A Self-Accelerating Packet Service Discipline for Low-Delay Service to Bursty Flows

Ricardo Pincheira

Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195
pinch@cs.washington.edu

Technical Report 94-05-07

ABSTRACT

In this paper, we consider the transmission of compressed video over networks. We develop a packet service discipline called Axel that provides very low service delays to bursty flows while providing guaranteed throughput to all flows in the system. In contrast to existing policies capable of providing bandwidth guarantees, Axel can provide low delay service to bursty flows without the need for resource over-reservation. We study the behavior of Axel through analysis and simulation, and conclude that Axel meets its stated goals. The policy is targeted at a network environment composed of a mixture of bursty, delay sensitive traffic and smoother, delay insensitive traffic. We expect such an environment to arise from a mixture of interactive, compressed video streams (eg., tele-conferencing) and non-interactive compressed video streams (eg., video-on-demand).

31 January 1994

1. Introduction

There is little doubt that in coming years, continuous-media flows such as digital video and audio will form an ever-increasing fraction of traffic in data networks. An important prerequisite for continuous media flows is guaranteed bandwidth; in 30 frames-per-second video, for example, a new frame must be available for consumption at the receiver every 33 milliseconds (see also[5]). Several service disciplines able to provide guaranteed bandwidth have been proposed in recent years [14,10,4,13]. Unfortunately, these policies are able to guarantee bandwidth by decoupling, to a large extent, the service that they give to a flow from the flow's arrival behavior. Such a decoupling is necessary to provide "firewalls" between flows and prevent any one flow from monopolizing system resources at the expense of other flows. However, it prevents a flow from receiving a higher rate of service when its bits arrive faster than expected. As a result, these policies penalize bursty flows and provide them with considerably higher service delays than smoother types of traffic.

Continuous media flows can be broadly divided into two classes: flows with stringent end-to-end delay requirements, and flows that are not delay sensitive. The first class arises from teleconferencing or other simultaneous tele-interactions. In an interactive conference, for example, maximum tolerable end-to-end delays are measured in the 100-200 millisecond range, and one can easily envision future methods of tele-interaction, such as distributed music rehearsal [1], in which end-to-end delay requirements are even more stringent. The second class arises from services such as video-on-demand. An end-to-end delay of several seconds is acceptable when watching a previously-recorded video clip or in a "live" but non-interactive application.

Due to the interaction of delay requirements with variable bit-rate (VBR) compression, the burstiness characteristics for these two classes of flows also differs. It is well known that applying variable bit-rate (VBR) compression to video yields a very bursty stream of data (eg, [9]). Though a bursty flow can be smoothed before it enters the network, smoothing results in additional delay. These delays are not significant for flows with loose end-to-end delay requirements. Indeed, when all the data to be transmitted is known in advance, as in video-on-demand services, the resulting flow can be smoothed to a practically constant data rate. By contrast, flows with stringent delay requirements preclude any significant degree of smoothing.

In sum, we can identify two distinct classes of continuous media flows: bursty flows with stringent end-to-end delay requirements, and smoother flows with loose end-to-end delay constraints. Since existing policies providing bandwidth guarantees penalize bursty flows, they are ill-suited to this environment.

In this paper we develop a packet service policy, called Axel, that offers very low end-to-end delays to bursty flows while providing smoother flows with bandwidth guarantees. It belongs to a class of service disciplines that we term *self-accelerating*, because they accelerate the rate of service of flows being

serviced at a rate that is too low relative to their actual arrival rate. They can thus provide bursty flows with very low service delays. We also require that self-accelerating policies provide bandwidth guarantees to all flows in the system and prevent any one flow from arbitrarily increasing its rate of service.

The idea of increasing a flow's rate of service so as to clear a burst of data more quickly is not new. [8] (reviewed in section 2.3.2) describes a policy called Pulse that tries to alleviate congestion quickly. It services packets according to a discipline that models the behavior of bursts. [2] discusses the relative merits of enforcing total isolation between flows vs. allowing statistical sharing when trying to provide bursty flows with low delay jitter (variability). The Axel policy developed here provides very low service delays to bursty flows by allowing a controlled amount of sharing; yet, by enforcing isolation, it provides all flows in the system with guaranteed throughput.

The rest of this paper is organized as follows. Section 2 introduces our system model and reviews some background material. Section 3 develops a self-accelerating policy called Axel. Section 4 uses simulation to evaluate the performance of Axel and compare it against several benchmark policies. Section 5 discusses potential improvements to Axel and suggests areas for future research. Lastly, section 6 concludes.

2. Definitions and Background

2.1. System Model

We assume a network in which packets originate at packet *sources*, traverse one or more packet *servers*, and terminate at packet *sinks*. A packet server k has an aggregate service capacity R_k. Sources, servers, and sinks are joined by links having infinite capacity.[1] As a condition for receiving a guaranteed level of service, each client flow enters into a contract with the network, in which the client promises to adhere to a certain traffic generation pattern and the network promises a certain level of service in return. We will use a very general characterization for each flow i, similar to that used in [3,10,2,1]. Each flow i advertises 3 parameters, r^i, $\hat{r}^i$, and W^i. r^i, the average rate, describes the flow's long-term average behavior and bandwidth requirement. $\hat{r}^i$, the peak rate, bounds the peak rate at which the flow will present packets to the network and is based on minimum inter-packet gaps. Hereafter, we will often use r^i and $\hat{r}^i$ to refer both to bit rates and packet rates. Whenever the distinction matters, the particular interpretation should be clear from context. We do not require that the network satisfy the peak rate constraints internally. $\hat{r}_i$ may increase if packets clump together at some server and are then output in a large burst.

The last parameter, W^i, describes the burstiness of the flow, as shown in figure 1. The solid curve shows the cumulative number of bits transmitted by some flow over its lifetime. The dotted line segments, which have slope r^i, show the number of bits the flow would have transmitted if it never produced bits

[1] The finite capacity of a physical link is captured by the server capacity.

