the two queues is as follows. Each queue is assigned a scheduling weight or rate, which is based on the fraction of link bandwidth allocated to the service mapped onto it, i.e., R_{EF} and $R_{AF/BE}$ for the EF and AF (and Best Effort) queues, respectively. The scheduler maintains a system virtual time, T_S, as well as service tags T_{EF} and $T_{AF/BE}$ for each queue, and schedules for transmission a packet from the queue with the smallest service tag. The service tag of a queue is updated each time a packet moves to the head of the queue, i.e., after transmission of the previous packet or when a packet arrives to an empty queue, and it is set to the system virtual time plus the transmission time of the new packet at the rate allocated to the queue. For instance when a packet of size L_k is transmitted from the EF queue, we have $T_{EF} = T_S + L_k/R_{EF}$. The system virtual time T_S is also updated to the service tag of the queue being served each time a new packet transmission starts. The system virtual time is reset to 0 each time both queues are empty. However, in order to avoid overflow of the system virtual time in the case of a heavily loaded system, we also update it when the most significant bit of the smallest service tag is 1. Specifically, whenever this condition is verified, we then set the most significant bit of both service tags and of the system virtual time to 0. This avoids wrap-around problems when a heavy load prevents the system from emptying and, therefore, resetting the virtual system time.

The use of a separate queue for EF packets, ensures that they see relatively small delays. This is because EF streams are rate limited, and the EF queue is guaranteed a service rate higher than the aggregate rate of all incoming streams; queue build-ups

[1] In this paper, we use the notation flow to denote the set of packets associated with a specific combination of attributes used to create a cache entry.

[2] A stream is the unit of resource allocation, and there can be many flows associated with a given stream and having, therefore, the same stream_id.

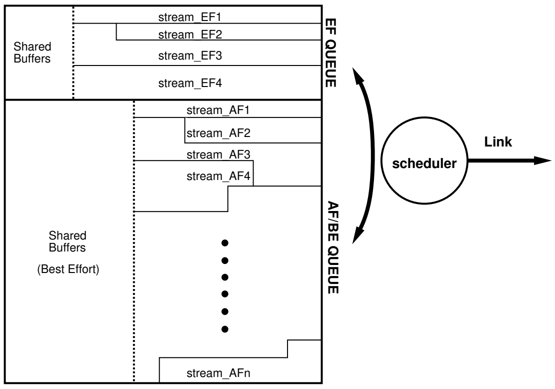


Fig. 4. QoS Support for Rate and Delay Guarantees.

are then unlikely to occur under such conditions. As mentioned before, this is a well understood approach to controlling delay, e.g., ATM CBR, and one which has been extensively studied in the context of providing real-time service on packet networks. In this paper, our goal is to verify that it can provide delay differentiation adequate for delay sensitive applications such as IP telephony, even in the context of a low-end edge device.

The second queue is shared by AF packets, network control traffic, and, as indicated above, traditional best effort traffic. This queue is denoted the AF/BE queue in the rest of this paper. Rate guarantees are provided within that queue using the buffer management method of [13], where a stream is allocated an amount of buffer proportional to the fraction of link bandwidth it is entitled to. The main benefits of this method is that rate guarantees can be provided to individual streams without incurring the complexity of a scheduler. Rate guarantees are provided simply by controlling which packets to accept in the buffer, which is a check which we need to perform in any case[3] and has minimal overhead. The same approach is also used in the EF queue to protect against potential misbehaving streams, and we proceed next to describe in more detail the operation of the buffer management mechanism.

A. Rate Guarantees Through Buffer Management

Each egress interface is configured with (logically as it is only for accounting purposes) a maximum number of system buffers that can be queued on it. This buffer pool is then divided into two separate sub-pools; one for the EF queue and one for the AF/BE queue. We denote their sizes as $\mathcal{B}_{EF}$ and $\mathcal{B}_{AF/BE}$. Rate guarantees are provided to individual streams within each queue by allocating a specific amount of buffer to each stream. For example, the amount of buffer B_k allocated to stream k in the AF queue to guarantee it a minimum rate of r_k is:

$$B_k = \mathcal{B}_{AF/BE} \times \frac{r_k}{R_{AF/BE}} \qquad (1)$$

The main intuition behind the above allocation is that transmission opportunities and, therefore, rate guarantees are in proportion to the buffer space occupied by a flow. In other words, if a flow consistently occupies a certain fraction of the buffer space, it will then get a corresponding fraction of the transmission opportunities, and hence of the link bandwidth (see [13] for a rigorous justification of this argument).

