Dynamic Window-Constrained Scheduling for Multimedia Applications *

Richard West and Karsten Schwan

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

Advances in network technologies have introduced opportunities for applications such as video conferencing, tele-medicine, and real-time multimedia applications. These applications require strict performance (or quality of service*) requirements on the information transferred across a network. This paper describes a new algorithm, called Dynamic Window-Constrained Scheduling (DWCS), designed to meet the service constraints on packets from multiple streams with different performance objectives.*

Using only two attributes, a deadline *and a* loss-tolerance *per packet stream, we show DWCS: (1) can limit the number of late packets over finite numbers of consecutive packets in loss-tolerant or delay-constrained, heterogeneous traffic streams, (2) does not require a-priori knowledge of the worst-case loading from multiple streams to establish the necessary bandwidth allocations to meet per-stream delay and loss-constraints, (3) can safely drop late packets in lossy streams without unnecessarily transmitting them, thereby avoiding unnecessary bandwidth consumption, and (4) can exhibit both fairness and unfairness properties when necessary. In fact, DWCS can perform fair-bandwidth allocation, static priority (SP) and earliest-deadline first (EDF) scheduling. We show the effectiveness of DWCS using a streaming video application, running over ATM.*

1 Introduction

Low latency, high bandwidth integrated services networks have introduced opportunities for new applications such as video conferencing, tele-medicine, and real-time multimedia applications. Many of these applications require strict performance (or *quality of service*) requirements on the information transferred across a network. Typically, these performance objectives are expressed as some function of throughput, delay, jitter and loss-rate[8]. With many multimedia applications, such as video-on-demand or streamed audio, it is important that information is received and processed at an almost constant rate, such as 30 frames per second for video information. However, some packets comprising a video frame or audio sample can be lost, resulting in little or no noticeable degradation in the quality of playback at the receiver. Similarly, a data source can lose a certain fraction of information during its transfer across a network as long as the receiver processes the received data to compensate for the loss. Consequently, loss-rate is an important performance measure for this category of applications. We define the term *loss-rate*[17] as the fraction of packets in a stream [1] either received later than allowed or not received at all at the destination.

One of the problems with using loss-rate as a performance metric is that it does not describe when losses are allowed to occur. For most loss-tolerant applications, there is usually a restriction on the number of *consecutive* packet losses that are acceptable. For example, losing a series of consecutive packets from an audio stream might result in the loss of a complete section of audio, rather than merely a reduction in the signal-to-noise ratio. A suitable performance measure in this case is a *windowed loss-rate*, i.e. loss-rate constrained over a finite range, or *window*, of consecutive packets. More precisely, an application might tolerate x packet losses for every y arrivals at the various service points across a network . Any service discipline attempting to meet these requirements must ensure that

*This work is supported in part by DARPA through the Honeywell Technology Center under contract numbers B09332478 and B09333218, and by the British Engineering and Physical Sciences Research Council with grant number 92600699.

[1] Corresponding to each stream is the notion of a 'flow', or logical connection.

the number of violations to the loss-tolerance specification is minimized (if not zero) across the whole stream.

In contrast to loss-constrained applications, computer data transferred between hosts using a file-transfer protocol cannot tolerate any loss at all. In this case, a more appropriate performance measure is mean delay, to ensure that the delay incurred by packets from this class of application is minimized.

In summary, integrated services networks must be able to support diverse performance objectives. Therefore, a suitable service discipline at the network access points must be able to schedule the transmission of packets from various streams so that the objectives of as many of the most important packets as possible are met.

This paper is concerned with meeting performance constraints on heterogeneous traffic streams originating from the same network node. To support per-connection (or stream) quality of service constraints, we propose a novel packet scheduling policy that resides at the base of an end-system quality of service (QoS) architecture. [2] The scheduler is responsible for multiplexing packets from multiple streams into the network, in an order compliant with each stream's quality of service (QoS) constraints.

We have implemented a novel packet scheduling algorithm, and a corresponding library system, on Solaris 2.5.1. The library system, called 'Dionisys', supports end-to-end quality of service guarantees[1, 10, 6], and can work with any one of a number of packet scheduling algorithms, including our algorithm, called Dynamic Window-Constrained Scheduling (DWCS). DWCS is targeted primarily at loss-tolerant networked applications, although it is equally useful for non-time-constrained applications. In fact, DWCS can also be used as a thread scheduling algorithm, to ensure that each thread receives x service quanta every y time quanta, while also supporting EDF and static priority threads.

DWCS is designed to maximize network bandwidth usage in the presence of multiple packets each with their own delay constraints and loss-tolerances. The per-packet delay and loss allowances must be provided as attributes after generating them from higher-level application constraints.

The algorithm requires two attributes per packet. At any time, a pair of attributes applies to all packets in the same stream. These attributes are described below:

- *Deadline* – this is the latest time a packet can *commence* service. The deadline is determined from a specification of the maximum allowable time between servicing consecutive packets in the same stream.
- *Loss-tolerance* – this is specified as a value x_i/y_i, where x_i is the number of packets that can be lost or transmitted late for every *window*, y_i, of consecutive packet arrivals in the same stream, i. For every y_i packet arrivals in stream i, a minimum of $y_i - x_i$ packets must be scheduled on time, while at most x_i packets can miss their deadlines and be either dropped or transmitted late, depending on whether or not the attribute-based QoS for the stream allows some packets to be lost.

Dionisys is used as a platform for testing various packet scheduling policies. Using Dionisys, we show (1) DWCS can limit the number of late packets over finite numbers of consecutive packets in loss-tolerant or delay-constrained, heterogeneous traffic streams, (2) DWCS does not require a-priori knowledge of the worst-case loading from multiple streams to establish the necessary bandwidth allocations to meet per-stream delay and loss-constraints, (3) DWCS can safely drop late packets in lossy streams, thereby avoiding unnecessary bandwidth consumption, and (4) DWCS can exhibit both fairness and unfairness properties when necessary. In fact, DWCS can perform fair-bandwidth allocation, static priority (SP) and earliest-deadline first (EDF) scheduling. We demonstrate, using a simple video-server application, the effectiveness of DWCS.

