Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism

David D. Clark[1]
Laboratory for Computer Science
Massachusetts Institute of Technology
ddc@lcs.mit.edu

Scott Shenker Lixia Zhang
Palo Alto Research Center
Xerox Corporation
shenker, lixia@parc.xerox.com

Abstract

This paper considers the support of real-time applications in an *Integrated Services Packet Network* (ISPN). We first review the characteristics of real-time applications. We observe that, contrary to the popular view that real-time applications necessarily require a fixed delay bound, some real-time applications are more flexible and can adapt to current network conditions. We then propose an ISPN architecture that supports two distinct kinds of real-time service: *guaranteed* service, which is the traditional form of real-time service discussed in most of the literature and involves pre-computed worst-case delay bounds, and *predicted* service which uses the measured performance of the network in computing delay bounds. We then propose a packet scheduling mechanism that can support both of these real-time services as well as accommodate datagram traffic. We also discuss two other aspects of an overall ISPN architecture: the service interface and the admission control criteria.

1 Introduction

The current generation of telephone networks and the current generation of computer networks were each designed to carry specific and very different kinds of traffic: analog voice and digital data. However, with the digitizing of telephony in ISDN and the increasing use of multi-media in computer applications, this distinction is rapidly disappearing. Merging these sorts of services into a single network, which we refer to here as an *Integrated Services Packet Network* (ISPN), would yield a single telecommunications infrastructure offering a multitude of advantages, including vast economies of scale, ubiquity of access, and improved statistical multiplexing. There is a broad consensus, at least in the computer networking community, that an ISPN is both a worthy and an achievable goal. However, there are many political, administrative, and technical hurdles to overcome before this vision can become a reality.

One of the most vexing technical problems that blocks the path towards an ISPN is that of supporting real-time applications in a packet network. Real-time applications are quite different from standard data applications, and require service that cannot be delivered within the typical data service architecture. In Section 2 we discuss the nature of real-time applications at length; here, however, it suffices to observe that one salient characteristic of the real-time applications we consider is that they require a bound on the delivery delay of each packet[2]. While this bound may be statistical, in the sense that some small fraction of the packets may fail to arrive by this bound, the bound itself must be known *a priori*. The traditional data service architecture underlying computer networks has no facilities for prescheduling resources or denying service upon overload, and thus is unable to meet this real-time requirement.

Therefore, in order to handle real-time traffic, an enhanced architecture is needed for an ISPN. We identify four key components to this architecture. The first piece of the architecture is the nature of the commitments made by the network when it promises to deliver a certain quality of service. We identify two sorts of commitments, *guaranteed* and *predicted*. Predicted service is a major aspect of our paper. While the idea of predicted service has been considered before, the issues that surround it have not, to our knowledge, been carefully explored.

The second piece of the architecture is the service interface, i.e., the set of parameters passed between the source and the network. The service interface must include both the characterization of the quality of service the network will deliver, fulfilling the need of applications to know when their packets will arrive, and the characterization of the source's traffic, thereby allowing the network to knowledgeably allocate resources. In this paper we attempt to identify the critical aspects of the service interface, and offer a particular interface as an example. We address in passing the need for enforcement of these characterizations.

The third piece of the architecture is the packet scheduling behavior of network switches needed to meet these service commitments. We discuss both the actual scheduling algorithms to be used in the switches, as well as the scheduling information that must be carried in packet headers. This

[1] Research at MIT was supported by DARPA through NASA Grant NAG 2-582, by NSF grant NCR-8814187, and by DARPA and NSF through Cooperative Agreement NCR-8919038 with the Corporation for National Research Initiatives.

[2] Since the term *bound* is tossed around with great abandon in the rest of the paper, we need to identify several different meanings to the term. An *a priori* bound on delay is a statement that none of the future delays will exceed that amount. A *post facto* bound is the maximal value of a set of observed delays. Statistical bounds allow for a certain percentage of violations of the bound; absolute bounds allow none.

part of the architecture must be carefully considered; since it must be executed for every packet it must not be so complex as to effect overall network performance.

The final part of the architecture is the means by which the traffic and service commitments get established. Clearly, the ability of the network to meet its service commitments is related to the criteria the network uses to decide whether to accept another request for service. While we do not present a specific algorithm to regulate the admission of new sources, we show the relation between the other parts of our proposal and a general approach to the admission control problem.

There are also many architectural issues not directly related to the nature of real-time traffic; for instance, the issues of routing, datagram congestion control, and the interaction of administrative domains all pose interesting challenges. We do not address these issues in this paper, and any final architectural proposal for an ISPN must solve these longstanding problems. It is important to note, however, that we do not believe that the architectural choices we advocate here for real-time traffic unnecessarily restrict the scope of solutions to these other problems.

This paper has 12 Sections and an Appendix. In Section 2 we begin with a discussion of the nature of real-time traffic. In particular, we note that some real-time applications can adapt to current network conditions. This leads us to propose, in Section 3, that an ISPN support two kinds of real-time service commitments: *guaranteed* service and *predicted* service. In Section 4 we present a time-stamp based scheduling algorithm which is a nonuniformly weighted version of the *Fair Queueing* algorithm discussed in Reference [4], and then refer to a recent result due to Parekh and Gallager (see References [19, 20]) which states that, under certain conditions, this algorithm delivers guaranteed service in a network of arbitrary topology. We then turn, in Sections 5 and 6, to the scheduling algorithms best suited for providing predicted service. The predicted service scheduling algorithm incorporates two novel ideas; that of using FIFO service in a real-time context, and that of correlating the queueing delay of a packet at successive nodes in its path to reduce delay jitter. We combine these two scheduling algorithms in Section 7, presenting a unified scheduling algorithm which provides both guaranteed and predicted service. Given the current frenzy of activity in the design of real-time scheduling algorithms, we do not expect that the algorithm presented here will be the final word on the matter; however, we do hope that the insight embodied therein will be of lasting value. In particular, we think that the insight underlying our design, that it is necessary to distinguish between the two basic principles of *isolation* and *sharing*, is both fundamental and novel.

In Section 8 we return to the issue of the service interface. Since the service interface will be invoked by applications, we expect that a real-time service interface will outlive any particular underlying network mechanism. Thus, we have attempted in our proposal to produce an interface which is flexible enough to accommodate a wide variety of supporting mechanisms. Admission control policies are discussed briefly in Section 9, and the support of other service qualities is covered in Section 10.

In order to build up sufficient context to meaningfully compare our work to previously published work, we delay the detailed discussion of related work until Section 11. However, we wish to note here that our work borrows heavily from the rapidly growing literature on providing real-time service in packet networks. In particular, the works of Parekh and Gallager ([19, 20]), Jacobson and Floyd ([14]), and Lazar, Hyman, and Pacifici ([12, 13]) have all contributed to our design.

Finally, in Section 12, we conclude our paper with a review of our results and a brief discussion of related economic issues. The Appendix contains details relating to the simulation results that are presented in Sections 5-7.

2 Properties of Real-Time Traffic

2.1 A Class of Real-Time Applications

In the discussion that follows, we focus on a particular class of real-time applications which we dub *play-back* applications. In a play-back application, the source takes some signal, packetizes it, and then transmits it over the network. The network inevitably introduces some variation in the delay of each delivered packet. This variation has traditionally been called *jitter*. The receiver depacketizes the data and then attempts to faithfully play back the signal. This is done by buffering the incoming data to remove the network induced jitter and then replaying the signal at some designated *play-back* point. Any data that arrives before its associated *play-back* point can be used to reconstruct the signal; data arriving after the *play-back* point is useless in reconstructing the real-time signal. For the purposes of this paper, we assume that all such applications have sufficient buffering to store all packets which arrive before the play-back point; we return to this point in Section 10.

Not all real-time applications are *play-back* applications (for example, one might imagine a visualization application which merely displayed the image encoded in each packet whenever it arrived). However, we believe the vast majority of future real-time applications, including most video and audio applications, will fit this paradigm. Furthermore, non-play-back applications can still use the real-time network service provided by our architecture, although this service is not specifically tailored to their needs.

