The QoSbox: A PC-Router for Quantitative Service Differentiation in IP Networks *

Technical Report: University of Virginia, CS-2001-28

Nicolas Christin *Jörg Liebeherr*
Department of Computer Science
University of Virginia
Charlottesville, VA 22904

Abstract

We describe the design and implementation in UNIX-based PCs of the QoSbox, a configurable IP router that provides per-hop service guarantees on loss, delays and throughput to classes of traffic. There is no restriction on the number of classes or the specific service guarantees each class obtains. 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 enforce the desired service guarantees, but instead, (2) dynamically adapts packet forwarding and dropping decisions as a function of the instantaneous traffic arrivals; also, (3) the QoSbox can enforce both absolute bounds and proportional service guarantees on queueing delays, loss rates, and throughput at the same time. We evaluate the QoSbox in a testbed of PC-routers over a FastEthernet network, and show that the QoSbox is a possible solution allowing for incremental deployment to the problem of providing service differentiation in a scalable manner.

Key Words: High-Speed Networks, Quality-of-Service, Service Differentiation, BSD, PC-Routers.

*This work is supported in part by the National Science Foundation through grants ANI-9730103 and ANI-0085955.

1 Introduction

Since its creation in the early 1970s, the Internet has adopted a "best-effort" service, which relies on the following three principles: (1) No traffic is denied admission to the network, (2) all traffic is treated in the same manner, and (3) the only guarantee given by the network is that traffic will be transmitted in the best possible manner given the available resources, that is, no artificial delays will be generated, and no unnecessary losses will occur.

The best-effort service is adequate as long as the applications using the network are not sensitive to variations in losses and delays (e.g., electronic mail), the load on the network is small, and pricing by network providers is not service-based. These conditions held in the early days of the Internet, when the Internet only consisted of network connections between a handful of universities. However, since the late 1980s, these conditions do not hold anymore, for two main reasons. Firstly, an increasing number of different applications, such as real-time video, peer-to-peer networking, or the World-Wide Web, to name a few, have been using the Internet. These different applications have different needs in the service they must receive from the network. Secondly, the Internet has switched from a government-supported research network to a commercial entity in 1994. Economics indicate that there may be a need for service-based pricing schemes that can better recover cost and maximize revenue than a best-effort network [26]. These two factors have created a demand for different levels of service, and it can be argued that finding a solution to the problem of service differentiation in the network became critical to ensure the survival of the Internet.

The issues of service differentiation in the network are subsumed by the term *Quality-of-Service* (QoS). One could think that QoS issues can be resolved by increasing the capacity of the network. Unfortunately, this solution faces a major problem. The QoS received by an end-to-end traffic flow is bounded by the QoS received at the link with the smallest capacity (i.e., bottleneck) 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 problem, e.g., from the core to the edge of a network. In fact, the capacity of the network links of the Internet has steadily increased in the past couple of years and only resulted in an increase of the traffic that uses these links, without any QoS.

Numerous architectures for providing QoS have been proposed over the past decade. However, only very few proposals have actually been deployed to some extent, in large part due to scalability concerns. A large number of QoS proposals rely on complex arithmetic manipulations, which may cause scalability problems on the data path, and/or complex state information, which may cause scalability problems on the control path.

Nowadays, with the increasing computational power of regular PCs, and the understanding of the aforementioned scalability issues, one can use PCs as routers to implement prototypes of scalable QoS architectures. In this paper, we describe the design and implementation in UNIX-based PCs of the QoSbox, a configurable IP router that provides per-hop service guarantees to classes of traffic. The novel aspects of the QoSbox are that (1) the QoSbox does not rely on any external component such as traffic shaping or admission control to enforce the desired service guarantees, which allows for deploying a network of QoSboxes in an incremental manner, but instead, (2) adapts dynamically packet forwarding and dropping decisions dependent on the instantaneous traffic arrivals; and (3) the QoSbox can enforce both absolute bounds and proportional service guarantees on queueing delays, loss rates, and throughput at the same time.

This paper is organized as follows. In Section 2, we present an overview of the QoSbox, 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 discuss the limitations of our architecture and lessons learned from the implementation. In Section 6, we present the related work. Finally, we summarize the current status of the project and draw brief conclusions in Section 7.

2 The QoSbox

In this section, we discuss the architecture of the QoSbox. To that effect, we first present a high-level overview of the QoSbox. We then describe the requirements the QoSbox architecture has to satisfy, and outline the mechanisms the QoSbox relies on.

2.1 Overview

The QoSbox is a configurable IP router that provides per-hop service differentiation to classes of traffic flows with similar QoS requirements. More specifically, the QoSbox can provide any mix of the following per-class guarantees at a given output link:

- Upper bound on the queueing delay encountered by packets from the same class. For instance, Class-1 Delay $\leq$ 5 ms means that no Class-1 packet should experience a queueing delay exceeding 5 ms.
- Upper bound on the loss rate. For instance, Class-3 Loss Rate $\leq$ 1%.
- Lower bound on the throughput. For instance, Class-1 Throughput $\geq$ 10 Mbps.
- Proportional differentiation between the delays of two classes. For instance, Class-3 Delay $\approx$ 2·Class-2 Delay.
- Proportional differentiation between the loss rates of two classes. For instance, Class-2 Loss Rate $\approx$ 2·Class-1 Loss Rate.

All service guarantees are provided over a finite-length time interval, or *epoch*, 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 epoch.

The key difference between the QoSbox and other QoS architectures is that the QoSbox does not rely on any external mechanism or component to enforce the desired service guarantees. Different from the Differentiated Services architecture (DiffServ, [8]), for instance, there is no need for bandwidth brokers, or traffic shapers, which regulate the traffic arrivals at a given router. Instead, the QoSbox adapts packet scheduling and dropping decisions in function of the instantaneous traffic arrivals. The main limitation of an approach dynamically adapting to the instantaneous traffic arrivals is that, in cases caused for instance by a sudden large burst of traffic, it may be impossible to satisfy all delay bounds and loss rate bounds at the same time. In such cases, some bounds have to be temporarily relaxed. Thus, an order of relaxation of the service guarantees must be chosen at design time. We will further discuss the limitations of our architecture in Section 5.