Based on the above allocation, the decision to admit or reject packet j from stream k is a function of the packet length L_j, the current buffer occupancy b_k, and the buffer allocation B_k. If $b_k + L_j \leq B_k$, then the packet is admitted, enqueued for transmission in the corresponding queue (packet transmissions are FIFO within both the EF and AF/BE queues), and the buffer occupancy b_k is incremented by L_j. If on the other hand $b_k +$ $|_j > B_k$, stream k is already using more than its share of buffers. Therefore, accepting the new packet should only be done if it does not affect other streams sharing its queue. This check is based on two criteria. First and foremost, accepting the packet should not impact the rate guarantees of other streams. Second, excess resources should be distributed "fairly" across competing streams.

To realize these two goals, the buffer pool of each queue is logically partitioned into allocated buffers and shared buffers. Allocated buffers are the sum of the buffers allocated to all streams in the queue, while shared buffers represent the remainder obtained by subtracting this amount from the total buffer pool of the queue. Excess packets can only be accommodated in shared buffers. Furthermore, to ensure fairness in the usage of shared buffers, we use the "holes" method of [13] which is based on [14]. With this approach, an excess packet is accepted only if the resulting number of shared buffers occupied by its stream does not exceed the current number of remaining free shared buffers. For example, assume two streams, s_1 and s_2, which have been allocated 20% and 50% of the link bandwidth, and therefore have buffer allocations of $B_1 = 0.2B$ and $B_2 = 0.5B$, where B is the total buffer size. The size of the shared buffers pool is then $0.3B$, of which each stream can use at most $0.15B$. Furthermore, if both streams are each using $0.1B$ of shared buffers, i.e., $b_1 = 0.3B$ and $b_2 = 0.5B$, neither of them can then grab an additional shared buffer, although $0.1B$ shared buffers remain available. There is, therefore, some inefficiency in the use of shared buffers, but it rapidly diminishes as the number of streams grows, and is the price paid for this simple enforcement of fairness.

Once an excess packet is accepted, the buffer occupancy of its stream is incremented and the shared buffer count is decremented. Buffer counts are also updated at packet transmission times, together with the shared buffer count whenever the stream to which the departed packet belonged had a buffer occupancy above its allocation. The latter ensures that buffers are preferentially released to the shared buffer pool. The overall structure of the QoS component is shown in Figure 4, which also shows that in the current implementation the BE traffic is only allowed to access shared buffers. This is the configuration we assume in our experiments, but it could easily be modified to provide a minimum rate guarantee to BE packets.

For the sake of clarity, the above discussion glossed over a number of details related to the exact update of buffer counts as well as the impact of discrepancies between byte counts and packet/buffer counts. In particular, rate guarantees based on buffer allocation require that buffer accounting be based on the

[3] As illustrated in [13], scheduling without buffer management has little or no effect.

number of bytes that a stream currently has waiting for transmission. This is because bytes are the units of relevance when it comes to consumption of link bandwidth. On the other hand, as was mentioned earlier, implementation limitations in our system impose a fixed buffer size independent of packet size. This introduces some additional problems when it comes to accounting for the amount of buffer allocated and used by a stream.

Specifically, an allocation and accounting which assumes that each buffer corresponds to its byte equivalent in terms of available storage space, can be overly optimistic. In particular, it allows rate-wise conformant streams to grab an excessive number of packet buffers, if they only transmit small packets. This will in turn deplete the buffer pool, so that packet buffers are unavailable to other conformant streams. Alternatively, assuming a worst case scenario where each buffer is only used to store a minimal size packet, is overly pessimistic. It would result in rejecting requests for rate guarantees because of what would be (incorrectly) perceived as insufficient buffer space. There is no ideal solution to this problem, as it requires identifying an appropriate trade-off between the distribution of packet sizes and the corresponding consumption of packet buffers.

In our implementation, we address this issue through the specification of a configurable parameter (BUFF_SIZE), that defines the size, byte-wise, that we assign for accounting purpose to a packet buffer. This parameter is used to translate the packet buffer pool allocated to each queue, into a corresponding byte count on which the buffer computations for rate guarantees, i.e., equation (1) is based. In addition, BUFF_SIZE determines the minimum packet size for accounting purposes, i.e., packets smaller than BUFF_SIZE are counted as being of size BUFF_SIZE. This is similar to the approach used in the Integrated-Service model [15], which allows the specification of a "minimum policed unit" to account for possible per-packet overhead.

For example, this means that if the AS/BE queue is allocated 50 packet buffers, it is considered as having a buffer capacity of $50 \times$ BUFF_SIZE. As a result, if stream k asks for a rate guarantee r_k of 20% of the bandwidth allocated to the AF/BE queue, its buffer allocation is $B_k = 10 \times$ BUFF_SIZE. Given that the packets of stream k are counted as being of size BUFF_SIZE or larger, stream k is limited to an allocation of at most 10 packet buffers. This means that stream k is guaranteed its transmission rate r_k only if sends packets of size BUFF_SIZE or more, and could get a lower throughput if it transmits many packets of size less than BUFF_SIZE. In Section V, we experiment with the sensitivity of this scheme to the value of BUFF_SIZE.

