The following paper was originally published in the
Proceedings of the USENIX Annual Technical Conference (NO 98)
New Orleans, Louisiana, June 1998

A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers

Kenjiro Cho
Sony Computer Science Laboratory, Inc.

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

A Framework for Alternate Queueing: Towards Traffic Management by PC-UNIX Based Routers

Kenjiro Cho

Sony Computer Science Laboratory, Inc.
Tokyo, Japan 1410022
kjc@csl.sony.co.jp

Abstract

Queueing is an essential element of traffic management, but the only queueing discipline used in traditional UNIX systems is simple FIFO queueing. This paper describes ALTQ, a queueing framework that allows the use of a set of queueing disciplines. We investigate the issues involved in designing a generic queueing framework as well as the issues involved in implementing various queueing disciplines. ALTQ is implemented as simple extension to the FreeBSD kernel including minor fixes to the device drivers. Several queueing disciplines including CBQ, RED, and WFQ are implemented onto the framework to demonstrate the design of ALTQ. The traffic management performance of a PC is presented to show the feasibility of traffic management by PC-UNIX based routers.

1 Introduction

Traffic management is of great importance to today's packet networks. Traffic management consists of a diverse set of mechanisms and policies, but the heart of the technology is a packet scheduling mechanism, also known as queueing. Sophisticated queueing can provide performance bounds of bandwidth, delay, jitter, and loss, and thus, can meet the requirements of real-time services. Queueing is also vital to best-effort services to avoid congestion and to provide fairness and protection, which leads to more stable and predictable network behavior. There has been a considerable amount of research related to traffic management and queueing over the last several years.

Many queueing disciplines have been proposed and studied to date by the research community, mostly by analysis and simulation. Such disciplines are not, however, widely used because there is no easy way to implement them into the existing network equipment. In BSD UNIX, the only queueing discipline implemented is a simple tail-drop FIFO queue. There is no general method to implement an alternative queueing discipline, which is the main obstacle to incorporating alternative queueing disciplines.

On the other hand, the rapidly increasing power of PCs, emerging high-speed network cards, and their dropping costs make it an attractive choice to implement an intelligent queueing on PC-based routers. Another driving force behind PC-based routers is flexibility in software development as the requirements for a router are growing.

In view of this situation, we have designed and built ALTQ, a framework for alternate queueing. ALTQ allows implementors to implement various queueing disciplines on PC-based UNIX systems. A set of queueing disciplines are implemented to demonstrate the traffic management abilities of PC-UNIX based routers.

ALTQ is designed to support a variety of queueing disciplines with different components: scheduling strategies, packet drop strategies, buffer allocation strategies, multiple priority levels, and non-work conserving queues. Different queueing disciplines can share many parts: flow classification, packet handling, and device driver support. Therefore, researchers will be able to implement a new queueing discipline without knowing the details of the kernel implementations.

Our framework is designed to support both research and operation. Once such a framework is widely deployed, research output by simulation can be easily implemented and tested with real machines and networks. Then, if proved useful, the discipline can be brought into routers in practical service. Availability of a set of queueing disciplines will raise public awareness of traffic management issues, which in turn raises research incentives to attack hard problems.

Another important issue to consider is deployment of the framework. To this end just a framework is not enough. In order to make people want to use it, we have to have concrete queueing disciplines which have specific applications. Therefore, we have implemented

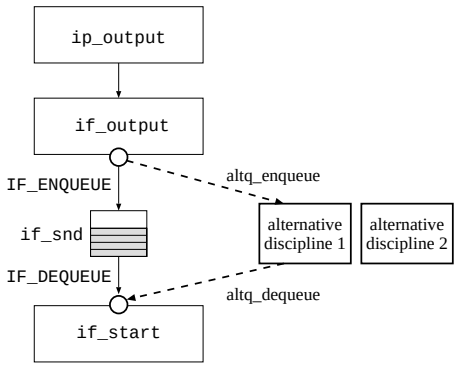

Figure 1: Alternate Queueing Architecture

Class-Based Queueing (CBQ) [6] and Random Early Detection (RED) [5] targeting two potential user groups. CBQ can be used for integrated services. RSVP [22] is a resource reservation protocol for integrated services; RSVP itself is a signaling protocol to set up the traffic control module of the routers along a path. The RSVP release from ISI [9] does not have a traffic control module so that there are great demands for a queueing implementation capable of traffic control. Although CBQ was not originally designed for use with RSVP, CBQ has been used by Sun as a traffic control module for RSVP on Solaris and the source code has been available [20].

RED is for active queue management of best-effort services [2]. Active queue management has been extensively discussed to avoid congestion in the Internet. Although CBQ can be used for this purpose, RED seems more popular since RED does not require flow states or class states, and thus, is simpler and more scalable. Again, no implementation is available at hand.

Another important factor for widespread acceptance is simplicity and stability of the implementation, which caused us to emphasize practicality instead of elegance while designing ALTQ.

In summary, the goals of our prototype are three-fold:

- provide a queueing research platform.
- provide a traffic control kernel for RSVP.
- make active queue management available.

In this paper, we will show the design and implementation of ALTQ, and report the traffic management performance of the prototype on FreeBSD [7].

2 ALTQ

2.1 ALTQ Design

The basic design of ALTQ is quite simple; the queueing interface is a switch to a set of queueing disciplines as shown in Figure 1. Alternate queueing is used only

```
s = splimp();
if (IF_QFULL(&ifp->if_snd)) {
    IF_DROP(&ifp->if_snd);
    splx(s);
    m_freem(m);
    return(ENOBUFS);
}
IF_ENQUEUE(&ifp->if_snd, m);
if ((ifp->if_flags & IFF_OACTIVE) == 0)
    (*ifp->if_start)(ifp);
splx(s);
```

Figure 2: Enqueue Operation in *if_output*

for the output queue of a network interface. The input queue has less importance because traffic control functions only at the entrance to a bottleneck link. In BSD UNIX, an output queue is implemented in the abstract interface structure *ifnet*. The queue structure, *if_snd*, is manipulated by *IF_ENQUEUE()* and *IF_DEQUEUE()* macros. These macros are used by two functions registered in *struct ifnet*, *if_output* and *if_start*; *if_output* defined for each link type performs the enqueue operation, and *if_start* defined as part of a network device driver performs the dequeue operation [13].

