The QoSbox: Quantitative Service Differentiation in BSD Routers[*]

Nicolas Christin
Information Networking Institute and
CyLab Japan
Carnegie Mellon University
1-3-3-17 Higashikawasaki-cho
Chuo-ku, Kobe 650-0044, Japan
nicolasc@cmu.edu

Jörg Liebeherr
The Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto
10 King's College Road
Toronto, ON M5S 3G4, Canada
jorg@comm.utoronto.ca

Abstract

We describe the design and implementation of the QoSbox, a configurable IP router that provides per-hop service differentiation on loss, delays and throughput to classes of traffic. The novel aspects of the QoSbox are that (1) the QoSbox does not rely on any external component (e.g., no traffic shaping and no admission control) to provide the desired service differentiation, but instead, (2) dynamically adapts packet forwarding and dropping decisions as a function of the instantaneous traffic arrivals and allows for temporary relaxation of some service objectives; also, (3) the QoSbox can enforce both absolute and proportional service differentiation on queuing delays, loss rates, and throughput at the same time. We focus on a publicly available implementation of the QoSbox in BSD-based PC-routers. We evaluate our implementation in a testbed of BSD routers over a FastEthernet network, and we sketch how the QoSbox can be implemented in high speed architectures.

Keywords: Quality-of-Service Implementations, Service Differentiation, PC-Routers, BSD, High-Speed Networks.

[*]Most of this work was done while both authors were with the University of Virginia. This work was supported in part by the National Science Foundation through grants ANI-9730103 and ANI-0085955.

1 Introduction

The capacity of the network links of the Internet has steadily increased in the past decade, and has mostly resulted in an increase of the traffic that uses these links, without any improvement in the service received. Stated differently, increasing the capacity of the network far beyond what is needed to accomodate typical loads, a practice known as *overprovisioning*, does not always translate in improved quality-of-service (QoS). Indeed, the service received by an end-to-end traffic flow is bounded by the service received at the link with the smallest capacity on the end-to-end path. Thus, augmenting the capacity of some links only moves the bottleneck to another part of the network, and consequently, only changes the location of the service bottleneck, e.g., from the core to the edge of a network. In particular, access links now appear to be one of the most likely points of congestion in the network [38].

An approach pursued by some Internet Service Providers (ISPs) to improve the average service received by the traffic they serve, is to simply filter out "undesirable," resource-consumming traffic. However, network measurements (e.g., [44]) indicate that end users may actually be particularly interested in resource-consumming traffic such as IP telephony or peer-to-peer transfers, which provides an economic incentive for most ISPs to avoid traffic filtering, lest they lose an important segment of their customer base.

A more desirable solution, at least from the end users' perspective, is to have the network support service differentiation between different classes of traffic. Numerous service architectures for enhancing the best-effort service provided by the Internet have been propose over the past decade (see [19] for an overview). However, only very few proposals have actually been deployed to some extent, in large part due to the complexity of the required arithmetic manipulations or the amount of state information to be maintained in network routers.

We have previously shown that one can offer relatively strong service differentiation, by solely relying on dynamically adaptive scheduling and buffer management at the bottleneck links of the network [12]. The architecture we proposed hinges on a specific packet handling algorithm to be implemented in routers, and does not require admission control, traffic policing, or network reservations, which could make it more easily deployable than some of the proposed alternatives (e.g., [8]).

The main contribution of the present paper is to demonstrate, through the description of a proof-of-concept implementation, that the router algorithms we propose are efficient enough for practical use at typical access link speeds. To that effect, we describe an implementation in BSD-based PCs of the QoSbox, an IP router providing per-hop service differentiation to classes of traffic. We show that this implementation, available to the public through the KAME [2] and ALTQ distributions, can be readily deployed in access networks with links in the order of a few hundreds of megabits per second, and we outline how the algorithms used can be further modified to be implemented in switch architectures governing high-speed links, in the order of several gigabits per second, at the expense of some reduced accuracy in the service differentiation.

Different from most QoS architectures, the QoSbox does not rely on any external component such as traffic shaping or admission control to enforce the desired service differentiation, which allows for deploying a network of QoSboxes in an incremental manner. Instead, the QoSbox dynamically adapts packet forwarding and dropping decisions dependent on the instantaneous traffic arrivals. The QoSbox can enforce both absolute bounds and proportional service differentiation on queuing delays, loss rates, and throughput at the same time. Since there is no admission control, it may not be feasible to enforce all absolute bounds at all times, in which case, the QoSbox relaxes some bounds according to an order specified a priori.

This paper is organized as follows. In Section 2, we present an overview of the proposed service architecture, and describe the mechanisms used by the QoSbox. In Section 3, we discuss implementation issues. In Section 4, we assess the efficiency of the QoSbox in a testbed of PC-routers. In Section 5, we present the related work. Finally, we summarize the current status of the project and draw brief conclusions in Section 6.

2 The QoSbox

In this section, we discuss the architecture of the QoSbox. To that effect, we first summarize the objectives and characteristics of the service the QoSbox offers. We then discuss the mechanisms implemented in the QoSbox to realize the proposed service.

2.1 Overview

Our objective is to provide a service architecture that can provide strong service differentiation in a large-scale network. More precisely, we want the service architecture to have the following properties: (1) it must be deployable in an incremental manner, without ever degrading the average level of service provided in the network, (2) it must scale to networks of large size, and (3) it must support quantitative service differentiation, including absolute bounds.

To meet these constraints, we propose a per-hop service, which provides service differentiation between classes of traffic flows with similar service requirements. The service is enforced independently at each router, no communication between routers is required, and routers do not rely on any external components such as admission control or traffic policers. Instead, service differentiation is enforced using packet scheduling and dropping algorithms designed for dynamic adaptation to changing traffic arrivals.

In comparison to a more traditional architecture, such as a static rate allocation used in conjunction with admission control, our proposed approach presents two advantages. First, measurements studies, as summarized in [43], have shown that Internet traffic is generally highly variable. Consequently, the backlog at bottleneck links is generally highly variable. A static configuration of the worst-case backlog, as needed by existing schedulers such as Class-Based Queuing (CBQ, [22]) when delay bounds are offered, therefore results in under-utilizing network resources, and it is desirable to use instead an architecture dynamically adapting to changes in the traffic load. Second, because each QoSbox operates independently of other routers, there is no communication or signaling overhead, and setting up a QoSbox to govern a given link does not require any knowledge of the characteristics or topology of the rest of the network.

The main limitation of an approach completely avoiding admission control and traffic policing is that satisfying all service bounds at all times may be impossible. For example, upon arrival of a sudden large burst of traffic, it may not be feasible to satisfy all absolute bounds (e.g., delay and loss rate bounds) simultaneously, in which case some bounds have to be temporarily relaxed. Thus, an order of relaxation of the service objectives must be chosen at configuration time, for instance, allowing a violation of a delay bound in favor of a loss rate bound. In the QoSbox, we give precedence to loss differentiation over delay (or throughput) differentiation. The rationale for our choice is that for TCP and TCP-friendly traffic, losses are used as a congestion indicator, and therefore result in throughput reduction. In other words, loss differentiation may have a stronger overall impact on the service received by TCP-friendly flows than delay (or throughput) differentiation.

Because the service paradigm we propose potentially enttails relaxation of service bounds, our approach is different from traditional deterministic service architectures such as Integrated Services (IntServ, [8]) or Differentiated Services (DiffServ, [6]), which provide strict service guarantees. Within the limits set by the potential relaxation of guarantees, the service model the QoSbox supports nevertheless allows for emulation of the Assured Forwarding (AF, [23]) and Expedited Forwarding (EF, [15]) services of the DiffServ architecture, and even attempts to generalize the service differentiation provided by both AF and EF. In particular, the QoSbox can support loss, delay and throughput bounds given to each class. In addition, the QoSbox supports proportional differentiation on loss and delay between classes of traffic, so that AF-like relative differentiation between classes of service can be quantified.

QoSbox configuration We illustrate in Fig. 1 the service differentiation the QoSbox can provide through an example of a QoSbox configuration file. The configuration file contains the properties of the output interface(s) on which QoSbox traffic control must be performed. In the example of Fig. 1, the interface concerned, `fxp0`, has a bandwidth of 100 Mbps and a buffer size of 200 packets. The field `jobs` indicates that traffic control is performed by a variant of the JoBS algorithm (Joint Buffer Management and Scheduling, [35]) which is the scheduling/dropping algorithm central to the QoSbox.

The next set of configuration commands, in lines (2)–(5) contains a description of the service offered to

```
(1) interface fxp0\
    bandwidth 100M qlimit 200 jobs
(2) class jobs fxp0 high_class NULL\
    priority 0\
    adc 2000 rdc -1 alc 0.01 rlc -1 arc 10M
(3) class jobs fxp0 med2_class NULL\
    priority 1\
    adc -1 rdc 2 alc -1 rlc 2 arc -1
(4) class jobs fxp0 med1_class NULL\
    priority 2\
    adc -1 rdc 2 alc -1 rlc 2 arc -1
(5) class jobs fxp0 low_class NULL\
    priority 3 default\
    adc -1 rdc -1 alc -1 rlc -1 arc -1
(6) filter fxp0 high_class 0 0 0 0 0 tos 1
(7) filter fxp0 med2_class 0 0 0 0 0 tos 2
(8) filter fxp0 med1_class 0 0 0 0 0 tos 3
(9) filter fxp0 low_class  0 0 0 0 0 tos 4
```

Figure 1: **Example of a QoSbox configuration file.** The configuration file defines (1) the properties of the output interface, (2) the service each class of traffic receives and (3) the filters used by the classifier to map packets to given classes of traffic. Line numbers are not part of the configuration file, but are used here for readability purposes.

each class. In this example, Class `high_class` is given a delay bound of 2000 microseconds (`adc 2000`), a loss bound of 1% (`alc 0.01`), and a minimum throughput of 10 Mbps (`arc 10M`). This class is not subject to proportional delay or loss differentiation, as indicated by the `rdc -1` and `rlc -1` commands. The `priority` field simply indicates the class index, but does not denote a priority order. Conversely, classes `med2_class`, `med1_class`, `low_class` are not offered delay or loss bounds, but are subject to proportional delay and loss differentiation with a factor of 2. This means that packets from the class `med1_class`, should get queuing delays twice as long as those experienced by `med2_class` packets. Likewise, packets belonging to `low_class`, should get queuing delays twice as long as those experienced by `med1_class` packets. Note that the parameters taken by `rdc` or `rlc` can be any positive value, including values less than one.