The following section describes related work, while Section 3 describes DWCS in detail. Section 4 presents an experimental evaluation of DWCS using a streaming video application where applicable. Finally, the conclusions and future work are discussed in Sections 5 and 6, respectively.

2 Related Work

Fair scheduling algorithms[7, 25, 9] (some of which are now implemented in hardware[18]) attempt to allocate $1/N$ of the available bandwidth among N streams or flows. Any idle time, due to one or more flows using less than its allocated bandwidth, is divided equally among the remaining flows. This concept generalizes to weighted fairness in which bandwidth must be allocated in proportion to the *weights* associated with individual flows.

Weighted Fair Queueing, WFQ, (and FQ) emulates a bit-by-bit round-robin service by scheduling packets in increasing order of their finish times. Unfortunately,

[2] This work is part of the Adaptive Systems project, underway at the Georgia Institute of Technology.

deriving the finish time of a packet involves an expensive computation to calculate a virtual time $v(t)$[26], which is the round number that would be in progress at time t if the packets were being serviced in a bit-by-bit weighted round-robin manner. WFQ approximates to Generalized Processor Sharing (GPS)[14], which cannot be implemented since it requires preemption of the network link on an arbitrarily small time scale. In fact, there can be large discrepancies between the service provided by WFQ and GPS[2].

Bennett and Zhang[3] improve upon WFQ, by developing a new algorithm called Worst-case Fair Weighted Fair Queueing (WF^2Q). In a WF^2Q system, the next packet chosen for service at time τ is the first queued packet that would complete service in the corresponding GPS system, *assuming the chosen packet would have already started, and possibly finished, receiving service in the GPS system at* τ.

Like WF^2Q, Start-Time Fair Queueing (SFQ)[15, 16] is a computationally efficient algorithm that allocates bandwidth fairly regardless of variation in server rate. It also supports hierarchical link-sharing for networks with multiple traffic types having heterogeneous performance objectives.

Start-Time Fair Queueing associates both a start and a finish tag with each packet, and schedules packets in increasing order of their start tags. Furthermore, SFQ outperforms WFQ in its ability to provide fairness among traffic flows using variable bit rate servers, as well as being computationally less expensive. In SFQ, the computation of virtual-time, $v(t)$, is inexpensive, since it only involves examining the start tag of the packet in service. For n flows at a server, SFQ has a complexity of $O(log(n))$ per packet. By contrast, WFQ requires computation of $v(t)$ by simulating a bit-by-bit round robin server in real-time. The computation of $v(t)$ for WFQ is expensive and requires the server capacity, B, to be constant. In a hierarchical link-sharing environment, where a server is shared by multiple traffic types of various priorities, the link can actually appear to have fluctuating service rate for low priority traffic. Hence, SFQ is valid for variable-rate servers while WFQ is only valid for fixed-rate servers, deeming WFQ inappropriate for hierarchical link-sharing.

Stoica et al[20] have recently proposed a proportional share algorithm (EEVDF) for real-time and time-shared systems. This algorithm is like fair scheduling algorithms for bandwidth allocation, but is targeted at CPU scheduling. It uses two virtual times (unlike SFQ, which uses just one), an eligible time and a deadline time. Service is granted to an eligible request with the earliest virtual deadline, where a request is eligible if it has received the same amount of service in a real system as it would in a fluid-flow system, like GPS. Although EEVDF uses two virtual times, it has the desirable property of keeping the lag, that is the difference between actual and optimal service in a proportional share system, at a constant.

DWCS solves two problems not addressed by fair-queueing algorithms. First, all fair-queueing algorithms use only *weights* to compute bandwidth-allocations for different streams. To accurately determine the correct ratios of weights to yield the necessary bandwidth allocations, the weights must be computed from a-priori knowledge of the maximum number of streams (or traffic classes) that can be active simultaneously. It is not always possible, or appropriate, to assume full a-priori and static knowledge about the maximum number of streams that must be serviced at the same network service-point. DWCS can provide the correct bandwidth allocations to streams, without prior knowledge of the maximum number of concurrent streams, as long as a suitable admission control policy establishes that sufficient resources are available when new streams require service. Second, real-time traffic requires more than just the bandwidth guarantees offered by fair-queueing algorithms, because packet deadlines can be violated if each packet in a stream is not serviced at the right time. Deadlines and loss-tolerances can be independently varied in DWCS, yielding a spectrum of queue-orderings from static-priority, to fair bandwidth allocation, to EDF. In contrast, SFQ always ensures fair bandwidth allocation, even when we always want to give precedence to one stream.

Hamdaoui and Ramanathan[11] have simulated an algorithm that services multiple streams, in an attempt to ensure at least m customers (packets or threads) in a stream (or process) meet their deadlines for every k consecutive customers from the same stream (or process). This algorithm is similar to DWCS but DWCS can also degrade to static priority (SP) and earliest-deadline first (EDF) scheduling. Furthermore, DWCS can support heterogeneous traffic (both deadline and non-deadline constrained) as well as a mixture of static priority traffic (e.g., for out-of-band data) and fair-bandwidth allocated traffic, that simultaneously compete for service. The characteristics of DWCS will be shown by experiments, which are described in Section 4.

Black[5] describes a scheduling approach in the Nemesis operating system that enables processes to receive a proportional (weighted-fair) share of CPU time. Similarly, Waldspurger and Weihl developed an algorithm called stride scheduling[22], which applies the properties of network-based fair queueing to processor scheduling. As stated above, DWCS differs in its ability to be unfair when necessary, thereby providing uninterrupted service to one traffic stream and, afterwards, weighted-fair service to other streams.