IV. Testbed Setup and Experiments

In this section, we briefly describe the setup we use to test our implementation of the service differentiation capabilities outlined in the previous section. Our test setup is shown in Figure 5, and consists of a number of routers (`Tim`, `McKinley`, `Rocky`, `Himalayas`, and `Alps`) interconnected by means of E1 ($\approx$ 2 Mbits/sec) links running the PPP protocol. The routers we use are IBM 2210 (`Tim`, `McKinley`, and `Alps`) and 2216 (`Rocky` and `Himalayas`) models as indicated on the figure, which are running a modified version of their forwarding code that incorporates our QoS extensions. The main reasons for using these platforms are not only because of the availability of their forwarding code, but more important because they are representative of many edge devices currently in use in IP networks, i.e., they have a structure similar to that of Figure 2. As one of our goals is to demonstrate that such edge devices can be easily upgraded to support QoS capabilities, it is important that we validate our claims in a realistic setting.

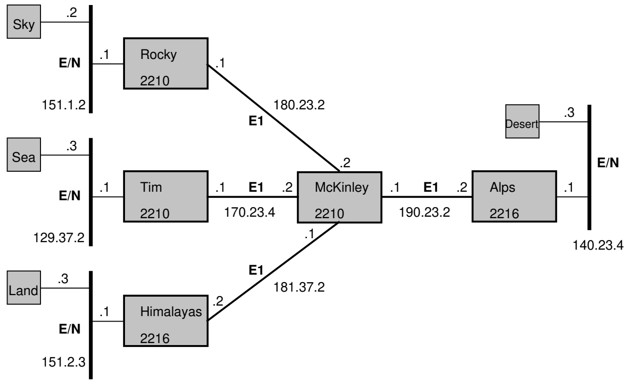


Fig. 5. Experimental Testbed.

Testing of service guarantees is carried out using a number of FreeBSD end-systems, which we use as traffic sources (`Sea`, `Sky`, and `Land`) and sinks (`Desert`). Traffic is generated by running MGEN ver. 3.0 [16] on our end-systems, with different configuration parameters so as to exercise a range of load and traffic patterns. In our test cases, we use the PERIODIC and POISSON settings of MGEN to generate streams of packets, where packet arrivals are either periodic or follow a Poisson distribution (see [16] for details). Periodic arrivals provide a "cleaner" estimate of our ability to give rate guarantees, i.e., they provide a more stable comparison basis, and may be representative of some real-time streams. On the other hand, the traffic patterns generated using the POISSON setting of MGEN may be somewhat more representative of real traffic.

Measurements to test our QoS guarantees are performed on the PPP E1 link between `McKinley` and `Alps`, i.e., we test the ability of our QoS enhanced forwarding code to enforce service differentiation on interface `190.23.2.1` on `McKinley`. We describe below the series of test cases we use, which all rely on generating a combination of BE, AF, and EF streams from our three traffic sources, and having them converge on `McKinley`'s egress interface `190.23.2.1`. The end system `Desert` serves as a common traffic sink for all streams, and is used to obtain various performance estimates, e.g., throughput and end-to-end delay, for the different streams. In particular, delay estimates are obtained after synchronizing the clocks on the four end-systems using NTP ver. 3 [17]. The end-system `Desert` runs an NTP server to which the clocks of the other three end-systems are synchronized. Note that in order to obtain reasonable delay estimates (0.5 msec), it is necessary to leave the system in operation for extended periods of time (over 24 hours) to properly calibrate the drifts between the different clocks.

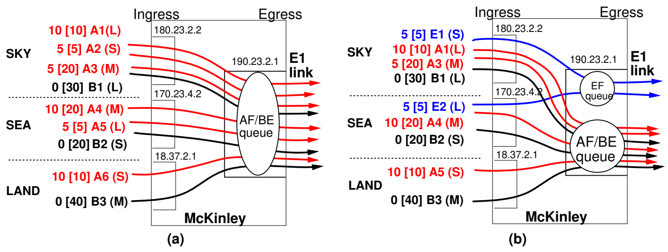


Fig. 6. Test Cases.

A. *Test Cases*

The test cases we run aim at assessing how well our implementation meets the design goals stated earlier, i.e.,

- Minimize overhead associated with QoS guarantees,
- Ensure basic rate guarantees and service isolation, and
- Provide some level of delay differentiation.

In addition, we also measure sensitivity to several other system parameters such as the total number of packet buffers available, and the value chosen for the parameter BUFF_SIZE discussed in Section III.

Our first test is intended to evaluate the relative overhead introduced by our QoS extensions. This is accomplished by inserting profiling statements that measure the time spent in various processing modules along the data path. The time stamps are of sub-microsecond granularity and are taken by reading a real-time clock which is an integral part of the router CPU. Specifically, we load the instrumented forwarding code on our test router, `McKinley`, and measure the time taken for a complete forwarding operation through both the fast and slow paths (see Figure 3). We also measure the execution time of the specific instructions corresponding to the QoS decisions and checks we have added. Using these measurements, we can evaluate the relative overhead introduced by QoS support, when compared to basic best-effort forwarding. Results of those measurements are reported in the next section.