Last, commands in lines (6)–(9) specify how packets should be classified. In this example, the only classification criterion is the DiffServ Codepoint (DSCP, [37]) of the IP header. In the present example, a value of 0x03 in the DSCP field of an incoming packet indicates that the packet belongs to Class `med1_class`. The configuration file presented above assumes the use of IPv4, but the implementation of the QoSbox presented in this paper also supports IPv6. In the case of IPv6, the DSCP corresponds to the IPv6 Traffic Class octet.

Note that we do not necessarily advocate the set of service differentiation parameters we use in this example. In all likelihood different ISPs would use different parameters depending on their needs and the type of differentiation they would like to offer. For instance, one could have several classes with different absolute bounds, or only proportional differentiation, or any mix of guarantees. One of the main features of the QoSbox is that the specific service semantics are decided by the network operator running the QoSbox, by means of a configuration file as we just discussed.

QoSbox deployment QoSboxes should primarily be deployed at bottleneck links. Indeed, if a link is never congested, incoming packets can be transmitted at once, using a First-Come-First-Served queuing policy, thereby getting a high-grade service (no loss, low delay, high throughput). Hence, there is no reason to artificially delay the packets supposed to travel on the link to satisfy service objectives. Reports on the utilization of backbone links indicate that backbone links are used at about 60% of their capacity on average [49], which tends to show that bottlenecks are rarely present in the cores of the networks. Thus, we can identify two locations where a bottleneck can occur: at the (ingress or egress) access links of a network,[1] where a large number of end users may share a common trunk, and at the interconnection links between two networks.

At interconnects between two networks, there may be an economic incentive for ISPs to use the service differentiation offered by a QoSbox to enforce the service level agreements (SLA) they sign as part of their peering contracts with other ISPs, rather than trying to remove the bottleneck by upgrading a potentially costly line (e.g., a transoceanic link). We can make a more straightforward economic case for using QoSboxes to govern access links (as in the case of Network A in the figure). Indeed, by differentiating incoming traffic, ISPs can avoid dimensioning their entire network to accomodate worst-case traffic spurts coming from outside their domains, which may be costly.

We point out that there are a number of questions regarding deployment that we do not answer in the present paper. For instance, this paper does not address the configuration problem of selecting specific service differentiation parameters at each hop in order to achieve end-to-end differentiation. Indeed, by design, the QoSbox only provides local, per-hop service differentiation. However, the service provided by a QoSbox, used in conjunction with routing mechanisms that can perform route-pinning, such as Multiprotocol Label Switching (MPLS, [42]), can be used to infer end-to-end service, and select the most appropriate route

[1]By "network", we refer here to what is generally called an Autonomous System (AS).

for a particular application given the service demands. A thorough inspection of the interaction of traffic engineering techniques (e.g., routing) with the QoSbox to provide end-to-end service guarantees is outside the scope of this paper. In addition, the economic feasibility of providing end-to-end service guarantees across multiple service providers remains an open problem. Likewise, we do not consider in this paper the issue of path replication and its effect on throughput bound selection, even though path replication techniques, such as one-to-one protection, are increasingly used in the context of load-balancing and core provisioning. Last, even though our BSD implementation can be used for packet marking, we assume that packet marking is handled separately, for instance by the end applications, as in [16].

2.2 Requirements

We next describe two requirements our service architecture has to meet so that it can be used in large size networks. First, the state information kept in the routers should be small (control path scalability). Second, the processing time for packet classification and scheduling should be small as well (data path scalability). The growing amount of traffic present in the core and at the edges of the network also suggests that there is a need for utilizing the existing network resources as efficiently as possible, for instance, maximizing link utilization. We use the distinction between control and data paths to refine the requirements the QoSbox has to satisfy.

Control path requirements. To limit the state information maintained on the control path, admission control of traffic and/or traffic shaping are not permitted. In other words, no *a priori* traffic description is made available to the QoSbox, which therefore has to dynamically adapt to the traffic demand. In a similar effort to reduce control path complexity, we require that QoSboxes do not communicate control or signaling information with their peers. In addition to limiting the processing overhead due to signaling, a second advantage linked to this requirement is a more efficient use of bandwidth.

Data path requirements. We limit the complexity of the operations performed on the data path by first requiring that no per-flow packet classification be performed. Per-flow classification means that each incoming packet must be inspected, in order to be separated from packets belonging to other flows. As a second data path requirement, we need to ensure that the complexity of the scheduling primitives is independent of the number of packets or flows backlogged at the router. Third, in an effort to maximize the utilization of the network resources, the QoSbox must be able to support best-effort traffic. Therefore, (1) scheduling in the QoSbox must be work-conserving, meaning that the backlog of packets at a QoSbox increases only at times when the output link is busy transmitting, and (2) the dropping primitives must minimize packet losses. Last, for performance purposes, it is desirable to have an algorithm that can be parallelized.

The objective of the aforementioned requirements is to limit the overall computational overhead incurred

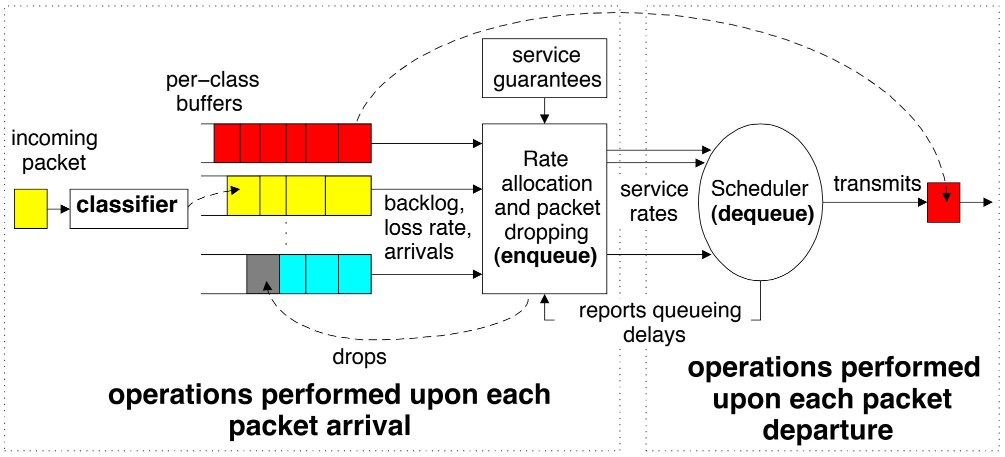

per−class buffers
incoming packet
classifier
service guarantees
Rate allocation and packet dropping (enqueue)
backlog, loss rate, arrivals
service rates
Scheduler (dequeue)
transmits
reports queueing delays
drops
operations performed upon each packet arrival
operations performed upon each packet departure

Figure 2: **Architecture of an output queue in the QoSbox.** The three main components are the packet classifier, in charge of storing incoming packets in the proper per-class buffer, the rate allocation and dropping algorithm, and the scheduler, which forwards packets according to the service rates allocated.

by the algorithm we propose to implement service differentiation in the QoSbox, regardless of the traffic demand, so that the algorithm can be implemented in routers. In this paper, we consider that computational overhead is negligible if all operations performed on a per-packet basis (packet classification, packet enqueuing, and packet dequeuing) can be carried out in less time than the average time needed to transmit a packet on the output link.

2.3 Mechanisms

The mechanisms for service differentiation in the QoSbox are packet dropping and packet scheduling at the output queues of the QoSbox. In other words, the QoSbox is an Output Queuing architecture. One cannot a priori assume Output Queuing in production routers, which generally use Virtual Output Queuing (VOQ) or Combined Input-Output Queuing (CIOQ). However, previous research contributions showed it was possible to emulate Output Queuing routers with CIOQ routers [13] and specific arbitration mechanisms, or to approximate Output Queuing at high-speeds using feedback control [20]. In the PC-routers that we use for our implementation, we can assume Output Queuing only if the input queues are empty, that is, only if the CPU is not overloaded [36]. We will later show that the overhead associated to our proposed mechanisms is limited enough so that we should not face overload conditions in typical access networks where our implementation in BSD routers can be deployed. From now on, we assume that traffic control is only performed at the output queues and solely focus on the operations performed at the output queues.

All output queues in the QoSbox have the same architecture, which is outlined in Fig. 2. Each class of traffic is associated with a per-class buffer. When a packet is passed to the interface governing the output link, a classifier looks up which class the packet belongs to and places the packet in the appropriate per-class buffer. Note that the classifier does not need to identify the flow to which the incoming packet belongs, but only the class to which the incoming packet belongs. The per-class buffers have a finite size selected by the network operator as follows. The maximum amount of traffic that can be held in each per-class buffer can be fixed to a constant (*separate buffers*), or, alternatively, the maximum total amount of traffic backlogged can be bounded (*shared buffer*).

After the incoming packet has been placed in a per-class buffer, the rate allocation and packet dropping algorithm adjusts the service rates allocated to each class of traffic and possibly drops packets in order to enforce the desired service. The computation of the service rates and packet drops is based on the current backlog, arrivals, loss rate on the one hand, and on queuing delays reported by the scheduler on the other hand. If needed, packets are dropped from the tail of each per-class buffer. Even though our implementation in PC-routers does not support them at the present time, more elaborate dropping algorithms can also be used. A study of alternative dropping strategies for the QoSbox is however outside the scope of the present paper, and is instead reported in [11].