Play-back real-time applications have several service requirements that inform our design proposal. First, since there is often real-time interaction between the two ends of an application, as in a voice conversation, the application performance is sensitive to the data delivery delay; in general lower delay is much preferable. Second, in order to set the play-back point, the application needs to have some information (preferably an absolute or statistical bound) about the delays that each packet will experience. Third, since all data is buffered until the play-back point, the application is indifferent as to when data is delivered as long as it arrives before the play-back point[3]. This turns out to be a crucial point, as it allows us to delay certain packets which are in no danger of missing their play-back point in favor of packets which are. Fourth, these play-back applications can often tolerate the loss of a certain fraction of packets with only a minimal distortion in the signal. Therefore, the play-back point need not be so delayed that absolutely every packet arrives beforehand.

2.2 The Nature of Delay

The delay in the network derives from several causes. There is in practice a large fixed component to the delay, caused

[3] This is where we invoke the assumption, mentioned previously, that the receiver has sufficient buffers.

by the propagation of the packet at the speed of light, and the delay in transmission at each switch point waiting for the entire packet to arrive before commencing the next stage of transmission. (*Cut-through* networks avoid this delay by starting transmission before receipt is complete; most packet networks are not cut-through.) Added to this fixed delay is a variable amount of delay related to the time that each packet spends in service queues in the switches. This variation, or *jitter*, is what must be bounded and minimized if adequate real-time service is to be achieved.

Queueing is a fundamental consequence of the statistical sharing that occurs in packet networks. One way to reduce jitter might be to eliminate the statistical behavior of the sources. Indeed, one misconception is that real-time sources cannot be bursty (variable in their transmission rate), but must transmit at a fixed invariant rate to achieve a real-time service. We reject this idea; allowing sources to have bursty transmission rates and to take advantage of statistical sharing is a major advantage of packet networks. Our approach is thus to bound and characterize the burstiness, rather than eliminate it.

The idea of statistical sharing implies that there are indeed several sources using the bandwidth; one cannot share alone. Our approach to real-time traffic thus looks at the aggregation of traffic as fundamental; the network must be shared in such a way that clients (1) get better service than if there were no sharing (as in a circuit switched or TDM network) and (2) are protected from the potentially negative effects of sharing (most obviously the disruption of service caused by sharing with a mis-behaving source that overloads the resource).

2.3 Dealing with Delay

An application, in order to set its play-back point knowledgeably, needs to know some bound on the delay, plus an estimate of the fraction of packets missing that bound. This information forms the nucleus of the network's service specification in the service interface (to be discussed more fully in Section 8).

Some real-time applications will use an *a priori* delay bound advertised by the network to set the play-back point and will keep the play-back point fixed regardless of the actual delays experienced. These we dub *rigid* applications. For other applications, the receiver will measure the network delay experienced by arriving packets and then adaptively move the play-back point to the minimal delay that still produces a sufficiently low loss rate. We call such applications *adaptive*. Notice that adaptive applications will typically have an earlier play-back point than rigid applications, and thus will suffer less performance degradation due to delay. This is because the client's estimate of the *post facto* bound on actual delay will likely be less than the *a priori* bound pre-computed by the network. On the other hand, since the adaptation process is not perfect and may occasionally set the play-back point too early, adaptive applications will likely experience a higher loss rate.

The idea of adaptive applications is not relevant to circuit switched networks, which do not have jitter due to queueing. Thus most real-time devices today, like voice and video codecs, are not adaptive. Lack of widespread experience may raise the concern that adaptive applications will be difficult to build. However, early experiments suggest that it is actually rather easy. Video can be made to adapt by dropping or replaying a frame as necessary, and voice can adapt imperceptibly by adjusting silent periods. In fact, such adaptive approaches have been employed in packetized voice applications since the early 70's ([23]); the VT ([2]) and VAT ([15]) packet voice protocols, which are currently used to transmit voice on the Internet, are living examples of such adaptive applications[4]. It is important to note that while adaptive applications can adjust to the delivered delays over some range, there are typically limits to this adaptability; for instance, once the delay reaches a certain level, it becomes difficult to carry out interactive conversations.

Another useful distinction between network clients is how tolerant they are to brief interruptions in service. This level of tolerance is not just a function of the application, but also of the end users involved. For instance, a video conference allowing one surgeon to remotely assist another during an operation will not be tolerant of any service interruption, whereas a video conference-based family reunion might happily tolerate interruptions in service (as long as it was reflected in a cheaper service rate).

We can thus characterize network clients along two axes: adaptive or rigid, and tolerant or intolerant. It is unlikely that an intolerant network client is adaptive, since the adaptive process will likely lead, in the event of rapidly changing network conditions, to a brief interruption in service while the play-back point is re-adjusting. Furthermore, a tolerant client that is rigid is merely losing the chance to improve its delay. Such a combination of tolerance and rigidity would probably reflect the lack of adaptive hardware and software, which we believe will soon be cheap and standard enough to become fairly ubiquitous. We are thus led to the prediction that there will be two dominant classes of traffic in the network: intolerant and rigid clients, and tolerant and adaptive clients. We predict that these two classes will likely request very different service commitments from the network. Thus, these basic considerations about delay and how clients deal with it have produced a taxonomy of network clients that guides the goals of our architecture.

Before turning to the issue of service commitments, let us note that one of the key differences between real-time applications and the traditional datagram applications lies in the nature of the offered traffic. Data traffic is typically sporadic and unpredictable. In contrast, real-time applications often have some intrinsic packet generation process which is long lasting compared to the end-to-end delays of the individual packets. This process is a consequence of the specifics of the application; for example the coding algorithm for video, along with the nature of the image, will determine the packet generation process. Furthermore, this generation process can often be characterized as conforming to some traffic filter or bound (such as the token bucket filter which we describe in Section 4). For instance, the traffic generated by certain video codecs can often be characterized by some peak rate of packet generation. When a network has some knowledge of the traffic load it will have to carry, it can allocate its resources in a much more efficient manner.

3 Service Commitments

Clearly, for a network to *make* a service commitment to a particular client, it must know beforehand some characterization of the traffic that will be offered by that client. For the network to reliably *meet* its service commitment, the

[4] Yet another example of an early adaptive packet voice application is described in Reference [5].

client must meet its traffic commitment (i.e., its traffic must conform to the characterization it has passed to the network). Thus, the service commitment made to a particular client is predicated on the traffic commitment of that client. The question is, what else is the service commitment predicated on (besides the obvious requirement that the network hardware function properly)?

One kind of service commitment, which we will call *guaranteed* service, depends on no other assumptions. That is, if the network hardware is functioning and the client is conforming to its traffic characterization, then the service commitment will be met. Notice that this level of commitment does *not* require that any other network clients conform to their traffic commitments. Guaranteed service is appropriate for intolerant and rigid clients, since they need absolute assurances about the service they receive.

However, guaranteed service is not necessarily appropriate for tolerant and adaptive clients. Adaptive clients, by adjusting their play-back point to reflect the delays their packets are currently receiving, are gambling that the network service in the near future will be similar to that delivered in the recent past. Any violation of that assumption in the direction of increased delays will result in a brief degradation in the application's performance as packets begin missing the play-back point. The client will then readjust the play-back point upward to reflect the change in service, but there will necessarily be some momentary disruption in service. This will occur even if the network is meeting its nominal service commitments (based on the bounds on the service), because an adaptive application is typically ignoring those *a priori* bounds on delay and adapting to the current delivered service.

Thus, as long as the application is gambling that the recent past is a guide to the near future, one might as well define a class of service commitment that makes the same gamble. Our second kind of service commitment is called *predicted* service. This level of commitment has two components. First, as stated above, the network commits that if the past is a guide to the future, then the network will meet its service characterization. This component embodies the fact that the network can take into account recent measurements of the traffic load in estimating what kind of service it can deliver reliably. This is in marked contrast to the worst-case analysis that underlies the guaranteed service commitment. Second, the network attempts to deliver service that will allow the adaptive algorithms to minimize their play-back points. (This is the same as saying that the service will attempt to minimize the *post facto* delay bound.) Obviously, when the overall network conditions change, the quality of service must also change; the intent of the second component of the commitment is that when network conditions are relatively static, the network schedules packets so that the current *post facto* delay bounds (which are typically well under the long-term *a priori* bounds that are part of the service commitment) are small.

Notice that predicted service has built into it very strong *implicit* assumptions about the behavior of other network clients by assuming that the network conditions will remain relatively unchanged, but involves very few *explicit* assumptions about these other network clients; i.e., their current behavior need not be explicitly characterized in any precise manner. Thus, for predicted service, the network takes steps to deliver consistent performance to the client; it avoids the hard problem, which must be faced with guaranteed service, of trying to compute *a priori* what that level of delivered service will be.