Peha and Tobagi[17] propose a *cost-based scheduling* algorithm (CBS) for packet-switched networks. Time-varying cost-functions are specified for each traffic class and the objective is to schedule packets from different classes to minimize total cost. Each packet's cost at any time can be calculated independently of any other queued packets, meaning the time complexity to order the packets at the head of n different streams is $O(n)$. Priorities $p_i(t)$ are assigned to packets each time the scheduler executes, at time t, and the highest priority packet is scheduled for transmission.

The problem with CBS is that the calculation of $p_i(t)$ is not trivial, except for simple cost-functions. Specifying service constraints using cost-functions can be complicated. By comparison, DWCS requires each packet to be specified simply by a (loss-tolerance, deadline) tuple. Furthermore, with DWCS, loss-tolerances and deadlines can be translated easily to weights, so that bandwidth can be allocated to streams based on the ratios of stream weights. This allows DWCS to act as a fair-scheduling algorithm, whereas it is not clear how to achieve this with CBS.

3 Dynamic Window-Constrained Scheduling

Dynamic Window-Constrained Scheduling (DWCS) orders packets for transmission based on the *current* values of their loss-tolerances and deadlines. Precedence is given to the packet at the head of the stream with the lowest loss-tolerance[3]. Packets in the same stream (for example, those packets forming part of one or more larger messages with the same QoS requirements) all have the same original and current loss-tolerances. Packets in the same stream are scheduled in their order of arrival. Whenever a packet misses its deadline, the loss-tolerance for all packets in the same stream, s, is adjusted to reflect the increased importance of transmitting a packet from s. This approach avoids starving the service granted to a given packet stream, and attempts to increase the importance of servicing any packet in a stream likely to violate its original loss constraints. Conversely, any packet serviced before its deadline causes the loss-tolerance of other packets (yet to be serviced) in the same stream to be increased, thereby reducing their priority.

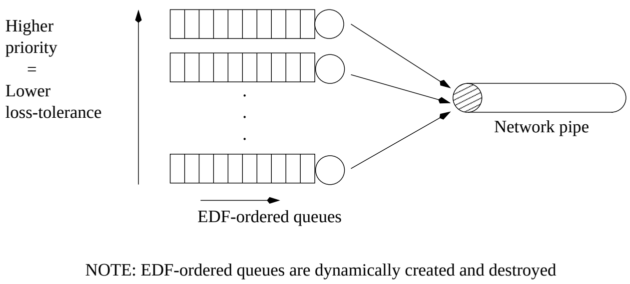

Figure 1: DWCS packet scheduling.

The DWCS algorithm actually works as follows: it places packets into logical priority queues, with packets in the same queue ordered earliest-deadline first (see Figure 1). Priorities are assigned to queues based on the current loss-tolerances of all buffered packets. Thus, all packets in the same queue have the same current loss-tolerance. Packets are scheduled for transmission from the head of the highest-priority queue, where the highest priority queue is the one holding the lowest loss-tolerant packets[4]. The number of logically-active queues depends on the number of distinct packet loss-tolerances among all buffered packets. This implies that new queues may be created dynamically and that existing queues are destroyed when they are empty. The loss-tolerance of a packet changes over time, depending on whether or not another (earlier) packet from the same stream has been scheduled for transmission by its deadline. If a packet cannot be scheduled by its deadline, it is either transmitted late (with adjusted loss-tolerance) or it is dropped and the deadline of the next packet in the stream is adjusted to compensate for the latest time it could be transmitted, assuming the dropped packet *was* transmitted as late as possible.

Table 1 shows the rules for ordering pairs of packets in different streams. Recall that all packets in the same stream are queued in their order of arrival. If two packets have the same non-zero loss-tolerance, they are ordered earliest-deadline first (EDF) in the same queue. If two packets have the same non-zero loss-tolerance and deadline they are ordered lowest loss-numerator x_i first, where x_i/y_i is the current loss-tolerance for all packets in stream i. By ordering on the lowest loss-numerator, precedence is given to the packet in the

[3] The exact rules for ordering packets for transmission will be described later.

[4] There can be more than one logical priority queue with a loss-tolerance of zero, depending on the value of the loss-denominator for packets in the corresponding queue. This will become apparent in the ensuing text.

Pairwise Packet Ordering
Lowest loss-tolerance first
Same non-zero loss-tolerance, order EDF
Same non-zero loss-tolerance & deadlines, order lowest loss-numerator first
Zero loss-tolerance & denominators, order EDF
Zero loss-tolerance, order highest loss-denominator first
All other cases: first-come-first-serve

Table 1: Precedence amongst pairs of packets

stream with *tighter* loss constraints, since fewer consecutive packet losses can be tolerated. If two packets have zero loss-tolerance and their loss-denominators are both zero, they are ordered EDF, otherwise they are ordered highest loss-denominator first. In such a circumstance, it is possible that a stream may lose more packets than its loss-tolerance specification allows. By increasing the denominator in this case, the algorithm attempts to favor the adversely affected packet stream, bringing the amortized loss for packets in that stream back to the original loss-tolerant value. If it is paramount that a stream never loses more packets than its loss-tolerance permits, then admission control must be added to the network manager, to avoid accepting connections whose QoS constraints cannot be met due to existing connections' service constraints.

Every time a packet in stream i is transmitted, the loss-tolerance of i is adjusted. Likewise, other streams' loss-tolerances are adjusted *only if* any of the packets in those streams miss their deadlines as a result of queueing delay. Consequently, DWCS requires worst-case $O(n)$ time to select the next packet for service from those packets at the head of n distinct streams. However, the average case performance can be far better, because not all streams always need to have their loss-tolerances adjusted after a packet transmission.