Service differentiation is provided over a finite-length time interval, called a *busy period*, whose beginning is defined as the last time the output queue was not backlogged. Similarly, the loss rate and throughput are computed over the current busy period. The choice of the current busy period as the monitoring interval for computing loss rates represents a trade-off. We expect busy periods to be generally short, which enables the output link to react quickly to changes of the traffic load, and limited state information to be maintained. As a disadvantage, at times of low load, when busy periods are very short, providing service differentiation only with information on the current busy period can be unreliable. Also, in times of persistent backlogs resulting in long busy periods, large bursts of packet drops may be allowed if traffic has not been dropped previously. However, at underloaded links transmission queues are mostly idle and all service classes receive a high-grade service, and long busy periods without drops require the offered load to be roughly constant. Near-constant offered loads rarely occur with self-similar arrivals as observed in the Internet. Thus, we believe that choosing the current busy period as the interval over which we compute loss rates and provide service differentiation is a sensible choice.

The key difference between the mechanisms used in the QoSbox and mechanisms used in other service architectures is that here, a single algorithm is in charge of adapting to the current traffic demand both the service rates and loss rates of all classes. The main argument in favor of using a scheme combining rate allocation and buffer management is that rate allocation and buffer management both address the same issue of managing the transmission queue at a given router. The only difference between the two concepts

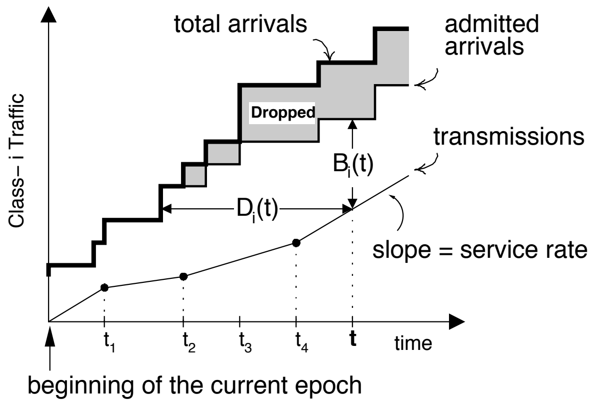


Figure 3: **Overview of the dynamic service rate allocation and packet dropping algorithm.** D_i is the delay and B_i is the backlog. The service rate is dynamically adjusted at times t_1, t_2 and t_4 in function of the traffic arrivals, and traffic is dropped at times t_2 and t_3.

lies in the fact that rate allocation manages the head of the transmission queue, deciding the rate at which packets leave, while buffer management manages the end of the transmission queue, deciding if new packets can be admitted to the queue. To further illustrate the rationale for combining rate allocation and buffer management, consider a given class i subject to a delay bound. The queuing delay D_i experienced by the packet at the tail of the Class-i buffer is equal to B_i/r_i if B_i is the Class-i backlog, and r_i the service rate allocated to Class i. Thus, to reduce D_i in order to satisfy the delay bound, one can either increase r_i, or decrease B_i. In other words, rate allocation and packet dropping can both be used as "handles" to provide the desired service differentiation.

As presented in Fig. 3, we take a quantitative view of traffic to determine how to use these handles. For each class, the QoSbox keeps track of the traffic arrivals, the arrivals that were admitted, and the transmissions since the beginning of the current busy period. As shown in Fig. 3, at any time, the amount of traffic dropped, from which the loss rate can be derived, is the vertical distance between the total arrivals curve and the admitted arrivals curve. Similarly, the delay in Fig. 3 is the horizontal distance between the admitted arrivals curve and the transmission curve, and the backlog is the vertical distance between these two curves, as illustrated for time t in Fig. 3. Additionally, the service rate is the slope of the transmission curve. With these parameters, the rate allocation and dropping algorithm can dynamically decide whether to adjust the service rate of a given class, or drop packets. In Fig. 3, service rates are adjusted at times t_1, t_2 and t_4, and packets are dropped at times t_2 and t_3.

To calculate the service rate allocation and determine if packets need to be dropped, the algorithm first computes, upon each packet arrival, the minimum service rate that is required to transmit the entire backlog of each class within the specified delay bounds. For each class, the minimum service rate is lower bounded by the throughput bound. If this minimum service rate is larger than the current rate allocation for some classes, the algorithm tries to redistribute the service rates allocated to each class so that no delay bound violations occur. If no service rate allocation can satisfy all delay bounds, or if a buffer overflow is detected, the algorithm reduces the backlog of classes by dropping packets according to the loss differentiation specified. Then, within the range of feasible service rates for each class, the algorithm selects a rate adjustment that ensures that proportional delay differentiation will be met. Despite the apparent complexity of the algorithm, all these operations can be efficiently implemented. In Section 2.4, we will describe in more details the rate allocation and packet dropping algorithm, which is an essential piece of the QoSbox architecture.

The service rates calculated by the rate allocation algorithm must be translated into packet forwarding decisions, which is the task of the packet scheduler. Schedulers translating service rates into packet forwarding decisions, such as Packetized-Generalized Processor Sharing (P-GPS, [39]) or Virtual Clock [50], have been proposed in the early 1990s. However, these schedulers were designed for cases where the rate allocation is static, that is, the service rates allocated to each class of traffic only change when a class joins or leaves the scheduler. In the context of a dynamic service rate adaptation such as the one we propose, these algorithms have a worst-case complexity of $O(N)$ where N is the number of packets backlogged in the system, and may thus be computationally too expensive to be carried out at high speeds. For this reason, we propose an $O(Q)$ heuristic, where Q is the number of classes in the system, inspired by the Deficit-Round Robin algorithm [45].

A counter recording the amount of traffic that has been sent in each class since the beginning of the current busy period, $Xmit_i$, is maintained by the scheduler. An auxiliary counter R_i^{out} is updated every time a packet enters the output queue, with

$$R_i^{out} \leftarrow R_i^{out} + r_i \cdot \Delta t \ , \tag{1}$$

where Δt corresponds to the amount of time that has elapsed since the last update of R_i^{out}. R_i^{out} corresponds to the amount of traffic that would have been transmitted since the beginning of the current busy period if packet scheduling perfectly matched the service rate allocation. Every time the output link is available for transmission of a packet, the scheduler computes, for each class, the difference $R_i^{out} - Xmit_i$. Denoting by k the index of the class for which this difference is maximum, meaning that Class k is the "most behind" its allocated service rate, the scheduler chooses to transmit the packet at the head of the Class-k buffer, and records the queuing delay experienced by the transmitted packet, by taking the difference between the current time and the time this packet was enqueued.

```
 (1) function enqueue(pkt)
 (2)     if (output_link_is_idle())
 (3)         reset_all_variables();
 (4)         transmit(pkt);
 (5)     else
 (6)         insert_tail(pkt);
 (7)         accounting(pkt);
 (8)         if (not_backlogged_anymore() or now_backlogged())
 (9)             reset_rates();
(10)         while (buffer_overflow())
(11)             i = select_dropped_class();
(12)             drop(i);
(13)         compute_min_rates();
(14)         while ($\sum$min_rates $>$ C and can_drop())
(15)             i = select_dropped_class();
(16)             drop(i);
(17)             compute_min_rates();
(18)         adjust_rates();
(19) return (dropped);
```

Figure 4: **Rate allocation and packet dropping in the QoSbox.** This sequence of operations is performed immediately after a packet arriving at the output queue has been classified. Line numbers are printed for readability purposes.

2.4 Details of Rate Allocation and Packet Dropping

After having presented an overview of the QoSbox architecture, we now delve into the details of the algorithm for dynamic rate allocation and packet dropping, which, as shown in Fig. 2, is the central component of the QoSbox. Note that the proposed algorithm is only an instance of a class of algorithms that can be designed to dynamically adapt to an unknown traffic arrival pattern.

We present in Fig. 4 the pseudo-code associated with the operations carried out by the rate adjustment and dropping algorithm we propose. We refer to [12] for a justification of the operations discussed in this paragraph. The `enqueue` function described in Fig. 4 is called upon every packet arrival at the output queue of the QoSbox.

First, the algorithm checks the status of the output link. If the output link is not busy transmitting any packets, a new busy period starts with the arrival of the packet. Since all service differentiation is provided over the current busy period, all counters (on arrivals and transmissions) are reset, and the incoming packet is immediately forwarded. This test ensures that scheduling in the QoSbox is work-conserving. Conversely, if the output link is busy, the current busy period started in the past. In that case, the sequence of operations carried out starts by a check on the traffic mix present in the QoSbox. If, since the last packet arrival, some classes are not backlogged anymore, or if the incoming packet belongs to a class that was not previously backlogged, all service rates are reset: classes which are not backlogged are assigned a service rate equal to zero, while classes which are backlogged are all assigned the same service rate. This test is needed since, as we will see later, service rates are *adjusted* instead of being allocated. Therefore, an initial value on the service rates has to be explicitly allocated.

Next, the algorithm tests if the incoming packet causes a buffer overflow. In the case of a shared buffer, the test consists in checking that the total number of packets backlogged is less than a given buffer size. In the case of separate buffers, the test consists in checking that the number of Class-i packets backlogged, where i is the class index of the incoming packet, is less than a given value. The algorithm drops packets until the buffer overflow is resolved. To that effect, the `select_dropped_class` function is called. This function computes, for each class, the difference between the actual loss rate of the class and the loss rate it should experience to satisfy to the proportional loss differentiation, and returns the class index of the class for which this difference is the smallest, meaning that the actual loss rate of that class is the "most behind" the value it should have to realize proportional loss differentiation. Excluded from that computation are classes for which dropping a packet would cause a violation of a loss rate bound. If no class can be dropped without yielding a violation of a loss rate bound, the `select_dropped_class` function selects the class for which the violation is the smallest. Completing the dropping operation, the packet at the tail of the buffer of the selected class is discarded.[2]

After possible buffer overflows have been resolved, the algorithm updates the arrivals and admitted arrivals counters, then calls the function `compute_min_rates` which determines the minimum service rate each class must be allocated to meet its delay and throughput bounds. For each backlogged Class i, the minimum service rate needed to meet the delay bound of Class i is $\frac{B_i}{d_i - D_i}$, where B_i represents the backlog of Class i, d_i represents the delay bound on Class i, and D_i represents the time the oldest Class-i packet still backlogged in the system has experienced [12]. For each backlogged Class i, the minimum rate needed to meet the absolute delay bound and the lower bound on throughput is therefore the maximum

[2]One could drop the packet at the head of the buffer instead. Such a "Drop-From-Front" [33] strategy has the advantage of lowering the queuing delays of all packets still backlogged, but has the major disadvantage of introducing a much more complex coupling between queuing delays and loss rates.

of $\frac{B_i}{d_i - D_i}$ and f_i where f_i is the minimum throughput desired for Class i. Finally, if $d_i - D_i \leq 0$ and $B_i > 0$, meaning that a delay bound violation has occurred, the minimum rate is set to C, the capacity of the output link. This limit case means that the entire Class-i backlog is transmitted as soon as possible in an effort to resolve the situation in a timely manner. When more than one class experiences a delay violation, all classes that have experienced violations get an equal share of the output link, so that in the extreme case where all classes suffer delay violations, the scheduling becomes equivalent to per-class fair-queuing. Non-backlogged classes are assigned a minimum service rate of zero.