We have thus defined two sorts of real time traffic, which differ in terms of the service commitment they receive. There is a third class of traffic that we call *datagram* traffic, to which the network makes no service commitments at all, except to promise not to delay or drop packets unnecessarily (this is sometimes called *best effort* service).

We now have the first component of our architecture, the nature of the service commitment. The challenge, now, is to schedule the packet departures at each switch so that these commitments are met. For the sake of clarity, we first consider, in Section 4, how to schedule guaranteed traffic in a network carrying only guaranteed traffic. In Sections 5 and 6 we then consider how to schedule predicted traffic in a network carrying only predicted traffic. After we have assembled the necessary components of our scheduling algorithm we then, in Section 7, present our unified scheduling algorithm which simultaneously handles all three levels of service commitment.

As we present these scheduling schemes, we also lay the groundwork for the other key pieces of the architecture, the specifics of the service interface (which must relate closely to the details of the service commitment) and the method to control the admission of new sources.

4 Scheduling Algorithms for Guaranteed Traffic

In this section we first describe a traffic filter and then a scheduling algorithm that together provide guaranteed service.

As discussed briefly in Section 3, a network client must characterize its traffic load to the network, so that the network can commit bandwidth and manage queues in a way that realizes the service commitment. We use a particular form of traffic characterization called a *token bucket* filter. A token bucket filter is characterized by two parameters, a rate r and a depth b. One can think of the token bucket as filling up with tokens continuously at a rate r, with b being its maximal depth. Every time a packet is generated p tokens are removed from the bucket, where p is the size of the packet. A traffic source conforms to a token bucket filter (r, b) if there are always enough tokens in the bucket whenever a packet is generated.

More precisely, consider a packet generation process with t_i and p_i denoting the generation time and size, respectively, of the i'th packet. We say that this traffic source conforms to a token bucket filter (r, b) of rate r and depth b if the sequence n_i defined by $n_0 = b$ and $n_i = MIN[b, n_{i-1} + (t_i - t_{i-1})r - p_i]$ obeys the constraint that $n_i \geq 0$ for all i. The quantities n_i, if nonnegative, represent the number of tokens residing in the bucket after the i'th packet leaves. For a given traffic generation process, we can define the nonincreasing function $b(r)$ as the minimal value such that the process conforms to a $(r, b(r))$ filter.

In recent years, several *time-stamp* based algorithms have been developed. These algorithms take as input some preassigned apportionment of the link expressed as a set of rates r^α (where α labels the flows); the resulting delays depend on the bucket sizes $b^\alpha(r^\alpha)$.

One of the first such time-stamp algorithms was the *Fair Queueing* algorithm introduced in Reference [4]. This algorithm was targeted at the traditional data service architecture, and so involved no preallocation of resources (and thus had each $r^\alpha = \mu$ where μ denotes the link speed).

In addition, a weighted version of the Fair Queueing algorithm (which we refer to as WFQ), in which the r^α need not all be equal, was also briefly described in Reference [4][5]. The *VirtualClock* algorithm, described in References [25, 26], involves an extremely similar underlying packet scheduling algorithm, but was expressly designed for a context where resources were preapportioned and thus had as a fundamental part of its architecture the assumption that the shares r^α were arbitrary. Parekh and Gallager, in Reference [19], reintroduce the WFQ algorithm under the name of *packetized generalized processor sharing* (PGPS). They have proven the important result that this algorithm, under certain conditions, can deliver a guaranteed quality of service ([20]). We present a brief summary of the WFQ algorithm below, since we make use of it in our overall scheduling algorithm; see References [4, 20] for more details.

First, consider some set of flows and a set of *clock rates* r^α. The clock rate of a flow represents the relative share of the link bandwidth this flow is entitled to; more properly, it represents the proportion of the total link bandwidth which this flow will receive when it is active. By assigning it a clock rate r^α the network commits to provide to this flow an effective throughput rate no worse than $(\mu r^\alpha)/(\sum_\beta r^\beta)$ where the sum in the denominator is over all currently active flows.

This formulation can be made precise in the context of a fluid flow model of the network, where the bits drain continuously out of the queue. Let t_i^α and p_i^α denote the generation time and size, respectively, of the i'th packet arriving in the α'th flow. We define the set of functions $m^\alpha(t)$, which characterize at any time the backlog of bits which each source has to send, and set $m^\alpha(0) = 0$. We say that a flow is active at time t if $m^\alpha(t) > 0$; let $A(t)$ denote the set of active flows. Then the dynamics of the system are determined as follows. Whenever a packet arrives, m must discontinuously increase by the packet size: $m^\alpha(t^+) = m^\alpha(t^-) + p_i$ if $t = t_i^\alpha$, where $m^\alpha(t^+)$ and $m^\alpha(t^-)$ refer to right hand and left hand limits of m^α at t. At all other times, we know that the bits are draining out of the queues of the active flows in proportion to the clock rates of the respective flows:

$$\frac{\partial m^\alpha(t)}{\partial t} = \frac{\mu r^\alpha}{\sum_{\beta \in A(t)} r^\beta} \text{ if } \alpha \in A(t)\,,\ \frac{\partial m^\alpha(t)}{\partial t} = 0 \text{ if } \alpha \notin A(t)$$

This completely characterizes the dynamics of the fluid flow model. Parekh and Gallager have shown the remarkable result that, in a network with arbitrary topology, if a flow gets the same clock rate at every switch and the sum of the clock rates of all the flows at every switch is no greater than the link speed, then the queueing delay of that flow is bounded above by $b^\alpha(r^\alpha)/r^\alpha$. Intuitively, this bound is the delay that would result from an instantaneous packet burst of the token bucket size being serviced by a single link of rate r^α; the queueing delays are no worse than if the entire network were replaced by a single link with a speed equal to the flow's clock rate r^α. This result can be motivated by noting that if flow α's traffic were put through a leaky bucket filter of rate r^α at the edge of the network[6], then the flow would not suffer any further queueing delays within the network since the instantaneous service rate given to this flow at every switch along the path would be at least r^α. Thus, all of the queueing delay would occur in the leaky bucket filter and, since the flow obeys an $(r^\alpha, b^\alpha(r^\alpha))$ token bucket filter, the delay in the leaky bucket filter would be bounded by $b^\alpha(r^\alpha)/r^\alpha$. Notice that the delay bound of a particular flow is *independent* of the other flows' characteristics; they can be arbitrarily badly behaved and the bound still applies. Furthermore, these bounds are strict, in that they can be realized with a set of *greedy* sources which keep their token buckets empty.

The previous paragraphs describe WFQ in the fluid flow approximation. One can define the packetized version of the algorithm in a straightforward manner. Define $\delta_i^\alpha(t)$ for all $t \geq t_i^\alpha$ as the number of bits that have been serviced from the flow α between the times t_i^α and t. Associate with each packet the function $E_i^\alpha(t) = (m^\alpha(t_i^\alpha) - \delta_i^\alpha(t))/r^\alpha$ where we take the right-hand limit of m; this number is the level of backlog ahead of the packet i in the flow α's queue divided by the flow's share of the link, and can be thought of as an *expected* delay until departure for the last bit in the packet. The packetized version of WFQ is merely, at any time t when the next packet to be transmitted must be chosen, to select the packet with the minimal $E_i^\alpha(t)$. This algorithm is called a time-stamp based scheme because there is an alternative but equivalent formulation in which each packet is stamped with a *time-stamp* as it arrives and then packets are transmitted in increasing order of time-stamps; see References [4, 20] for details on this formulation. Parekh and Gallager have shown that a bound, similar to the fluid flow bound, applies to this packetized algorithm as well. However, the formulae for the delays in the packetized case are significantly more complicated; see Reference [20] for details.

To understand the relation between the clock rate r, the bucket size $b(r)$ and the resultant delay, consider what happens to a burst of packets. The packet that receives the highest queueing delay is the last packet of a burst. The bound on the jitter is proportional to the size of the burst and inversely proportional to the clock rate. The means by which the source can improve the worst case bound is to increase its clock rate r to permit the burst to pass through the network more quickly.