An efficient implementation of DWCS is as follows: packet streams with the same inter-packet deadlines are grouped into the same class. Assuming all deadlines are multiples of t time units apart, the scheduler only needs to check whether to adjust loss-tolerances every t time units for all streams in the class with packet deadlines t time units apart. Similarly, the scheduler only needs to check whether to adjust loss-tolerances half as frequently, once every $2t$ time units, for all streams in the class with packet deadlines $2t$ time units apart, and so on. To select the next packet for service, the scheduler can simply scan a linear-list of the head packets in each stream.

For streams that can lose packets, any packets in these streams that have missed their deadlines are simply discarded. For a stream that cannot lose packets, the deadline serves to minimize queueing delay before eventual transmission of all packets in that stream. The loss-tolerance value for such streams serves to avoid transmitting too many late packets.

DWCS can also support streams comprising packets without deadlines. In this case, the original loss-tolerances act as static priorities, whereby the current loss-tolerance for a stream is always the same as it original value. The previous table (Table 1) of pairwise packet-orderings still applies to pairs of packets without deadlines, except such packets can be considered to have the same deadlines (which can be considered to be infinite).

We now describe how loss-tolerances are adjusted. Let x_i/y_i denote the original loss-tolerance for all packets in stream i. Let x'_i/y'_i denote the current loss-tolerance for all queued packets in stream i. Let x'_i denote the current loss-numerator, while x_i is the original loss-numerator for packets in stream i. y'_i and y_i denote current and original loss-denominators, respectively. Before a packet stream is serviced, its current and original loss-tolerances are equal. For all buffered packets in the same stream i as the packet most recently transmitted before its deadline, adjust the loss numerators and denominators as follows:

if $(y'_i > x'_i)$ then $y'_i = y'_i - 1$;
if $(x'_i = y'_i = 0)$ then $x'_i = x_i; y'_i = y_i$;

For all buffered packets in the same stream i as the packet most recently transmitted, where packets in i do not have deadlines, do not adjust their loss-tolerances. That is: $y'_i = y_i; x'_i = x_i$.

For all buffered packets, if any packet in stream $j | j \neq i$ misses its deadline:

if $(x'_j > 0)$ then
    $x'_j = x'_j - 1; y'_j = y'_j - 1$;
    if $(x'_j = y'_j = 0)$ then $x'_j = x_j; y'_j = y_j$;
if $(x'_j = 0)$ then
    if $(x_j > 0)$ then $y'_j = y'_j + \lceil \frac{y_j - x_j}{x_j} \rceil$;
    if $(x_j = 0)$ then $y'_j = y'_j + y_j$;

As an example, consider $n = 3$ streams of packets, s_1, s_2 and s_3 (see Figure 2). Let the original loss-tolerances of each stream be 1/2, 3/4 and 6/8, respectively. Let the deadlines of the first packets in each stream be 0 and let each successive packet p in stream i, have a deadline one time unit later than predecessor $(p-1)$ in the same stream i. That is, $deadline_{1_i} = 0$ and $deadline_{p_i} = deadline_{(p-1)_i} + 1$, $\forall i, 1 \leq i \leq n, p \in Z^{+}$[5]. Assume that the service time of each packet is one time unit. If each stream has a packet arrive for service once every time unit, the total load on the scheduler is 3.0 from all three streams. However, due to the loss-tolerances of each stream, the minimum demand from all streams is $\sum_{i=1}^{n} \frac{(1-l_i)C_i}{T_i}$, where l_i is the loss-tolerance, C_i is the service time (or transmission delay) of each and every packet in stream i, and T_i is the inter-arrival time for packets in stream i. For this example, with the three streams having the above loss-tolerances, the effective scheduler load can be as low as 1.0 if we carefully discard (or service late) appropriate late packets from each stream. Thus, it may still be possible to service all three streams while meeting the appropriate losses from each stream.

From Figure 2, the first packet to be scheduled in this case will be from s_1, because s_1 has the lowest loss-tolerance. Since the serviced packet does not miss its deadline, the new (current) loss-tolerance of s_1 will be set to 1/1. As a result, we can still allow the loss of the next packet in s_1 and not violate the original loss tolerance. Hence, the rationale for adjusting the loss-tolerance in this way. At time $t = 1$, the first packet in s_1 has been serviced but the first packets in s_2 and s_3 have each missed their deadlines. As a result, the first packet in each of these streams is dropped and the new loss-tolerances for s_2 and s_3 are set to 2/3 and 5/7, respectively. This change in loss-tolerance compensates for one less allowable packet loss over a range of one fewer packets than in the original loss-tolerance specification. At time $t = 1$, the packet at the head of s_2 with $deadline = 1$ has the highest priority, so it is serviced next. This causes the packets with $deadline = 1$ from s_1 and s_3 to miss their deadlines. Observe that s_1's loss-tolerance is set back to its original value at this point, because it was temporarily set to 0/0, which is meaningless. Notice that at time $t = 5$, a packet in s_2 gets serviced. When two packets have the same non-zero loss-tolerance and deadline, and their loss-numerators are the same, ties can be broken arbitrarily. This example shows a packet from the lowest-numbered stream being serviced first. At time $t = 8$, the schedule repeats itself. Observe that over the first eight packets serviced, s_1 transmits four packets and loses four, consuming 50% of the bandwidth, and both s_2 and s_3 transmit two packets and lose six, each consuming 25% of the bandwidth. Furthermore, one packet from s_1 is serviced every two time units (or packet service times), one packet from s_2 is serviced every four time units, and two packets from s_3 are serviced every eight time units. Hence, the loss-tolerances from all three streams are met.

[5] Z^{+} is the set of positive integers.