To illustrate the difference between the QoSbox and other available QoS architectures, suppose one wants to provide a given class of traffic, say Class 1, a minimum throughput of $f_1 = 10$ Mbps and a delay bound of $d_1 = 10$ ms at a given router in the network. Using existing schedulers such as Class-Based

```
(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 2 alc -1 rlc 2 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 guarantees 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.

Queueing (CBQ, [17]), the router needs to know the maximum Class-1 backlog that can be present at any time, $\max_t B_1(t)$, and infer the minimum throughput $\max_t B_1(t)/d_1$ required to ensure that the delay bound is never violated. Since the minimum throughput a class must receive is a parameter statically configured in CBQ, such an approach requires to have a static description of the traffic arrivals available at network configuration time, which in turn requires to have a traffic shaper located at the ingress of the network that ensures actual traffic arrivals conform to the traffic description. Conversely, using a QoSbox, the network operator only needs to specify the QoS parameters, by means of a configuration file, and turn on the QoSbox to obtain the desired service guarantees.

We give an example of such a QoSbox configuration file in Fig. 1. 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 [21] which is the scheduling/dropping algorithm central to the QoSbox. The next set of configuration commands, in lines (2)–(5) contains the service guarantees offered to each class. In this example, we see that Class `high_class` is given a delay bound, indicated by the keyword `adc`, of 2000 microseconds, no proportional delay differentiation, as specified by the field `rdc -1`, a loss bound of 1%, configured by the command `alc 0.01`, no proportional loss differentiation (`rlc -1`) and a guaranteed throughput of 10 Mbps (`arc 10M`). The `priority` field simply indicates the class index. Classes `med1_class`, `med2_class`, `med3_class` are not offered delay or loss bounds, but are subject to proportional delay and loss differentiation, using a factor of two in each case. For instance, `med1_class` packets should get queueing delays twice as long as those experienced by `med2_class` packets. Last, commands in lines (6)–(9) specify how packets should be classified. In this example, we see that the only classification criterion is the Type-of-Service (TOS) field of the IP header, recently renamed DiffServ Codepoint (DSCP, [22]). For instance, a value of 0x03 in the DSCP field of an incoming packet indicates that the packet belongs to Class `med1_class`.

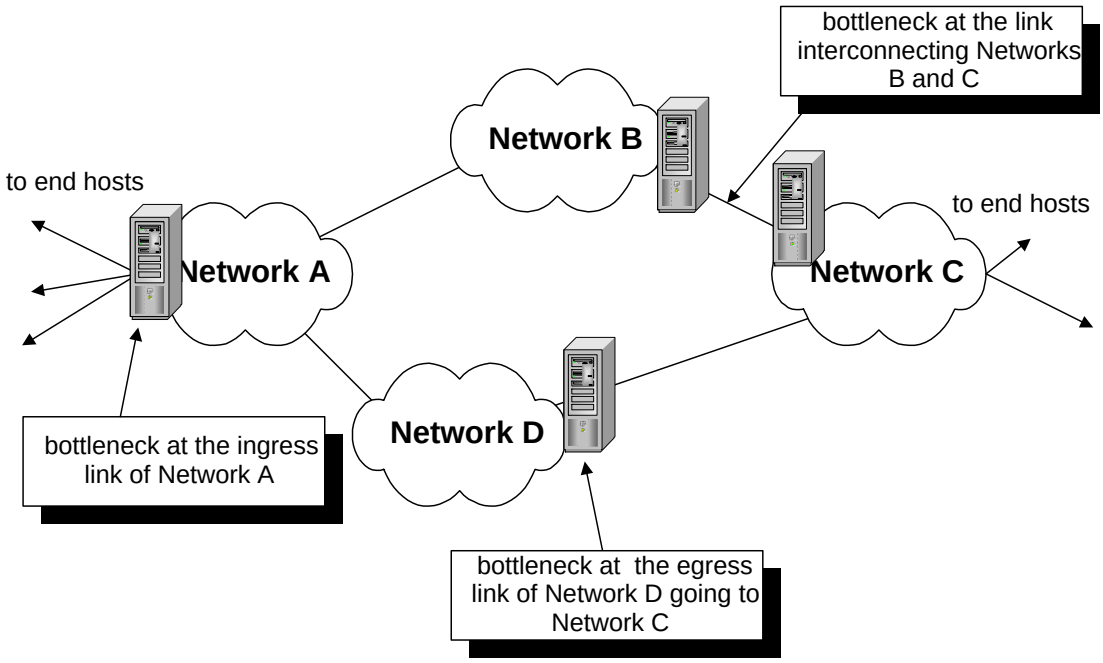


Figure 2: **Positioning of the QoSboxes in a network of networks such as the Internet.** The bottleneck links are those at the entrance of Network A, connecting Networks B and C, and a link in the backbone of Network D.

From this example, we infer that there are two immediate advantages of using a QoSbox. First, the current trend of using TCP or TCP-friendly congestion control mechanisms [15] yields highly variable sending rates at the sources. Consequently, the backlog at bottleneck links is generally highly variable. A static configuration of the worst-case, as in CBQ, therefore results in under-utilizing network resources such as bandwidth [30], and it is desirable to use instead an architecture dynamically adapting to changes in the traffic load. Second, since the QoSbox does not require any external component, deployment and configuration are relatively easy.

We present an example in Fig. 2 to explain how the QoSbox can be deployed. First, we remark that QoS is only needed at potential bottleneck points. Indeed, if a link is never congested, incoming packets can be transmitted at once, using a First-Come-First-Served queueing 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 guarantees. Reports on the utilization of backbone links indicate that backbone links are used at about 60% of their capacity on average [29], which tends to show that bottlenecks are rarely present in the cores of the networks. Thus, we can identify three locations where a bottleneck can occur: at the ingress link of a network, where all users may share a common trunk, as shown in the case of Network A[1] in Fig. 2, at the exchange points between two networks, such as the link between Networks B and C, or at the egress link of a network, as shown for Network D. Placing QoSboxes to regulate the traffic at the bottleneck links can ensure that QoS is enforced where it is needed, without generating any overhead for links that are mostly idle. Additionally, the design of the QoSbox allows for incremental deployment,

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

by first placing QoSboxes at the most severe bottlenecks, and, if feasible, generalize the use of QoSboxes to links that are less likely to be congested.

Another potential use of the QoSbox concerns end-to-end service guarantees. By design, the QoSbox only provides local, per-hop service guarantees. However, the service provided by a QoSbox, used in conjunction with routing mechanisms that can perform route-pinning, such as MPLS [25], can be used to infer end-to-end service guarantees, and select the most appropriate route 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.

2.2 Requirements

With the objective of providing an easily deployable service architecture such as described above, the QoSbox has to exhibit scalability properties, which are enforced by two scalability requirements. First, the state information kept in the routers should be small (scalability on the control path). Second, the processing time for QoS classification and scheduling should be small as well (scalability on the data path). The growing number of flows present in the core 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 ensure scalability 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 guarantee 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.