The next series of tests is meant to evaluate the ability of the implementation to enforce service differentiation. For those tests, traffic patterns and loads are chosen so that the router is not processor limited, but instead bandwidth on the E1 link between `McKinley` and `Alps` is the scarce resource. Figure 6 shows the test scenarios we have used for that purpose. Packet streams are generated from our traffic sources, and arrive on the three ingress interfaces at `McKinley` before heading to the common egress interface `192.23.2.1`. In all tests, the letter B is used for BE streams, A for AF streams, and E for EF streams. Associated with each stream are two numbers: The first gives the rate guarantee, if any, for the stream, while the second number, in brackets, specifies the actual amount of traffic that the flow generates. The numbers are given in percentage of the egress link bandwidth. The scenarios shown in Figures 6(a) and 6(b), include a mixture of streams with and without reservations, and whose performance will be measured under different conditions. Streams differ in terms of their packet sizes, the traffic they generate, and their reservation, if any.

In the first case of Figure 6(a), all the streams with reservations are of type AF, so that the ability of the scheduler and the EF queue to provide improved delay performance is not tested. Instead, the focus is on assessing the influence on both throughput and service guarantees, of the parameter BUFF_SIZE for streams with different packet sizes. This is accomplished by varying the value of BUFF_SIZE and observing its impact on the throughput of both AS and BE streams with different packet sizes. As indicated in the figure, we use streams with three different packet sizes: 1000 bytes (L), 500 bytes (M), and 200 bytes (S). The next test uses again the scenario and streams of Figure 6(a), but this time varies the size of the packet buffer pool on interface `190.23.2.1`. This value is a configurable parameter, i.e., a specific portion of the system memory can be allocated to any given interface, and as pointed out in [13] may affect the ability to deliver rate guarantees through buffer management. The test is performed assuming a value of BUFF_SIZE= 500 bytes, which was found to be a reasonable choice in the previous experiment. These last two tests also provide useful information on how successful the simple buffer management mechanism is at ensuring rate guarantees and redistributing excess resources across different streams. The discussions of Section V highlight these issues, and describe the behavior of the system when both PERIODIC and POISSON settings are used by MGEN to generate various traffic patterns.

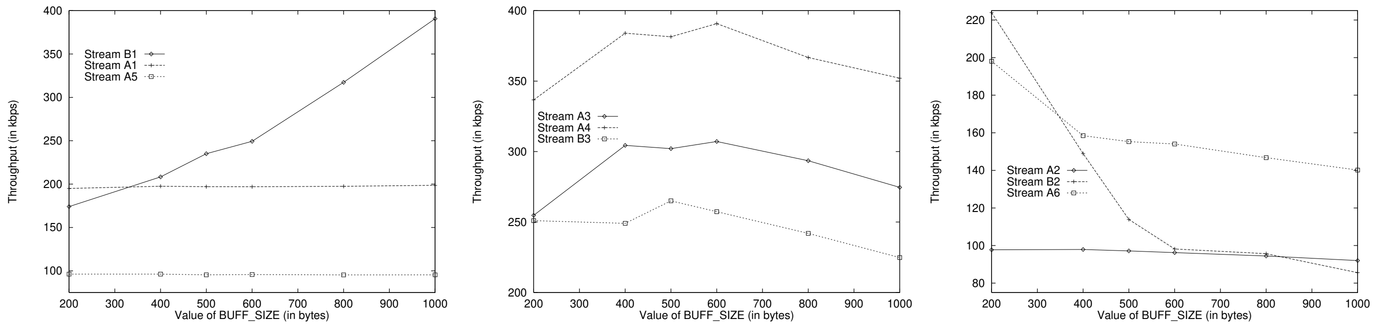
The last test is intended to verify that using a separate queue for the EF traffic is an adequate mechanism for providing EF flows with improved delay guarantees. In order to assess if this is indeed the case, we modify the scenario of Figure 6(a) and move two of the AS flows to the EF queue. We then measure any improvement in delay that they see. This new configuration is illustrated in Figure 6(b). Note that in agreement with the earlier assumption regarding EF flows, the two AF flows (A2 and A5) moved to the EF queue are conformant, i.e., the traffic they generate conforms to the rate they have reserved.

V. Experimental Results

This section reports on the measurement results for the test cases described in the previous section, and discusses their implications and the conclusions one can draw from them.