As long as the sum of the minimum service rates exceeds the capacity of the output link, packets have to be dropped to decrease the minimum service rates required. Packets are dropped in the same manner as in the case of a buffer overflow, and after each drop, since the Class-i backlog B_i has decreased, the minimum service rates are recomputed. It may happen that no packet can be dropped without violating a loss bound. This condition is tested by the `can_drop` primitive. In such a case, it is impossible to meet at the same time all bounds on delays and on loss rates. As discussed earlier, for the implementation we present in this paper, we make the choice of giving higher importance to loss rate bounds than to delay bounds, and thus, the algorithm stops dropping. We point out that while our implementation uses delay bound relaxation, the algorithm only requires a relaxation order to be provided, without imposing any specific order. In fact, choosing the opposite order, i.e., favoring delay bounds over loss rate bounds, can be done in our implementation by simply removing the `can_drop()` test in line (14) of the pseudocode given in Fig. 4. We plan in the future to make the relaxation order a configurable option in our implementation.

After service rates and packet drops have been computed to meet absolute differentiation (or some bounds have been relaxed), the final operation carried out in the `enqueue` function adjusts the service rates so that proportional delay differentiation is achieved. In our earlier work [12], we showed that the service rate adjustment could be performed by a single multiplication $\Delta r_i = K \cdot e_i$, where Δr_i denotes the adjustment in the service rate of Class-i, K is a time-dependent factor and e_i translates the difference between the queuing delay encountered the last Class-i packet to have been transmitted and the value this queuing delay should have had to satisfy the desired proportional delay differentiation. Hence, $e_i = 0$ indicates perfect proportional delay differentiation. The coefficient K is common to all classes, and its computation takes into account the fact that $r_i + \Delta r_i$ has to be greater than the minimum rate of Class i.

3 Implementation details

Next, we describe the details of our implementation. We first focus on the specifics of the implementation we carried out for BSD kernels using the Alternate Queuing framework (ALTQ, [9]). We provide a short review of ALTQ, and then turn to a discussion of the operations performed by our implementation. In our

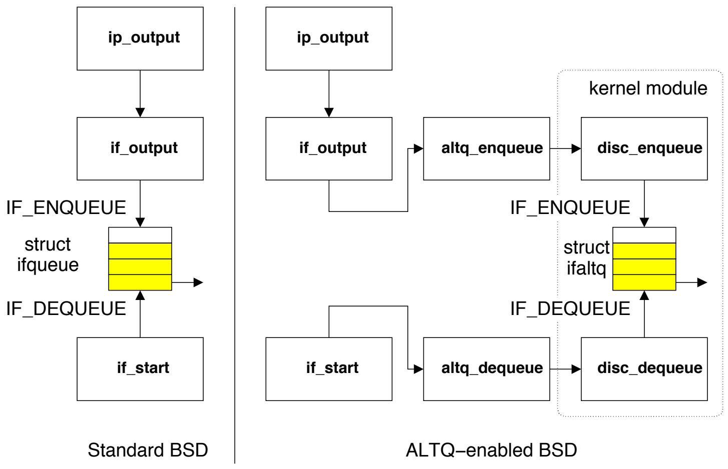


Figure 5: **Functions and structures associated with the output queue in BSD and ALTQ-enabled BSD.** Each queueing discipline implemented in ALTQ consists of a kernel module, represented by the dotted box.

PC-router implementation, all operations are sequential. Even without exploiting potential parallelism in the algorithm, we will show in Section 4 that our implementation can operate at line speeds in the order of 100-500 Mbps in a 1 GHz PC if the number of classes is small.

3.1 ALTQ

Our implementation of the QoSbox for PC-routers builds on ALTQ. ALTQ is an extension to the FreeBSD, OpenBSD and NetBSD operating system kernels. In addition to various bug fixes to networking device drivers, ALTQ provides a modular framework for replacing the default FIFO queueing discipline of network interfaces by custom-designed queueing disciplines.

In BSD kernels, an output networking interface is governed by the `if_output` and `if_start` functions, which enqueue and dequeue packets from the transmission queue, respectively. The transmission queue is represented by the `ifqueue` structure and is shown on the left in Figure 5. An incoming packet is passed to `ip_output` which, after looking up the route, filling the IP header, and possibly fragmenting the packet passes it to `if_output`; `if_output` enqueues the packet in the `ifqueue` structure. When the output link is available for transmission, a packet is dequeued from the `ifqueue` structure by the `if_start` function.

As shown on the right in Figure 5, ALTQ replaces the operations performed by `if_output` and `if_start` by user-defined transmission queue structures and functions included in dynamically loadable kernel modules. Each kernel module implements a specific queueing discipline. A custom packet queue structure (`struct ifaltq`) is used as a replacement to the `ifqueue` structure to implement the transmission queue. Transmission queue management is realized by enqueue and dequeue functions specific to each queueing discipline, as denoted by `disc_enqueue` and `disc_dequeue` in the figure. For instance, the enqueue and dequeue functions of our the QoSbox are called `enqueue` and `dequeue`, respectively. Queueing disciplines that have been developed for ALTQ include Class-Based Queueing (CBQ, [22]), Hierarchical Fair Service Curve (HFSC, [48]), Random Early Detection (RED, [21]) and Blue [18].

Additionally, ALTQ provides a classifier that is used to map incoming packets to classes of traffic. In the QoSbox, classification only consists of selecting in which buffer to store the packet by looking up a class index contained in the DSCP, as marked upstream, and adding a priority field tag to the packet buffer. In a BSD implementation, packets are stored in memory buffers called `mbuf`'s. The `mbuf` structure can be modified to record information, such as the priority field tag we need to insert, in addition to the packet header and payload

We refer to [9] for more details on the implementation of ALTQ.

3.2 Packet Processing

All mechanisms specific to the QoSbox are realized by `enqueue` and `dequeue`. We next describe both functions in detail.

The `dequeue` function The `dequeue` function implements the packet scheduler described in the last paragraph of Section 2.3. The `dequeue` function reads the value of the variables R_i^{out} and $Xmit_i$ for each class (two memory accesses for each class), finds the class for which the difference $R_i^{out} - Xmit_i$ is maximized (one integer subtraction, and one integer comparison per class), and dequeues the packet located at the head of the corresponding per-class buffer. Thus, `dequeue` uses Q integer subtractions and Q comparisons, for a total of $2Q$ arithmetic operations, and performs $2Q + 1$ memory accesses, one for reading each of the variables R_i^{out} and $Xmit_i$, and one for accessing the packet to be dequeued.

The `enqueue` function The main cause of processing overhead in the QoSbox stems from the operations occurring during the enqueueing of an incoming packet, that is, the operations performed by the rate allocation and packet dropping algorithm. This algorithm is implemented by the `enqueue` function in ALTQ and relies on several arithmetic operations (e.g., computation of K). In a network simulator, these operations can be performed using double precision floating-point numbers. In the case of a kernel-level implementation, floating-point operations should be avoided, because the hardware floating-point unit (FPU) is generally not supported in the kernel, and floating-point operations using the FPU emulation library are extremely slow.

The computations in `enqueue` only use fixed-point arithmetic. In our implementation, all quantities are expressed using 64-bit unsigned integers, which requires that we adopt some specific units. Delays are expressed in clock ticks, service rates are expressed in bytes per clock tick scaled by a factor of 2^{32}, and loss rates are expressed as fractions of 2^{32}. These units can achieve a satisfactory degree of precision.

Let us now consider the first set of operations, in lines (2)–(4), which are executed when a new busy period starts. A memory access to check the status of the output link is followed by a reset, for each class, of the four variables corresponding to the total arrivals (A_i), the admitted arrivals (R_i^{in}), the counter R_i^{out}, and the transmissions $Xmit_i$, for a total of $4Q$ arithmetic operations. The current time is recorded as the start of a busy period (which requires one assignment operation), and, for the class of the incoming packet, we set A_i, R_i^{in}, R_i^{out} and $Xmit_i$ equal to the size of the incoming packet, which results in another four assignment operations. Lines (2)–(4) thus require $4(Q + 1) + 1$ assignments and arithmetic operations before transmitting the packet.

If the output link is not idle, the incoming packet is added to a per-class buffer, based on the value of its priority field tag. The `insert_tail` function stores the arrival time of the packet in a timestamp, which is inserted at the tail of a timestamp list.[3] To record the arrival time, `enqueue` needs to access the CPU clock. A simple solution would be to use the `microtime()` function provided in BSD, which has a microsecond granularity. Using `microtime()` increases portability of the implementation, because all BSD systems implement the `microtime()` function since 4.4-BSD. However, `microtime()` may not have a fine enough granularity, and requires a periodic adjustment to account for possible clock skews. Also, using `microtime()` generates significant overhead.[4] A more efficient solution is to directly read the timestamp counter (TSC) register available in the Pentium series processors [28], and compatible architectures, such as AMD processors. This register is an unsigned 64-bit precision integer, and gives the number of cycles elapsed since the machine has been turned on. The resolution of the TSC register is much finer than that provided by `microtime()`. A similar counter (processor cycle counter, PCC) can be found on DEC Alpha architectures, but only provides a 32-bit precision [14]. We read the TSC or PCC registers if they are available, and if not, roll back to `microtime()` to ensure portability of our implementation. Despite potential variations in the duration of each clock cycle in recent processors, due for instance to power management, reading A cycle counter provides time measurements accurate enough for our implementation.