The objective of the aforementioned requirements is to ensure that there exists some algorithm that can adapt dynamically to the traffic demand, without suffering from any significant computational overhead, regardless of the traffic demand. In this paper, we consider that computational overhead is negligible if all operations performed on a per-packet basis (packet classification, packet enqueueing, and packet dequeueing) can be carried out in less time than the average time needed to transmit a packet on the output link.

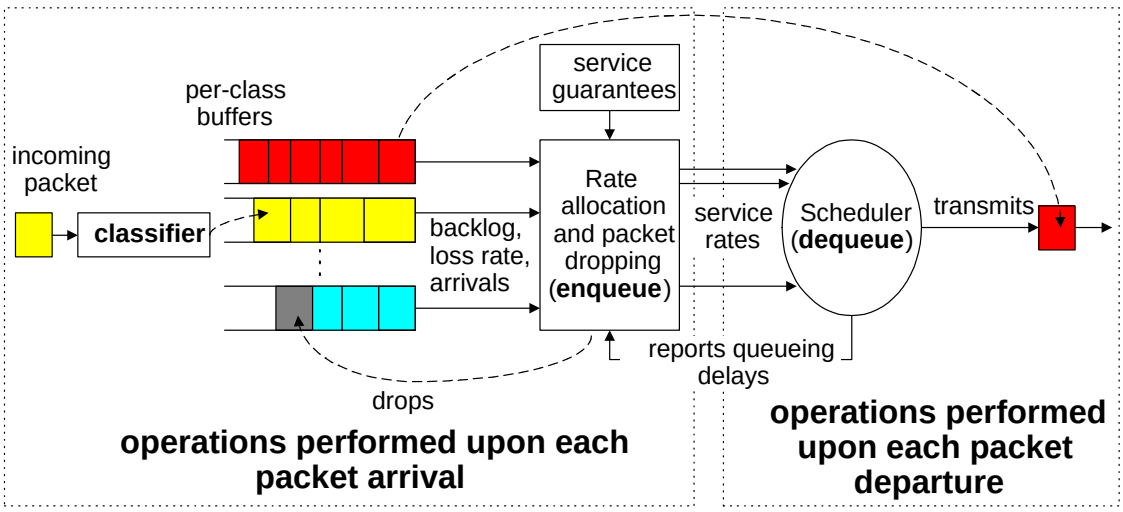


Figure 3: **Architecture of an output queue in the QoSbox.** The three main components are the packet classifier, in charge of storing incoming packets in the proper per-class buffer, the rate allocation and dropping algorithm, and the scheduler, which forwards packets according to the service rates allocated.

2.3 Mechanisms

The mechanisms for service differentiation in the QoSbox are packet dropping and packet scheduling at the output queues of the QoSbox. This is sufficient to achieve the desired service differentiation since traffic control functions only at the entrance of a bottleneck link. In shared-memory routers such as PC-routers, the throughput between the input queues and the output queues of a router is essentially determined by processor speed and memory access speed, and is generally several orders of magnitude larger than the throughput between the output queues and the next routers, which are limited by the output link capacities. Therefore, the input queues are generally empty, and traffic control cannot be performed at the input queues. Hence, from now on, we solely focus on the operations at the output queues.

All output queues in the QoSbox have the same structure, thus, without loss of generality, we now consider a specific output queue. In Fig. 3, we outline the architecture of such an output queue. 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. We will see in Section 3 that such an operation is simple enough to be performed at high speeds. 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 guarantees. The computation of the service rates and packet drops is based on the current backlog, arrivals, loss rate on the one hand, and on queueing delays reported by the scheduler on the other hand. If needed, packets are dropped from the tail of each per-class buffer.

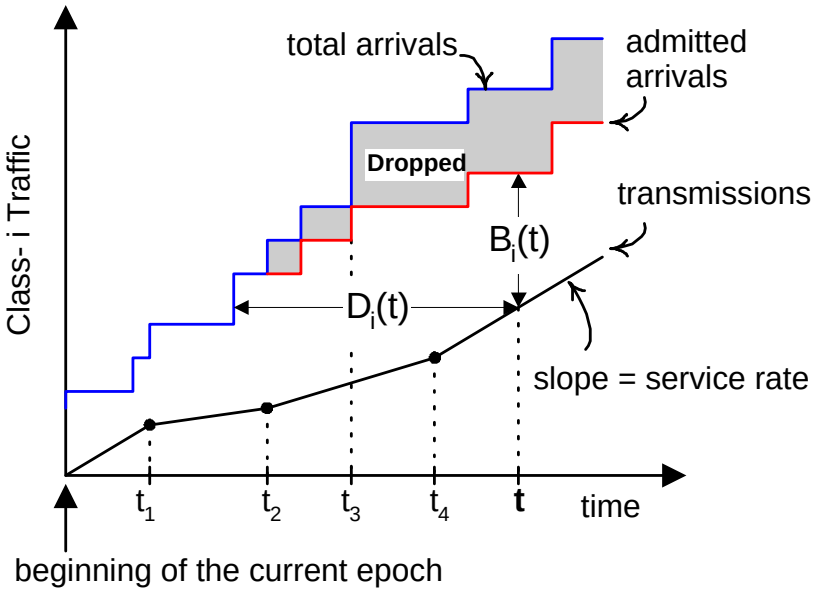


Figure 4: **Overview of the dynamic service rate allocation and packet dropping algorithm.** D_i is the delay and B_i is the backlog. The service rate is dynamically adjusted at times t_1, t_2 and t_4 in function of the traffic arrivals, and traffic is dropped at times t_2 and t_3.

The key difference between the mechanisms used in the QoSbox and mechanisms used in other QoS 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 such a combined scheme lies in the dependency between the backlog of a given class, the delay that packets from this class experience, and the service rate allocated to this class. Indeed, the queueing 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 a 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 service differentiation.

As represented in Fig. 4, 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 epoch. As shown in Fig. 4, 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. 4 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. 4. 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. 4, 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 guarantee. 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 guarantees specified. Then, within the range of feasible service rates for each class, the algorithm selects a rate adjustment which ensures that proportional delay guarantees will be met. Despite the apparent complexity of the algorithm, all these operations can be carried out at high speeds with negligible overhead. 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-GPS [23] or Virtual Clock [31], have been proposed in the early 1990s. However, these schedulers were designed for cases where the rate allocation is essentially 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 [27].

A counter recording the amount of traffic that has been transmitted in each class since the beginning of the current epoch, $Xmit_i$, is maintained by the scheduler. In parallel, every time a packet enters the output queue, an auxiliary counter R_i^{out} is updated as follows

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

where Δt corresponds to the amount of time that has elapsed since the last update of R_i^{out}. R_i^{out} thus corresponds to the amount of traffic that would have been transmitted since the beginning of the current epoch 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 theoretical transmission rate, the scheduler chooses to transmit the packet at the head of the Class-k buffer, and records the queueing delay experienced by the transmitted packet, by taking the difference between the current time and the time this packet was enqueued.

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. 3, 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. 5 the pseudo-code associated with the operations carried out by the rate adjustment and dropping algorithm we propose. We refer to [12] for a theoretical justification of the operations discussed in this paragraph. The `enqueue` procedure described in Fig. 5 is called upon every packet arrival at the output queue of the QoSbox.