A. *Relative Overhead*

As mentioned earlier, the first goal of our tests was to assess the relative overhead incurred by introducing QoS support. This involved measuring the path length of both the standard forwarding path and the QoS enhancements we introduced. Whenever there was a cache hit (see Figure 3) the average processing time in the forwarding path was 154 μsec/packet, of which the QoS extensions that we added amounted to a total of 23 μsec/packet. Thus the QoS extensions took around 15% of the total fast path processing. The slow path on the other hand took around 419 μsec/packet and so the QoS extensions were only about 5% of the slow path operations. These numbers were obtained by averaging measurements over multiple traffic patterns and loads, to exercise all possible combinations of QoS instructions. Variations across scenarios were minimal, so that the above numbers should be representative of the actual per packet cost increase. Note that the number obtained for the basic forwarding code, translates into a maximum throughput of



(a) Large packets

(b) Medium packets

(c) Small packets

Fig. 7. Sensitivity of Throughput and Rate Guarantees to BUFF_SIZE.

around 6500 packets/sec through the edge router. This is a small number, but representative of the type of low-end device we are considering. It is important to keep this in mind, in particular in the context of the comparisons of the next sections.

The main conclusion from the above path length comparison is that the impact of the slightly greater path length of the QoS code is very minor. This is not unexpected as significant efforts were spent in both the design and the implementation, to keep the overhead of QoS as small as possible. Furthermore, there are many other components besides the forwarding code, that contribute to limiting the raw throughput through a box. In particular, the device driver code that is responsible for pushing packets out on the links, is often a major fraction of the total path length. As a result, while the added path length due to the QoS code may translate into a small absolute increase in packet processing time, the relative magnitude of this increase will most likely be even less noticeable. This was actually verified through a number of throughput comparisons, which revealed only minor differences of the order of a few percent between the best-effort and the QoS enabled forwarding codes.

B. Strength of QoS Support

The previous section indicated that the cost of QoS support, as implemented in the test router, was relatively small. The next question is to determine how good QoS support actually is, i.e., how effective it is at enforcing service differentiation. As discussed before, answering this question is the purpose of the scenario shown in Figure 6(a).

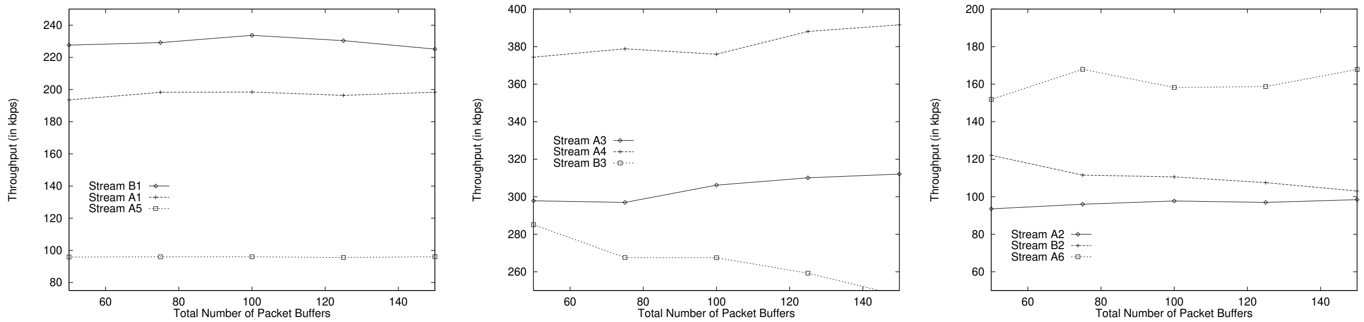
B.1 Sensitivity of QoS Support to BUFF_SIZE

The first test is targeted at estimating the sensitivity of QoS support to the value of BUFF_SIZE, used to translate packet buffers into equivalent number of bytes. Because packets smaller than BUFF_SIZE are assumed to be of size BUFF_SIZE, we expect that the value used for this parameter will affect differently flows with different packet sizes. In order to test this, we run the scenario of Figure 6(a) for values of BUFF_SIZE of 200, 400, 500, 600, 800, and 1000 bytes, and the results are reported in Figure 7 for each type of streams, i.e., streams with large (1000 bytes), medium (500 bytes), and small (200 bytes) packets.

AF streams A1 and A5, which have large packets, are found to exhibit little sensitivity to changes in the value of BUFF_SIZE (see Figure 7(a)). This is because both are conformant and remain so independent of the value of BUFF_SIZE, since their packets are never counted as being larger than they really are. For flows with large packets, the main disadvantage of small values of BUFF_SIZE, is that the total shared buffer space appears smaller than it really is, and as a result smaller bursts and/or amounts of excess traffic can be accommodated. Flows A1 and A5 being conformant are not really affected by this. On the other hand, the best effort flow B1 sees a substantial improvement as BUFF_SIZE increases. This increase is caused by the corresponding increase in shared buffers, which are the only buffers that flow B1 can access.