The `accounting()` function called in line (7) adds the size of the (class-i) incoming packet to both A_i and R_i^{in}, and proceeds to compute the current loss rate $p_i = (A_i - R_i^{in})/A_i$ for each class i, which requires a total of $2Q$ arithmetic operations (one subtraction, and one integer division per class).

[3]One could also use an extra tag added to the `mbuf` holding the packet to mark its arrival time.

[4]As of 4.4-BSD, a `nanotime()` function is also available. `nanotime()` provides nanosecond granularity, but suffers the same overhead and clock skew adjustment problems as `microtime()`.

Once the packet has been added to its per-class buffer, the test in lines (8) and (9) in Figure 4 checks if the set of classes with a backlog in the per-class buffers has changed. If there is a change, service rates of all classes are reset: classes with no backlog get a service rate of zero, while backlogged classes equally share the capacity of the output link. In our implementation, there are Q tests to check which classes are backlogged, and, if a change in the backlogged classes is detected, the `reset_rates` call consists of Q assignments.

Next, in lines (10)–(12), `enqueue` drops packets in case a buffer overflow occurs. The `while` loop in lines (11) and (12) is executed in the worst-case for each backlogged packet. However, since this while loop is executed upon each packet arrival, the number of iterations is in fact bounded by the number of packets that have arrived since the last packet arrival. In a PC-router, it is extremely rare, if not impossible, in practice, to have simultaneous packet arrivals. In other switch architectures, the number of iterations is bounded by the internal speed-up of the switch, which is generally less than the number of line cards in the switch, and remains a relatively small number.

The overhead of the function `select_dropped_class` in line (11) depends on the type of loss differentiation offered to the incoming packet(s). If only absolute loss bounds are offered, `select_dropped_class` checks that dropping the incoming packet(s) does not violate loss bounds. This requires looking up the incoming packet size S, and computing the value for the loss rate if the incoming packet is dropped, which can be done with one addition and one integer division, by adding S/A_i to p_i. If no class can be dropped without violating a loss bound, the addition and division are performed for each class, along with a comparison to the loss rate bound, for a total of $2Q$ arithmetic operations, and Q comparisons. Then, `select_dropped_class` has to find the class for which the violation is minimal. This requires another Q subtractions ($L_i - (p_i + S/A_i)$ for each class i), and another Q comparisons to find the minimal violation. So, the worst-case corresponds to a total of $3Q$ arithmetic operations and $2Q$ comparisons.

To find from which class to drop to satisfy proportional service differentiation, the `select_dropped_class` function implements the technique described in Section 2.4, by minimizing a quantity $p_i - p^*$, where p^* is a weighted arithmetic mean of all loss rates (details can be found in [12]). Computing a weighted arithmetic mean over Q elements requires Q multiplications, $Q-1$ additions and one integer division, for a total of $2Q$ operations. Computing all the $p_i - p^*$ requires in turn to perform Q subtractions, and finding the smallest $p_i - p^*$ requires another Q comparisons. Thus, all in all, finding the class that needs to be dropped for enforcing proportional differentiation requires $4Q$ operations. If both proportional loss differentiation and a loss rate bound are offered to all classes, the worst-case overhead amounts to a total of $4Q$ operations for proportional differentiation and $3Q + 2Q = 5Q$ for absolute differentiation, as discussed before, that is, $9Q$ operations.

The computation in line (13) determines the service rates $r_i^{\min}$ needed for meeting the delay and throughput bounds. For each class i, the computation of the minimum service rate required for meeting a delay bound involves looking up the timestamp of the head-of-the-line packet in the considered class, reading the machine clock, and performing a subtraction ($d_i - D_i$) and a division ($B_i/(d_i - D_i)$). These operations are augmented by a comparison ($\max\{B_i/(d_i - D_i), f_i\}$) in case a throughput bound f_i is also present.

The `while` loop in lines (14)–(17) drops packets until a feasible rate allocation exists or all loss rate bounds are reached. The chosen relaxation order imposes that loss rate bounds have higher precedence than delay bounds, which is enforced by the `can_drop()` test in line (14). Note that all operations in the `while` loop in lines (14)–(17), including the functions `select_dropped_class` and `compute_min_rates`, can be executed once for each backlogged packet in the worst case. To reduce the total number of operations, we propose to replace lines (14)–(17) by a call to a function called `greedy_alloc`, based on a heurisitic initially presented in [35]. Instead of choosing which class to drop to respect proportional loss differentiation, `greedy_alloc` redistributes the service rates in a greedy manner to minimize the difference between service rates and minimum service rates, and then, for each class whose service rate is less than the minimum service rate required, drops as much traffic as needed to have the minimum service rate equal to the service rate allocated. The reduced complexity offered by the function `greedy_alloc` comes at at the expense of a potential relaxation of proportional loss differentiation when packets have to be dropped to meet delay bounds. `greedy_alloc` uses Q arithmetic operations to redistribute the service rates. Most of the overhead in `greedy_alloc` comes from the number of operations required when dropping packets. Each time a packet is dropped, a memory access is performed, and four arithmetic operations are used to update the variables R_i^{in}, R_i^{out} and $Xmit_i$. In the worst case, all backlogged class-i packets are dropped. However, note that the worst-case is highly unlikely to happen since `enqueue` is called upon each packet arrival, and it is highly improbable that a single arrival causes a change in the load so drastic that all packets have to be discarded. In practice, at most a handful of packets are dropped every time the `greedy_alloc` function is called. We point out that the number of packets dropped can significantly increase when the frequency at which `greedy_alloc` is called decreases (e.g., when `greedy_alloc` is called every 1,000 arrivals), but, at the same time, calling `greedy_alloc` less frequently gives the function more time to complete.

The last step, described in line (18) adjusts the service rates subject to proportional delay differentiation constraints, and has an algorithmic complexity of $O(Q)$. However, `adjust_rates` itself hinges on computing the coefficient K. The computation of K, whose actual expression is given in [12], requires a number or tests when the traffic mix changes, and to look at most of the state variables (arrivals, departures, backlogs) maintained by the system. In particular, it can be shown that the computation of K requires

$Q^2 + 5Q$ arithmetic operations in the worst case [10]. Hence, while the computation of K is simple enough to be carried out on the fast path of a relatively low-speed router such as our BSD-router implementation, this computation may have to delegated to a background computation in high-speed routers.

3.3 Overhead Reduction

We have described our implementation in PC-routers running a BSD kernel. We next describe how the overhead of our implementation can be reduced for higher performance switches.

The overhead of our implementation is primarily caused by three operations in the `enqueue` function: the adjustment of the service rates for proportional differentiation, performed by the call to the function `adjust_rates`, the computation of the minimum service rates required for meeting delay bounds, implemented by the call to `compute_min_rates`, and the reallocation of service rates and packet drops for meeting delay bounds, realized by the call to `greedy_alloc`.

An option to reduce the computational overhead is to reduce the frequency at which `adjust_rates`, `compute_min_rates`, and `greedy_alloc` are called. This comes at the expense of degraded performance with respect to service differentiation. The degradation in performance may remain acceptable for high sampling frequencies. For instance, we showed in [35] that updating the service rate allocation every T arrivals, with T in the order of 10–100, managed to achieve almost the same results as adjusting the service rates upon each packet arrival.

When the calls to `adjust_rates`, `compute_min_rates`, and `greedy_alloc` are performed only every T arrivals, we can split `enqueue` into operations that have to be performed on a per-packet basis, that is, lines (1)–(12) in Figure 4, from sampled operations, that is, lines (13)–(18). Because no per-packet operation follows a sampled operation, sampled operations can be delegated to a co-processor. As a practical example, in an architecture such as the Intel IXP 1200 network processor [1], sampled operations such as `adjust_rates` could be performed on the StrongARM processor, whereas per-packet operations should be performed by the micro-engines.

In addition to sampling some operations, simplifications to the arithmetic operations involved can be carried out at the expense of flexibility. For instance, one may want to restrict the proportional differentiation factors to powers of two, so that all the multiplications used for proportional differentiation can be replaced by bit-shifting operations. One may also consider fixed sampling intervals for computing loss rates, instead of the current busy period, to avoid integer divisions in the loss rate computations. Last, we use byte counters in our implementation in PC-routers, but one may instead elect to use packet counters, which require increment/decrement operations instead of additions/subtractions for accounting.

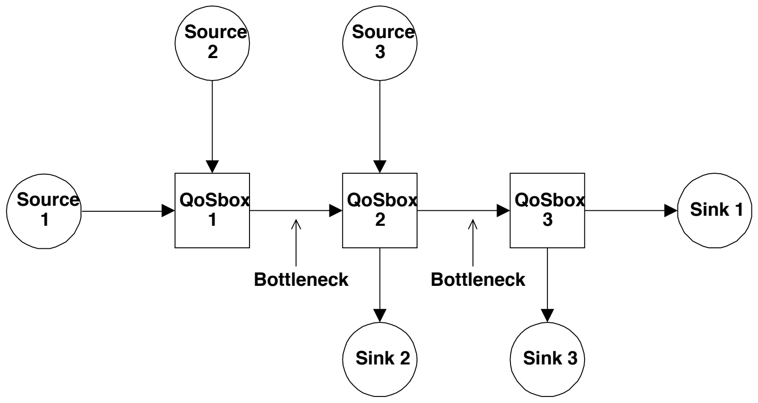


Figure 6: **Network Topology.** All links have a capacity of 100 Mbps. We measure the service provided by QoSboxes 1 and 2 at the indicated bottleneck links.

4 Performance Evaluation

We present experimental measurements of our implementation on a testbed of PC-routers used as QoSboxes. The PCs are Dell PowerEdge 1550 with 1 GHz Intel Pentium-III processors and 256 MB of RAM. The system software is FreeBSD 4.3 and ALTQ 3.0. Each system is equipped with five 100 Mbps-Ethernet interfaces.