```
(1)  procedure enqueue(pkt)
(2)      if (output_link_is_idle())
(3)          reset_all_counters();
(4)          transmit(pkt);
(5)      else
(6)          if (class_left() or class_joined())
(7)              reset_rates();
(8)          while (buffer_overflow())
(9)              i = select_dropped_class();
(10)             drop(i);
(11)         compute_min_rates();
(12)         while (∑min_rates > C and can_drop())
(13)             i = select_dropped_class();
(14)             drop(i);
(15)             compute_min_rates();
(16)         adjust_rates();
(17) return;
```

Figure 5: **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.

First, the algorithm checks the status of the output link. If the output link is not busy transmitting any packets, a new epoch starts with the arrival of the packet. Since all service guarantees are provided over the current epoch, 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 epoch 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 value set *a priori* by the network operator. 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 repeatedly 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 guarantees, 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" its theoretical value. 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 calls the function `compute_min_rates` which determines the minimum service rate each class must be allocated to meet its delay bound and throughput guarantee. 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 of $\frac{B_i}{d_i - D_i}$ and f_i where f_i is the minimum throughput guaranteed to 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. 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. In the design of our algorithm we make the choice of giving higher importance to loss rate bounds than to delay bounds, and thus, the algorithm stops dropping. This design choice is motivated by the fact that over 90% of the traffic on the Internet uses the TCP transport protocol [5]. Due to TCP congestion control mechanisms [6], a TCP source reduces its sending rate upon detection of a packet drop. Thus, relaxing a loss rate bound may have the side-effect of starving some TCP flows, while relaxing a delay bound does not have any such undesirable side-effect.

The final operation carried out in the `enqueue` procedure adjusts the service rates so that proportional delay differentiation is achieved. Earlier work [12] showed that the service rate adjustment could be performed by a simple 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 queueing delay encountered the last Class-i packet to have been transmitted and the theoretical value this queueing delay should have had to satisfy proportional delay differentiation guarantees. 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

Next, we describe a set of issues we addressed during the course of our implementation. First, we ensure that the algorithms used in the QoSbox have low complexity, which is a crucial prerequisite for an implementation with negligible computational overhead. However, having a low algorithmic complexity does

[2] One could drop the packet at the head of the buffer instead. Such a "Drop-From-Front" [20] strategy has the advantage of lowering the queueing delays of all packets still backlogged, but has the major disadvantage of introducing a much more complex coupling between queueing delays and loss rates.

not necessarily guarantee that the computational overhead is negligible. Indeed, the algorithmic complexity evaluation does not take into account the computational cost of some operations, for instance, floating-point divisions, which may consume a large number of CPU cycles. Hence, we next focus on the specifics of the implementation we carried out for BSD kernels using the ALTQ [10] package. To that effect, we first provide a short review of how ALTQ operates, before turning to a discussion of the implementation constraints we had to tackle.

3.1 Algorithmic Complexity

Recall from Section 2 the three main components in the QoSbox architecture: the packet classifier, which maps incoming packets to per-class buffers, the algorithm which adjusts service rates and drop packets, and the packet scheduler which maps the service rate allocation to scheduling decisions.

The only operation performed by the packet classifier is a lookup of the class index carried by an incoming packet, to determine in which buffer the packet should be stored. This lookup is performed by a linear search, thus the algorithmic complexity associated with the packet classifier is $O(Q)$ in the worst-case, where Q is the number of classes supported by the QoSbox. In general, we expect Q to be small, e.g., between 2 and 16, and thus, packet classification can be carried out at high speeds. Similarly, the packet scheduler should be implementable at high speeds, since, by design, the scheduler used has a worst-case complexity of $O(Q)$.

Therefore, the algorithmic complexity in the QoSbox is directly linked to the operations occurring during the enqueueing of an incoming packet, that is, the operations performed by the rate allocation and packet dropping algorithm. Clearly, the proposed algorithm is more complex than its counterparts relying on static descriptions of traffic arrivals. The question is then to know whether this algorithm is still simple enough to be implementable at high speeds.

The first set of operations, represented by lines (2)–(7) in Figure 5, only operates on a per-class basis, and all operations are linear. Thus, the worst-case complexity of this set of operations is $O(Q)$. Next, in lines (8)–(10), packets are dropped in case a buffer overflow occurs. The `while` statement indicates that the worst-case complexity of the operation is $O(SQ)$, with S equal to the maximum number of packets backlogged in each class. However, since this operation is performed upon each packet arrival, S is in fact bounded to the number of packets that have arrived since the last packet arrival. Thus, $S > 1$ only if more than one packet arrive exactly at the same time in the output queue, which is extremely rare, if not impossible, in practice. Therefore, the average-case complexity of lines (8)–(10) is $\Theta(Q)$, which indicates that this segment of code can likely be implemented at high speeds. The computation of the minimum service rates, in line (11), only takes $O(Q)$ as well.