Figure 7(b) reports similar results for streams with medium size packets. However, the conclusions are somewhat different from those for streams with large packet sizes. In particular, we now observe greater sensitivity of the AF flows to the value of BUFF_SIZE. In particular, we observe variations in throughput after BUFF_SIZE becomes larger than the packet size of the streams, i.e., when it exceeds 500 bytes. We observe this behavior for streams A3 and A4, which experience a slight decrease in throughput after BUFF_SIZE increases beyond 500 bytes. This is because their packets are now being counted as being larger than what they are, and this limits their ability to access shared buffers. Note that because both streams generate substantial amounts of excess traffic, they do rely on shared buffers and actually achieve throughputs well in excess of their reservations, which are 100kbps and 200kbps, respectively. Both streams also experience an increase in throughput when BUFF_SIZE first increases from 200 to 400 bytes. This is because such an increase translates into a larger shared buffer pool, which allows more of the flows excess traffic to get through. The best effort stream B3 sees a similar trend in throughput variations, but somewhat less pronounced because it can only access the shared buffers.

Finally, Figure 7(c) shows the impact of varying BUFF_SIZE for streams with small packets. As is expected, such streams get rapidly penalized as BUFF_SIZE increases since all their packets are counted as being much larger than they are. As a result, streams that are conformant now appear non-conformant, and must rely on shared buffers for many of their packets. This penalty is somewhat compensated by the fact that increasing



(a) Large packets

(b) Medium packets

(c) Streams with small packets

Fig. 8. Sensitivity of Throughput and Rate Guarantees to Total Buffer Size (Number of Packet Buffers).

BUFF_SIZE correspondingly increases the amount of shared buffers. However, as can be seen from Figure 7(c), this is not sufficient to offset the accounting penalty that small packets incur. This penalty is more severe the higher the initial reservation, as the stream needs to gain access to a correspondingly larger number of shared buffers in order to maintain its throughput. This is why stream A2, which has a reservation of only 100kbps, sees a smaller degradation than stream A6 which has a reservation twice as large.

The main conclusion from this section, is that the setting of BUFF_SIZE needs to be taken into consideration when making reservations. Relatively large values, e.g., over 500 bytes appear preferable overall, and are also desirable in order to accommodate a reasonable number of reservations. However, sufficient shared buffers should be kept aside in order to ensure that streams with small packets are not overly penalized.

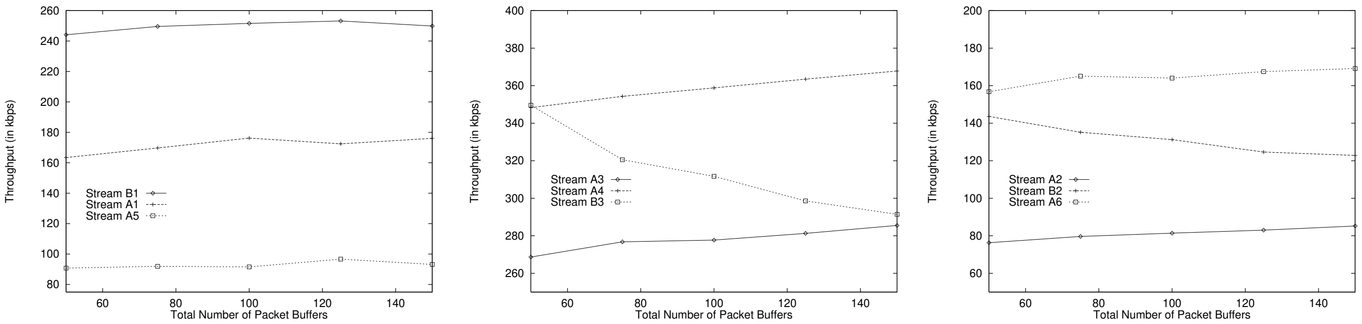
B.2 Sensitivity of QoS Support to Total Buffer Size

The other system parameter that is likely to affect performance is the the total number of packet buffers available on an interface. To assess its impact, we conduct a set experiments similar to those of the previous section. The results are given in Figure 8, where the total number of packet buffers available is varied from 50 to 150 packet buffers. The results are again plotted separately for streams with large, medium, and small packets. A value of BUFF_SIZE equal to 500 bytes was assumed for all the measurements.

Overall, we see that while there is minimal sensitivity to the total amount of buffers, and that variations are relatively contained with only small differences for streams with different packet sizes. Streams with small and medium size packets still see slightly larger variations than those with large packet sizes, but they are hardly noticeable. Note that all reserved streams essentially have throughputs at or above their reservations. Even stream A6 which is penalized by its small packets (each 100 bytes packet is counted as being of size BUFF_SIZE$=500$ bytes) and relatively large reservation (200kbps), gets close to its reservation (the throughput numbers of Figure 8(c) are only based on payload, and do not account for the IP/UDP headers).