We first determine if and how well the QoSbox provides the desired service differentiation on a per-node basis. We then present an evaluation of the overhead associated to the enqueue and dequeue operations of the QoSbox. The objective is to show that the implementation of the QoSbox in BSD-based PC-routers is a solution that can be readily deployed in medium-speed access networks with capacities in the order of a few hundreds megabits per seconds, and to confirm that the operations responsible for most of the overhead are those that can be performed as background tasks in higher-performance architectures. Complementary experiments comparing the algorithms used in the QoSbox with alternative proposals for service differentiation are available in [10].

4.1 Configuration

We consider a local network topology with multiple nodes and point-to-point Ethernet links, as shown in Fig. 6. All links are full-duplex and have a capacity of $C = 100$ Mbps. Three PCs are set up as routers, indicated in Fig. 6 as QoSbox 1, 2 and 3. Other PCs are acting as sources and sinks of traffic. The topology has two bottlenecks: the link between QoSboxes 1 and 2, and the link between QoSboxes 2 and 3. The buffer at the output link of each router is shared, and its total size is set to $B = 200$ packets.

Class	Differentiation parameters				
	d_i	L_i	f_i	k_i	k'_i
1	8 ms	1 %	–	–	–
2	–	–	35 Mbps	2	2
3	–	–	–	2	2
4	–	–	–	N/A	N/A

Table 1: **Desired service differentiation.** d_i denotes a delay bound, L_i denotes a bound on the loss rate, f_i denotes a minimum throughput bound, and k_i (resp. k'_i) denotes the desired ratio of delays (resp. loss rates) between Class $(i+1)$ and Class i. The desired differentiation is identical at all QoSboxes.

Class	No. of flows	Type	
		Protocol	Traffic
1	6	UDP	On-off
2	6	TCP	Greedy/On-off
3	6	TCP	Greedy/On-off
4	6	TCP	Greedy/On-off

Table 2: **Traffic mix.** The traffic mix is identical for each source-sink pair. The on-off UDP sources send bursts of 20 packets during an on-period, and have a 150 ms off-period. TCP sources are greedy during time intervals $[0s, 10s]$, $[20s, 30s]$, and $[40s, 50s]$, and transmit chunks of 8 KB with a pause of 175 ms between each transmission during time intervals $[10, 20s]$, $[30, 40s]$, and $[50s, 60s]$. TCP sources run the *NewReno* congestion control algorithm.

We consider four traffic classes with service differentiation parameters as summarized in Table 1. Class 1 gets absolute service differentiation, while Classes 2, 3 and 4 get proportional service differentiation. In addition, Class 2 is offered a minimum throughput bound.

Sources 1, 2 and 3 send traffic to Sinks 1, 2 and 3, respectively. Each source transmits traffic from all four classes. The traffic mix, the number of flows per class, and the characterization of the flows is identical for each source, and as shown in Table 2. Traffic is generated using the *netperf* v2.1pl3 traffic generator [29]. Each source transmits six flows from each of the classes. Class 1 traffic consists of on-off UDP flows, and the other classes consist of TCP flows. UDP sources start transmitting packets with a fixed size of 1024 Bytes at time $t = 0$ until the end of the experiment at $t = 60$ seconds. We configured the TCP sources to be greedy during time intervals $[0s, 10s]$, $[20s, 30s]$ and $[40s, 50s]$. In the remaining time intervals, the TCP sources send chunks of 8 KB of data and pause for 175 ms between the transmission of each chunk. The chosen traffic mix results in an highly variable offered load at QoSboxes 1 and 2, which we present in

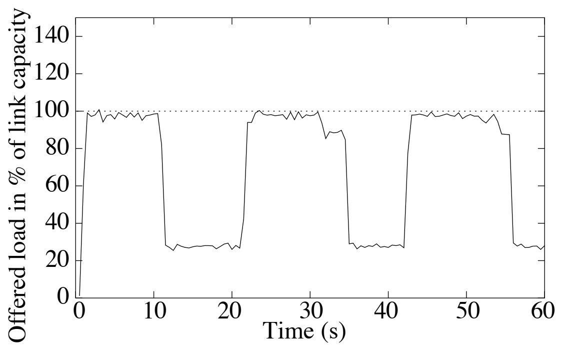


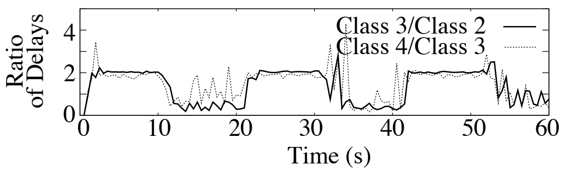
Figure 7: **Offered Load.** The graph shows the offered load at QoSbox 1. The offered load is similar at QoSbox 2.

Fig. 7. Additionally, since the source/sink pairs are connected by different routes, the round-trip times of the different TCP flows in the network are different, reducing the probability of having synchronization effects between TCP sources that belong to different source/sink pairs.

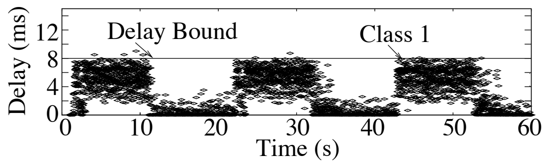
This experimental setup allows us 1) to gauge the reactivity of the QoSbox to rapidly changing traffic conditions, 2) to observe how the QoSbox behaves over relatively long busy periods (in the order of ten seconds), 3) to test a relatively representative set of service differentiation parameters, which is likely to require temprorary relaxations of some service bounds, and 4) to see how the QoSbox reacts to a mix of TCP and UDP traffic. We therefore believe that the experiment tests the most important aspects of the QoSbox, but point out that additional experiments with greedy TCP sources and near-constant loads are available in [10].

4.2 Service Guarantees

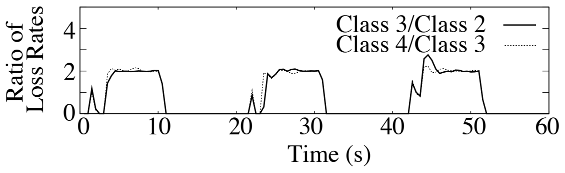
In Figs. 8 and 9, we present our measurements of the service received at the bottleneck links of QoSboxes 1 and 2, respectively. Figs. 8(a) and 9(a) depict the ratios of the delays of Classes 4 and 3, and the ratios of the delays of Classes 3 and 2. Each datapoint is an average over a sliding window of size 0.5 s. The plots show that the target value of $k = 2$ (from Table 1) is achieved when the load is high. Conversely, when the link is underloaded, we observe oscillations in the ratios of delays. This result is due to the fact that proportional delay differentiation cannot be achieved by a work-conserving scheduler when the link is underloaded. In fact, all classes experience queuing delays close to zero when the link is underloaded, and one can therefore argue that there is no need for differentiation since all classes get a high-grade service. We also see that, at times $t = 0$, $t = 20$, and $t = 40$, when the load increases abruptly over a short period of time, the delay differentiation is realized within one datapoint (0.5 s), which shows that the algorithm used in the QoSbox



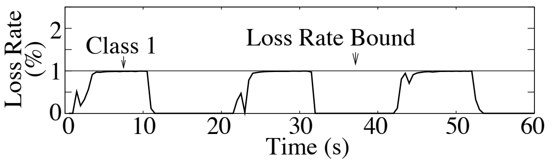
(a) Ratios of Delays.



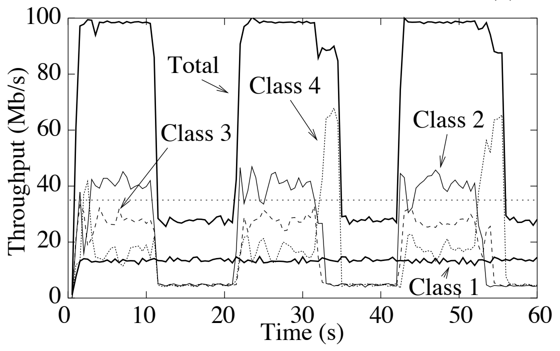
(b) Class-1 Delays (individual).



(c) Ratios of Loss Rates.



(d) Loss Rates.



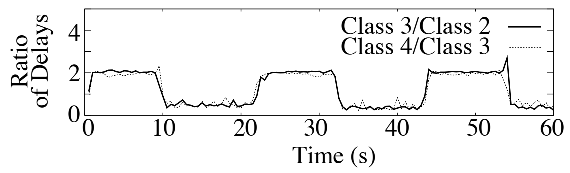
(e) Throughput.

Figure 8: **QoSbox 1.** The graphs show the service obtained by each class at the output link of QoSbox 1.

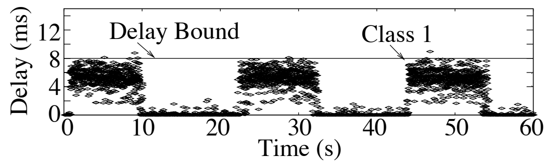
is efficient at providing differentiation reasonably quickly.

In Figs. 8(b) and 9(b) we show the individual delays of Class-1 packets at QoSboxes 1 and 2. The delay bound of $d_1 = 8$ ms is satisfied most of the time. We note that there are a few ($< 1.5\%$) delay bound violations. These delay bound violations are due to the fact that it is impossible to satisfy delay and loss bounds at the same time, since traffic arrivals are not regulated. As explained in Section 2, when such is the case, loss differentiation is given precedence over delay differentiation. Note however, that no Class-1 packet ever experiences a delay higher than 10 ms at either QoSbox 1 or 2. Delay values of other classes, not shown here, are in the range 10–50 ms.

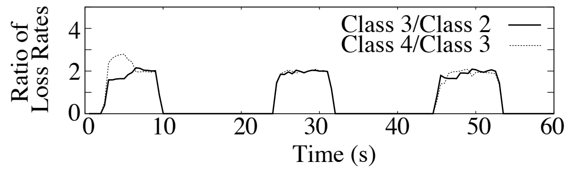
Figs. 8(c) and 9(c) represent plots of ratios of loss rates averaged over a sliding window of size 0.5 s, and show that proportional loss differentiation is realized, with the desired factor $k' = 2$, at times of packet losses. Figs. 8(d) and 9(d) show the loss rate experienced by Class-1 traffic, and we see that, even at times of packet drops, the loss rate of Class 1 remains below the loss bound of 1%. Loss rates of other classes (not shown) are generally below 1%, which indicates that most traffic is dropped to satisfy the delay bound on Class 1, rather than due to buffer overflows.