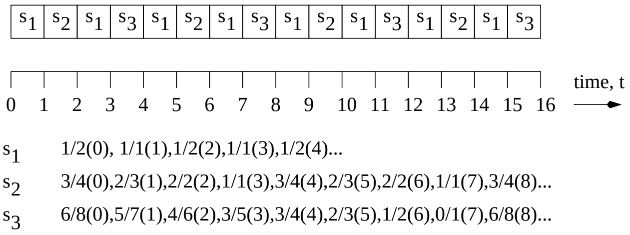

Figure 2: Example DWCS scheduling of 3 streams, s_1, s_2 and s_3. Each packet has a deadline 1 time unit later than its predecessor in the same stream. Deadlines are shown in brackets and loss-tolerances are shown as x/y. All packets take 1 time unit to be serviced.

DWCS is a flexible scheduling algorithm, having the ability to model a number of real-time and non-real-time policies. Moreover, DWCS can act as a fair queueing, static priority and earliest-deadline first scheduling algorithm, as well as provide service for a mix of static and dynamic priority traffic streams. We now describe these features of DWCS in more detail.

Earliest-Deadline First Scheduling using DWCS

When all packets in each stream have zero loss-tolerances (specifically, each stream has an original loss-tolerance of 0/0), DWCS degrades to EDF. Intuitively, this makes sense, since all streams have the same importance so their corresponding packets are serviced based upon the time remaining to their deadlines. It can be shown that if all deadlines can be met, EDF guarantees to meet all deadlines[4]. If packets are dropped after missing their deadlines, EDF is optimal with respect to loss-rate in discrete-time G/D/1 and continuous-time M/D/1 queues[13].

Static Priority Scheduling using DWCS

If no packets in any streams have deadlines (ie., they effectively have infinite deadlines), DWCS degrades to static priority (SP). Static-priority scheduling is optimal for a weighted mean delay objective, where weighted mean delay is a linear combination of the delays experienced by all packets[12]. In DWCS, the current loss-tolerances associated with each packet in every stream are always equal to their original loss-tolerances, and each packet's loss-tolerance serves as its static priority. Namely, for packets with infinite deadlines, the term "loss-tolerance" is really a misnomer, since no packets are actually lost. As expected, precedence is given to the packet with the lowest "loss-tolerance" (i.e., highest priority). For packets with infinite deadlines, DWCS has the ability to service non-time-constrained packets in static priority order to minimize weighted mean delay.

Fair Scheduling using DWCS

Fair Queueing derivatives such as SFQ have the ability to share bandwidth among n message streams such that each stream receives a weighted fair-share of available bandwidth. Specifically, let w_i be the weight of message stream i and $B_i(t_1, t_2)$ be the aggregate service (in bits) of i in the interval $[t_1, t_2]$. If we consider two message streams, i and j, the normalized service (by weight) received by each stream will be $\frac{B_i(t_1,t_2)}{w_i}$ and $\frac{B_j(t_1,t_2)}{w_j}$, respectively. The aim is to ensure that $|\frac{B_i(t_1,t_2)}{w_i} - \frac{B_j(t_1,t_2)}{w_j}|$ is as close to zero as possible, considering that packets are indivisible entities and an integer number of packets might not be serviced during the interval $[t_1, t_2]$.

DWCS also has the ability to meet weighted fair allocation of bandwidth. Figure 2 shows how three streams are allocated bandwidth in accordance with their loss-tolerances. Figure 3 shows an example of bandwidth allocation among two streams, s_1 and s_2, comprising packets of different lengths. In the latter example, s_1 and s_2 each require 50% of the available bandwidth. The service times for each and every packet in streams s_1 and s_2 are 5 time units and 3 time units, respectively.

Given stream weights, w_i, in a fair bandwidth-allocating algorithm, like SFQ, we can calculate the loss-tolerances and deadlines that must be assigned to streams in DWCS to give the equivalent bandwidth allocations. This is done as follows:

1. Determine the minimum time window, Δ_{min}, over which bandwidth is shared proportionally among n streams, each with weight $w_i | 1 \leq i \leq n, w_i \in Z^+$:

   First, let C_i be the service time of each packet in stream i. (This assumes all packets in any one stream are the same length). Let $\omega = \sum_{i=1}^{n} w_i$ and let η_i be the number of packets from stream i serviced in some arbitrary time window Δ. (Note that $\eta_i C_i$ is the total service time of stream i over the interval Δ, and $\sum_{i=1}^{n} \eta_i C_i = \Delta$. Furthermore, Δ is assumed sufficiently large to ensure bandwidth allocations amongst all n streams in *exact* proportions to their weights). This implies that $\frac{\eta_i C_i}{\Delta} = \frac{w_i}{\omega}$.

   If w_i is a factor of ωC_i, let $\gamma_i = \frac{\omega C_i}{w_i}$, else let $\gamma_i = \omega C_i$.

   Then $\Delta_{min} = lcm(\gamma_1, ..., \gamma_n)$, where $lcm(a, b)$ is the lowest-common-multiple of a and b.

2. For DWCS, set $deadline_{1_i} = 0$, and $deadline_{p_i} = deadline_{(p-1)_i} + C_i$, for each packet p_i in stream i, where $p \in Z^+$.

3. To calculate the loss-tolerance, l_i, of packets in stream i, let $l_i = \frac{x_i}{y_i}$, where:

   $y_i = \frac{\Delta_{min}}{C_i}$, and $x_i = \frac{\Delta_{min}}{C_i} - \eta_i'$,
   and $\eta_i' = \frac{\eta_i \Delta_{min}}{\Delta} = \frac{w_i \Delta_{min}}{\omega C_i}$.

If deadlines are assigned as in step 2, we can translate packet loss-tolerances back into stream *weights*, w_i, as follows:

$w_i = \frac{y_n(y_i - x_i)}{y_i(y_n - x_n)}$, where $0 < \frac{x_i}{y_i} < 1$.