Another aspect that Figure 8 illustrates, is that the scheme we use to control access to shared buffers is reasonably successful at ensuring fair access to the unreserved link bandwidth. Specifically, the total level of reservations from flows A1 to A6 is 900kbps, which implies that there is 1,100 kbps of unreserved bandwidth. There are 5 streams that generate excess traffic, namely streams A3 and A4 as well as the best effort streams B1, B2, and B3. As a result, we would expect each of them to get about 220 kbps of the unreserved bandwidth. From Figure 8(a), we see that flow B1 does indeed get a little over 220 kbps. Similarly, Figure 8(b) shows that flow A3 gets close to 320 kbps (for large number of packet buffers) instead of its reservation of only 100 kbps, and flow A4 gets nearly 400 kbps while it only reserved 200 kbps. Similarly, flow B3 ends up getting close to 220 kbps (again for large buffers). The situation is a little bit different for flows with small packets because of the impact of BUFF_SIZE, which is set to 500 bytes. This means that the buffer accounting for those flows is as if they were transmitting at 2.5 times their actual rate (recall that they use 200 bytes packets). If we account for this overhead, we then see from Figure 8(c) that the approximately 100 kbps that flow B2 gets, actually correspond to about 250 kbps. Again, this is reasonably close to our target of 220 kbps.

We can, therefore, conclude that the buffer allocation and sharing mechanism is successful at providing rate guarantees as well as ensuring fair access to excess bandwidth. However, based on the measurement results of Figure 8, it appears that having a reasonably large number of packet buffers is useful when it comes to enforcing fairness. In particular, the results of Figure 8 show small improvements for streams with small and medium packet sizes as the total number of packet buffers increases. This is because more packet buffers translates into more shared buffers, which streams with small and medium size packets can more readily benefit from. However, it should be noted that an increase in the total buffer space does not correspond to an equal increase in shared buffers. This is because reserved buffers need to be allocated in *proportion* to the total buffer space. Increasing the total buffer count, therefore, scales all buffer reservations accordingly, so that only a small portion of the additional buffers is actually added to the shared buffers. although the shared buffer space increases, the larger reserved buffers combined with the (holes) accounting method used to control access to shared buffers, actually improves the odds of



(a) Large packets (b) Medium packets (c) Small packets

Fig. 9. Throughput and Rate Guarantees for POISSON Arrivals With Varying Total Buffer Space.

reserved streams to access shared buffers. As a result, best effort streams actually see a slight decrease in throughput, especially for small and medium packet sizes. This decrease matches the corresponding increase that reserved streams see.

Increasing buffer space can in general be beneficial, and one dimension where we would expect to see some benefits of increasing the total buffer count, is in terms of handling some amount of burstiness in the streams. In order to estimate this sensitivity, we repeat the previous tests using the POISSON setting of MGEN instead of the PERIODIC one. In other words, we use the same packet arrival rates, but the inter-arrival times are now exponentially distributed instead of being constant. The value of BUFF_SIZE is maintained at 500 bytes for these experiments as well. The results of the experiment are shown in Figure 9, and confirm that for flows with small and medium sized packets, there is some benefit to increasing the total buffer size. For flows with large packets, there does not seem to be any significant improvement in throughput. This is attributable to the fact that the larger packet size also implies larger bursts, and there is still not enough shared buffer left to accommodate these larger bursts.

The main conclusion from the above series of tests is that some reasonable amount of shared buffers needs to be available in order to provide rate guarantees to bursty flows. This is not unexpected and is a phenomenon that has been documented in numerous traffic management studies. The buffer management schemes on which we rely can be easily extended to also account for burstiness (see [13] for details), and we plan on adding this capability in the next release. However, such an extension comes at the price of reserving substantially larger amounts of buffer, and we want to first experiment further with the use of shared buffers as the base mechanism for handling burstiness. Additionally, larger buffers potentially result in larger delays which can be quite significant for low-speed links.

C. Delay Guarantees

Our last experiment is aimed at verifying our ability to provide better delay to EF streams. For that purpose, we take two of the conformant streams of Figure 6(a), streams A2 and A5, and move them to the EF queue as shown in Figure 6(b) (now flows E1 and E2). We then compare the delays experienced by the two streams in each configuration. A value of BUFF_SIZE= 500 bytes was used together with a total of 100 packet buffers. The results are reported in Table I, which shows that the use of the EF queue succeeds at significantly reducing the delays (by a factor of more than 3). Interestingly, stream A2 which had a worse delay than stream A5 in the AF/BE queue, has a better delay when the two flows are moved to the EF queue. This is because of the smaller packet size of stream A2. In the AF/BE queue, this benefit is not significant as it is multiplexed with many other streams. However, the difference in packet sizes becomes apparent in the EF queue that only flows A2 and A5 use.

Flow	Delay in AF queue	Delay in EF queue
A2	180 msec	44 msec
A5	169 msec	49 msec

TABLE I

DELAY COMPARISON BETWEEN AF AND EF QUEUES.

It should be noted that the numbers reported in the table are for end-to-end delays between the source and destination end systems. As a result, they include additional contributors than the router `McKinley`, that we use to provide service differentiation. Overall, while the delays we see for EF streams are not small, we believe that they are reasonable given the relatively low speed of the link we used (2Mbps), and they should be adequate for real-time applications.