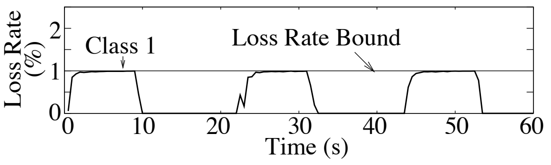
(a) Ratios of Delays.



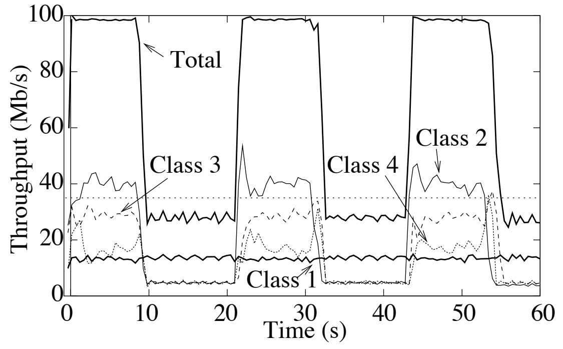
(b) Class-1 Delays (individual).



(c) Ratios of Loss Rates.



(d) Loss Rates.



(e) Throughput.

Figure 9: **QoSbox 2.** The graphs show the service obtained by each class at the output link of QoSbox 2.

Last, Figs. 8(e) and 9(e) present throughput measurements obtained by each class, averaged over a sliding window of size 0.5 s, as well as the aggregate throughput at QoSboxes 1 and 2. We see that whenever Class 2 sources send traffic at a rate of at least 35 Mbps, the desired minimum throughput of 35 Mbps on Class 2 is enforced at QoSboxes 1 and 2. Furthermore, we see that the QoSboxes manage to transmit data at 100 Mbps when needed. Hence, we can infer that the time needed to run the `enqueue` and `dequeue` functions is less than the average transmission time of a packet, and thus, that the overhead associated to the algorithms running in the QoSbox can be considered negligible for this experiment.

In summary, this experiment on a network with multiple bottlenecks and varying load shows that the QoSbox achieves the desired service differentiation and utilizes the entire link capacity when needed.

4.3 Limitations of the QoSbox

The experiment we proposed evidenced how delay bounds were relaxed in favor of loss bounds, which is the main limitation of the QoSbox compared to other proposals relying on admission control.

In addtiion, we mentioned in Section 2 that long busy periods could result in poor loss differentiation, due to the loss rates being computed over long intervals. Huang and Guérin recently exhibited through simulation how such a scenario affects the differentation provided by the JoBS algorithm used in the QoSbox [25]. In their example, they notably found that adequate loss differentiation could be delayed by as much as 400 ms. While particularly interesting to exhibit the limitations of the algorithms we use in the QoSbox, the experiment proposed in [25] relies on on-off constant bit rate sources with an infinitely long busy period, which may be fairly unusual traffic patterns.

4.4 Overhead

We saw that our implementation can fully utilize the capacity of a 100 Mbps link, without overloading the QoSbox. We next present an analysis of the overhead of our implementation, where we attempt to predict the data rates that can be supported by the PC-router implementation of the QoSbox, and where we measure the sensitivity of our implementation to the number of service constraints and to the number of classes. We will show measurements of number of cycles consumed by the `enqueue` and `dequeue` functions for four different sets of service differntiation parameters, tested for four traffic classes.

Set 1:	Same parameters as in Table 1.
Set 2:	Set 1 with absolute bounds from Set 1 removed.
Set 3:	Set 2 with proportional differentiation from Set 1 removed.
Set 4:	No service differentiation.

In the measurements we determine the number of cycles consumed for the `enqueue` and `dequeue` functions, and we indicate the contribution of each function call in `enqueue` to the total overhead. The TSC register of the Pentium processor is read at the beginning and at the end of each of the monitored functions, for each execution of the function.

We compiled our implementation with a code optimizer, in our case, we use the *gcc* v2.95.3 compiler [47] with the "-O2" flag set. The results of our measurements, collected under FreeBSD 4.5 and ALTQ 3.1, are presented in Table 3, where we include the machine cycles consumed by `enqueue` and `dequeue`, and the cycles spent in each of the functions called by `enqueue`. The measurements are averages of over 500,000 datagram transmissions on a heavily loaded link, using the same topology as in Figure 6. The measurements in Table 3 were collected at Router 1. Measurements collected at Router 2 showed deviations of no more than $\pm 5\%$ compared to Router 1. The row containing "FIFO" denotes the overhead of the FIFO queueing discipline in ALTQ, and is used to measure the overhead created by ALTQ itself.

Set	enqueue		dequeue		**Pred.**
	$\overline{X}$	s	$\overline{X}$	s	f_{pred} (Mbps)
1	**11323**	3140	**1057**	316	**291**
2	**10723**	2305	**1092**	340	**305**
3	**3039**	1512	**1138**	348	**864**
4	**2573**	668	**1078**	343	**988**
FIFO	**25**	66	**221**	147	–

Table 3: **Overhead and predicted maximum throughput.** This table presents, for four different sets of service differentiation parameters, the average number of cycles ($\overline{X}$) consumed by the `enqueue` and `dequeue` operations, the standard deviation (s), and the predicted throughput f_{pred} (in Mbps) that can be achieved. In the 1 GHz PCs we use, one cycle roughly corresponds to one nanosecond.

Since the `enqueue` and `dequeue` functions are invoked once for each IP datagram, we can predict the maximum throughput of a PC-router to be

$$f_{pred} = \frac{F}{n_{\text{enqueue}} + n_{\text{dequeue}}} \cdot \overline{P} \, , \qquad (2)$$

where F denotes the CPU clock frequency in Hz, n_{enqueue} denotes the number of cycles consumed by the `enqueue` function, n_{dequeue} denotes the number of cycles consumed by the `dequeue` function, and $\overline{P}$ is the average size of a datagram. The equation given above assumes that clock cycles have a fixed duration, neglects bus contention and operations that occur in an interrupt context (e.g., arrival of a packet at the input link), and does not take into account the cost of packet classification. Thus, Eqn. (2) is an optimistic estimate. In the case of our implementation in 1 GHz PCs, we have $F = 10^9$. Data from a recent report [3] indicates that the packet size distribution on the Internet is multimodal, and that the average size of a packet on the Internet is $\overline{P} \approx 451$ bytes. Using these values for $\overline{P}$ and F in the above equation shows that, in the four sets of constraints considered, we estimate that our implementation can be run at data rates of at least 291 Mbps.[5]

While common PCs can nowadays send/forward traffic at close to 1 Gbps, especially when using large packet sizes, we point out that most bottleneck links in access networks operate at far more modest speeds. For instance, a T-1 line coming out of a small corporate office operates at 1.54 Mbps, and could easily be governed by a QoSbox. For faster bottleneck links (e.g., ISP-ISP interconnects), one would need to

[5]Previous measurements in [12] exhibited slightly more significant overhead under the same type of arrivals, but were collected using an older version of our software, and an older version of the FreeBSD operating system.

Set	`accounting`		`select dropped_class`		`compute min_rates`		`greedy_alloc`		`adjust_rates`	
	$\overline{X}$	s	$\overline{X}$	s	$\overline{X}$	s	$\overline{X}$	s	$\overline{X}$	s
1	777	393	46	585	582	419	3128	667	6471	1512
2	1052	350	44	616	219	135	–	–	6696	1087
3	774	401	1138	1005	568	434	3172	1044	–	–
4	948	382	793	190	202	121	–	–	–	–

Table 4: **Overhead distribution.** This table presents, for the four considered sets of service differentiation parameters, the average number of cycles ($\overline{X}$) consumed by each of the functions called by `enqueue`, and the standard deviation (s). In the 1 GHz PCs we use, one cycle roughly corresponds to one nanosecond.

implement some of the approximations and the parallelism discussed in Section 3.3. Finding an upper bound on the speed at which our algorithms can run, using current hardware, is an open problem, to which the measurements presented here only give a partial answer.

Table 4 indicates that a dominant portion of the overhead is linked to the presence of proportional delay differentiation (Sets 1 and 2). In particular, Table 4 confirms most of the overhead is incurred by functions that can be sampled, notably, the implementation in `greedy_alloc` of the greedy algorithm for the redistribution of the service rates in presence of absolute delay bounds, and the rate adjustment for proportional delay differentiation, in `adjust_rates`. The row corresponding to Set 4 gives some insight as to the cost of the operations that have to be performed on a per-packet basis: without calls to `greedy_alloc` and `adjust_rates`, the overhead is almost negligible. As pointed out in Section 3.3, we can sample `greedy_alloc` and `adjust_rates` so that these operations are performed every T arrivals, where T is in the order of 10–100, without large performance penalties in terms of service differentiation. Taking into account the possibility of sampling the two most time-consumming functions, a simple extrapolation of the results of Table 4 indicates that the overhead, albeit not negligible, appears to be reasonable.

Last, varying the number of classes in Set 1, we gathered the overhead of the `enqueue` and `dequeue` functions in Table 5. Table 5 indicates that, as discussed in Section 3, the overhead of the `enqueue` function appears linear in the number of classes. The small discrepancy observed for $Q = 2$ is linked to the absence of proportional differentiation in that experiment. Measurements for the `dequeue` function tend also to exhibit linearity, but with a much higher constant cost independent of the number of classes.

Number of classes Q	`enqueue`	`dequeue`
2	3094	1000
4	11323	1057
6	15090	1091
8	20656	1224

Table 5: **Overhead in function of the number of classes.** This table shows that the overhead, expressed as an average number of cycles, appears to be linear in the number of classes.

5 Related Work

Tools that facilitate the implementation and configuration of queuing disciplines in PCs have been devised since the days of UNIX System V, with STREAMS [40]. More recent packages, such as ALTQ [9], Netgraph [17], the x-kernel [27], Click [30] and Dummynet [41] allow for implementing sophisticated queuing disciplines in Linux or BSD.