One might think that just replacing *IF_ENQUEUE()* and *IF_DEQUEUE()* will suffice, but, unfortunately, it is not the case. The problem is that the queueing operations used inside the kernel are not only enqueueing and dequeueing. In addition, surprisingly many parts of the kernel code assume FIFO queueing and the *ifqueue* structure.

To illustrate the problem, let's take a look at the enqueue operation in a typical *if_output* in Figure 2. The code performs three operations related to queueing.

1. check if the queue is full by *IF_QFULL()*, and if so, drop the arriving packet.
2. enqueue the packet by *IF_ENQUEUE()*.
3. call the device driver to send out the packet unless the driver is already busy.

The code assumes the tail-drop policy, that is, the arriving packet is dropped. But the decision to drop and the selection of a victim packet should be up to a queueing discipline. Moreover, in a random drop policy, the drop operation often comes after enqueueing an arriving packet. The order of the two operations also depends on a queueing discipline. Furthermore, in a non-work conserving queue, enqueueing a packet does not mean the packet is sent out immediately, but rather, the driver should be invoked later at some scheduled timing. Hence, in order to implement a generic queueing interface, we have no choice but to replace part of the code in *if_output* routines.

There are also problems in *if_start* routines. Some drivers peek at the head of the queue to see if the driver

has enough buffer space and/or DMA descriptors for the next packet. Those drivers directly access *if_snd* using different methods since no procedure is defined for a peek operation. A queueing discipline could have multiple queues, or could be about to dequeue a packet other than the one at the head of the queue. Therefore, the peek operation should be part of the generic queueing interface. Although it is possible in theory to rewrite all drivers not to use peek operations, it is wise to support a peek operation, considering the labor required to modify the existing drivers. A discipline must guarantee that the peeked packet will be returned by the next dequeue operation.

IF_PREPEND() is defined in BSD UNIX to add a packet at the head of the queue, but the prepend operation is intended for a FIFO queue and should not be used for a generic queueing interface. Fortunately, the prepend operation is rarely used—with the exception of one popular Ethernet driver of FreeBSD. This driver uses *IF_PREPEND()* when there are not enough DMA descriptors available, to put back a dequeued packet. We had to modify this driver to use a peek-and-dequeue method instead.

Another problem in *if_start* routines is a queue flush operation to empty the queue. Since a non-work conserving queue cannot be emptied by a dequeue loop, the flush operation should be defined.

In summary, the requirements of a queueing framework to support various queueing disciplines are:

- a queueing framework should support enqueue, dequeue, peek, and flush operations.
- an enqueue operation is responsible for dropping packets and starting drivers.
- drivers may use a peek operation, but should not use a prepend operation.

2.2 ALTQ Implementation

Our design policy is to make minimal changes to the existing kernel, but it turns out that we have to modify both *if_output* routines and *if_start* routines because the current queue operations do not have enough abstraction. Modifying *if_start* means modifications to drivers and it is not easy to modify all the existing drivers. Therefore, we took an approach that allows both modified drivers and unmodified drivers to coexist so that we can modify only the drivers we need, and incrementally add supported drivers. This is done by leaving the original queueing structures and the original queueing code intact, and adding a hook to switch to alternate queueing. By doing this, the kernel, unless alternate queueing is enabled, follows the same sequence using the same references as in the original—with the exception of test of the hook. This method has other advantages; the system can fall back to the original queueing if something goes wrong. As a result, the system becomes more reliable, easier to use, and easier to debug. In addition, it is compatible with the existing user programs that refer to *struct ifnet* (e.g., ifconfig and netstat).

Queueing disciplines are controlled by *ioctl* system calls via a queueing device (e.g., */dev/cbq*). ALTQ is defined as a character device and each queueing discipline is defined as a minor device of ALTQ. To activate an alternative queueing discipline, a privileged user program opens the queue device associated with the discipline, then, attaches the discipline to an interface and enables it via the corresponding *ioctl* system calls. When the alternative queueing is disabled or closed, the system falls back to the original FIFO queueing.

Several fields are added to *struct ifnet* since *struct ifnet* holds the original queue structures, and is suitable to place the alternate queueing fields. The added fields are a discipline type, a common state field, a pointer to a discipline specific state, and pointers to discipline specific enqueue/dequeue functions.

Throughout the modifications to the kernel for ALTQ, the *ALTQ_IS_ON()* macro checks the ALTQ state field in *struct ifnet* to see if alternate queueing is currently used. When alternate queueing is not used, the original FIFO queueing code is executed. Otherwise, the alternative queue operations are executed.

Two modifications are made to *if_output* routines. One is to pass the protocol header information to the enqueue operation. The protocol header information consists of the address family of a packet and a pointer to the network layer header in the packet. Packet classifiers can use this information to efficiently extract the necessary fields from a packet header. Packet marking can also be implemented using the protocol header information. Alternatively, flow information can be extracted in the network layer, or in queueing operations without the protocol header information. However, if flow information extraction is implemented in the network layer, the *if_output* interface should be changed to pass the information to *if_output* or auxiliary data should be added to *mbuf* structure, which affects fairly large part of the kernel code. On the other hand, if flow information extraction is implemented entirely in enqueue operations, it has to handle various link-level headers to locate the network layer header. Since we have to modify *if_output* routines anyway to support the enqueue operation of ALTQ, our choice is to save the minimum information in *if_output* before prepending the link header and pass the protocol header information to the enqueue operation.

The second modification to *if_output* routines is to support the ALTQ enqueue operation as shown in Figure 3. The ALTQ enqueue function is also responsible for drop-

```
    s = splimp();
#ifdef ALTQ
    if (ALTQ_IS_ON(ifp)) {
        error = (*ifp->if_altqenqueue)(ifp, m,
                        &pr_hdr, ALTEQ_NORMAL);
        if (error) {
            splx(s);
            return (error);
        }
    }
    else {
#endif
        if (IF_QFULL(&ifp->if_snd)) {
            IF_DROP(&ifp->if_snd);
            splx(s);
            m_freem(m);
            return(ENOBUFS);
        }
        IF_ENQUEUE(&ifp->if_snd, m);
        if ((ifp->if_flags & IFF_OACTIVE) == 0)
            (*ifp->if_start)(ifp);
#ifdef ALTQ
    }
#endif
    splx(s);
```

Figure 3: Modified Enqueue Operation in *if_output*

```
#ifdef ALTQ
    if (ALTQ_IS_ON(ifp))
        m = (*ifp->if_altqdequeue)(ifp,
                                ALTDQ_DEQUEUE);
    else
#endif
        IF_DEQUEUE(&ifp->if_snd, m);
```

Figure 4: Modified Dequeue Operation in *if_start*

ping packets and starting the driver.