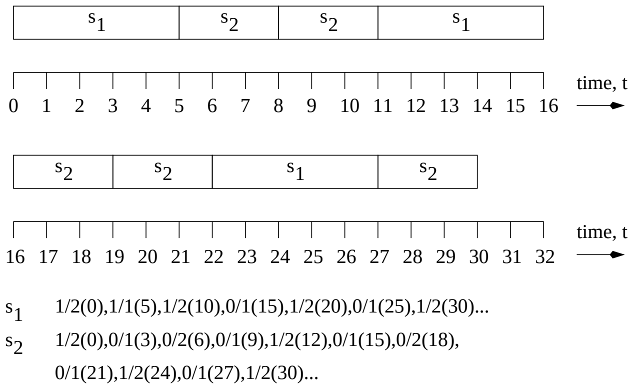

Figure 3: Example DWCS scheduling of 2 streams, s_1, s_2, so that each stream receives a 50% share of bandwidth when both streams are active. Packets in s_1 take 5 time units to be serviced, while those in s_2 take 3 time units. Note that the fine-grained loss-constraints of each stream are no longer met but each stream gets 50% of the bandwidth every 30 time units.

DWCS Support for Heterogeneous Traffic Streams

DWCS can service multiple traffic streams each having their own loss-tolerances and packet deadlines. It is possible to vary loss-tolerances and deadlines independently, to achieve a range of service properties. Each traffic stream can be characterized by its own loss-tolerance and packet deadlines, independent of other traffic streams. With DWCS, it is possible to dynamically and independently vary loss-tolerances and deadlines for each and every stream.

DWCS can support a mix of static and dynamic priority traffic streams. Some streams may have packets with infinite deadlines[6], in which case (as stated earlier) the loss-tolerance serves as a static priority. However, consider the problem of servicing two streams, s_1 and s_2, under the following conditions. Let s_1 have an original loss-tolerance of 0/0 and packet deadlines 1 time unit after their arrival times. Let s_2 have a 1/4 loss-tolerance (acting as a static priority), and each packet in the stream has an infinite deadline. Furthermore, assume that the time to service a packet from either stream is one time unit. The problem with this example is that, whenever s_1 has a packet that requires service, it will always be serviced as soon as possible, even though there is time to service 1 packet from s_2 followed by a packet, p, from s_1, so that p in s_1 still meets its deadline.

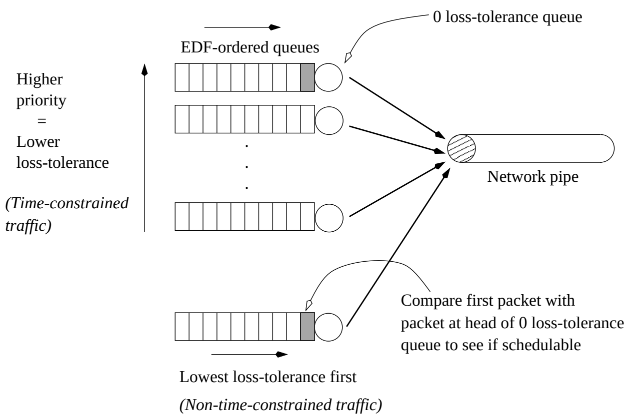

Figure 4: DWCS with separate logical queueing of non-time-constrained traffic.

An adjustment to the algorithm (see Figure 4) in this case is to maintain a separate logical queue, q_{ntc}, for all non-time-constrained packets (with ∞ deadlines) and order them lowest loss-tolerance first. Any two packets in different streams having the same loss-tolerances are ordered so that the shortest packet is scheduled first. All other packets are placed in queues with other packets that have the same loss-tolerances and are ordered EDF as in the original algorithm. If there is a non-empty 0/0 loss-tolerance queue for time-constrained traffic, q_{tc}, the packet at the head of this queue is compared against the packet at the head of q_{ntc}. If q_{ntc} is not empty and the first packet can be serviced before the first packet in q_{tc} will miss its deadline, it is serviced in an attempt to keep delay of non-time-constrained packets to a minimum.

[6] Streams with infinite deadlines are non-time-constrained.

4 Experimental Evaluation

DWCS and other scheduling algorithms are evaluated in the context of the Dionisys end-system[23]. All experiments were performed on a cluster of SparcStation Ultra II Model 2148s. The scheduler runs as a periodic, real-time daemon process (that must be executed by the superuser). At initialization time, the scheduler sets up a 16MB shared memory segment[7] for the scheduler queue and corresponding data structures. Figure 5 shows a typical scenario in which the packet schedulers are evaluated. A video-server forks one child (application) process for every client connection. The child processes then stream packets of MPEG-1 video frames across an ATM network to clients, $c_i|1 \leq i \leq n$, on remote hosts. Each video-server child process places video frames in a shared memory region, ready for packetization and scheduling by the Dionisys system. Each client has the ability to supply service attributes that can be translated into suitable values for the scheduler[8]. At any point in time there are between one and n active streams $s_i|1 \leq i \leq n$. These streams are repeated after exponentially-distributed idle periods. The mean idle period is varied to control the average demand on the scheduler.

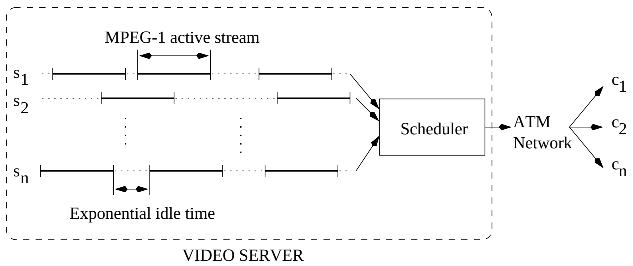

Figure 5: A video-server, servicing up to n streams concurrently. Packets from each stream are fed to a scheduler which multiplexes them over an ATM link to appropriate clients, c_1 to c_n.

[7] The size of the shared memory segment is configurable via the Dionisys library. We chose 16MB to hold enough backlogged, packetized MPEG-1 frames to observe the scheduler's performance under load.