Since the bounds given in guaranteed service must be worst-case bounds (i.e. the bounds must apply for all possible behaviors of the other sources), the primary function of a scheduling algorithm designed to deliver guaranteed service is to *isolate* flows from each other, so that a flow can have only a limited negative effect on other flows. The WFQ scheme isolates each source from the others by providing it a specified share of the bandwidth under overload conditions. The work of Parekh and Gallager provides a way for the source to compute the maximum queueing delay which its packets will encounter, provided that the source restricts itself to a (r, b) token bucket filter. But the network's scheduling algorithm does not depend on this filter. Indeed, an important point about this form of guaranteed service is that the traffic filters do not play any role in packet scheduling.

Given that the WFQ algorithm can deliver guaranteed service, why not use this algorithm to provide predicted service as well? Recall that the goal of guaranteed service was to provide worst-case guarantees, which are required by intolerant and rigid applications. The goal of predicted service is not to provide worst-case delay bounds, but rather to minimize the actual *post facto* delay bounds thereby allowing tolerant and adaptive applications to reel in their play-back point. In the next section, we show that the WFQ algo-

[5] The weighted version of Fair Queueing is mentioned on page 24 of Reference [4], though not referred to by the name Weighted Fair Queueing.

[6] In a fluid flow version of a leaky bucket of rate r, the bits drain out at a constant rate r and any excess is queued.

rithm, with its emphasis on isolation, is not well suited for this task. Thus, algorithms specifically designed to provide predicted service must be found.

5 Scheduling Algorithms for Predicted Service

We motivate the development of our scheduling algorithm by considering the following *gedanken experiment*. Consider a single-link network carrying some number of clients, and assume that all sources conform to some traffic filter such as the token buckets described above. Furthermore, assume that all the clients are bursty sources, and wish to mix their traffic so that in the aggregate they achieve a better use of bandwidth and a controlled delay. How does one best schedule the packets to achieve low *post facto* delay bounds (or, equivalently, minimal play-back points)?

What behavior does the WFQ algorithm induce? Whenever there is a backlog in the queue, packets leave the queue at rates proportional to their clock rates. Consider a moment when all sources are transmitting uniformly at their clock rates except for one which emits a burst of packets. The WFQ algorithm would continue to send the packets from the uniform sources at their clock rates, so their packets are not queued for any significant time whereas the backlog of packets from the bursty source will take a long time to drain. Thus, a burst by one source causes a sharp increase in the delay seen by that source, and has minimal effects on the delays seen by the other sources. The mean delay will be rather low, assuming the network link is not over-committed, but a burst will induce jitter directly, and mostly, affecting only the source that emitted the burst.

WFQ provides for a great degree of isolation, so that sources are protected from other sources' bursts. Is this the best approach to obtaining the lowest play-back point when a number of sources are sharing a link? We argue that this isolation, while necessary for providing guaranteed service, is counterproductive for predicted service.

The nature of play-back real-time applications allows the scheduling algorithm to delay all packets up to the play-back point without adversely affecting the application's performance. Thus, one can think of the play-back point as a deadline. For such problems, the standard earliest-deadline-first scheduling algorithm, as described in Reference [17], has been proven optimal. However, in our *gedanken experiment* the play-back points are not set *a priori*, as in the above reference, but are rather the result of the clients adapting to the current level of delay.

Let us consider a simple example where a class of clients have similar service desires. This implies that they are all satisfied with the same delay jitter; thus they all have the same play-back point and thus the same deadline. If the deadline for each packet is a constant offset to the arrival time, the deadline scheduling algorithm becomes, surprisingly, FIFO; the packet that is closest to its deadline is the one that arrived first. Hyman, Lazar, and Pacifici, in Reference [13], also make this observation that FIFO is merely a special case of deadline scheduling.

Consider what happens when we use the FIFO queueing discipline instead of WFQ. Now when a burst from one source arrives, this burst passes through the queue in a clump while subsequent packets from the other sources are temporarily delayed; this latter delay, however, is much smaller than the delay that the bursting source would have received under WFQ. Thus, the play-back point need not be moved out as far to accommodate the jitter induced by the burst. Furthermore, the particular source producing the burst is not singled out for increased jitter; all the sources share in all the jitter induced by the bursts of all the sources. Recall that when the packets are of uniform size, the total queueing delay in any time period (summed over all flows) is independent of the scheduling algorithm. The FIFO algorithm splits this delay evenly, whereas the WFQ algorithm assigns the delay to the flows that caused the momentary queueing (by sending bursts). When the delays are shared as in FIFO, in what might be called a multiplexing of bursts, the *post facto* jitter bounds are smaller than when the sources are isolated from each other as in WFQ. This was exactly our goal; under the same link utilization, FIFO allows a number of sources aggregating their traffic to obtain a lower overall delay jitter.

scheduling	mean	99.9 %ile
WFQ	3.16	53.86
FIFO	3.17	34.72

Table 1: The mean and 99.9'th percentile queueing delays (measured in milliseconds, which is a single packet transmission time) for a sample flow under the WFQ and FIFO scheduling algorithms. The link is 83.5% utilized.

In order to test our intuition, we have simulated both the WFQ and FIFO algorithms. The Appendix contains a complete description of our simulation procedure; we only present the results here. We consider a single 1 Mbit/sec link being utilized by 10 flows, each having the same statistical generation process. In Table 1 we show the mean and 99.9'th percentile queueing delays for a sample flow (the data from the various flows are similar) under each of the two scheduling algorithms. Note that while the mean delays are about the same for the two algorithms, the 99.9'th percentile delays are significantly smaller under the FIFO algorithm. This confirms our analysis above.

The FIFO queue discipline has generally been considered ineffective for providing real-time service; in fact, it has been shown in certain circumstances to be the *worst* possible algorithm ([1]). The reason is that if one source injects excessive traffic into the net, this disrupts the service for everyone. This assessment, however, arises from a failure to distinguish the two separate objectives of any traffic control algorithm, *isolation* and *sharing*. Isolation is the more fundamental goal; it provides guaranteed service for well-behaved clients and quarantines misbehaving sources. But sharing, if it is performed in the context of an encompassing isolation scheme, performs the very different goal of mixing traffic from different sources in a way that is beneficial to all; bursts are multiplexed so that the *post facto* jitter is smaller for everyone. The FIFO scheme is an effective sharing scheme, but it does not provide any isolation. WFQ, on the other hand, is an effective method for isolation. If we organize the traffic into classes of clients with similar service requirements, we find that this reasoning leads to a nested scheme in which the queuing decision is in two steps: a first step to insure isolation of classes, and then a particular sharing method within each class.

FIFO is not the only interesting sharing method. Another sharing method is priority, which has a very different behavior than FIFO. The goal of FIFO is to let every source

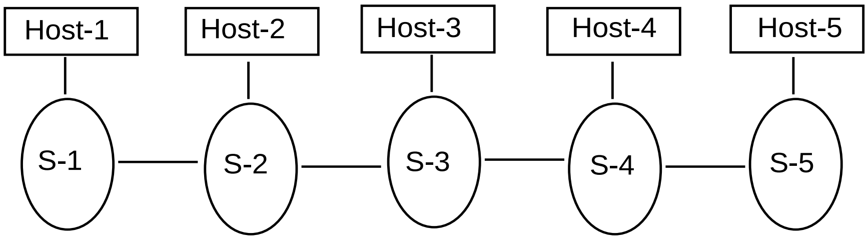

Figure 1: Network topology used for data in Table 2.

	Path Length							
	1		2		3		4	
scheduling	mean	99.9 %ile	mean	99.9 %ile	mean	99.9 %ile	mean	99.9 %ile
WFQ	2.65	45.31	4.74	60.31	7.51	65.86	9.64	80.59
FIFO	2.54	30.49	4.73	41.22	7.97	52.36	10.33	58.13
FIFO+	2.71	33.59	4.69	38.15	7.76	43.30	10.11	45.25

Table 2: The mean and 99.9'th percentile queueing delays (measured in milliseconds, which is a single packet transmission time) for four sample flows of different path lengths under the WFQ, FIFO, and FIFO+ scheduling algorithms. The network configuration is shown in Figure 1. Each inter-switch link is 83.5% utilized.

in a common class share equally in the jitter. In priority, one class acquires jitter of higher priority classes, which consequently get much lower jitter. In one direction priority is considered a sharing mechanism, but in the other it is an isolation mechanism, i.e. lower priority traffic can never affect the performance of higher priority one.

Why might a priority algorithm be of mutual benefit? The benefit of lower jitter is obvious; the benefit of higher jitter would presumably be a lower cost for the service. A source with more tolerance for jitter (or for higher overall delay) might be very happy to obtain a cheaper service in exchange for taking the jitter of some other sources.