Similarly, *if_start* routines are modified to use alternate queueing as shown in Figure 4. A peek operation and a flush operation can be done by calling a dequeue routine with *ALTDQ_PEEK* or *ALTDQ_FLUSH* as a second parameter. Network device drivers modified to support ALTQ can be identified by setting the *ALTQF_READY* bit of the ALTQ state field in *struct ifnet*. This bit is checked when a discipline is attached.

3 Queueing Disciplines

3.1 Overview of Implemented Disciplines

First, we briefly review the implemented disciplines. The details of the mechanisms, simulation results, and analysis can be found elsewhere [6, 21, 5, 14, 3, 11, 12].

CBQ (Class-Based Queueing)

CBQ was proposed by Jacobson and has been studied by Floyd [6]. CBQ has given careful consideration to implementation issues, and is implemented as a STREAMS module by Sun, UCL and LBNL [21]. Our CBQ code is ported from CBQ version 2.0 and enhanced.

CBQ achieves both partitioning and sharing of link bandwidth by hierarchically structured classes. Each class has its own queue and is assigned its share of bandwidth. A child class can borrow bandwidth from its parent class as long as excess bandwidth is available.

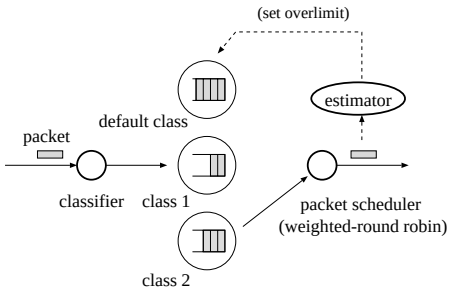

Figure 5: CBQ Components

Figure 5 shows the basic components of CBQ. CBQ works as follows: The classifier assigns arriving packets to the appropriate class. The estimator estimates the bandwidth recently used by a class. If a class has exceeded its predefined limit, the estimator marks the class as overlimit. The scheduler determines the next packet to be sent from the various classes, based on priorities and states of the classes. Weighted-round robin scheduling is used between classes with the same priority.

RED (Random Early Detection)

RED was also introduced by Floyd and Jacobson [5]. RED is an implicit congestion notification mechanism that exercises packet dropping or packet marking stochastically according to the average queue length. Since RED does not require per-flow state, it is considered scalable and suitable for backbone routers. At the same time, RED can be viewed as a buffer management mechanism and can be integrated into other packet scheduling schemes.

Our implementation of RED is derived from the RED module in the NS simulator version 2.0. Explicit Congestion Notification (ECN) [4], a packet marking mechanism under standardization process, is experimentally supported. RED and ECN are integrated into CBQ so that RED and ECN can be enabled on a class queue basis.

WFQ (Weighted-Fair Queueing)

WFQ [14, 3, 11] is the best known and the best studied queueing discipline. In a broad sense, WFQ is a discipline that assigns a queue for each flow. A weight can be assigned to each queue to give a different proportion of the network capacity. As a result, WFQ can provide protection against other flows. In the queueing research community, WFQ is more precisely defined as the specific scheduling mechanism proposed by Demers et al. [3] that is proved to be able to provide worst-case end-to-

end delay bounds [15]. Our implementation is not WFQ in this sense, but is closer to a variant of WFQ, known as SFQ or stochastic fairness queueing [12]. A hash function is used to map a flow to one of a set of queues, and thus, it is possible for two different flows to be mapped into the same queue. In contrast to WFQ, no guarantee can be provided by SFQ.

FIFOQ (First-In First-Out Queueing)

FIFOQ is nothing but a simple tail-drop FIFO queue that is implemented as a template for those who want to write their own queueing disciplines.

3.2 Implementation Issues

There are issues and limitations which are generic when porting a queueing discipline to the ALTQ framework. We discuss these issues in this section, but details specific to particular queueing disciplines are beyond the scope of this paper.

Implementing a New Discipline

We assume that a queueing discipline is evaluated by simulation, and then ported onto ALTQ. The NS simulator [18] is one of a few simulators that support different queueing disciplines. The NS simulator is widely used in the research community and includes the RED and CBQ modules.

To implement a new queueing discipline in ALTQ, one can concentrate on the enqueue and dequeue routines of the new discipline. The FIFOQ implementation is provided as a template so that the FIFOQ code can be modified to put a new queueing discipline into the ALTQ framework. The basic steps are just to add an entry to the ALTQ device table, and then provide open, close, and ioctl routines. The required *ioctls* are attach, detach, enable, and disable. Once the above steps are finished, the new discipline is available on all the interface cards supported by ALTQ.

To use the added discipline, a privileged user program is required. Again, a daemon program for FIFOQ included in the release should serve as a template.

Heuristic Algorithms

Queueing algorithms often employ heuristic algorithms to approximate the ideal model for efficient implementation. But sometimes properties of these heuristics are not well studied. As a result, it becomes difficult to verify the algorithm after it is ported into the kernel.

The *Top-Level link-sharing* algorithm of CBQ suggested by Floyd [6] is an example of such an algorithm. The algorithm employs heuristics to control how far the scheduler needs to traverse the class tree. The suggested heuristics work fine with their simulation settings, but do not work so well under some conditions. It requires time-consuming efforts to tune parameters by heuristics. Although good heuristics are important for efficient implementation, heuristics should be carefully used and study of properties of the employed heuristics will be a great help for implementors.

Blocking Interrupts

Interrupts should be blocked when manipulating data structures shared with the dequeue operation. Dequeue operations are called in the device interrupt level so that the shared structures should be guarded by blocking interrupts to avoid race conditions. Interrupts are blocked during the execution of *if_start* by the caller.

Precision of Integer Calculation

32-bit integer calculations easily overflow or underflow with link bandwidth varying from 9600bps modems to 155Mbps ATM. In simulators, 64-bit double precision floating-point is available and it is reasonable to use it to avoid precision errors. However, floating-point calculation is not available or not very efficient in the kernel since the floating-point registers are not saved for the kernel (in order to reduce overhead). Hence, algorithms often need to be converted to use integers or fixed-point values. Our RED implementation uses fixed-point calculations converted from floating-point calculations in the NS simulator. We recommend performing calculations in the user space using floating-point values, and then bringing the results into the kernel. CBQ uses this technique. The situation will be improved when 64-bit integers become more commonly used and efficient.