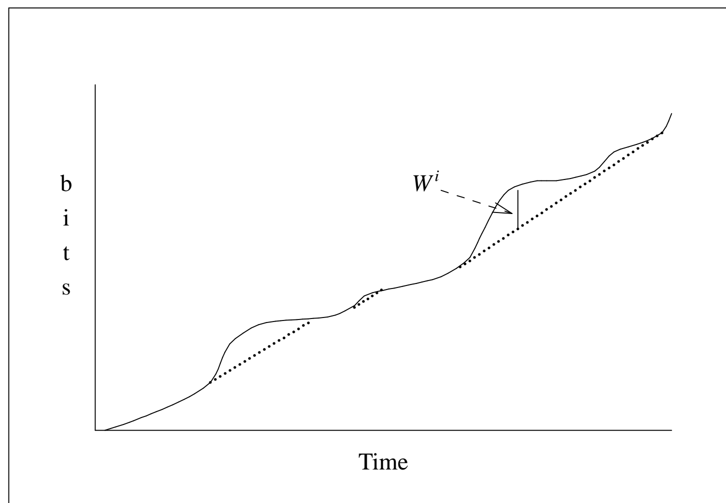

Figure 1

faster than its claimed average rate during active periods. W^i bounds the difference between these two quantities; in the figure, W^i is simply the maximum vertical distance between the solid and dotted curves. W^i is a measure of burstiness because it bounds the size of a burst.

An equivalent definition of W^i is to consider an idealized server that always services flow i at an instantaneous rate r^i. Then W^i bounds flow i's queue length at the ideal server. A flow with an average rate r^i can be forced to abide to a given $(W^i, r^i, \hat{r}^i)$ parameterization with a leaky-bucket filter [11].

Because of the dual interpretation for W^i, we will sometimes call W^i the *maximum workahead* of the flow (when viewed from the standpoint of data generation) or the the *maximum backlog* of the flow (when viewed from the standpoint of an ideal server).

Let the servers that a flow traverses be denoted by $0, 1, 2 ... m - 1$, and its sink be denoted m. At each hop k a burstiness W^i_k can be defined in a manner analogous to the definition given for the flow's burstiness at the edge of the network by considering the output from the previous hop. W^i is then W^i_0. The *input burstiness* of the flow at a particular server k is W^i_k, while its *output burstiness* is W^i_{k+1}.

Two of the policies we will discuss here, Virtual Clock (section 2.3.1 [14]) and Pulse (section 2.3.2 [8]), use a parameter AI^i instead of W^i to describe the burstiness of a flow. AI^i is called the *averaging interval*, and has the following interpretation: if a flow's behavior is monitored over successive disjoint intervals of length AI^i, then in every such interval the flow transmits at most $AI^i r^i$ bits. For these policies, we define a suitable value for AI^i via $AI^i = \dfrac{W^i}{r^i}$. This conversion corresponds to a flow transmitting W^i bits at the beginning of an averaging interval.

2.2. Classes of Flows

In the introduction, we alluded to the existence of two classes of flows, one composed of bursty flows requiring very low end-to-end latencies, and another composed of very smooth flows able to tolerate relatively high end-to-end latencies. We call the first of these classes the *bursty* or A flows; they are characterized by high values for $\frac{W^i}{r^i}$ and $\frac{\hat{r}^i}{r^i}$. We call the second class of flows the *smooth* or D flows; they are characterized by low values for $\frac{W^i}{r^i}$ and $\frac{\hat{r}^i}{r^i}$. The A and D designations arise from the fact that whenever a bursty flow generates a burst, its service rate should be temporarily accelerated in order to provide it with low end-to-end latency, while the service rate of delay-tolerant smooth flows can be temporarily decelerated while part of their resources are devoted to improving the service given to A flows.

2.3. Related Work

2.3.1. Virtual Clock

The Axel packet service policy developed in section 3 uses many of the ideas of Virtual Clock [14]. Virtual Clock belongs to the class of *timestamp-based service disciplines*. These operate by assigning a timestamp to the packets of each flow traversing the server and servicing the packets in timestamp order. In Virtual Clock, the timestamp for flow i is generated using the following rule. $vclock^i$ (called $auxVC^i$ in [14]) is a state variable that tracks the evolution of the timestamp; $vtick^i$ is a constant set to $1/r^i$ and is the "tick" size at which $vclock^i$ ticks; and *RealTime* stands for wall-clock time at the packet server.

Initially, on seeing the first bit from flow i, set $vclock^i = RealTime$

For each packet received from flow i, let l be the packet length. Then do as follows:

- $vclock^i = max(RealTime, vclock^i)$
- $vclock^i = vclock^i + l\ vtick^i$
- Stamp the packet with $vclock^i$

Note that the timestamps of a flow transmitting close to its claimed rate r^i closely track *RealTime*, whereas the timestamps of a flow transmitting faster than expected evolve faster than *RealTime*. Such a misbehaving flow quickly loses priority to flows abiding to their claimed average rate, because its timestamps will be larger. Note also that the resynchronization of *vclock* with *RealTime* via $vclock^i = max(RealTime, vclock^i)$ prevents a flow from arbitrarily reducing its timestamp values relative to those of flows that transmit close to their claimed rate. Consequently, no flow can arbitrarily increase its priority over other flows.

Through these two mechanisms, Virtual Clock is able to provide each flow i with a guaranteed throughput. Indeed, if n flows, $1, \ldots n$, traverse a server k having aggregate service capacity R_k, then provided that $R_k \geq \sum_{j=1}^{n} r^j$, Virtual Clock provides each flow i with average throughput at least r^i. More

generally, the long term rate of service received by flow i is $\frac{r^i}{\sum_{j=1}^{n} r^j} R_k$. [2]

Though Virtual Clock provides guaranteed throughput to all flows in the system, it clearly penalizes bursty flows, even when they adhere to their claimed long-term average throughput r^i. Whenever a bursty flow generates a burst, its *vclock* advances rapidly and the flow loses priority to smoother flows. For this reason, Virtual Clock does not work well in an environment composed of delay-intolerant, bursty *A* flows and delay-tolerant, smooth *D* flows.

2.3.2. Pulse

The Pulse policy [8] aims to minimize congestion by attempting to clear bursts quickly. Pulse generalizes Virtual Clock by using two values for the timestamp increment, $vtick^i_{large} \geq 1/r^i$ and $vtick_{small} \leq 1/r^i$. The vclock is incremented using $vtick^i_{small}$ until a certain number of bits are received. The vclock is then incremented once with $vtick^i_{large}$. The two values of vtick are thus used alternatingly. Pulse models the behavior of on/off-type sources quite closely and can be expected to provide them with low service delays.

For the performance study of section 4, we set $vtick^i_{small}$ to $\frac{1}{\hat{r}^i}$ and $vtick^i_{large}$ to $\frac{W^i}{r^i} - W^i \; vtick^i_{small}$. Initially, the flow has W^i "credits" for the smaller timestamp increment, which it consumes as bits arrive. Once the flow exhausts its credits, the timestamp is incremented once with the large vtick and the flow receives an additional W^i credits.

Pulse also differs from Virtual Clock in that the *vclock* is not resynchronized with *RealTime* if it lags behind. That is, unlike Virtual Clock, the Pulse timestamping rule does not include the step $vclock = max(vclock, RealTime)$.