One can think in general of scheduling algorithms as representing methods for *jitter shifting*, in which explicit actions are taken to transfer the jitter among flows in a controlled and characterized way. One could invent a wide range of scheduling schemes that reorder the queue in specific ways, as we discuss in the section on related work. They should all be examined from two perspectives. First, how and to what extent do they perform isolation? Second, how and to what extent do they provide sharing?

6 Multi-Hop Sharing

One of the problems with the FIFO algorithm is that if we generalize our *gedanken experiment* to include several links, then the jitter tends to increase dramatically with the number of hops, since the packet has a separate opportunity for uncorrelated queueing delays at each hop.

In fact, it is not clear that this increase in jitter need occur. Going through more hops provides more opportunities for sharing, and hence more opportunities for reducing jitter. The key is to correlate the sharing experience which a packet has at the successive nodes in its path. We call this scheme FIFO+. FIFO+ is qualitatively very similar to the *least slack* scheduling algorithms for manufacturing systems discussed in Reference [18].

In FIFO+, we try to induce FIFO-style sharing (equal jitter for all sources in the aggregate class) across all the hops along the path to minimize jitter. We do this as follows. For each hop, we measure the average delay seen by packets in each aggregate class at that switch. We then compute for each packet the difference between its particular delay and the class average. We add (or subtract) this difference to a field in the header of the packet, which thus accumulates the total offset for this packet from the average for its class. This field allows each switch to compute when the packet *should* have arrived if it were indeed given average service. The switch then schedules the packet as if it arrived at this average expected time; this is done by ordering the queue by these expected arrival times rather than by the actual arrival times.

To test this algorithm, we have simulated its performance on a network as shown on Figure 1. This network has four equivalent 1 Mbit/sec inter-switch links, and each link is shared by 10 flows. There are, in total, 22 flows; all of them have the same statistical generation process (described in the Appendix) but they travel different network paths. 12 traverse only one inter-switch link, 4 traverse two inter-switch links, 4 traverse three inter-switch links, and 2 traverse all four inter-switch links. Table 2 displays the mean and 99.9'th percentile queueing delays for a single sample flow for each path length (the data from the other flows are similar). We compare the WFQ, FIFO, and FIFO+ algorithms (where we have used equal clock rates in the WFQ algorithm). Note that the mean delays are comparable in all three cases. While the 99.9'th percentile delays increase with path length for all three algorithms, the rate of growth is much smaller with the FIFO+ algorithm.

As the simulation shows, the effect of FIFO+, as compared to FIFO, is to slightly increase the mean delay and jitter of flows on short paths, slightly decrease the mean delay and significantly decrease the jitter of flows on long paths, which means that the overall delay bound goes down and the precision of estimation goes up on long paths. When we compare the implementation of the two schemes, they differs in one important way – the queue management discipline is no longer trivial (add the packet to the tail of the queue for the class) but instead requires that the queue be ordered by deadline, where the deadline is explicitly computed by taking the actual arrival time, adjusting this by the offset in the packet header to find the *expected* arrival time, and then using this to order the queue. This has the possibility of a more expensive processing overhead, but we believe that efficient coding methods can implement this in software with the same performance as current packet switches achieve.

We have now extended our predicted service class to multiple hops, using FIFO+ as an explicit means to minimize the jitter by obtaining as much benefit as possible from sharing. Recall that for guaranteed service, the service is specified by the worst-case bounds and thus the focus is on scheduling algorithms that provide *isolation* between the various flows. In contrast, for predicted service we assume that (1) adequate isolation is being provided by the enforcement of traffic filters before or at the entrance to the network rather than by the scheduling algorithm itself, and (2) network conditions, rather than being worst-case, are rather similar to those of the recent past. For predicted service, the challenge is to *share* the link effectively in a way that *minimizes* the play-back point. As we have seen, FIFO is a effective sharing mechanism. The modification of FIFO+ merely extends the concept of sharing from sharing between flows at a single hop to sharing between hops.

7 Unified Scheduling Algorithm

In the previous three sections we have presented scheduling algorithms that each handle a single kind of service commitment. In this section we combine these algorithms into a unified scheduling algorithm that handles guaranteed, predicted, and datagram service.

Consider a set of real-time flows, some requesting guaranteed service and some requesting predicted service, and also a set of datagram sources. We first describe the scheduling algorithm as implemented at each switch and then discuss how this fits into our overall service architecture.

The scheduling algorithm at a single switch is quite straightforward. The basic idea is that we must isolate each of the guaranteed service flows from each other and from the predicted service flows (and the datagram traffic). Therefore we use the time-stamp based WFQ scheme as a framework into which we fit the other scheduling algorithms[7]. Each guaranteed service client α has a separate WFQ flow with some clock rate r^{α}. All of the predicted service and datagram service traffic is assigned to a pseudo WFQ flow, call it flow 0, with, at each link, $r^0 = \mu - \sum_{\alpha} r^{\alpha}$ where the sum is over all the guaranteed service flows passing through that link. Inside this flow 0, there are a number of strict priority classes, and within each priority class we operate the FIFO+ algorithm. Once we have assigned each predictive service flow (and also the datagram traffic) to a priority level at each switch, the scheduling algorithm is completely defined. We now discuss how this algorithm fits into our overall service architecture.

We have discussed the function of the FIFO+ scheme above. What is the role of the priority classes? Remember above that the effect of priority is to shift the jitter of higher priority class traffic to the lower priority classes. We assign datagram traffic to the lowest priority class. There are K other priority levels above the datagram priority level.

At the service interface, we provide K widely spaced target delay bounds D_i for predicted service (at a particular switch). The priorities are used to separate the traffic for the different K classes. These bounds D_i are *not* estimates of the actual delivered delays. Rather, they are *a priori* upper bounds and the network tries, through admission control policies, to keep queueing delays at each switch for a particular class i well below these bounds D_i. We mentioned earlier that adaptive applications have limits to their adaptability; these bounds D_i are indicative of such limits. A predicted service flow is assigned a priority level at each switch (not necessarily the same level in every switch); the *a priori* delay bound advertised to a predicted service flow is the sum of the appropriate D_i along the path. The delay bound advertised to a guaranteed service flow is the Parekh-Gallager bound.

This scheme has the problem that, since delay is additive, asking for a particular D_i at a given switch does not directly mean that D_i is the target delay bound for the path as a whole. Rather, it is necessary to add up the target delays at each hop to find the target upper bound for the path. We expect the true *post facto* bounds over a long path to be significantly lower than the sum of the bounds D_i at each hop. But we suggest that, since this is an adaptive service, the network should not attempt to characterize or control the service to great precision, and thus should just use the sum of the D_i's as the advertised bound.

Consider in more detail how the priority scheme works. If the highest priority class has a momentary need for extra bandwidth due to a burst by several of its sources, it steals the bandwidth from the lower classes. The next class thus sees as a baseline of operation the aggregate jitter of the higher class. This gets factored together with the aggregate burstiness of this class to produce the total jitter for the second class. This cascades down to the datagram traffic, which gets whatever bandwidth is leftover and suffers from the accumulated jitter. As we argue later, the datagram traffic should probably be given an average rate of at least 10% or so, both to insure that it makes some progress on the average and to provide a reasonable pool of bandwidth for the higher priority traffic to borrow from during momentary overloads.

For a lower priority class, what source of jitter will dominate its observed behavior: its intrinsic aggregate behavior or the jitter shifted from the higher priority classes? If the target goals for jitter are widely spaced (and for the purpose of rough estimation as we suggested above they probably need be no closer than an order of magnitude) then the exported jitter from the higher priority class should be an order of magnitude less than the intrinsic behavior of the class, and the classes should usually operate more or less independently. Thus, a particular class is isolated from the lower priority classes by the priority scheduling algorithm and is in effect isolated from the higher priority classes because their jitter will be so much smaller than that of the particular class.

We have simulated this unified scheduling algorithm using the same simulation configuration as used for Table 2, that has 22 real-time flows with identical statistical generation processes but which traverse different network paths. To these 22 real-time flows we added 2 datagram TCP connections. In this simulation, 5 of the real-time flows are guaranteed service clients; 3 of these have a clock rate equal

[7] We should note that there are perhaps other isolating mechanisms that could be substituted for WFQ.