Knowing Transfer Completion

Queueing disciplines may need to know the time when a packet transmission is completed. CBQ is one of such disciplines. In BSD UNIX, the *if_done* entry point is provided as a callback function for use when the output queue is emptied [13], but no driver supports this callback. In any case, a discipline needs to be notified when each packet is transferred rather than when the queue is emptied. Another possible way to know transfer completion is to use the callback hook of a memory buffer (e.g., *mbuf cluster*) so that the discipline is notified when the buffer is freed. The CBQ release for Solaris uses this technique. The problem with this method is that the callback is executed when data finishes transferring to the interface, rather than to the wire. Putting a packet on the wire takes much longer than DMAing the packet to the interface. Our CBQ implementation does not use these callbacks, but estimates the completion time from the packet size when a packet is queued. Though this is not an ideal solution, it is driver-independent and provides estimates good enough for CBQ.

Time Measurements

One should be aware of the resolution and overhead of getting the time value. Queueing disciplines often need to measure time to control packet scheduling. To get wall clock time in the kernel, BSD UNIX provides a *microtime()* call that returns the time offset from 1970 in microseconds. Intel Pentium or better processor has a 64-bit time stamp counter driven by the processor clock, and this counter can be read by a single instruction. If the processor clock is 200MHz, the resolution is 5 nanoseconds. The PC based UNIX systems use this counter for *microtime()* when available. Alternatively, one can directly read the time stamp counter for efficiency or for high resolution. Even better, the counter value is never adjusted as opposed to *microtime()*. The problem is that processors have different clocks so that the time stamp counter value needs to be normalized to be usable on different machines. Normalization requires expensive multiplications and divisions, and the low order bits are subject to rounding errors. Thus, one should be careful about precision and rounding errors. Since *microtime()* requires only a microsecond resolution, it is coded to normalize the counter value by a single multiplication and to have enough precision. *Microtime()* takes about 450 nanoseconds on a PentiumPro 200MHz machine. The ALTQ implementation currently uses *microtime()* only.

Timer Granularity

Timers are frequently used to set timeout-process routines. While timers in a simulator have almost infinite precision, timers inside the kernel are implemented by an interval timer and have limited granularity. Timer granularity and packet size are fundamental factors to packet scheduling.

To take one example, the accuracy of the bandwidth control in CBQ relies on timer granularity in the following way: CBQ measures the recent bandwidth use of each class by averaging packet intervals. CBQ regulates a class by suspending the class when the class exceeds its limit. To resume a suspended class, CBQ needs a trigger, either a timer event or a packet input/output event. In the worst case scenario where there is no packet event, resume timing is rounded up to the timer granularity. Most UNIX systems use 10 msec timer granularity as default, and CBQ uses 20 msec as the minimum timer.

Each class has a variable *maxburst* and can send at most *maxburst* back-to-back packets. If a class sends *maxburst* back-to-back packets at the beginning of a 20 msec cycle, the class gets suspended and would not be resumed until the next timer event—unless other event triggers occur. If this situation continues, the transfer rate becomes

$$rate = packetsize \times maxburst \times 8 \div 0.02$$

Now, assume that *maxburst* is 16 (default) and the packet size is the link MTU. For 10baseT with a 1500-byte MTU, the calculated rate is 9.6Mbps. For ATM with a 9180-byte MTU, the calculated rate is 58.8Mbps.

A problem arises with 100baseT; it is 10 times faster than 10baseT, but the calculated rate remains the same as 10baseT. CBQ can fill only 1/10 of the link bandwidth. This is a generic problem in high-speed network when packet size is small compared to the available bandwidth. Because increasing *maxburst* or the packet size by a factor of 10 is problematic, a fine-grained kernel timer is required to handle 100baseT. Current PCs seem to have little overhead even if timer granularity is increased by a factor of 10. The problem with 100baseT and the effect of a fine-grained timer are illustrated in Section 4.3.

Depending solely on the kernel timer is, however, the worst case. In more realistic settings, there are other flows or TCP ACKs that can trigger CBQ to calibrate sending rates of classes.

Slow Device with Large Buffer

Some network devices have large buffers and a large send buffer adversely affects queueing. Although a large receive buffer helps avoid overflow, a large send buffer just spoils the effect of intelligent queueing, especially when the link is slow. For example, if a device for a 128Kbps link has a 16KB buffer, the buffer can hold 1 second worth of packets, and this buffer is beyond the control of queueing. The problem is invisible under FIFO queueing. However, when better queueing is available, the send buffer size of a device should be set to the minimum amount that is required to fill up the pipe.

4 Performance

In this section, we present the performance of the implemented disciplines. Note that sophisticated queueing becomes more important at a bottleneck link, and thus, its performance does not necessarily correspond to the high-speed portion of a network.

We use primarily CBQ to illustrate the traffic management performance of PC-UNIX routers since CBQ is the most complex and interesting of the implemented disciplines. That is, CBQ is non-work conserving, needs a classifier, and uses the combination scheduling of priority and weighted-round robin. However, we do not attempt to outline the details specific to CBQ.

4.1 Test System Configuration

We have measured the performance using three PentiumPro machines (all 200MHz with 440FX chipset) running FreeBSD-2.2.5/altq-1.0.1. Figure 6 shows the test system configuration. Host A is a source, host B is a router, and host C is a sink. CBQ is enabled only

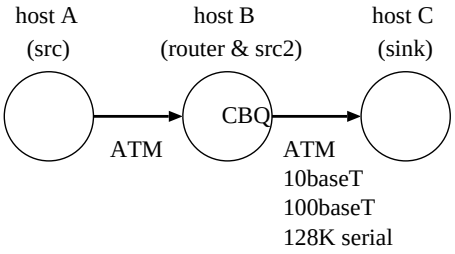

Figure 6: Test System Configuration

on the interface of host B connected to host C. The link between host A and host B is 155Mbps ATM. The link between host B and host C is either 155M ATM, 10baseT, 100baseT, or 128K serial line. When 10baseT is used, a dumb hub is inserted. When 100baseT is used, a direct connection is made by a cross cable, and the interfaces are set to the full-duplex mode. Efficient Network Inc. ENI-155p cards are used for ATM, Intel EtherExpress Pro/100B cards are used for 10baseT and 100baseT. RISCom/N2 cards are used for a synchronous serial line.