In lines (12)–(15), we find a second `while` loop, which may present higher computational overhead than the first `while` loop discussed above. Here again, the worst-case complexity of this operation is $O(SQ)$ where SQ is the number of packets backlogged in the output queue, but, different from the buffer overflow resolution, we cannot guarantee that the `while` loop is executed only once on average. We thus propose a simple optimization that can be used to ensure a worst-case complexity of $O(Q)$. Instead of choosing which class to drop to respect proportional loss differentiation, one can redistribute 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, drop as much traffic as needed to have the minimum service rate equal to the service rate allocated. This approach, initially

presented in [21], has the advantage of having a worst-case complexity of $O(Q)$, but has the disadvantage of relaxing proportional loss differentiation guarantees when packets are dropped to meet delay bounds.

The final step, described in line (16), which adjusts the service rates subject to proportional delay differentiation constraints has an algorithmic complexity of $O(Q)$. However, some concerns may be raised by the fact that the `adjust_rates` operation relies on the computation of the parameter K, common to all classes. We showed in [12] that the expression of K only required to use some per-class parameters (backlog, delay, minimum service rate and previous service rate allocation). In fact, the computation of K has a worst-case complexity of $O(Q)$, and only requires two divisions. Thus, this segment of code should not cause implementation concerns. It can be pointed out, though, that the rate adjustment for proportional delay differentiation does not need to be performed for every packet arrival. In [21], performing such an adjustment 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. Hence, if processor time is scarce, sampling the rate adjustment may be an option, which comes at the expense of degraded performance with respect to QoS guarantees.

With the assurance that the algorithms at stake present low algorithmic complexity ($\Theta(Q)$ in the average-case, with Q the number of classes), we now turn to the actual implementation.

3.2 ALTQ

The implementation in PC-routers of the QoSbox uses the Alternate Queueing framework (ALTQ, [10]), which is an extension to the FreeBSD, OpenBSD and NetBSD operating system kernels. In addition to various bug fixes to networking devices drivers, ALTQ provides a modular framework for replacing the default FIFO queueing discipline by custom-designed queueing disciplines.

ALTQ operates on output interfaces as follows. 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. As illustrated in Figure 6, ALTQ replaces these functions and structure by user-defined transmission queue structures and functions included in dynamically loadable kernel modules that implement a specific queueing discipline, such as CBQ. Additionally, ALTQ provides a classifier that is used to map incoming packets to classes of traffic. We refer to [10] and [11] for more details on the implementation of ALTQ.

3.3 Implementation Constraints

From what precedes, ALTQ basically provides a way for replacing the standard `enqueue` and `dequeue` functions at an output interface. However, even if the `enqueue` and `dequeue` functions in the QoSbox have relatively low algorithmic complexity, their implementation is not straightforward, since a kernel-level implementation of these functions is subject to three constraints. First, the only clocking mechanism available is the CPU clock. Second, we cannot arbitrarily modify the structure of the IP header. Third, floating-point arithmetic is not usable at the kernel level for efficiency purposes, and counter overflows may occur and have to be addressed. We now describe in more details the approach we used in our implementation to overcome these constraints.

Time measurements. To measure delays and throughput, the algorithm needs to access the CPU clock. A simple solution is to use the `microtime()` system call provided in BSD, which returns the time offset

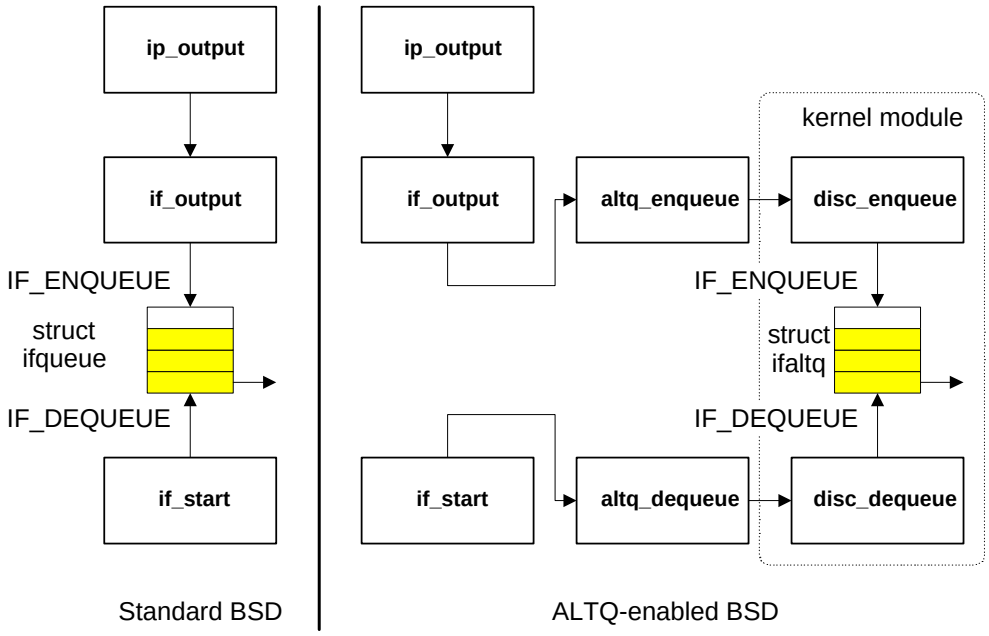


Figure 6: **Functions and structures associated with the output queue in BSD and ALTQ-enabled BSD.** An incoming packet is passed to `ip_output` which, after looking up the route, filling the IP header, and possibly fragmenting the packet passes it to `if_output`; `if_output` enqueues the packet in a `ifqueue` structure. When the output link is available for transmission, a packet is dequeued from the `ifqueue` structure by the `if_start` function. ALTQ provides a way of replacing the operations performed by `if_output` and `if_start` by enqueue and dequeue functions specific to a particular queueing discipline, as denoted by `disc_enqueue` and `disc_dequeue`, as well as a custom-designed packet queue structure (`struct ifaltq`) as a replacement to the `ifqueue` structure.

from 1970 in microseconds. This approach has the advantage of portability, since all BSD systems implement the `microtime()` system call since 4.4-BSD, but has several drawbacks: `microtime()` only provides microsecond granularity, the value returned by `microtime()` is periodically adjusted to account for possible clock skews, and using `microtime()` generates the overhead of the system call (approximately 450 nanoseconds on a PentiumPro 200 MHz [10]). [3] A more efficient solution is to directly read the timestamp counter (TSC) register available in the Pentium series processors [14], and compatible architectures such as recent AMD processors (e.g., Athlon). This register is an unsigned 64-bit precision integer, and gives the number of cycles elapsed since the machine has been turned on. Thus, the resolution of the counter 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 [13]. Our approach is to read the TSC or PCC registers if they are available, and if not, roll back to `microtime()` to ensure portability of our implementation.