Guaranteed Service						Predicted Service					
type	path length	delay measure mean	99.9 %ile	max	P-G bound	type	path length	delay measure mean	99.9 %ile	max
Peak	4	7.92	14.31	15.94	23.53	High	4	3.12	8.21	10.71
Peak	2	3.80	8.12	8.79	11.76	High	2	1.61	5.89	7.04
Average	3	54.35	242.49	292.46	611.76	Low	3	18.04	100.55	120.88
Average	1	35.40	217.91	255.46	588.24	Low	1	6.94	72.46	106.56

Table 3: The mean, 99.9'th percentile, and maximum queueing delays (measured in milliseconds, which is a single packet transmission time) in eight sample flows under the unified scheduling algorithm. The network configuration is shown in Figure 1. Each inter-switch link is utilized over 99%.

to their peak packet generation rate (we denote such flows by Guaranteed-Peak) and the other 2 have a clock rate equal to their average packet generation rate (we denote such flows by Guaranteed-Average). See the Appendix for details on the statistical generation process and the values of the average and peak rates. The remaining 17 real-time flows are predicted service clients served by two priority classes, 7 flows are in the high priority class (we denote such flows by Predicted-High) and the other 10 flows are in the low priority class (we denote such flows by Predicted-Low). If we look at the traffic traversing each link, it consists of one datagram connection and 10 real-time flows: 2 Guaranteed-Peak, 1 Guaranteed-Average, 3 Predicted-High, and 4 Predicted-Low.

Results for eight sample real-time flows in this simulation are presented in Table 3. The mean, 99.9'th percentile, and the maximum delays are displayed for all real-time flows, and the Parekh-Gallager (P-G) delay bounds are shown for the guaranteed service flows. We see that all of the guaranteed service flows received worst-case delays that were well within the Parekh-Gallager bounds. Not surprisingly, the Guaranteed-Peak flows experienced much lower delays than the Guaranteed-Average flows. Similarly, the Predicted-High flows experienced lower delays than the Predicted-Low flows. For the given load pattern described here, the delays of the Predicted-High flows were lower than those of the comparable Guaranteed-Peak flows, and the delays of the Predicted-Low flows were lower than those of the comparable Guaranteed-Average flows; however, this relation between the delays of the two classes is an artifact of the particular load pattern and is not necessarily indicative of a general pattern.

Not shown in Table 3 is the performance of the datagram traffic which experienced a very low drop rate, around 0.1%. The overall utilization of the network was over 99%, with 83.5% of this being real-time traffic. It is important to note that if all of the real-time flows had requested guaranteed service with a clock rate equal to their peak rate, the network could accommodate many fewer real-time flows and the utilization due to real-time traffic would be reduced to roughly 50%. Thus, providing predicted service allows the network to operate with a higher degree of real-time traffic than would be allowed by a pure guaranteed service offering the same delay bounds. These results, though woefully incomplete, are qualitatively consistent with our analysis.

We are currently attempting to more fully validate our design through simulation, and we hope to report on our progress in a subsequent publication. Note that much of the challenge here is determining how to evaluate our proposal. There is no widely accepted set of benchmarks for real-time loads, and much of the novelty of our unified scheduling algorithm is our provision for predicted service, which can only be meaningfully tested in a dynamic environment with adaptive clients.

We have now completed the first parts of our architecture. We have described a model for the low-level packet forwarding algorithm, which is a sharing discipline inside an isolation discipline, and we have provided a particular example of such a scheme, which provides both of our service commitment models, guaranteed and predicted. The scheme provides several predicted service classes with different delay bounds, and uses a particular technique (FIFO+) to provide low jitter, and to provide a jitter bound that does not vary strongly with the number of hops in the paths.

8 Service Interface

As a part of the definition of the unified scheduling algorithm, we have also defined our service interface. In fact, there are two forms for the service interface, one for guaranteed service and another for predicted service.

For guaranteed service, the interface is simple: the source only needs to specify the needed clock rate r^α, then the network guarantees this rate. The source uses its known value for $b^\alpha(r^\alpha)$ to compute its worst case queueing delay. If the delay is unsuitable, it must request a higher clock rate r^α. The network does no conformance check on any guaranteed service flow, because the flow does not make any traffic characterization commitment to the network.

For predicted service, the service interface must characterize both the traffic and the service. For the characterization of the traffic we have the source declare the parameters (r, b) of the token bucket traffic filter to which it claims its traffic will conform. Note that in the guaranteed case the client did not need to inform the network of its bucket size[8] b. Separately, the source must request the needed service. This involves selecting a suitable delay D and a target loss rate L the application can tolerate. The network will use these numbers to assign the source to an aggregate class at each switch for sharing purposes. Thus, for predicted service, the parameters of the service interface are the filter rate and size (r, b) and the delay and loss characteristics (D, L).

To provide predicted service, the network must also enforce the traffic commitments made by the clients. Enforcement is carried out as follows. Each predicted service flow is checked at the edge of the network (i.e., the first switch

[8] This assumes that the switches always have sufficient buffering; knowledge of the bucket size b is required for the switch to manage the buffers in cases of overflow.

the traffic passes through) for conformance to its declared token bucket filter; nonconforming packets are dropped or tagged. This conformance check provides the necessary isolation that is mandatory for entering a shared world. After that initial check, conformance is never enforced at later switches; this is because any later violation would be due to the scheduling policies and load dynamics of the network and not the generation behavior of the source.

In the case of the predicted service, specifying the token bucket traffic filter also permits the network to estimate if it can carry the new source at the requested rate and burstiness and still meet the service targets for this, and all of the existing, flows. This is the function of the last part of the architecture, the flow admission control computation.

9 Admission Control

While we stated earlier that we would not address the negotiation process for the establishment of service commitments, we must at least address the conditions under which a network accepts or denies a request for service, without necessarily specifying the exact dynamics of that exchange.

There are two criteria to apply when deciding whether or not to admit additional flows into the network. The first admission control criterion is that we should reserve no more than 90% of the bandwidth for real-time traffic, thereby letting the datagram traffic have access to at least 10% of the link; while the numerical value, 10%, of this quota is completely *ad hoc* and experience may suggest other values are more effective, we do believe that it is crucial to have such a quota. This quota ensures that the datagram service remains operational at all times; having the datagram traffic completely shut out for arbitrarily long periods of time will likely put impossible demands on the datagram transport layers. In addition, the datagram quota ensures that there is enough spare capacity to accommodate sizable fluctuations in the guaranteed and predicted service traffic. The second admission control criterion is that we want to ensure that the addition of a flow does not increase any of the predicted delays over the bounds D_i.

We now give an example, albeit superficial, of how one might make these criteria specific. Let $\hat{\nu}$ denote some measure of the utilization on a link due to real-time traffic (in general, the hat symbol denotes measured quantities), let $\hat{d}_i$ denote some measure of the delay of the traffic in class i, and let μ denote the link speed. In this example admission control criterion, a flow promising to conform to a token bucket traffic filter (r, b) can be admitted to priority level i if (1) $r + \hat{\nu} < .9\mu$, and (2) $b < (D_j - \hat{d}_j)(\mu - \hat{\nu} - r)$ for each class j which is lower than or equal in priority to level i. The first condition guarantees that there is at least 10% of the link left over for datagram traffic. The second condition is a heuristic designed to ensure that the delays will not violate the bounds D_j once the new flow is admitted even if the new flow displays worst-case behavior. The key to making the predictive service commitments reliable is to choose appropriately conservative measures for $\hat{\nu}$ and $\hat{d}_j$; these should not just be averages but consistently conservative estimates. Knowing how conservative to make the $\hat{\nu}$ and $\hat{d}_j$ may involve historical knowledge of the size of fluctuations in network traffic and delay on various links.

This example is overly sketchy, and we have yet to simulate to see how this particular implementation of admission control would function in a dynamic network. We offer it solely as an illustration of the considerations involved in designing an admission control policy. It is clear that the viability of our proposal rests on our ability to formulate an admission control policy which will make the predicted service class sufficiently reliable; specifying and validating such an admission control policy is the focus of our current work.

We have the following additional general comments on admission control policies. It is not clear how precise such an algorithm needs to be. If there is enough bandwidth to meet most customer needs, and if only a small fraction of traffic needs the most demanding of the predicted service, then a rough estimate may be adequate. In addition, we are offering a general method which involves *measuring* the behavior of the existing real-time traffic, rather than using the traffic model specified in the service interface, in deciding whether to admit new traffic. We use the worst-case traffic model only for the new source, which we cannot otherwise characterize; once the new flow starts running, we will be able to measure the aggregate traffic with the new flow and base further admission decisions on the most recent measurement. This approach is important for two reasons. First, since the sources will normally operate inside their limits, this will give a better characterization and better link utilization. Second, it matches what the clients themselves are doing, as they adapt the play-back point to the observed network traffic. Having the network and the endpoints assess the traffic in similar ways is likely to better produce reasonable behavior.