Therefore, the implementation of QoS architectures using PC-routers is not new. For instance, the ALTQ package itself supports natively the CBQ and HFSC [48] schedulers. However, without external admission control, the ALTQ implementations of these QoS schedulers are in practice essentially used to control the bandwidth individual users can receive.

With respect to building fully functional QoS networks, one can cite the attempts at creating DiffServ networks using PC-routers. Implementations of DiffServ components in the Linux 2.1 kernel are for instance discussed in [7]. The authors of [7] integrate traffic policing and scheduling/dropping in the same router, which generates significant overhead, and, as a result, traffic can only be forwarded at approximately 20 Mbps. A similar effort to implement DiffServ components in the Linux kernels has been recently pursued by [5], and as an application of Click [30].

A very different effort is pursued by flow-aware networking [31], which attempts to provide per-flow differentiation to active (i.e., backlogged) flows. The building block of flow-aware networking, is that, at any given time the number of flows simultaneously backlogged in a router is small (in the order of a few hundreds), so that it is possible per-flow differentiation. An implementation in Linux, called the Cross Protect router [32], is available. The major differences with our approach are their use of per-flow service differentiation, and per-flow admission control. We report an investigation of the interaction of flow-aware networking and class-based services as proposed in the QoSbox elsewhere [11].

From the algorithmic perspective, the notion of using a dynamic rate allocation to react to traffic arrivals is not new, and was for instance proposed in [24, 34, 46]. However, these approaches do not fully exploit the interaction between dropping and scheduling, and instead rely on admission control and traffic policing to enforce service differentiation. More recently, the Alternative Best-Effort (ABE, [26]) service has been proposed as a service architecture that does not hinge on admission control. ABE considers only two classes of traffic, and uses the Duplicate Scheduler with Deadlines (DSD) to provide strict delay bounds to one class of traffic, at the expense of a higher loss rate. ABE does not quantify the differentiation in the loss rates obtained by both classes. Similar to the QoSbox, no traffic policing or admission control are required.

6 Conclusion

We discussed the design of the QoSbox, a configurable IP router that provides proportional and absolute service differentiation to classes of traffic on a per-hop basis, by dynamically adapting to the traffic demand. The number of classes or the service differentiation each class obtains are only limited by the computational power available. No admission control or traffic policing is required, and there is no communication between routers.

We presented the implementation of the QoSbox in BSD-based PC-routers, and showed through analysis and measurements that the PC-router implementation is a solution that can be readily used in low/medium-speed access networks with bandwidths in the order of a couple of hundred megabits per second. We also identified a number of approximations and techniques that can be used to implement the QoSbox in high-speed routers used in network cores and interconnects between autonomous systems.

We evaluated the potential of the QoSbox using a testbed PC-routers, and showed that the QoSbox is a promising alternative to traditional architectures relying on admission control and traffic policing for providing quantitative service differentiation.

A version of the QoSbox for BSD kernels is available to the public, along with the source code and documentation at `http://qosbox.cs.virginia.edu/software.html`. The software has been available under the BSD license since late October 2001, and is now distributed as part of ALTQ 3.1, and the KAME [2] snap-kits. Inclusion in the base BSD distributions as part of ALTQ is under consideration.

7 Acknowledgments

We wish to thank Kenjiro Cho for his invaluable advice during the course of the ALTQ implementation.

References

[1] Intel's IXP 1200 network processor. `http://developer.intel.com/design/network/products/npfamily/ixp1200.htm`.

[2] The KAME project. `http://www.kame.net`.

[3] Packet sizes and sequencing, May 2001. `http://www.caida.org/outreach/resources/learn/packetsizes`.

[4] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. IETF RFC 2581, April 1999.

[5] W. Almesberger, J. H. Salim, and A. Kuznetsov. Differentiated services on Linux, June 1999. IETF draft, draft-almesberger-wajhak-diffserv-linux-01.txt. See also `http://diffserv.sourceforge.net`.

[6] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. IETF RFC 2475, December 1998.

[7] R. Bless and K. Wehrle. Evaluation of differentiated services using an implementation under Linux. In *Proceedings of IWQoS'99*, pages 97–106, London, UK, June 1999.

[8] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview. IETF RFC 1633, July 1994.

[9] K. Cho. A framework for alternate queueing: towards traffic management by PC-UNIX based routers. In *Proceedings of USENIX'98 Annual Technical Conference*, pages 247–258, New Orleans, LA, June 1998.

[10] N. Christin. *Quantifiable Service Differentiation for Packet Networks*. PhD thesis, University of Virginia, August 2003.

[11] N. Christin and J. Liebeherr. Marking algorithms for service differentiation of TCP traffic. *Computer Communications*, 28(18):2058–2069, November 2005.

[12] N. Christin, J. Liebeherr, and T. F. Abdelzaher. A quantitative assured forwarding service. In *Proceedings of IEEE INFOCOM'02*, volume 2, pages 864–873, New York, NY, June 2002.

[13] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar. Matching output queueing with a combined input-output queued switch. In *Proceedings of IEEE INFOCOM'99*, volume 3, pages 1169–1178, New York, NY, March 1999.

[14] Compaq Computer Corporation. *Alpha Architecture Handbook*, 4th edition, 1998.

[15] B. Davie, A. Charny, J. Bennett, K. Benson, J.-Y. Le Boudec, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis. An expedited forwarding PHB. IETF RFC 3246, March 2002.

[16] C. Dovrolis and P. Ramanathan. Dynamic class selection: From relative differentiation to absolute QoS. In *Proceedings of ICNP'01*, pages 120–128, Riverside, CA, November 2001.

[17] J. Elischer and A. Cobbs. The Netgraph networking system. Technical report, Whistle Communications, January 1998. `http://www.elischer.com/netgraph/`.

[18] W.-C. Feng, D. Kandlur, D. Saha, and K. Shin. Blue: A new class of active queue management algorithms. Technical Report CSE-TR-387-99, University of Michigan, April 1999.

[19] V. Firoiu, J.-Y. Le Boudec, D. Towsley, and Z.-L. Zhang. Theories and models for Internet quality of service. *Proceedings of the IEEE, Special Issue on Internet Technology*, 90(9):1565–1591, August 2002.

[20] V. Firoiu, X. Zhang, E. Gündüzhan, and N. Christin. Providing service guarantees in high-speed switching systems with feedback output queuing. *IEEE Transactions on Parallel and Distributed Systems*, 2006. In print.

[21] S. Floyd and V. Jacobson. Random early detection for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, July 1993.

[22] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, August 1995.

[23] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured forwarding PHB group. IETF RFC 2597, June 1999.

[24] I. Hsu and J. Walrand. Dynamic bandwidth allocation for ATM switches. *Journal of Applied Probability*, (33):758–771, September 1996.

[25] Y. Huang and R. Guérin. A simple FIFO-based scheme for differentiated loss guarantees. In *Proceedings of IEEE/IFIP IWQoS'04*, pages 96–105, Montreal, QC, Canada, June 2004.

[26] P. Hurley, J.-Y. Le Boudec, P. Thiran, and M. Kara. ABE: providing low delay service within best effort. *IEEE Networks*, 15(3):60–69, May 2001. See also `http://www.abeservice.org`.

[27] N. Hutchinson and L. Peterson. The x-kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[28] Intel Corporation. *Pentium Pro Family Developer's Manual. Volume III: Operating System Writer's Guide*, 1995.

[29] R. Jones. *netperf*: a benchmark for measuring network performance - revision 2.0. Information Networks Division, Hewlett-Packard Company, February 1995. See also `http://www.netperf.org`.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[31] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts. Minimizing the overhead in implementing flow-aware networking. In *Proceedings of ACM ANCS'05*, Princeton, NJ, October 2005.

[32] A. Kortebi, S. Oueslati, and J. Roberts. Cross-protect: implicit service differentiation and admission control. In *Proceedings of IEEE HPSR'04*, pages 56–60, Phoenix, AZ, April 2004.

[33] T.V. Lakshman, A. Neidhardt, and T. Ott. The drop from front strategy in TCP and in TCP over ATM. In *Proceedings of IEEE INFOCOM'96*, pages 1242–1250, San Francisco, CA, March 1996.

[34] R. Liao and A. Campbell. Dynamic core provisioning for quantitative differentiated service. In *Proceedings of IWQoS'01*, pages 9–26, Karlsruhe, Germany, June 2001.

[35] J. Liebeherr and N. Christin. JoBS: Joint buffer management and scheduling for differentiated services. In *Proceedings of IWQoS'01*, pages 404–418, Karlsruhe, Germany, June 2001.

[36] J. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[37] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers. IETF RFC 2474, December 1998.

[38] K. Papagiannaki, D. Veitch, and N. Hohn. Origins of microcongestion in an access router. In *Proceedings of the 5th Passive and Active Measurement Workshop (PAM'04)*, pages 126–136, Juan-les-Pins, France, April 2004.

[39] A. Parekh and R. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[40] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[41] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, January 1997.

[42] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. IETF RFC 3031, January 2001.

[43] Z. Sahinoglu and S. Tekinay. Self-similar traffic and network performance. *IEEE Communications Magazine*, 37(1):48–52, January 1999.

[44] S. Sen and J. Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment (IMW'02)*, pages 137–150, Marseille, France, November 2002.

[45] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, June 1996.

[46] P. Siripongwutikorn, S. Banerjee, and D. Tipper. Adaptive bandwidth control for efficient aggregate qos provisioning. In *Proceedings of IEEE GLOBECOM'02*, volume 3, pages 2634–2638, Taipei, Taiwan, November 2002.

[47] R. Stallman. *Using and Porting the GNU Compiler Collection (gcc)*. iUniverse Inc., 2000.

[48] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *Proceedings of ACM SIGCOMM'97*, pages 249–262, Cannes, France, August 1997.

[49] N. Taft, S. Bhattacharyya, J. Jetcheva, and C. Diot. Understanding traffic dynamics at a backbone POP. In *Proceedings of SPIE ITCOM Workshop on Scalability and Traffic Control in IP Networks*, number 4526, Denver, CO, August 2001.

[50] L. Zhang. Virtual clock: A new traffic control algorithm for packet switched networks. *ACM Transactions on Computer Systems*, 9(2):101–125, May 1991.