IP header limitations. The second issue to be addressed is the fact that the structure of the IP header cannot be modified at will in a kernel-level implementation. This limitation translates into two constraints for implementing the queueing scheme we described in Section 2. First, the only field available in an IPv4 packet header to indicate to which class of traffic a particular packet belongs is the DSCP. In practice, we rely on the ALTQ classifier to read the DSCP and classify the packet in the corresponding per-class buffer. This operation is a linear search, and is thus computationally efficient since the number of classes is expected to be small.

Second, we need to record the arrival times of each packet, since arrival times are required to compute queueing delays which are a key metric in the QoSbox algorithms. To that effect, we can use the IP header timestamp option field, but, in case an IP packet is fragmented, only the first fragment will carry the timestamp. Thus, we prefer the following solution. Each class of traffic is associated with a linked list of timestamps, which is manipulated as follows. Every time a Class-i packet arrives at the output interface, the current time is recorded and enqueued at the tail of the timestamp list. Whenever a Class-i packet is transmitted, the timestamp located at the head of the Class-i timestamp list is removed and used for computation of the queuing delay experienced by the transmitted packet, by simply subtracting the timestamp from the current time. Finally, when a Class-i packet is dropped, the timestamp at the tail of the Class-i timestamp list is discarded as well. Using these operations on the timestamp lists, each Class-i timestamp list matches exactly the corresponding Class-i packet buffer, because interrupts are disabled during the `enqueue` and `dequeue` procedures, and therefore, no race condition can occur.

Arithmetic limitations. The last constraint on the implementation regards arithmetic operations. The rate allocation and dropping algorithm relies on several arithmetic operations (e.g., computation of K). In a network simulator such as *ns-2* [4], 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 due to the prohibitive cost of storing and restoring the FPU state upon each arithmetic computation, and floating-point operations using the FPU emulation library are extremely slow compared to fixed-point arithmetic that can be performed in hardware.

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

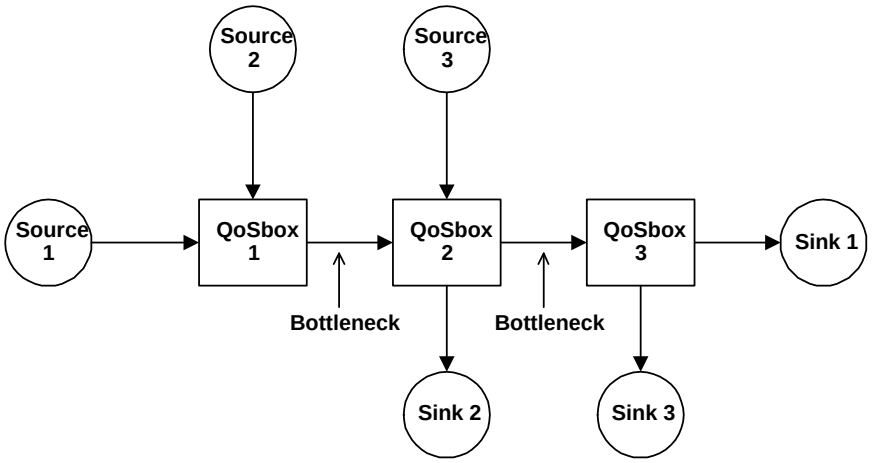


Figure 7: **Network Topology.** All links have a capacity of 100 Mbps. We measure the service provided by QoSboxes 1 and 2 at the indicated bottleneck links.

Class	**Service Guarantees**				
	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: **Service guarantees.** d_i denotes a delay bound, L_i denotes a bound on the loss rate, f_i denotes a minimum throughput guarantee, and k_i (resp. k_i') denotes the desired ratio of delays (resp. loss rates) between Class $(i+1)$ and Class i. The guarantees are identical at all QoSboxes.

Hence, we should only use fixed-point arithmetic in the computations required by the QoSbox algorithms. To that effect, we use 64-bit unsigned integers, which have a high precision, and, yet, can be manipulated with relatively low-cost arithmetic operations. Using fixed-point arithmetic requires to adopt some internal units that differ from "standard units". Namely, in our implementation, 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}. With these units, we can achieve precise calculations using only 64-bit unsigned integers. 64-bit unsigned integers have a very large maximum integer ($\approx 2 \cdot 10^{19}$), but, for the sake of robustness of our implementation, we reset all counters and assume a new epoch starts whenever a counter overflow occurs.

We have shown that the algorithms in the QoSbox have a relatively low computational complexity, and can be realized using low-cost instructions, which allowed us to implement a prototype of the QoSbox in UNIX-based PCs.

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.

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 [1] 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.

4.1 Configuration

We consider a local network topology with multiple nodes and point-to-point Ethernet links, as shown in Fig. 7. All links are full-duplex and have a capacity of $C = 100$ Mbps. Three PCs are set up as routers, indicated in Fig. 7 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.

We consider four traffic classes with service guarantees as summarized in Table 1. Class 1 gets absolute service guarantees, while Classes 2, 3 and 4 get proportional service differentiation.

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 [19]. 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 Fig. 8.

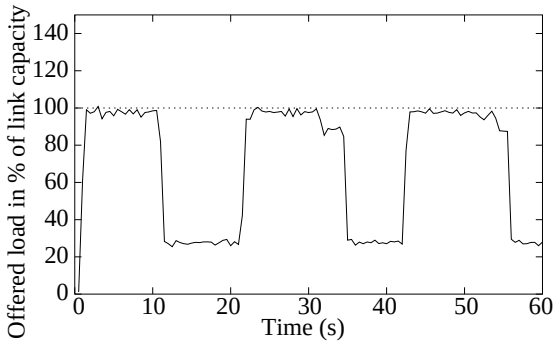


Figure 8: **Offered Load.** The graph shows the offered load at QoSbox 1. The offered load is similar at QoSbox 2.

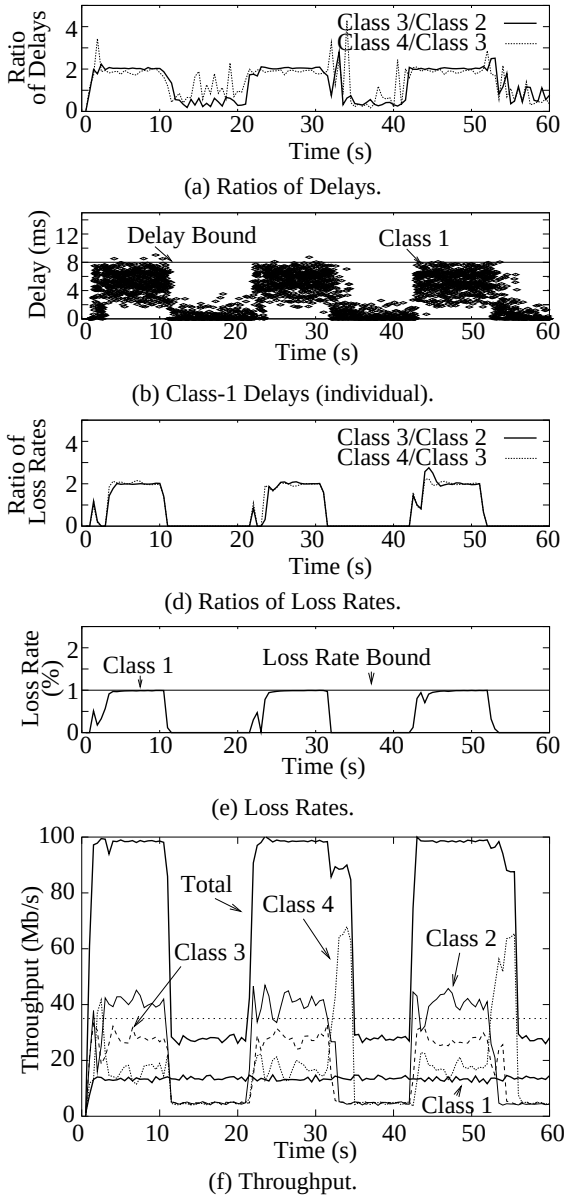
4.2 Service Guarantees

In Figs. 9 and 10, we present our measurements of the service received at the bottleneck links of QoSboxes 1 and 2, respectively. Figs. 9(a) and 10(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 queueing 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 almost immediately, which shows that the algorithm used in the QoSbox is efficient at providing differentiation as soon as possible.