[8] Since an MPEG-1 stream consists of I, P and B-frames, an application might divide the stream into 3 sub-streams and allocate separate delay and loss constraints to each sub-stream. For simplicity, in our experiments, we do not divide streams based on their frame-type, thereby treating all frame-types equally.

4.1 Fair-Bandwidth Allocation (Link Sharing)

DWCS has the ability to mimic a range of well-known scheduling policies. From Section 3, it should be clear that DWCS can model EDF and SP scheduling. However, DWCS can also achieve fair-bandwidth allocation.

The first experiment compares DWCS to SFQ in its ability to achieve fair-bandwidth allocation amongst n streams in the shortest time-frame possible. Weights are assigned to the streams in SFQ, and the corresponding loss-tolerances and deadlines, for DWCS, are computed using the method in Section 3.

Four streams requiring service, each carry MPEG-1 data, comprising bursts of 150 frames with an average arrival rate of 30 frames per second, followed by idle periods with a mean inter-burst gap of 1 second. The burst periods average 5 seconds and these bursts are repeated 10 times per stream, for a total of 1500 frame (or packet) arrivals per stream. All packets are eventually transmitted, even if they are late. The scheduler interval is $40mS$, so 25 frames can be serviced in 1 second[9]. This scenario overloads the scheduler and forces a build-up of back-logged arrivals.

Figure 6(a) shows the bandwidths (bit service rates) of the four streams over a 50 second period, for SFQ and DWCS. The weights for streams $s_1, ..., s_4$ are 1, 1, 2 and 4, respectively. The corresponding loss-tolerances are 7/8, 14/16, 6/8 and 4/8. In the steady-state, SFQ and DWCS behave almost identically, each servicing s_1 and s_2 at about 110 Kbps, s_3 at about 220 Kbps, and s_4 at about 440 Kbps. Figure 6(b) shows the scenario for two streams. Observe that after all 1500 packets of s_2 are serviced, the service rate of s_1 rises. In this case, the scheduler is run once every $50mS$ to cause sufficient back-logging.

The number of deadlines missed and the mean packet delay are approximately the same for DWCS and SFQ. Figure 7(a) shows the number of deadlines missed for the two-stream example above. Note that in transmitting all 1500 packets, some packets can miss their deadlines more than once. Figure 7(b) shows mean delay increasing as a function of the packets sent due to overload on the schedulers. The delay curve for s_1 increases at a decreasing rate since it becomes the only active stream after all 1500 packets from s_2 have been serviced. The large delays in Figure 7(b) are due to significant back-logging of the scheduler.

By minimizing the window of time over which bandwidth is allocated in proportion to weights of streams in SFQ, SFQ actually manages to meet the loss-tolerances and deadlines of packets almost as successfully as DWCS. However, DWCS explicitly uses deadlines and loss-tolerances for packet streams. Hence, bandwidth can be allocated to streams to meet these constraints, independent of the constraints on other streams, assuming enough bandwidth is available, which is the case when $\sum_{i=1}^{n} \frac{(1-l_i)C_i}{T_i} \leq 1.0$. Note that in the experiments above, at no time did DWCS violate the loss-tolerances on any of its streams even though the scheduler was overloaded and had to service some packets late.

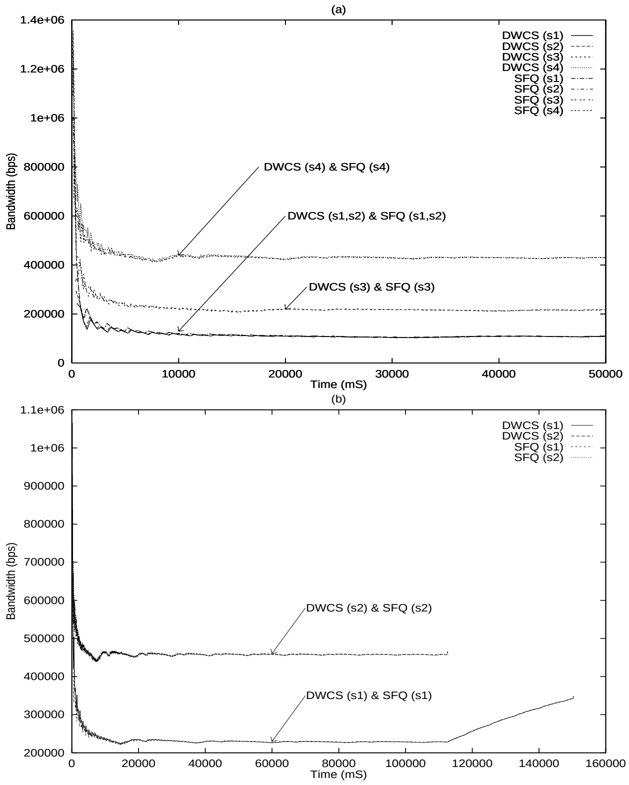

Figure 6: (a) Bandwidth allocations for 4 streams, $s_1, ..., s_4$, with weights, 1, 1, 2 and 4, respectively. The corresponding loss-tolerances are 7/8, 14/16, 6/8 and 4/8. (b) Bandwidths of two streams, s_1 and s_2, with weights 1 and 2, and loss-tolerances 2/3 and 1/3, respectively.

4.2 Bandwidth Allocation with Out-of-Band Traffic

DWCS has the ability to support static priority traffic in the presence of fair-bandwidth allocated traffic. Consider the scenario where stream s_1 requires twice

[9] For these experiments, the packet service times are assumed to be equal to the scheduler period, since the scheduler services at most one packetized frame each time it executes.