2.3.3. Fifo+

Fifo+ [2] is a modification of ordinary Fifo. Fifo can be implemented as a timestamp-based service discipline by using a packet's actual arrival time at the server as its timestamp. Fifo+ modifies the Fifo timestamp by the deviation between the packet's actual delay in reaching the server from its source and the delay that the packet would have encountered if, at each of the upstream servers it traversed, it had been delayed by that server's average observed delay. A packet that has received higher than average delay in traversing upstream servers receives a smaller timestamp than it would under Fifo, while one that has received lower than average delay is assigned a larger timestamp.

---

[2] As [10] points out, however, Virtual Clock is vulnerable to a "punishment" phenomenom and does not guarantee each *packet* of flow i a service rate r^i. The closely related Weighted Fair Queueing or PGPS policy [10,4], can guarantee this rate even to individual packets of a flow (within a very small factor to account for priority inversion — a server services each packet non-preemptively to completion). The PGPS timestamping rule is syntactically identical to that of Virtual Clock, but its slightly different semantics for the time associated with a server prevent punishment.

Under Fifo+, end-to-end delays change relatively slowly, and one can predict a flow's expected end-to-end delays from the observed mean service delays at the servers it traverses. Our interest in Fifo+ stems from the fact that it attempts to give similar end-to-end delays to all packets of a flow. To the extent that its packets encounter similar delays, a flow's original arrival characteristics are approximately preserved. Since any increase or decrease in a flow's original burstiness implies an increase in its end-to-end delay, one would expect Fifo+ to service bursty flows with relatively low delay.

3. A Self-Accelerating Service Discipline

This section introduces the concept of self-acceleration in slightly more precise terms than has been done to this point and develops a self-accelerating service policy called Axel. In introducing Axel, we will first present a policy that does not work well, and then modify it to yield Axel. Axel provides very low end-to-end delays to bursty flows, yet also provides bandwidth guarantees to every flow in the system and prevents a malicious flow from monopolizing system resources. This will be demonstrated in section 4 which presents the results of a simulation study.

Consider a flow i traversing a server, and suppose that over a small time interval the flow's input rate into the server is $\rho^i_{in} \geq 0$ while its output (service) rate is $\rho^i_{out} \geq 0$. A *self-accelerating service policy* is one that increases ρ^i_{out} whenever $\rho^i_{in} > \rho^i_{out}$. Clearly, a usable policy cannot increase ρ^i_{out} to an arbitrary degree or for an indefinite period of time, for otherwise, the flow could monopolize resources. Thus, in defining a self-accelerating service policy we also require that any period of acceleration be bounded, so that no flow can arbitrarily increase its quality of service at the expense of other flows. This last condition excludes policies such as ordinary first-come, first-served (Fifo) from the class of self-accelerating policies.

The idea underlying our development of a self-accelerating service policy is the observation that any discrepancy in ρ^i_{in} and ρ^i_{out} is reflected in the number of bits back-logged (queued) inside the server. In particular, if ρ^i_b denotes the rate at which the backlog changes, with positive values denoting an increase in backlog and negative values a decrease, we have

$$\rho^i_{in} = \rho^i_{out} + \rho^i_b$$

Suppose that a flow is serviced using Virtual Clock, and consider an idealized case in which the flow is in a steady-state, with $\rho^i_{in} = \rho^i_{out} = r^i$ and $\rho^i_b = 0$. If the flow bursts, the timestamp will evolve at the relative rate $\frac{\rho^i_{in}}{r^i}$, or, substituting $\rho^i_{out} + \rho^i_b$ for ρ^i_{in}, $1 + \frac{\rho^i_b}{r^i}$, where 1 is the rate at which *RealTime* evolves. $\frac{\rho^i_b}{r^i}$ is thus the rate at which the flow's timestamp surges ahead of *RealTime*, and, in some sense, reflects the difference between the rate at which the vclock is evolving and the rate at which it "should" evolve if the flow's service rate were to match its input rate. This holds to the extent that the timestamp value associated with a bit bounds the time by which the bit is serviced.

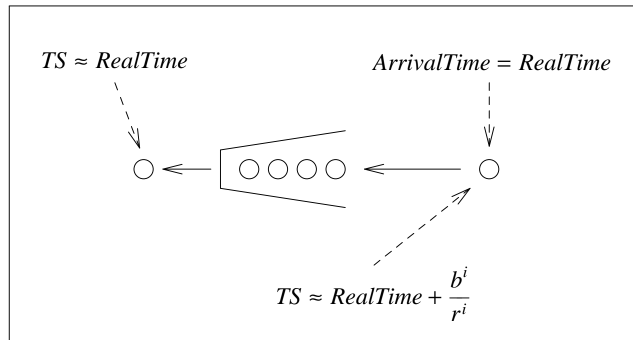

Figure 2

Figure 2 illustrates. The figure shows the state of some flow *i*'s bits following a burst by the flow. b^i denotes the current backlog in bits for the flow at the server and *TS* denotes the timestamp of the indicated bit. The figure shows an idealized case in which flow *i* had no back-logged bits at the server before bursting and the timestamp value of the bit being serviced closely matches *RealTime*. This latter assumption is somewhat unrealistic, since whenever the server is (momentarily) overloaded, the timestamp of the bit being served will tend to be greater than *RealTime*, while in cases in which the server is not operating at full capacity, its timestamp may actually surge ahead of *RealTime*.[3] But clearly, in some sense $\frac{b^i}{r^i}$ reflects the difference between the flow's vclock and where it "should" be if the flow were to be serviced at a rate matching its arrival rate. This observation suggests modifying the timestamps generated by Virtual Clock by $\frac{b^i}{r^i}$.

Suppose that the server services flows according to Virtual Clock, and call the generated timestamps the *baseline timestamps*. Consider the following modification to the timestamp actually assigned to the flow's packets:[4]

$$Timestamp = baselineTimestamp - \frac{min(b^i, W^i)}{r^i}$$

This modified timestamping rule has a number of interesting properties:

- *A flow (momentarily) transmitting at a rate higher than its specified rate receives service at a higher rate*. Note that the vtick between the timestamps assigned to two successive bits is $\frac{1}{r^i} - \frac{\Delta_b^i}{r^i}$, where Δ_b^i is the change in b^i between the arrival of the two bits. If a flow transmits faster than it is serviced, its backlog b^i will increase. Δ_b^i will be positive, and the vtick will be reduced, speeding up its rate of service.