In Figs. 9(b) and 10(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 may be 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 guarantees are given precedence over delay guarantees. 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. We refer to [12] for additional plots.

Figs. 9(c) and 10(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. 9(d) and 10(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 guarantee of 1%. Loss rates of other classes (not shown) are generally below 1% [12] which indicates that traffic is mostly dropped to satisfy the delay bound on Class 1.

Last, Figs. 9(e) and 10(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 minimum throughput guarantee of 35 Mbps on Class 2 is enforced at QoSboxes 1 and 2. Furthermore, we see that the QoSboxes manage to transmit data



(a) Ratios of Delays.

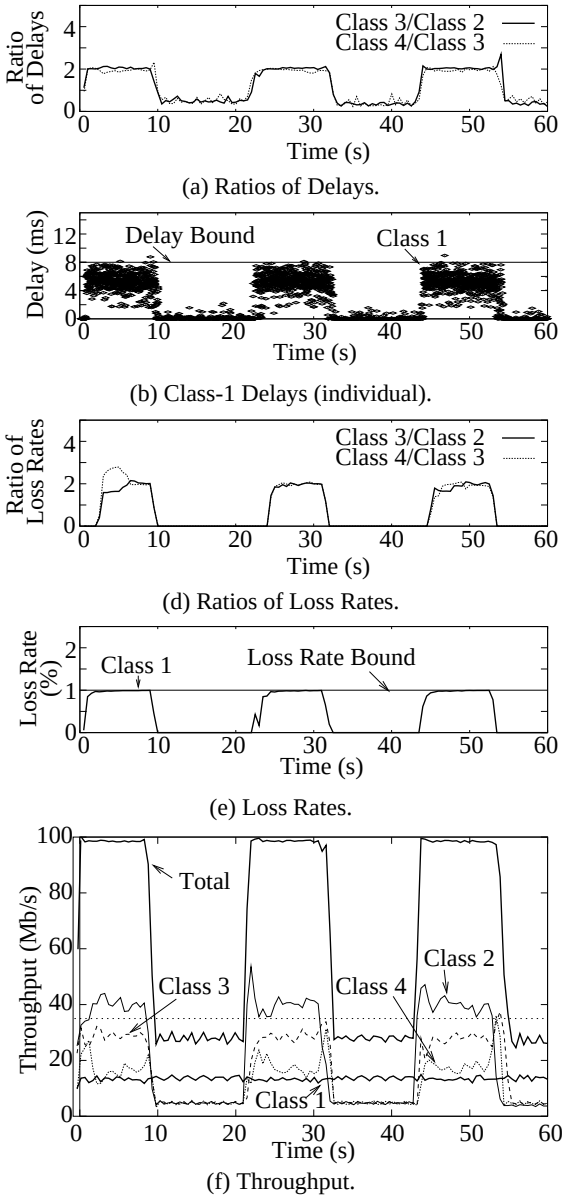
(b) Class-1 Delays (individual).

(d) Ratios of Loss Rates.

(e) Loss Rates.

(f) Throughput.

Figure 9: **QoSbox 1.** The graphs show the service obtained by each class at the output link of QoSbox 1.



(a) Ratios of Delays.

(b) Class-1 Delays (individual).

(d) Ratios of Loss Rates.

(e) Loss Rates.

(f) Throughput.

Figure 10: **QoSbox 2.** The graphs show the service obtained by each class at the output link of QoSbox 2.

Set	enqueue		dequeue		Pred.
	$\overline{X}$	s	$\overline{X}$	s	f_{pred} (Mbps)
1	15347	2603	4053	912	186
2	11849	2798	3580	970	234
3	2671	1101	3811	826	557
4	2415	837	3810	858	580

Table 3: **Overhead Measurements.** This table presents, for the four considered sets of service guarantees, 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 corresponds to one nanosecond.

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 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 this implementation of the QoSbox, and where we measure the sensitivity of our implementation to the number of service constraints. We will show measurements of the enqueue and dequeue operations for four different sets of service guarantees, tested for four traffic classes.

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

In the measurements we determine the number of cycles consumed for the enqueue and dequeue procedures. The TSC register of the Pentium processor is read at the beginning and at the end of the procedures, for each execution of the procedure.

We compiled our implementation with a code optimizer, in our case, we use the *gcc* v2.95.3 compiler with the "-O2" flag set. The results of our measurements are presented in Table 3, where we include the machine cycles consumed by the enqueue and dequeue operations. The measurements are averages of over 500,000 datagram transmissions on a heavily loaded link, using the same topology as in Fig. 7. The measurements in Table 3 were collected at QoSbox 1. Measurements collected at QoSbox 2 showed deviations of no more than $\pm 5\%$ compared to QoSbox 1.

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

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

where F denotes the CPU clock frequency in Hz, n_{enqueue} denotes the number of cycles consumed by the enqueue operation, n_{dequeue} denotes the number of cycles consumed by the dequeue operation, and $\overline{P}$ is the average size of a datagram. The equation given above neglects operations that occur in an interrupt context (e.g., arrival of a packet at the input link) and is thus an estimate. Note, however, that operations performed in an interrupt context must have a negligible overhead for the router to operate properly. In the case of our implementation in 1 GHz PCs, we have $F = 10^9$. Data from a recent report [5] indicates that the average size of an IP datagram on the Internet is $\overline{P} = 451.11$ 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 186 Mbps.

We next evaluate the sensitivity of the performance as a function of the number of constraints. Note from Section 2 that the number of cycles consumed by the dequeue operation is independent of the set of constraints. From Table 3, we see that the overhead associated to the absolute service guarantees (Set 3) is approximately 10% compared to a set with no service guarantees (Set 4). The overhead is 29% when comparing a set with absolute and proportional service guarantees (Set 1) to a set with proportional guarantees only (Set 2). Thus, the overhead incurred by absolute constraints is dependent on the presence of proportional guarantees. This result shows that the computation of the coefficient K, used for proportional differentiation, is more complex when absolute guarantees are present. Proportional guarantees seem to incur more overhead than absolute guarantees, which is essentially due to the computations that need to be performed to dynamically update the value of the coefficient K. However, in the set of constraints considered, there is a larger number of classes with proportional guarantees than classes with absolute guarantees, and thus, more computations are needed to enforce proportional guarantees.

5 Discussion

We next provide a brief discussion of the limitations and ongoing work in the design and implementation of the QoSbox, based on our implementation and experiments.

First, we note that our implementation of the QoSbox using PC-routers has some limitations inherent to the fact that this implementation is a prototype. For example, the flexibility provided by the use of regular PCs as routers comes at the expense of a lack of dedicated hardware, such as multiprocessor systems that can provide true parallelism for tasks such as service rate calculation or packet dequeueing. We are currently working on optimizing the code implemented in our prototype, in order to compensate for this lack of parallelism. With the preliminary experiments run on our testbed, and illustrated in Section 4, we are confident that the algorithms used in the QoSbox do not present any conceptual limitation that would prevent us from obtaining throughputs in the order of 1 Gbps, using specialized hardware such as the Intel IXP 1200 programmable router [2].