VI. SUMMARY

In this paper, we have described a *lightweight* software implementation for offering some basic service differentiation capabilities in the context of simple edge devices. Our goal was to investigate the ability of such a platform to offer support for service differentiation and QoS guarantees with minimal impact on its forwarding performance.

We carried this investigation on a real router platform, where we developed and tested a set of enhancements aimed at providing rate guarantees and delay differentiation. These two types of service guarantees were chosen based on the general direction of the service proposals currently being discussed in the

IETF Diff-Serv Working Group. The measurements performed on our implementation showed that support for basic QoS guarantees can be achieved in edge devices with minimal impact on overall performance. In addition, we showed that the buffer management approach of [13] was indeed capable of providing reasonably accurate rate guarantees and fair distribution of excess resources. We also verified that a simple design based on two queues and a rudimentary SCFQ scheduler, can provide adequate delay differentiation to meet the requirements of most real-time applications. There are clearly many aspects and behaviors of our implementation that require further investigation. However, we believe that these initial results provide strong evidences that support for QoS guarantees can be incrementally deployed on most existing edge devices, and with a minimal impact on performance. We hope that such evidences can foster the rapid deployment of QoS capabilities in the Internet.

ACKNOWLEDGMENTS

The authors would like to thank Govindaraj Sampathkumar and Gaurang Shah for valuable discussions as well as technical help in the implementation effort.

REFERENCES

[1] 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.

[2] Y. Bernet, J. Binder, S. Blake, Mark Carlson, E. Davies, B. Ohlman, D. Verma, Z. Wang, and W. Weiss, "A framework for differentiated services," Internet Draft, `draft-ietf-diffserv-framework-01.txt`, November 1998, (Work in Progress).

[3] D. Black, S. Blake, Mark Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," Internet Draft, `draft-ietf-diffserv-arch-02.txt`, November 1998, (Work in Progress).

[4] K. Nichols, S. Blake, F. Baker, and D. L. Black, "Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers," Internet Draft, `draft-ietf-diffserv-header-04.txt`, November 1998, (Work in Progress).

[5] R. Guérin, S. Blake, and S. Herzog, "Aggregating RSVP-based QoS requests," Internet Draft, `draft-guerin-aggreg-rsvp-00.txt`, November 1997, (Work in Progress).

[6] Y. Bernet, R. Yavatkar, P. Ford, F. Baker, L. Zhang, K. Nichols, and M. Speer, "A framework for use of RSVP with Diff-Serv networks," Internet Draft, `draft-ietf-diffserv-rsvp-01.txt`, November 1998, (Work in Progress).

[7] T. V. Lakshman and D. Stiliadis, "High speed policy-based packet forwarding using efficient multi-dimensional range matching," in *Proceedings of SIGCOMM*, Vancouver, British Columbia, CANADA, August 1998, (To appear).

[8] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast scalable algorithms for level four switching," in *Proceedings of SIGCOMM*, Vancouver, British Columbia, CANADA, August 1998, (To appear).

[9] V. Jacobson, K. Nichols, and K. Poduri, "An expedited forwarding PHB," Internet Draft, `draft-ietf-diffserv-phb-ef-01.txt`, November 1998, (Work in Progress).

[10] S. Sathaye, "ATM Forum Traffic Management Specification Version 4.0," ATM Forum 95-0013, December 1995.

[11] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski, "Assured forwarding PHB group," Internet Draft, `draft-ietf-diffserv-af-03.txt`, November 1998, (Work in Progress).

[12] S. J. Golestani, "Network delay analysis of a class of fair queueing algorithms," *IEEE J. Select. Areas in Commun.*, vol. 13, no. 6, pp. 1057–1070, August 1995.

[13] R. Guérin, S. Kamat, V. Peris, and R. Rajan, "Scalable QoS provision through buffer management," in *Proceedings of SIGCOMM*, Vancouver, British Columbia, CANADA, August 1998, (To appear).

[14] A. K. Choudhury and E. L. Hahne, "Dynamic queue length thresholds in a shared memory ATM switch," in *Proceedings of INFOCOM*, San Francisco, CA, April 1996, pp. 679–687.

[15] S. Shenker and J. Wroclawski, "General characterization parameters for integrated service network elements," Request For Comments (Proposed Standard) RFC 2215, Internet Engineering Task Force, September 1997.

[16] B. Adamson, "The Naval Research Laboratory (NRL) "multi-generator" (MGEN) toolset, ver. 3.0," Code is available from, `ftp://manimac.itd.nrl.navy.mil/Pub/MGEN/dist`, 1998.

[17] D. L. Mills, "Network Time Protocol (version 3): Specification, implementation, and analysis," Request For Comments (Draft Standard) RFC 1305, Internet Engineering Task Force, March 1992.