10 Other Service Qualities

There are a number of other service features that have been proposed in the context of real-time services. Here we wish to mention them, although we do not discuss exactly how to support them in the context of our scheme.

One goal is that if overload causes some of the packets from a source to miss their deadline, the source should be able to separate its packets into different classes, to control which packets get dropped. This idea can be incorporated into our scheme by creating several priority classes with the same target D_i. Packets tagged as "less important" go into the lower priority class, where they will arrive just behind the more important packets, but with higher priority than the classes with larger D_i. It is obvious that use of priority here can create a range of policies.

Another proposed service is that packets that are sufficiently late should be discarded internally, rather than being delivered, since in delivering them the network may use bandwidth that could have been better used to reduce the delay of subsequent packets. The offset carried in the packet in the FIFO+ scheme provides precisely the needed information: if a packet accumulates a very large jitter offset, it is a target for immediate discarding. This idea has been proposed elsewhere ([21]) but we observe that it fits naturally into the FIFO+ scheme.

A third service is that packets should be buffered in the network if they might otherwise arrive early (before the playback point) so that the end-node need not provide the buffering or estimate the current delay. We are not convinced that this service is useful in general. With current memory costs, buffering does not seem expensive. And while it might seem nice for the network to relieve the destination equipment from the need to estimate the delay, it cannot eliminate the need for the end to adapt to a change in the delay. The way in which the adaptation is done is application specific, and must drive the decision as to when to change the actual play-back point. Once we give the destination enough control to perform this act, it seems obvious that it is just as simple to have it perform the delay estimation as well.

11 Related Work

There has been a flurry of recent work on supporting real-time traffic in packet networks. We cannot hope to cover all of the relevant literature in this brief review; instead, we mention only a few representative references.

Though the WFQ scheduling algorithm was first described in Reference [4], Parekh and Gallager were the first to observe that, when the weights are chosen appropriately and the traffic sources conform to token bucket filters, the scheduling algorithm provides guaranteed service. WFQ is similar in spirit, though not in detail, to the Delay-EDD scheme proposed in Reference [7] and the MARS scheme proposed in References [12, 13], in that the use of a deadline for scheduling in Delay-EDD and MARS are analogous to the virtual departure time-stamps used in WFQ. However, the algorithms used to compute the time-stamps/deadlines are quite different in the three algorithms. Furthermore, the algorithms use rather different traffic filters to provide bounds. Delay-EDD uses peak-rate limits (and a condition on the average rate) whereas WFQ uses token buckets to provide guaranteed bounds. MARS has no explicit traffic filters and does not provide guaranteed bounds (i.e., no bounds that are independent of the other sources' behavior); rather, MARS has been shown through simulation with a particular set of statistical sources to obey certain *post facto* bounds.

WFQ, Delay-EDD, and MARS are work-conserving scheduling algorithms, in that the link is never left idle if there are packets in the queue. Several non-work-conserving scheduling algorithms have been proposed; for example, Stop-and-Go queueing ([8, 9]), Hierarchical Round Robin ([16]), and Jitter-EDD ([22]). All of these bear a superficial similarity to WFQ in that packets are scheduled according to some deadline or frame; the difference is that the packets are not allowed to leave early. These algorithms typically deliver higher average delays in return for lower jitter. See the review studies [24, 27] for a more detailed comparison of these schemes.

The Jitter-EDD ([6, 22]) algorithm make use of a delay field in the packet header to inform scheduling decisions, much like the FIFO+ algorithm. Also, we should note that the MARS scheduling algorithm uses FIFO scheduling within a class of aggregated traffic in a fashion very similar to our use of FIFO within each predicted service class. Furthermore, Reference [13] makes the same observation that deadline scheduling in a homogeneous class leads to FIFO. Reference [12] also observed that strict priority does not permit as many sources to share a link as a scheme that more actively manages jitter shifting. This work thus represents an example of queue management to increase link loading, as opposed to expanded service offerings.

The general architecture of most of the proposals in the literature, with Delay-EDD, Jitter-EDD, and HRR being examples, focus primarily on the delivery of what we have called guaranteed service to real-time traffic (with datagram traffic comprising the rest of the network load). Therefore these scheduling algorithms have been designed mainly to provide isolation between flows. MARS is an exception; MARS promotes sharing within a traffic class by using FIFO, and among different classes by a somewhat more complex scheme. Due to lack of isolation, however, MARS does not provide guaranteed service. The MARS algorithm, as well as the Statistical-EDD ([7]), attempt to achieve statistical bounds, but these bounds are still computed *a priori* (either through analytical approximation or through the simulation of a particular statistical source). There is implicit in these proposals the assumption that all real-time network clients are, in our taxonomy, intolerant and rigid. While the worst-case guaranteed bounds delivered by these mechanisms are appropriate for intolerant and rigid clients, we have argued that there will likely be many real-time clients who are both tolerant and adaptive.

There is only one other general architecture that has, as one of its goals, the delivery of service more appropriate for these tolerant and adaptive clients (and which we have called predicted service); this is an unpublished scheme due to Jacobson and Floyd which is currently being simulated and implemented. Their work shares with our predicted service mechanism the philosophy of measuring the current offered load and delivered service in order to decide if new service requests should be granted. Furthermore, their scheme also involves the use of priorities as a combine sharing/isolation mechanism. In contrast to our scheme, their scheme uses enforcement of traffic filters at every switch as an additional form of isolation, and they use round-robin instead of FIFO within a given priority level[9]. Moreover, there is no provision for guaranteed service in their mechanism.

References [10, 11] present admission control policies involving the concept of *equivalent capacity* and then discuss traffic filters (those references use the term access controls) related to those admission control policies. While much of the work is analytical, they also raise the possibility of using measurements of current network conditions to inform the various control policies.

12 Conclusion

This paper contains two contributions: an architecture and a mechanism. Our architecture is perhaps the more fundamental piece, in that it defines the problem and provides a framework for comparing various mechanistic alternatives. The main novelty of our architecture, which arose from the observation that many real-time applications can be made adaptive, is the explicit provision for two different kinds of service commitments. The guaranteed class of service is the traditional real-time service that is discussed in much of the literature. Guaranteed service is based on characterization of source behavior that then leads to static worst-case bounds. The predicted class of service, which is designed for adaptive and tolerant real-time clients, is less traditional and we believe that this paper represents the first exploration of the issues surrounding this type of service commitment. Predicted service replaces *a priori* traffic characterization with measurement in the network admission control algorithms. Applications using this service are likely to replace static play-back points based on *a priori* bounds with adaptive

[9]More specifically, they combine the traffic in each priority level into some number of *aggregate* groups, and do FIFO within each group (they use the term class, but in this paper we have used that term with a different meaning) and round-robin among the groups. The enforcement of traffic filters mentioned above is applied to each group.

play-back points based on delay measurements. We conjecture that with predictive service and adaptive clients we can achieve both higher link utilizations and superior application performance (because the play-back points will be at the *post facto* bounds, not the *a priori* worst-case bounds).

Our mechanism is both an existence proof that our architecture can be realized, and perhaps a useful artifact in its own right. The mechanism's scheduling algorithms are built around the recognition that the principles of *isolation* and *sharing* are distinct and both play important roles when sources are bursty and bandwidth is limited.

Isolation is fundamental and mandatory for any real-time traffic control algorithm. The network cannot make any commitments if it cannot prevent the unexpected behavior of one source from disrupting others. Sharing is important but not fundamental. If bandwidth were plentiful, effective behavior could be obtained by allocating to each source its peak rate; in this case sharing need not be considered. Note, however, that plentiful bandwidth does not eliminate the need for isolation, as we still need to ensure that each source does not use more than its allocated portion of the bandwidth. Thus, careful attention to sharing arises only when bandwidth is limited. In environments like LANs, it may be more cost-effective to over-provision than to implement intricate sharing algorithms. One should therefore embed sharing into the architecture only with caution.