The Netperf benchmark program [10] is used with $\pm 2.5\%$ confidence interval at 99% confidence level. We use TCP to measure the packet forwarding performance under heavy load. In contrast to UDP, which consumes CPU cycles to keep dropping excess packets, TCP quickly adapts to the available bandwidth, and thus does not waste CPU cycle. However, a suitable window size should be selected carefully according to the end-to-end latency and the number of packets queued inside the network. Also, one should be careful about traffic in the reverse direction, since ACKs play a vital role in TCP. Especially with shared media (e.g., Ethernet), sending packets could choke TCP ACKs.

4.2 CBQ Overhead

The overhead introduced by CBQ consists of four steps: (1) extract flow information from an arriving packet. (2) classify the packet to the appropriate class. (3) select an eligible class for sending next. (4) estimate bandwidth use of the class and maintain the state of the class. There are many factors which affect the overhead: structure of class hierarchy, priority distribution, number of classes, number of active classes, rate of packet arrival, distribution of arrival, and so on. Hence, the following measurements are not intended to be complete.

Throughput Overhead

Table 1 compares TCP throughput of CBQ with that of the original FIFO queueing, measured over different link types. A small CBQ configuration with three classes is used to show the minimum overhead of CBQ. There is no other background traffic during the measurement.

Table 1: CBQ Throughput

Link Type	orig. FIFO (Mbps)	CBQ (Mbps)	overhead (%)
ATM	132.98	132.77	0.16
10baseT	6.52	6.45	1.07
100baseT	93.11	92.74	0.40
loopback			
MTU 16384	366.20	334.77	8.58
MTU 9180	337.96	314.04	7.08
MTU 1500	239.21	185.07	22.63

Table 2: CBQ Latency

Link Type	queue type	request/ response (bytes)	trans. per sec	calc'd RTT (usec)	diff (usec)
ATM	FIFO	1, 1	2875.20	347.8	
	CBQ		2792.87	358.1	10.3
	FIFO	64,64	2367.90	422.3	
	CBQ		2306.93	433.5	11.2
	FIFO	1024,64	1581.16	632.4	
	CBQ		1552.03	644.3	11.9
	FIFO	8192,64	434.61	2300.9	
	CBQ		432.64	2311.1	10.2
10baseT	FIFO	1,1	2322.40	430.6	
	CBQ		2268.17	440.9	10.3
	FIFO	64, 64	1813.52	551.4	
	CBQ		1784.32	560.4	9.0
	FIFO	1024,64	697.97	1432.7	
	CBQ		692.76	1443.5	10.8

No significant CBQ overhead is observed from the table because CBQ packet processing can overlap the sending time of the previous packet. As a result, use of CBQ does not affect the throughput.

The measurements over the software loopback interface with various MTU sizes are also listed in the table. These values show the limit of the processing power and the CBQ overhead in terms of CPU cycle. CBQ does have about 7% overhead with 9180-byte MTU, and about 23% overhead with 1500-byte MTU. It also shows that a current PC can handle more than 300Mbps with bi-directional loopback load. That is, a PC-based router has processing power enough to handle multiple 100Mbps-class interfaces; CPU load will be much lower with physical interfaces since DMA can be used. On a 300MHz PentiumII machine, we observed the loopback throughput of 420.62Mbps with 16384-byte MTU.

Latency Overhead

Table 2 shows the CBQ overhead in latency over ATM and 10baseT. In this test, request/reply style transactions are performed using UDP, and the test measures how many transactions can be performed per second. The rightmost two columns show the calculated average round-trip time (RTT) and the difference in microseconds. Again, CBQ has three classes, and there is no background traffic. We see from the table that the increase of RTT by CBQ is almost constant regardless of

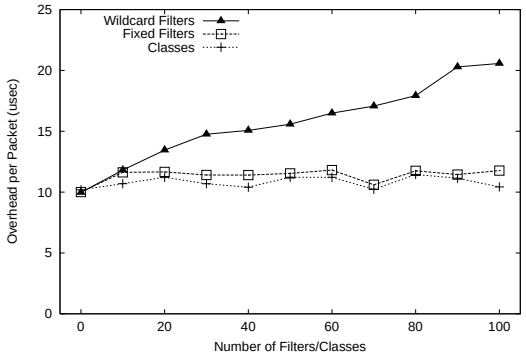

Figure 7: Effect of Number of Filters/Classes

packet size or link type, that is, the CBQ overhead per packet is about 10 microseconds.

Scalability Issues

CBQ is designed such that a class tree has relatively small number of classes; a typical class tree would have less than 20 classes. Still, it is important to identify the scalability issues of CBQ. Although a full test of scalability is difficult, the following measurements provide some insight into it. Figure 7 shows how the latency changes when we add additional filters or classes; up to 100 filters or classes are added. The values are differences in calculated RTT from the original FIFO queueing measured over ATM with 64-byte request and 64-byte response.

The "Wildcard Filters" plot and "Fixed Filters" plot in the graph show the effect of two different types of filters. To classify a packet, the classifier performs filter-matching by comparing packet header fields (e.g., IP addresses and port numbers) for each packet. In our implementation, class filters are hashed by the destination addresses in order to reduce the number of filter matching operations. However, if a filter doesn't specify a destination address, the filter is put onto the wildcard-filter list. When classifying a packet, the classifier tries the hashed list first, and if no matching is found, it tries the wildcard list. In this implementation, per-packet overhead grows linearly with the number of wildcard filters. A classifier could be implemented more efficiently, for example, using a directed acyclic graph (DAG) [1].

On the other hand, the number of classes doesn't directly affect the packet scheduler. As long as classes are underlimit, the scheduler can select the next class without checking the states of the other classes. However, to schedule a class which exceeds its share, the scheduler should see if there is a class to be scheduled first. Note that because the maximum number of overlimit classes is bound by the link speed and the minimum packet size, the overhead will not grow beyond a certain point.

When there are overlimit classes, it is obvious that CBQ performs much better than FIFO. We do not have numbers for such a scenario because it is difficult in our test configuration to separate the CBQ overhead from other factors (e.g., the overhead at the source and the destination hosts). But the dominant factor in latency will be the device level buffer. The measured latency will oscillate due to head-of-line blocking. The CBQ overhead itself will be by an order of magnitude smaller.