More importantly, service limitations are inherent to our decision to avoid to use admission control or traffic shaping. The example of Section 4 showed that, in some rare cases, due to our choice of not performing traffic regulation, enforcing both delay and loss rate bounds at the same time was not feasible.

Our current work focuses on preventing such cases from happening, by relying on TCP congestion control algorithms, which have the sending rate of a source decrease when a packet drop is detected. Thus, we conjecture that, by dropping proactively, i.e., before the traffic arrivals generate an infeasible system of QoS constraints, one can regulate traffic arrivals dynamically, and avoid conflicts between absolute service guarantees. Such a proactive approach has been successfully used in the context of active queue management, e.g., by the RED algorithm [16].

Lastly, we currently trust each incoming packet for carrying the right class index in the DSCP field of the IP header. While this assumption may hold in private networks, some form of security mechanism ensuring that each packet is appropriately marked is required in a public network. To that effect, performing a check on a sample of packets may be an option that we are interested in investigating. On a related note, we are also working on defining a standard set of DSCP values to denote QoSbox classes, in an effort to be compatible with the proposals for Differentiated Services architectures.

6 Related Work

The implementation of QoS architectures using PC-routers is not new. For instance, the ALTQ package itself supports natively the CBQ and HFSC [28] 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 [9]. The authors of [9] integrate traffic policing and scheduling/dropping in the same router. However, the DiffServ architecture only supports quantitative differentiation for the Assured Forwarding (AF) classes, e.g., one class gets higher loss rates than another class, without quantifying the service differentiation. Expediting Forwarding (EF) classes are provided with throughput guarantees, but require the use of admission control and traffic policing. Furthermore, in the implementation described by [9], 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 [7].

More recently, the Alternative Best-Effort (ABE, [18]) service has been proposed. 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. Note that 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. Implementations of DSD in Dummynet [24] for FreeBSD and in the Linux kernel are in progress [18].

7 Conclusion

We presented the design and implementation 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. There is no restriction on the number of classes or the service guarantees each class obtains, and no admission control or traffic policing is required, making the proposed QoS architecture easy to deploy. We evaluated the potential of the QoSbox using PC-routers, and showed that the QoSbox was a

promising solution to the problem of providing service differentiation in a scalable manner.

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. We are currently running extensive tests, and are pursuing a distribution as part of the ALTQ and KAME [3] packages.

Acknowledgments

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

References

[1] The FreeBSD project. http://www.freebsd.org.

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

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

[4] *ns-2* network simulator. http://www.isi.edu/nsnam/ns/.

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

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

[7] 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.

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

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

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

[11] K. Cho. Notes on the new ALTQ implementation, July 2000. Documentation included in the ALTQ 3.0 package. http://www.csl.sony.co.jp/person/kjc/software.html.

[12] N. Christin, J. Liebeherr, and T. Abdelzaher. A quantitative assured forwarding service. Technical Report CS-2001-21, University of Virginia, August 2001. ftp://ftp.cs.virginia.edu/pub/techreports/CS-2001-21.pdf. Short version to appear in *Proceedings of IEEE INFOCOM 2002*.

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

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

[15] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, August 1999.

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

[17] 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.

[18] 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`.

[19] 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`.

[20] 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.

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

[22] 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.

[23] A. K. Parekh and R. G. 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.

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

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

[26] S. Shenker, D. Clark, D. Estrin, and S. Herzog. Pricing in computer networks: reshaping the research agenda. *ACM Computer Communication Review*, 26(2):19–43, April 1996.

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

[28] 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.

[29] 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.

[30] D. E. Wrege, E. W. Knightly, H. Zhang, and J. Liebeherr. Deterministic delay bounds for VBR video in packet-switching networks: fundamental limits and practical trade-offs. *IEEE/ACM Transactions on Networking*, 4(3):352–362, June 1996.

[31] L. Zhang. Virtual clock: A new traffic control algorithm for packet switched networks. *ACM Trans. Comput. Syst.*, 9(2):101–125, May 1991.