---

[3] The fact that the timestamp of the bit being served can surge ahead of *RealTime* is precisely why Virtual Clock is vulnerable to the punishment phenomenon mentioned earlier. See [10].

[4] This rule will be modified slightly below.

- *No flow can increase its rate of service arbitrarily*. Once a flow's backlog reaches W^i, its timestamp increment reverts to that of the baseline vtick, or $\frac{1}{r^i}$.

- *No flow can increase its priority arbitrarily*, in the sense of arbitrarily reducing the value of its timestamps relative to those of the baseline policy and gaining advantage over other flows. The maximum priority that a flow can obtain is $\frac{W^i}{r^i}$.

- *The relative priority of two bits reflects the degree of smoothing that would occur under the baseline policy.* In the idealized case in which the server services the flow at an instantaneous rate r^i, the delay that a newly arriving bit will encounter is $\frac{b^i}{r^i}$. The difference in the expected service delay between two bits, $\frac{\Delta_b^i}{r^i}$, reflects the degree to which the two bits would be smoothed under the baseline policy. For this reason, we will sometimes refer to the difference in the delay encountered by successive bits or packets as the *smoothing factor*.

- *Over a time interval, if the flow transmits at a rate higher than claimed, the timestamp values assigned to the flow will reflect the ratio of the flow's actual arrival rate to its actual service rate.* This holds provided that throughout the time interval, $0 \le b^i \le W^i$. Consider a time interval of length t during which a flow i transmits a total of n bits at a rate $m\, r^i$, where $m > 1$. Without loss of generality, we may assume this interval starts at time 0. Suppose that during this time interval, the flow has been serviced at a rate pr^i, where $p > 0$. We may assume without loss of generality that the backlog b^i was 0 at the beginning of this time interval. At time t, therefore, the backlog will be $t\,(m - p)r^i$. Note also that at time t (once n bits are received), the baseline value for the timestamp will be $\frac{n}{r^i}$. Assuming the backlog $t\,(m - p)\,r^i \le W^i$, the actual timestamp at the end of the interval is therefore

$$\frac{n}{r^i} - \frac{t\,(m-p)\,r^i}{r^i}$$

  Since $t = \frac{n}{mr^i}$, this reduces to $\frac{pn}{mr^i}$. Since the timestamp value "should" be n/r^i (the baseline value), the timestamp increased only p/m of what the actual increase should have been, as claimed.

  This property implies that, provided the server is not momentarily overloaded, the flow's actual service rate over the time interval t will reflect the flow's actual arrival rate.

- *Timestamps can be made to increase monotonically*. The timestamp assignment rule described so far does *not* satisfy timestamp monotonicity, though the timestamp values will never decrease. The latter property holds because $\frac{b^i}{r^i}$ never increases faster than the baseline vclock. Two bits can receive the same timestamp, however.

  In a timestamp-based discipline, timestamp values must increase monotonically. Furthermore, the timestamps assigned to bits should not increase too slowly, for otherwise a flow could temporarily

monopolize the server. Forcing a minimum timestamp increment solves both problems. As a minimum timestamp increment, we will use $\frac{1}{\hat{r}^i}$, the inverse of the flow's claimed peak rate.

With the modifications required to ensure a minimum timestamp increment, the timestamp assignment rule becomes:

- compute the *baselineTimestamp* value according to Virtual Clock
- compute $t1 = baselineTimestamp - \frac{min(b^i, W^i)}{r^i}$
- compute $t2$ = previous timestamp value + $\frac{1}{\hat{r}^i}$
- set $timestamp = max(t1, t2)$

The above timestamping rule "defers" the acceleration it gives to a flow; the flow's rate of service isn't actually increased until after all packets ahead of those that the policy timestamps with the reduced timestamp increment are serviced. We thus call the service policy based on this rule the *Deferred* policy. It is worth pointing out that the Deferred policy reduces to the baseline timestamping rule in the event that there is never a backlog of bits to be serviced. Also, because the state used to generate the baseline timestamps is never modified, the algorithm reverts to the baseline timestamping algorithm following a period of non-zero backlog.

The Deferred policy does not work well. Though we will not present detailed results here, the Deferred policy can give *every* flow in the system, both *A* and *D*, considerably worse end-to-end latencies than Virtual Clock, and can increase their burstiness (the ratio of output to input burstiness) as well. The reason for this poor performance is that the Deferred policy *increases* the effective timestamp increment beyond the baseline *vtick* whenever the flow's backlog is decreasing, reducing the flow's rate of service. Thus, though the Deferred policy initially increases a flow's service rate following a burst, it reduces the service rate shortly thereafter, once the backlog starts to clear. This behavior leads to poor service, particularly when several flows bursting simultaneously heavily load the server. During the brief time in which a flow is accelerated following a burst, the server is heavily loaded, so that the flow receives poor service regardless of its timestamp increment. As the flow's backlog clears, the flow is decelerated, losing resources to flows that had not build up a heavy backlog.

Figure 3 illustrates this property of the timestamp evolution under the Deferred policy. It plots the per-packet timestamp increment vs. time for a bursty flow transmitting a number of constant-sized packets during each burst. We have omitted a significant number of values greater than 0.01. By contrast, although we do not show it here, under Virtual Clock, the timestamp increment is almost always[5] approximately 0.0005, the value of $\frac{1}{r^i}$ multiplied by the packet size.

[5] Except when resynchronizing with *RealTime*.