Table 3: Queueing Overhead Comparison

	FIFO	FIFOQ	RED	WFQ	CBQ	CBQ +RED
(usec)	0.0	0.14	1.62	1.95	10.72	11.97

Overhead of Other Disciplines

The latency overhead can be used to compare the minimum overhead of the implemented disciplines. Table 3 shows the per-packet latency overhead of the implemented disciplines measured over ATM with 64-byte request and 64-byte response. The values are differences in calculated RTT from the original FIFO queueing. The difference of the original FIFO and our FIFOQ is that the enqueue and dequeue operations are macros in the original FIFO but they are function calls in ALTQ.

Impact of Latency Overhead

Network engineers seem to be reluctant to put extra processing on the packet forwarding path. But when we talk about the added latency, we should also take queueing delay into consideration. For example, a 1KB packet takes 800 microseconds to be put onto a 10Mbps link. If two packets are already in the queue, an arriving packet could be delayed more than 1 millisecond. If the dominant factor of the end-to-end latency is queueing delay, sophisticated queueing is worth it.

4.3 Bandwidth Allocation

Figure 8, 9 and 10 shows the accuracy of bandwidth allocation over different link types. TCP throughputs were measured when a class is allocated 5% to 95% of the link bandwidth. The plot of 100% shows the throughput when the class can borrow bandwidth from the root class. As the graphs show, the allocated bandwidth changes almost linearly over ATM, 10baseT and a serial line. However, considerable deviation is observed over 100baseT, especially during the range from 15% to 55%.

The problem in the 100baseT case is the timer granularity problem described in Section 3.2. The calculated limit rate is 9.6Mbps, and the throughput in the graph stays at this limit up to 55%. Then, as the sending rate increases, packet events help CBQ scale beyond the limit. To back up this theory, we tested the performance of the kernel whose timer granularity is modified from 10ms to 1ms. With this kernel, the calculated limit rate is 96Mbps. The result, shown as *100baseT-1KHzTimer*,

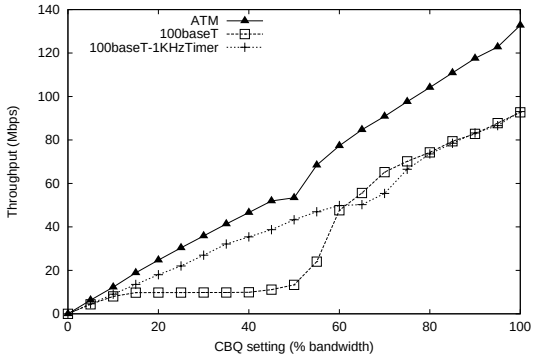

Figure 8: Bandwidth Allocation over ATM/100baseT

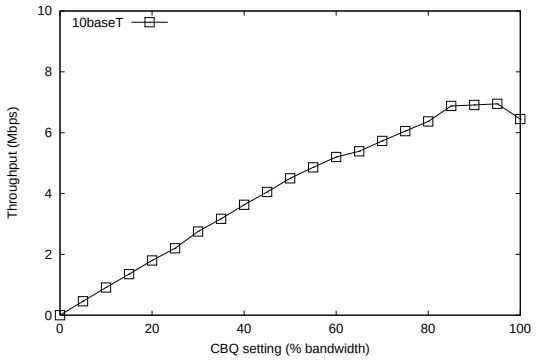

Figure 9: Bandwidth Allocation over 10baseT

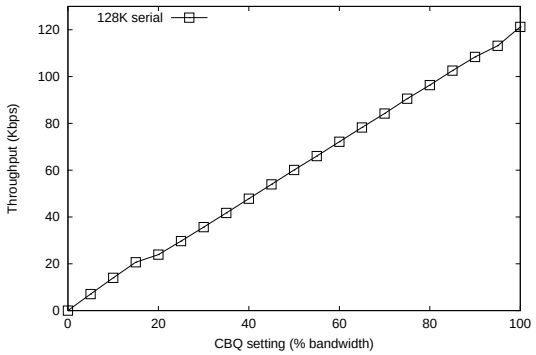

Figure 10: Bandwidth Allocation over 128K serial

is satisfactory, which also agrees with theory. Note that the calculated limit of the ATM case is 58.8Mbps, and we can observe a slight deviation at 50%, but packet events help CBQ scale beyond the limit. Also, note that 10baseT shows saturation of shared-media, and performance peaks at 85%. The performance of 10baseT drops when we try to fill up the link.

4.4 Bandwidth Guarantee

Figure 11 illustrates the success of bandwidth guarantee over ATM. Four classes, one each allocated 10Mbps, 20Mbps, 30Mbps and 40Mbps, are defined. A background TCP flow matching the default class is sent dur-

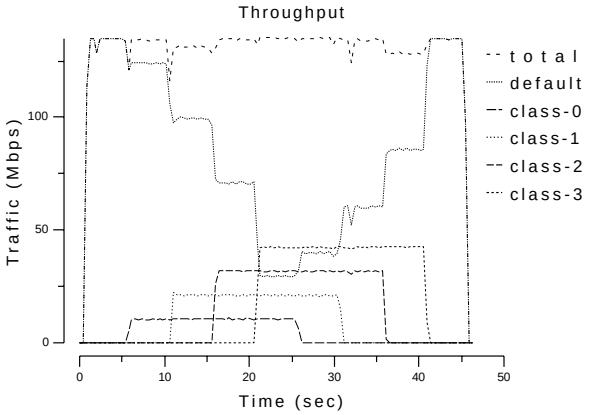

Figure 11: CBQ Bandwidth Guarantee

ing the test period. Four 20-second-long TCP flows, each corresponding to the defined classes, start 5 seconds apart from each other. To avoid oscillation caused by process scheduling, class-0 and class-2 are sent from host B and the other three classes are sent from host A. All TCP connections are trying to fill up the pipe, but the sending rate is controlled by CBQ at host B.

The *cbqprobe* tool is used to obtain the CBQ statistics (total number of octets sent by a class) every 400 msec via *ioctl*, and the *cbqmonitor* tool is used to make the graph. Both tools are included in the release.

As we can see from the graph, each class receives its share and there is no interference from other traffic. Also note that the background flow receives the remaining bandwidth, and the link is almost fully utilized during the measurement.

4.5 Link Sharing by Borrowing

Link sharing is the ability to correctly distribute available bandwidth in a hierarchical class tree. Link-sharing allows multiple organizations or multiple protocols to share the link bandwidth and to distribute "excess" bandwidth according to the class tree structure. Link-sharing has a wide range of practical applications. For example, organizations sharing a link can receive the available bandwidth proportional to their share of the cost. Another example is to control the bandwidth use of different traffic types, such as telnet, ftp, or real-time video.