We have proposed a particular scheme for sharing, which seems general enough that we propose that the control field (the jitter offset) be defined as part of the packet header. But we note that, if a subnetwork naturally produces very low jitters, it could just ignore the field and operate in some simple mode like FIFO. When a subnetwork has these very low natural jitters, it will not have enough queueing to remove most of the accumulated jitter anyway, and the error introduced by ignoring the field should be minor. Thus our sharing proposal is half architecture and half optional mechanism.

We conclude with one last observation: pricing must be a basic part of any complete ISPN architecture. If all services are free, there is no incentive to request less than the best service the network can provide, which will not produce effective utilization of the network's resources (see Reference [3] for a discussion of these issues). The sharing model in existing datagram networks deals with overload by giving everyone equally poor service; the equivalent in real-time services would be to refuse a high fraction of requests, which would be very unsatisfactory. Prices must be introduced so that some clients will request higher jitter service because of its lower cost. Therefore, real-time services must be deployed along with some means for accounting.

It is exactly this price discrimination that will make the predicted service class viable. Certainly predicted service is less reliable than guaranteed service and, in the absence of any other incentive, network clients would insist on guaranteed service and the network would operate at low levels of utilization and, presumably, high prices. However, if one can ensure that the reliability of predicted service is sufficiently high and the price sufficiently low, many network clients will prefer to use the predicted service. This will allow ISPN's to operate at a much higher level of utilization, which then allows the costs to be spread among a much larger user population.

13 Acknowledgments

This work is an attempt to clarify some ideas which were, and, to some extent, remain somewhat hazy and elusive. Any progress we have made is the result of our numerous discussions with colleagues who, through either vigorous disagreement or constructive agreement, have helped us arrive at this point. In particular, we would like to thank J. Davin, S. Deering, D. Estrin, S. Floyd, A. Heybey, V. Jacobson, A. Parekh, K. Sollins, and J. Wroclawski.

14 Appendix

In our simulations, we use a network simulator written by one of us (LZ) and used in a number of previous simulation studies ([3, 25, 27]). The sources of real-time traffic are two-state Markov processes. In each burst period, a geometrically distributed random number of packets are generated at some peak rate P; B is the average size of this burst. After the burst has been generated, the source remains idle for some exponentially distributed random time period; I denotes the average length of an idle period. The average rate of packet generation A is given by

$$A^{-1} = \frac{I}{B} + \frac{1}{P}$$

In all the simulations mentioned in this paper, we set $B = 5pkts$, $A = 85pkts/sec$, $P = 170pkts/sec$ and $I = 5/170sec$, so that the peak rate was double the average rate. Each traffic source was then subjected to an (r, b) token bucket filter with $r = 85pkts/sec$ and $b = 50pkts$, and any nonconforming packets were dropped at the source; in our simulations about 2% of the packets were dropped, so the true average rate was around $83.5pkts/sec$.

In the networks we simulate, each host is connected to the switch by an infinitely fast link. All inter-switch links have bandwidths of 1 Mbit/sec, all switches have buffers which can hold 200 packets, and all packets are 1000 bits. All the queueing delay measurements are shown in units of per packet transmission time (1msec) and all data is taken from simulations covering 10 minutes of simulated time.

For the data in Table 1, we simulated a single-link network; there were 10 flows sharing the link. The data in Tables 2 and 3 is based on the configuration in Figure 1 which has 5 switches, each attached to a host, and four inter-switch links. There are 22 flows, with each host being the source and/or receiver of several flows, and all of the network traffic travelling in the same direction. Each inter-switch link was shared by 10 flows. There were 12 flows of path length one, 4 flows of path length two, 4 flows of path length three, and 2 flows of path length four. For the data in Table 3, the flows were assigned to different service classes (as discussed in section 7) and two datagram TCP connections were added.

References

[1] R. Chipalkatti, J. Kurose, and D. Towsley. *Scheduling Policies for Real-Time and Non-Real-Time Traffic in a Statistical Multiplexer*, In **Proceedings of GlobeCom '89**, pp 774-783, 1989.

[2] S. Casner private communication, 1992.

[3] R. Cocchi, D. Estrin, S. Shenker, and L. Zhang. *A Study of Priority Pricing in Multiple Service Class Networks*, In **Proceedings of SIGCOMM '91**, pp 123-130, 1991.

[4] A. Demers, S. Keshav, and S. Shenker. *Analysis and Simulation of a Fair Queueing Algorithm*, In **Journal of Internetworking: Research and Experience**, 1, pp. 3-26, 1990. Also in Proc. ACM SIGCOMM '89, pp 3-12.

[5] J. DeTreville and D. Sincoskie. *A Distributed Experimental Communications System*, In **IEEE JSAC**, Vol. 1, No. 6, pp 1070-1075, December 1983.

[6] D. Ferrari. *Distributed Delay Jitter Control in Packet-Switching Internetworks*, preprint, 1991.

[7] D. Ferrari and D. Verma. *A Scheme for Real-Time Channel Establishment in Wide-Area Networks*, In **IEEE JSAC**, Vol. 8, No. 4, pp 368-379, April 1990.

[8] S. J. Golestani. *A Stop and Go Queueing Framework for Congestion Management*, In **Proceedings of SIGCOMM '90**, pp 8-18, 1990.

[9] S. J. Golestani. *Duration-Limited Statistical Multiplexing of Delay Sensitive Traffic in Packet Networks*, In **Proceedings of INFOCOM '91**, 1991.

[10] R. Guérin and L. Gün. *A Unified Approach to Bandwidth Allocation and Access Control in Fast Packet-Switched Networks*, To appear in **Proceedings of INFOCOM '92**.

[11] R. Guérin, H. Ahmadi, and M. Naghshineh. *Equivalent Capacity and Its Application to Bandwidth Allocation in High-Speed Networks*, In **IEEE JSAC**, Vol. 9, No. 9, pp 968-981, September 1991.

[12] J. Hyman and A. Lazar. *MARS: The Magnet II Real-Time Scheduling Algorithm*, In **Proceedings of SIGCOMM '91**, pp 285-293, 1991.

[13] J. Hyman, A. Lazar, and G. Pacifici. *Real-Time Scheduling with Quality of Service Constraints*, In **IEEE JSAC**, Vol. 9, No. 9, pp 1052-1063, September 1991.

[14] V. Jacobson and S. Floyd private communication, 1991.

[15] V. Jacobson private communication, 1991.

[16] C. Kalmanek, H. Kanakia, and S. Keshav. *Rate Controlled Servers for Very High-Speed Networks*, In **Proceedings of GlobeCom '90**, pp 300.3.1-300.3.9, 1990.

[17] C. Liu and J. Layland. *Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment*, In **Journal of ACM**, 20, pp. 46-61, 1973.

[18] S. Lu and P. R. Kumar. *Distributed Scheduling Based on Due Dates and Buffer Priorities*, In **IEEE Transactions on Automatic Control**, 36, pp 1406-1416, 1991.

[19] A. Parekh and R. Gallager. *A Generalized Processor Sharing Approach to Flow Control- The Single Node Case*, In **Technical Report LIDS-TR-2040**, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1991.

[20] A. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks*, In **Technical Report LIDS-TR-2089**, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, 1992.

[21] H. Schulzrinne, J. Kurose, and D. Towsley. *Congestion Control for Real-Time Traffic*, In **Proceedings of INFOCOM '90**.

[22] D. Verma, H. Zhang, and D. Ferrari. *Delay Jitter Control for Real-Time Communication in a Packet Switching Network*, In **Proceedings of TriCom '91**, pp 35-43, 1991.

[23] C. Weinstein and J. Forgie. *Experience with Speech Communication in Packet Networks*, In **IEEE JSAC**, Vol. 1, No. 6, pp 963-980, December 1983.

[24] H. Zhang and S. Keshav. *Comparison of Rate-Based Service Disciplines*, In **Proceedings of SIGCOMM '91**, pp 113-121, 1991.

[25] L. Zhang. *A New Architecture for Packet Switching Network Protocols*, In **Technical Report LCS-TR-455**, Laboratory for Computer Science, Massachusetts Institute of Technology, 1989.

[26] L. Zhang. *VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks*, In **ACM Transactions on Computer Systems**, Vol. 9, No. 2, pp 101-124, May 1991. Also in Proc. ACM SIGCOMM '90, pp 19-29.

[27] L. Zhang. *A Comparison of Traffic Control Algorithms for High-Speed Networks*, In **2nd Annual Workshop on Very High Speed Networks**, 1991.