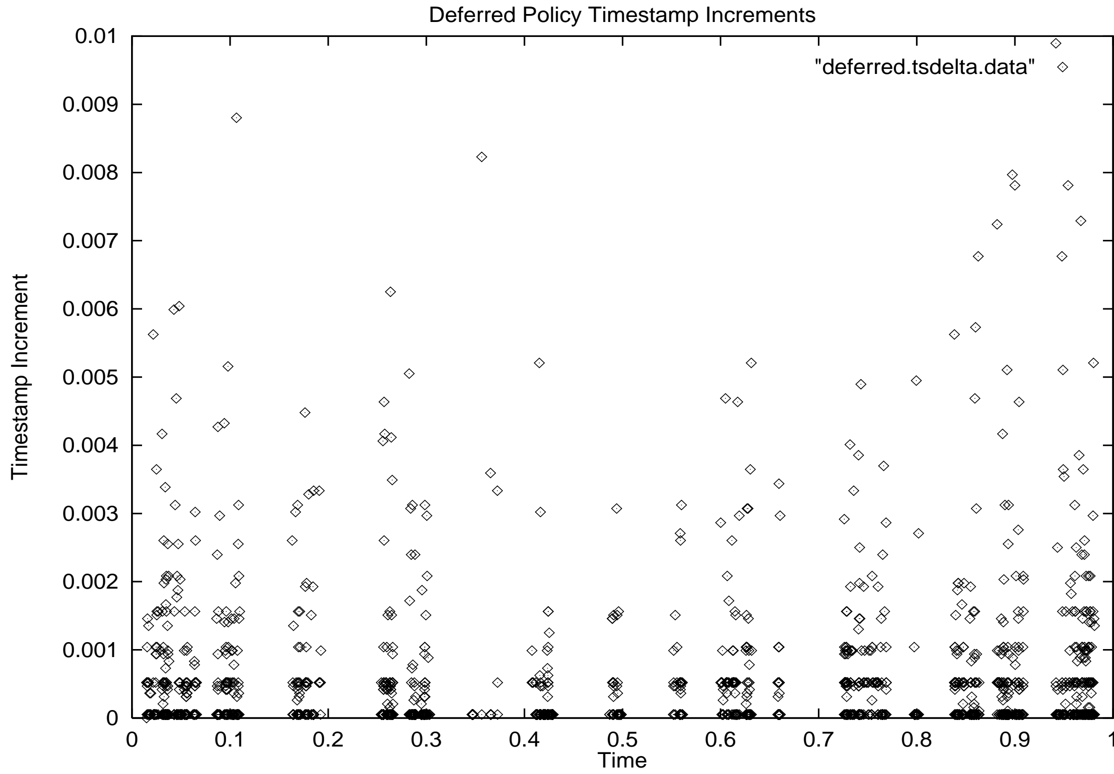

Figure 3

Initially, on seeing the first bit from flow i, set $vclock^i = baseVC^i = RealTime$.

Let l be the packet length of the jth packet received from flow i. Then do as follows:

- $$\delta_b^i = \begin{cases} \Delta_b^i & \text{if } b^i(j-1) < b^i(j) \le W^i \\ W^i - b^i(j-1) & \text{if } b^i(j-1) < W^i < b^i(j) \\ 0 & \text{otherwise} \end{cases}$$
- $vclock^i = max(vclock^i, RealTime)$

  $baseVC^i = max(baseVC^i, RealTime)$
- $vclock^i = \max\left(vclock^i + \frac{l}{\hat{r}^i}, vclock^i + l\ vtick^i - \frac{\delta_b^i}{r^i}\right)$

  $baseVC^i = vclock^i + l\ vtick^i$
- $vclock^i = \max\left(vclock^i, baseVC^i - \frac{W^i}{r^i}\right)$
- Stamp the packet with $vclock^i$

Figure 4

This analysis suggests modifying the Deferred policy so that the effective *vtick* never increases beyond the baseline *vtick*, $\frac{1}{r^i}$. Recall that the Deferred policy's timestamp increment is $\frac{1}{r^i} - \frac{\Delta_b^i}{r^i}$. The

proposed modification is to take as the timestamp increment $\frac{1}{r^i} - max(0, \frac{\Delta_b^i}{r^i})$. This rule, however, allows a flow to receive better service than it should under some circumstances. Even though a flow's timestamp values cannot become less than *RealTime* (because the vclock is resynchronized with *RealTime*), its timestamp values can become much smaller than those of the baseline timestamping rule. This can happen when a very bursty flow transmits faster than expected and its timestamps become greater than *RealTime*. By building up backlog, the flow can reduce its timestamp values by as much as $\frac{W^i}{r^i}$ relative to those of the baseline rule. By remaining idle until its backlog decreases below W^i, the flow receives another opportunity to reduce its timestamp values. By repeatedly building up backlog and then allowing it to drain, a flow can increase its priority almost arbitrarily

The solution to this problem is to prevent timestamp values from ever becoming smaller than $baselineTimestamp - \frac{W^i}{r^i}$. Figure 4 describes the complete timestamping rule, which we call Axel. $vclock^i$ is the state variable actually used to generate the timestamps. $baseVC^i$ is the baseline vclock and prevents a flow from arbitrarily increasing its priority. $vtick^i$ is $\frac{1}{r^i}$, the baseline vtick. $b^i(j)$ is the backlog when the jth packet is received, so that Δ_b^i is simply $b^i(j) - b^i(j-1)$. The variable δ_b^i implements bounded acceleration. If packet sizes are small relative to W^i, the definition of δ_b^i can be simplified by using 0 for both the second and third cases.

Axel retains all the positive qualities of the Deferred policy. A flow cannot increase its rate of service arbitrarily, because once its backlog reaches W^i, its vtick reverts to $\frac{1}{r^i}$. Also, because the vclock is resynchronized with *RealTime* and is not allowed to lag too far behind $baseVC^i$, a flow cannot arbitrarily reduce the value of its timestamps and increase its priority. Lastly, as in Virtual Clock, a misbehaving flow that transmits faster than claimed will quickly lose priority to flows adhering to their advertised rate. It is also worth pointing out that if $\hat{r}^i$ is set equal to r^i, Axel is equivalent to Virtual Clock.

The Axel policy performs very well. This is demonstrated in the next section.

4. Simulation Study

This section evaluates the performance of Axel via simulation.

4.1. Network Configuration and Methodology

Figure 5 shows the basic network configuration used for this study. It consists of a chain of 5 servers, numbered 0 to 4. Two *A* and six *D* flows traverse the entire chain of servers, from server 0 to server 4. These flows were used to study end-to-end latencies. Two *A* and five *D* cross-flows also traverse each server. Each of the cross-traffic flows traverses only one server.

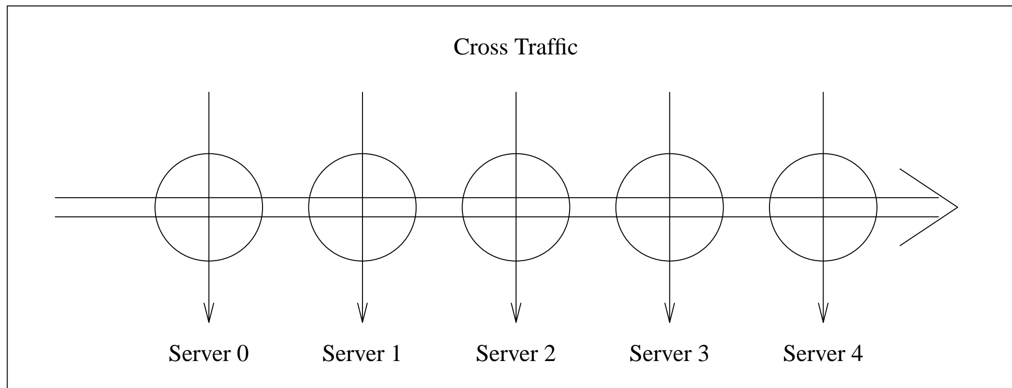

Figure 5

Large W Flow Characteristics						
Flow Type	r^i	actual rate	$\hat{r}^i$	W^i	packet size	burst size
A	92160	89834	921600	96000	48	64
D	92160	92198	92160	96000	48	1