The test configuration is similar to the two agency setting used by Floyd [6]. The class hierarchy is defined as shown in Figure 12 where two agencies share the link, and interactive and non-interactive leaf classes share the bandwidth of each agency. In the measurements, Agency X is emulated by host B and agency Y is emulated by host A. Four TCP flows are generated as in Figure 13. Each TCP tries to send at its maximum rate, except for the idle period. Each agency should receive its share of bandwidth all the time even when one of the leaf classes is idle, that is, the sum of class-0 and class-1 and the sum

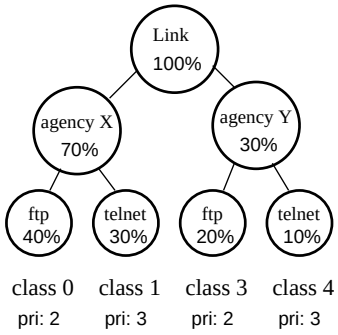

Figure 12: Class Configuration

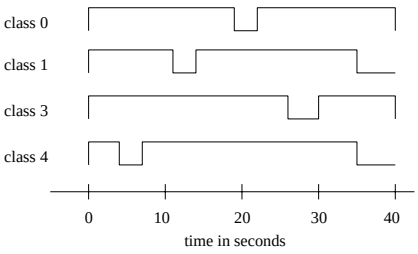

Figure 13: Test Scenario

of class-3 and class-4 should be constant.

Figure 14 shows the traffic trace generated by the same method described for Figure 11. The classes receive their share of the link bandwidth and, most of the time, receive the "excess" bandwidth when the other class in the same agency is idle. High priority class-4, however, receives more than its share in some situations (e.g., time frame:22–25). The combination of priority and borrowing in the current CBQ algorithm, especially when a class has a high priority but a small share of bandwidth, does not work so well as in the NS simulator [6]. To confirm the cause of the problem, we tested with all the classes set to the same priority. As Figure 15 shows, the problem of class-4 is improved. Note that, even if interactive and non-interactive classes have the same priority, interactive classes are likely to have much shorter latency because interactive classes are likely to have much fewer packets in their queues.

5 Discussion

One of our goals is to promote the widespread use of UNIX-based routers. Traffic management is becoming increasingly important, especially at network boundaries that are points of congestion. Technical innovations are required to provide smoother and more predictable network behavior. In order to develop intelligent routers for the next generation, a flexible and open software development environment is most important. We believe UNIX-based systems, once again, will play a vital role

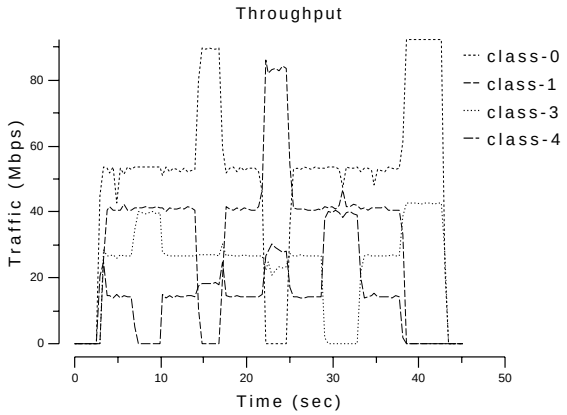

Figure 14: Link-Sharing Trace

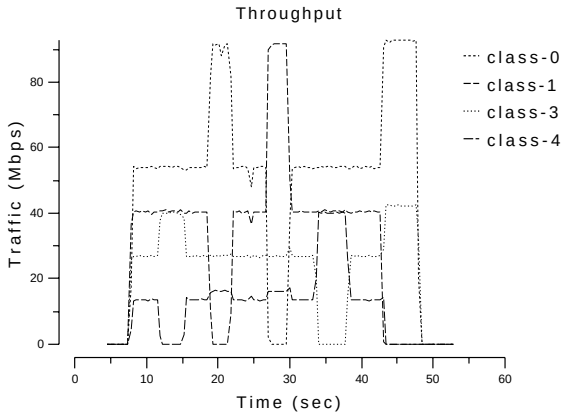

Figure 15: Link-Sharing Trace with Same Priority

in the advancement of technologies.

However, PC-UNIX based routers are unlikely to replace all router products in the market. Non-technical users will not use PC-UNIX based routers. High-speed routers or highly-reliable routers require special hardware and will not be replaced by PCs. Still, the advantage of PC-UNIX based routers is their flexibility and availability in source form, just like the advantage of UNIX over other operating systems. We argue that we should not let black boxes replace our routers and risk loosing our research and development environment. We should instead do our best to provide high-quality routers based on open technologies.

There are still many things that need to be worked out for the widespread use of PC-UNIX based routers. Network operators may worry about the reliability of PC-based routers. However, a reliable PC-based router can be built if the components are carefully selected. In most cases, problems are disk related troubles and it is possible to run UNIX without a hard disk by using a non-mechanical storage device such as ATA flash cards. Another reliability issue lies in PC components (e.g., cool-

ing fans) that may not be selected to be used 24 hours a day. There are a wide range of PC components and the requirements for a router are quite different from those for a desktop business machine. Some of the rack-mount PCs on the market are suitable for routers but SOHO routers need smaller chassis. We need PC hardware packages targeted for router use.

By the same token, we need software packages. It is not an easy task to configure a kernel to have the necessary modules for a router and make it run without a hard disk or a video card. Although there are many tools for routers, compilation, configuration, and maintenance of the tools are time consuming. Freely-available network administration tools seem to be weak but could be improved as PC-based routers become popular.

In summary, the technologies required to build quality PC-UNIX based routers are already available, but we need better packaging for both hardware and software. The networking research community would benefit a great deal if a line of PC-based router packages were available for specific scenarios, such as dial-up router, boundary router, workgroup router, etc.

6 Related Work

Our idea of providing a framework for queueing disciplines is not new. Nonetheless there have been few efforts to support a generic queueing framework. Research queueing implementations in the past have customized kernels for their disciplines [8, 19]. However, they are not generalized for use of other queueing disciplines.