Table 1

Small W Flow Characteristics						
Flow Type	r^i	actual rate	$\hat{r}^i$	W^i	packet size	burst size
A	76200	74384	921600	24000	48	64
D	76200	76231	76200	9600	48	1

Table 2

The packet sources used to drive the simulation were either Poisson or Poisson-based. We modeled each smooth source as a simple stream of constant-size packets with exponentially-distributed inter-packet gaps. We modeled bursty flows as on/off sources. During an active period, a bursty source produced a geometrically-distributed number of packets with constant inter-packet gap $\frac{1}{\hat{r}^i}$. Following an active period, the source remained quiescent for an exponentially-distributed time interval before becoming active again.

Though the limitations of Poisson and Poisson-related packet sources for driving network simulations, and particularly for modeling cross traffic, are well-known [7] , we believe our results are valid, in that they highlight the significant performance differences between Axel and other policies.

We simulated two principal cases: a “large-W” case, in which the advertised W^i’s of the flows were quite large, and a “small-W” case. Tables 1 and 2 describe the flows in the system for these two configurations. All quantities are in bytes and bytes/second, except for mean burst size, which is in packets. All flows were filtered with a leaky-bucket filter before entering the network. The actual rate columns show the actual rate after filtering, averaged across all flows of a particular type. For the small-W case, the “raw” average rate of the bursty flows before filtering was approximately 92,160 bytes/second.

For each configuration, we set the capacity of each server so that its utilization would be approximately 91.5%. Each server was allowed an unbounded amount of buffer space.

For each of the scenarios described below, we used the same packet streams to simulate each policy. However, we simulated each policy only once and did not compute confidence intervals. Except as noted below, the simulations cover a time interval of 100 seconds. After filtering, each source produced at least 153,000 packets in the small-W case, and at least 184,000 packets in the large-W case.

4.2. Performance Indices

We used two metrics to evaluate the performance of packet service policies: (1) the end-to-end delays of bursty flows traversing the entire chain of 5 servers. (2) The burstiness increase of the smooth flows, both at individual servers and over several hops. The burstiness increase is defined as the normalized ratio of output to input burstiness, $\frac{W_{out} - W_{in}}{W_{in}}$, after traversing one or more servers.

The burstiness increase of smooth flows is important because it has implications for buffer requirements. Greater burstiness also implies a hidden cost in end-to-end delay, to the extent that a flow made burstier in crossing the network must be smoothed at its destination.

4.3. Results

In addition to Axel, we studied the performance of several other policies: Pulse, PeakVC, Virtual Clock (VC), Fifo+, and Fifo. Fifo is ordinary first-come, first-served. PeakVC is Virtual Clock, but with the *vtick* based on the peak rate $\hat{r}^i$ instead of the average rate r^i. Pulse and Fifo+ were described in sections 2.3.2 and 2.3.3.

4.3.1. All Flows Behaving

When every flow abides by its advertised long-term average throughput r^i, Axel and PeakVC offer almost identically low delays to bursty flows, while Virtual Clock provides excellent service to the smoother flows at the expense of the bursty flows. Tables 3 and 4 show the bursty flow end-to-end delays and delay percentiles for the large-W case, while tables 5 and 6 show the corresponding data in the small-W case. In the large-W case, Axel has a clear performance advantage over all other policies. In the small-W case, the maximum Axel packet delays are only slightly better than under Fifo+. As table 6 shows, the performance advantage of Axel over Fifo+ increases if the application is willing to tolerate higher packet losses.

Tables 7 and 8 show the end-to-end delays of the smooth flows. Not surprisingly, these delays are quite large under Axel.

Bursty Flow End-to-End Packet Delays (ms) All Flows Behaving, Large W												
	Axel		Pulse		Peak VC		VC		Fifo+		Fifo	
Flow	mean	max	mean	max	mean	max	mean	max	mean	max	mean	max
0	1.1	23.5	2.0	186.3	1.1	24.1	53.4	378.4	19.3	61.1	19.0	82.6
1	1.1	29.7	20.6	443.4	1.1	24.9	47.5	545.5	19.1	62.0	18.7	84.6
max	1.1	29.7	20.6	443.4	1.1	24.9	53.4	545.5	19.3	62.0	19.0	84.6

Table 3

Bursty Flow End-to-End Packet Delay Percentiles (ms) All Flows Behaving, Large W						
	Axel		Fifo+		Pulse	
Flow	99%	99.9%	99%	99.9%	99%	99.9%
0	10.3	18.7	41.7	56.8	54.6	151.3
1	10.1	21.5	41.7	58.7	296.4	414.7

Table 4

Bursty Flow End-to-End Packet Delays (ms) All Flows Behaving, Small W												
	Axel		Pulse		Peak VC		VC		Fifo+		Fifo	
Flow	mean	max	mean	max	mean	max	mean	max	mean	max	mean	max
0	1.3	44.7	2.9	214.3	1.4	46.0	18.6	181.5	15.7	46.7	15.6	53.7
1	1.3	38.6	10.9	203.8	1.4	40.1	18.8	178.0	15.8	49.1	15.9	55.4
max	1.3	44.7	10.9	214.3	1.4	46.0	18.8	181.5	15.8	49.1	15.9	55.4

Table 5

Bursty Flow End-to-End Packet Delay Percentiles (ms) All Flows Behaving, Small W						
	Axel		Fifo+		Pulse	
Flow	99%	99.9%	99%	99.9%	99%	99.9%
0	8.4	37.8	33.1	41.5	22.4	145.9
1	9.9	24.3	34.3	43.3	132.6	175.5

Table 6

Tables 9 and 10 show the overall end-to-end burstiness increase and the per-server burstiness increase of smooth flows. The entries show the measured $\frac{W_{out} - W_{in}}{W_{in}}$ ratios, normalized so that 100 corresponds to a doubling in the burstiness. The row labeled “end” shows the overall burstiness increase of the smooth, end-to-end flow suffering the largest increase. The other entries show the largest burstiness increase at each server for each class of smooth flow. The columns marked “end” show the maximum burstiness increase among all the smooth end-to-end flows at each server. The columns marked “cross”

Smooth Flow End-to-End Packet Delays (ms) All Flows Behaving, Large W												
	Axel		Pulse		Peak VC		VC		Fifo+		Fifo	
Flow	mean	max	mean	max	mean	max	mean	max	mean	max	mean	max
2	11.9	123.8	10.2	120.4	11.9	123.8	0.7	49.3	17.3	61.2	16.2	84.6
3	14.4	171.0	11.1	183.0	14.4	171.0	0.8	43.8	17.3	62.0	16.1	84.4
4	5.0	79.0	2.5	68.7	5.1	79.0	0.2	29.7	17.3	61.9	16.1	84.7
5	53.2	291.0	63.4	334.9	53.2	291.0	4.8	128.2	17.3	62.0	16.1	84.6
6	7.1	99.6	3.4	92.2	7.0	99.6	0.3	32.3	17.3	62.0	16.1	84.5
7	18.8	193.8	6.6	124.9	19.0	193.8	1.2	58.7	17.3	62.0	16.1	84.6
max	53.2	291.0	63.4	334.9	53.2	291.0	4.8	128.2	17.3	62.0	16.2	84.7

Table 7

Smooth Flow End-to-End Packet Delays (ms) All Flows Behaving, Small W												
	Axel		Pulse		Peak VC		VC		Fifo+		Fifo	
Flow	mean	max	mean	max	mean	max	mean	max	mean	max	mean	max
2	17.3	133.9	19.0	144.8	17.3	133.9	6.4	92.0	14.4	49.0	13.8	55.5
3	17.6	135.0	19.0	157.0	17.6	135.0	7.6	109.0	14.4	49.0	13.7	55.4
4	24.0	200.9	32.1	198.2	24.0	200.9	2.1	73.7	14.4	48.8	13.7	55.6
5	8.0	126.1	6.2	113.4	8.0	126.7	0.6	47.6	14.4	48.8	13.7	55.4
6	48.2	265.7	14.9	158.8	48.2	265.5	36.1	234.7	14.3	48.7	13.7	55.4
7	20.6	191.6	13.6	179.7	20.6	191.4	16.5	181.4	14.4	49.0	13.7	55.4
max	48.2	265.7	32.1	198.2	48.2	265.5	36.1	234.7	14.4	49.0	13.8	55.6

Table 8

show the corresponding data for the smooth cross-flows. Not surprisingly, under Axel the burstiness increases are quite large. The large burstiness increases under Virtual Clock are surprising.

The maximum packet population observed at any of the servers with any of the policies during these simulations was approximately 730 packets for the small-W case and about 1230 packets for the large-W case.

Maximum Smooth Flow Burstiness Increase (percent) All Flows Behaving, Large W												
	Axel		Pulse		Peak VC		VC		Fifo+		Fifo	
Server	end	cross	end	cross	end	cross	end	cross	end	cross	end	cross
0	94	160	186	213	94	160	43	53	48	59	48	59
1	10	174	22	162	10	174	1	68	19	29	17	27
2	30	101	25	135	30	102	10	32	11	36	15	34
3	6	176	3	126	6	176	13	48	6	29	12	20
4	8	120	7	109	7	122	22	14	10	19	4	11
end	193	-	238	-	193	-	93	-	68	-	98	-

Table 9

Maximum Smooth Flow Burstiness Increase (percent) All Flows Behaving, Small W												
	Axel		Pulse		Peak VC		VC		Fifo+		Fifo	
Server	end	cross	end	cross	end	cross	end	cross	end	cross	end	cross
0	85	98	56	100	85	100	72	75	27	42	27	42
1	20	109	11	63	20	109	14	68	11	34	12	33
2	13	31	18	37	13	31	7	21	3	29	4	29
3	2	86	6	133	2	86	1	58	7	37	7	28
4	6	70	4	83	6	70	2	56	7	31	4	32
end	163	-	94	-	163	-	115	-	37	-	46	-

Table 10

4.3.2. Misbehaving Flows

To study the relative performance of the various policies in the presence of misbehaving flows, we set one of the bursty flows crossing server 2 (the middle server in the chain) to transmit much faster than claimed. We reduced the flow's inter-burst gap to $\frac{1}{10}$ of that used in the "all flows behaving" case. We also disabled leaky-bucket filtering for that flow. The misbehaving flow's actual rate for both the large- and small-W cases was approximately 490,000 bytes/second.

Tables 11 and 12 show the end-to-end delays of the bursty flows in this case. As can be seen, Axel effectively insulated the other flows in the system from the badly-behaved flow.

The results shown in the tables should be interpreted with some caution. The simulator used in this study currently allows only infinite buffer sizes at each server. For this reason, we simulated this configuration for only 10 seconds (instead of 100). Additionally, because no packets were dropped, the packets of the misbehaving flow did get serviced eventually, and did consume resources. With finite buffers and an appropriate packet-drop policy, most of the misbehaving flow's packets would have been dropped. Despite these caveats, it is clear that Axel can contain the effects of misbehaving flows.

Bursty Flow End-to-End Packet Delays (ms) Misbehaving Flow at Server 2, Large W												
	Axel		Pulse		Peak VC		VC		Fifo+		Fifo	
Flow	mean	max	mean	max	mean	max	mean	max	mean	max	mean	max
0	1.4	23.7	2.8	89.0	747.0	1548.8	34.3	174.0	774.4	1582.0	777.9	1580.7
1	1.6	25.0	1.2	74.0	762.1	1551.6	17.4	105.6	789.2	1584.4	793.1	1587.2
max	1.6	25.0	2.8	89.0	762.1	1551.6	34.3	174.0	789.2	1584.4	793.1	1587.2

Table 11

4.4. Summary

This section has shown that Axel performs extremely well. It provides bursty flows with very low end-to-end delays and by insulating flows from the ill-effects of misbehaving flows, provides every flow in the system with guaranteed throughput.

Bursty Flow End-to-End Packet Delays (ms) Misbehaving Flow at Server 2, Small W												
	Axel		Pulse		Peak VC		VC		Fifo+		Fifo	
Flow	mean	max	mean	max	mean	max	mean	max	mean	max	mean	max
0	1.9	29.0	4.6	131.4	990.3	1972.7	28.9	168.6	1020.6	2009.1	1021.9	2029.0
1	1.9	26.3	11.8	135.9	971.4	1969.7	34.2	173.7	1000.5	2014.9	1002.4	2030.5
max	1.9	29.0	11.8	135.9	990.3	1972.7	34.2	173.7	1020.6	2014.9	1021.9	2030.5

Table 12

5. Discussion, Limitations, and Future Work

The simulation results of section 4 and the informal analysis presented in section 3 demonstrate that Axel provides very low delay to bursty flows and provides bandwidth guarantees to all flows in the system. Axel is not perfect, however. In this section we briefly mention some possible improvements and suggest areas for further research.