ALTQ implements a switch to a set of queueing disciplines, which is similar to the protocol switch structure of BSD UNIX. A different approach is to use a modular protocol interface to implement a queueing discipline. STREAMS [17] and x-kernel [16] are such frameworks and the CBQ release for Solaris is actually implemented as a STREAMS module. Although it is technically possible to implement a queueing discipline as a protocol module, a queueing discipline is not a protocol and the requirements are quite different. One of the contributions of this paper is to have identified the requirements of a generic queueing framework.

A large amount of literature exists in the area of process scheduling but we are not concerned with process scheduling issues. Routers, as opposed to end hosts which run real-time applications, do not need real-time process scheduling because packet forwarding is part of interrupt processing. For end hosts, process scheduling is complementary to packet scheduling.

7 Current Status

The ALTQ implementation has been publicly available since March 1997. The current version runs on FreeBSD-2.2.x and implements CBQ, RED (including ECN), WFQ, and FIFOQ. The CBQ glues for ISI's RSVP are also included in the release. Most of the popular drivers including seven Ethernet drivers, one ATM driver, and three synchronous serial drivers can be used with ALTQ. ALTQ, as well as the original CBQ and RSVP, is still under active development.

ALTQ is used by many people as a research platform or a testbed. Although we do not know how many ALTQ users there are, our ftp server has recorded more than 1,000 downloads over the last 6 months. The majority of the users seem to use ALTQ for RSVP, but others do use ALTQ to control live traffic for their congested links and this group seems to be growing.

The performance of our implementation is quite satisfactory, but there are still many things to be worked out such as scalability issues in implementation and easier configuration for users.

We are planning to add new features, including support for IPv6, better support for slow links, better use of ATM VCs for traffic classes, and diskless configurations for more reliable router operations. Also, building good tools, especially traffic generators, is very important for development.

7.1 Availability

A public release of ALTQ for FreeBSD, the source code along with additional information, can be found at http://www.csl.sony.co.jp/person/kjc/software.html.

8 Conclusion

We have identified the requirements of a generic queueing framework and the issues of implementation. Then, we have demonstrated, with several queueing disciplines, that simple extension to BSD UNIX and minor fixes to drivers are enough to incorporate a variety of queueing disciplines. The main contribution of ALTQ is engineering efforts to make better queueing available for researchers and network operators on commodity PC platforms and UNIX. Our performance measurements clearly show the feasibility of traffic management by PC-UNIX based routers.

As router products have been proliferating over the last decade, network researchers have been losing research testbeds available in source form. We argue that general purpose computers, especially PC-based UNIX systems, have become once again competitive router platforms because of flexibility and cost/performance. Traffic management issues require technical innovations, and the key to progress is platforms which new ideas could be easily adopted into. ALTQ is an important step in that direction. We hope our implementation will stimulate other research activities in the field.

Acknowledgments

We would like to thank Elizabeth Zwicky and the anonymous reviewers for their helpful comments and suggestions on earlier drafts of this paper. We are also grateful to Sally Floyd for providing various information about CBQ and RED. We thank the members of Sony Computer Science Laboratory and the WIDE Project for their help in testing and debugging ALTQ. Hiroshi Kyusojin of Keio University implemented WFQ. The ALTQ release is a collection of outputs from other projects. These include CBQ and RED at LBNL, RSVP/CBQ at Sun, RSVP at ISI, and FreeBSD.

References

[1] Mary L. Baily, Burra Gopal, Michael A. Pagels, Larry L. Peterson, and Prasenjit Sarkan. Pathfinder: A pattern-based packet classifier. In *Proceedings of Operating Systems Design and Implementation*, pages 115–123, Monterey, CA, November 1994.

[2] B. Braden, D. Clark, J. Crowcroft, B. Davie, S. Deering, D. Estrin, S. Floyd, V. Jacobson, G. Minshall, C. Partridge, K. L. Peterson, S. Shenker Ramakrishnan, J. Wroclawski, and L. Zhang. Recommendations on queue management and congestion avoidance in the internet. RFC 2309, IETF, April 1998.

[3] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of SIGCOMM '89 Symposium*, pages 1–12, Austin, TX, September 1989.

[4] Sally Floyd. TCP and explicit congestion notification. *ACM Computer Communication Review*, 24(5), October 1994.

[5] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transaction on Networking*, 1(4):397–413, August 1993.

[6] Sally Floyd and Van Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), August 1995. Also available from http://www-nrg.ee.lbl.gov/floyd/papers.html.

[7] The FreeBSD Project. http://www.freebsd.org/.

[8] Amit Gupta and Domenico Ferrari. Resource partitioning for real-time communication. *IEEE/ACM Transactions on Networking*, 3(5), October 1995. Also available from http://tenet.berkeley.edu/tenet-papers.html.

[9] The RSVP Project at ISI. http://www.isi.edu/rsvp/.

[10] Rick Jones. *Netperf: A Benchmark for Measuring Network Performance*. Hewlett-Packard Company, 1993. Available at http://www.cup.hp.com/netperf/NetperfPage.html.

[11] Srinivasan Keshav. On the efficient implementation of fair queueing. *Internetworking: Research and Experience*, 2:157–173, September 1991.

[12] P. E. McKenney. Stochastic fairness queueing. In *Proceedings of INFOCOM*, San Francisco, CA, June 1990.

[13] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Co., 1996.

[14] John Nagle. On packet switches with infinite storage. *IEEE Trans. on Comm.*, 35(4), April 1987.

[15] Abhay Parekh. A generalized processor sharing approach to flow control in integrated services networks. LIDS-TH 2089, MIT, February 1992.

[16] Larry L. Peterson, Norman C. Hutchinson, Sean W. O'Malley, and Herman C. Rao. The x-kernel: A platform for accessing internet resources. *Computer*, 23(5):23–34, May 1990.

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

[18] McCanne S. and Floyd S. NS (Network Simulator). http://www-nrg.ee.lbl.gov/ns/, 1995.

[19] Ion Stoica and Hui Zhang. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *Proceedings of SIGCOMM '97 Symposium*, pages 249–262, Cannes, France, September 1997.

[20] Solaris RSVP/CBQ. ftp://playground.sun.com/pub/rsvp/.

[21] Ian Wakeman, Atanu Ghosh, Jon Crowcroft, Van Jacobson, and Sally Floyd. Implementing real-time packet forwarding policies using streams. In *Proceedings of USENIX '95*, pages 71–82, New Orleans, LA, January 1995.

[22] Lixia Zhang, Steve Deering, Deborah Estrin, Scott Shenker, and Daniel Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7:8–18, September 1993. Also available from http://www.isi.edu/rsvp/pub.html.