Accelerating bursty A flows necessarily diminishes the quality of service received by smooth D flows. Axel limits the maximum amount of time during which a flow can remain accelerated. It also insulates other flows from the effects of misbehaving flows and can offer throughput guarantees to all flows in the system. However, these guarantees relate to long-term behavior. Axel can temporarily reduce the service rate of smooth D flows to a very low level. There are at least two interesting areas for investigation on this front. First, at this time, we have not investigated analytically the worst-case service rate and burstiness increase that Axel gives to smooth flows. Second, it may be possible to develop a policy that guarantees even individual packets of D flows a certain minimum throughput, and that also guarantees D flows a certain maximum burstiness increase. A policy that doesn't reduce the service rate of D flows too low, too often may be able to give such guarantees and still provide A flows with very low service delays.

We have not considered admission control policies. Axel provides each flow with guaranteed bandwidth, but guidelines for best allocating resources among flows so as to trade-off delay vs. burstiness need to be developed. A similar tradeoff can be expected when varying the value of the advertised W^i, since it determines the length of time during which a flow can remain accelerated. We have not studied the performance impact of varying the advertised r^i and W^i. More generally, instead of using W^i to limit the time during which a flow can remain accelerated, one could introduce an additional parameter.

Axel determines whether or not to accelerate a flow based solely on the flow's actual arrival pattern. This is an advantage, to the extent that Axel can preserve a flow's original arrival characteristics. However, if the network alters these characteristics, the information needed by Axel to determine whether or not to serve a flow preferentially will be destroyed. It seems worthwhile to consider using additional information. One approach that we are currently investigating is to use information about a packet's actual measured service delays, in the style of Fifo+ or Jitter-EDD [12]. In accelerating a packet, Axel uses the value of $\frac{\Delta_b^i}{r^i}$. This is the *expected* smoothing factor at a particular server. One could also use information about the

actual smoothing that has occurred in upstream nodes. Fifo+ uses the deviation between a packet's actual and expected service delays in upstream nodes to modify the timestamp. The difference in this value between successive packets reflects the actual smoothing that has occurred. These differences could be used to supplement or replace $\frac{\Delta_b^i}{r^i}$, but their use would add considerable complexity to the policy. Pulse seems immune to this problem, because it does not use the flow's actual arrival characteristics in its timestamping rule.

Lastly, the Axel timestamping rule is rather complex. Compared to Virtual Clock, Axel must keep an additional vclock and must track the change in backlog of the flow. It also incurs some additional complexity in enforcing a minimum timestamp increment. On the other hand, the most significant source of complexity in timestamp-based policies appears to be the need for some sort of priority queue. In this sense, Axel falls in roughly the same complexity class as Virtual Clock, which appears to be a practical packet service discipline [6].

6. Conclusions

This paper has presented the Axel packet service policy, which is targeted at environments composed of two classes of flows: very bursty, delay intolerant flows, and very smooth, delay-tolerant flows. In coming years, we expect the ever-increasing volume of continuous-media traffic to produce precisely this sort of environment. Bursty, delay-intolerant flows will arise from interactive applications such as teleconferencing, while smooth, delay-insensitive flows will arise from non-interactive applications such as video-on-demand.

The informal analysis in section 3 and the simulation study presented in section 4 show that Axel provides bursty flows with very low service delays and provides all flows in the system with bandwidth guarantees. It also outperforms policies such as Pulse, Fifo+, Virtual Clock, and Virtual Clock using peak bandwidth allocation. Axel does not appear to be significantly more complex than Virtual Clock or Fifo+. We believe it is a significant contribution to the support of future network services.

References

1. David P. Anderson, Shin-Yuan Tzou, Robert Wahbe, Ramesh Govindan, and Martin Andrews, "Support for Continuous Media in the DASH System," *Proceedings 10th International Conference on Distributed Computing Systems,* pp. 54-61, Paris, France (May 28 - June 1 1990).

2. David D. Clark, Scott Shenker, and Lixia Zhang, "Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism," *Proceedings ACM SIGCOMM'92 in ACM Computer Communications Review,* 22, 4, pp. 14-26 (October 1992).

3. Rene L. Cruz, "A Calculus for Network Delay Part I: Network Elements in Isolation," *IEEE Transactions on Information Theory,* 37, 1, pp. 114-131 (January 1991).

4. Alan Demers, Srinivasan Keshav, and Scott Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *Proceedings ACM SIGCOMM '89,* pp. 1-12 (September 1989).

5. D. Ferrari, "Client Requirements for Real-Time Communication Services," *IEEE Communications Magazine,* 28, 11, pp. 65-72 (November 1990).

6. Srinivasan Keshav, "Congestion Control in Computer Networks," Technical Report No. UCB/CSD 91/649, Computer Science Division, University of California, Berkeley, CA (1991). PhD Thesis ..

7. Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson, "On the Self-Similar Nature of Ethernet Traffic," *Proceedings SIGCOMM'93 in ACM Computer Communication Review,* 23, 4, pp. 183-193 (October 1993).

8. Amarnath Mukherjee, Lawrence H. Landweber, and Theordore Faber, "Dynamic Time Windows and Generalized Virtual Clock: Combined Closed-Loop/Open-Loop Congestion Control," *Proceedings IEEE Infocom,* pp. 0322-0332, Florence, Italy (1992).

9. Pramod Pancha and Magda El Zarki, "A look at the MPEG video coding standard for variable bit rate video transmission," *Proceedings IEEE Infocom,* pp. 0085-0094, Florence, Italy (1992).

10. A. Parekh, "A Generalized Processor Sharing Approach to Flow Control in Integrated Systems Services Networks," Technical Report LIDS-TR-2089, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology (1992). PhD Thesis.

11. Jonathan S. Turner, "New Directions in Communications (or Which Way to the Information Age)," *IEEE Communications Magazine,* 24, 10, pp. 8-15 (Oct 1986).

12. D. Verma, H. Zhang, and D. Ferrari, "Guaranteeing Delay Jitter Bounds in Packet Switching Networks," *Proceedings IEEE TriComm,* Chapel Hill, NC (April 1991).

13. Hui Zhang and Srinivasan Keshav, "Comparison of Rate-Based Service Disciplines," *Proceedings SIGCOMM'91,* pp. 113-121.

14. Lixia Zhang, "VirtualClock: A New Traffic Control Algorithm for Packet-Switched Networks," *ACM Transactions on Computer Systems,* 9, 2, pp. 101-124 (May 1991).