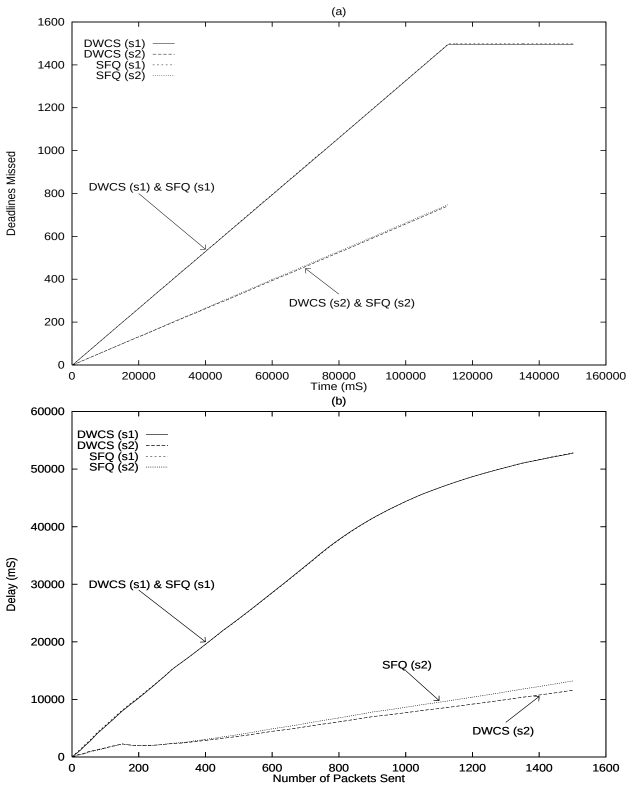

Figure 7: (a) Deadlines missed over time from two streams, s_1 and s_2, with weights 1 and 2, and loss-tolerances 2/3 and 1/3, respectively. (b) Mean packet delay as a function of the packets sent for the same two streams.

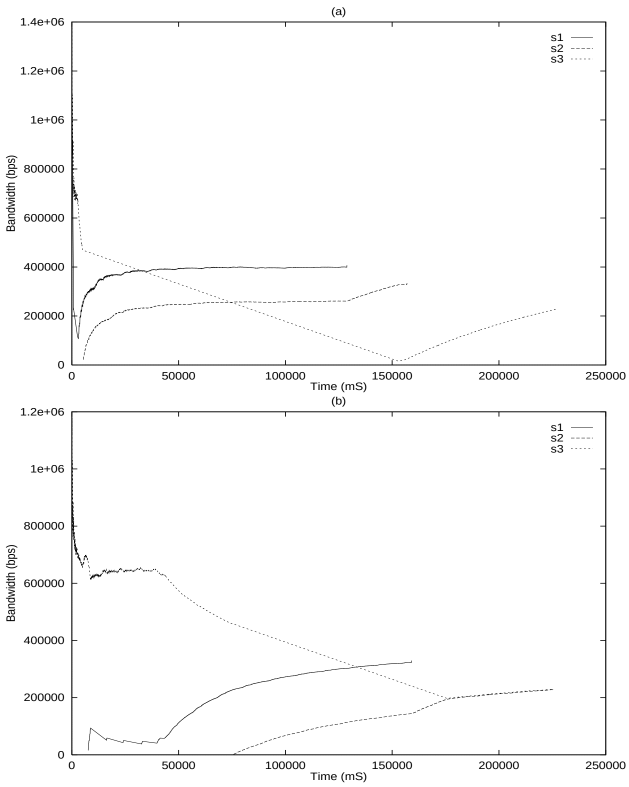

Figure 8: Bandwidth versus time for two dynamic priority streams, s_1 and s_2, and a static priority stream, s_3. (a) Loss-tolerances for s_1, s_2 and s_3 are 1/3, 2/3 and 0/100, respectively. (b) Loss-tolerances for s_1, s_2 and s_3 are 1/3, 2/3 and 0/1500, respectively.

as much bandwidth as s_2 when both streams are active. However, suppose stream s_3 is carrying time-critical (out-of-band) traffic that must be delivered to its destination with the shortest possible delay. In this case, we always want to grant service to s_3 when s_3 has packets for service, but when s_3 is not active, the bandwidth must be shared between s_1 and s_2.

Fair-scheduling algorithms cannot handle the above scenario but DWCS can, because for the duration of a burst of packets from s_3, that burst must be serviced exclusively. This violates the fairness properties for which SFQ has been designed. In contrast, with DWCS, by carefully choosing a loss-tolerance that reflects the highest priority for s_3, and by setting the packet deadlines to infinity, DWCS gives exclusive service to s_3 when it is active. However, as s_1 and s_2 are starved of service, their loss-tolerances *dynamically* decrease, thereby raising their priorities until it is possible that they have precedence over s_3. The loss-tolerance of s_3 traffic must be set so that the maximum burst size from s_3 does not require a longer uninterrupted service duration than the time to raise the priority of either s_1 or s_2 above that of s_3.

Figure 8(a) shows the results of servicing 3 streams, in which the first two, s_1 and s_2, are assigned dynamic priorities (ie., loss-tolerances of 1/3 and 2/3, respectively) that reflect their bandwidth shares, while s_3 is assigned a static priority of 0/100. Initially, s_3 receives a greater service rate than either s_1 or s_2, but as s_1 and s_2 are neglected, their loss-tolerances decrease.

If the loss-tolerance for s_3 is reduced even further, by increasing its denominator[10], larger durations of service time are granted to consecutive packet arrivals from s_3 (as shown in Figure 8(b) where the loss-tolerance of s_3 is 0/1500). Hence, we can fine-tune stream attributes, so that the loss-tolerance of out-of-band data reflects the time to service the largest traffic burst without interruption. Observe that in Figures 8(a) and (b), when the bandwidth curve for s_3 decreases, it is actually not being serviced. Meanwhile,

[10] Observe that $0/y_1$ is higher priority than $0/y_2$ if $y_1 > y_2$. This is one of the precedence rules described in Table 1.

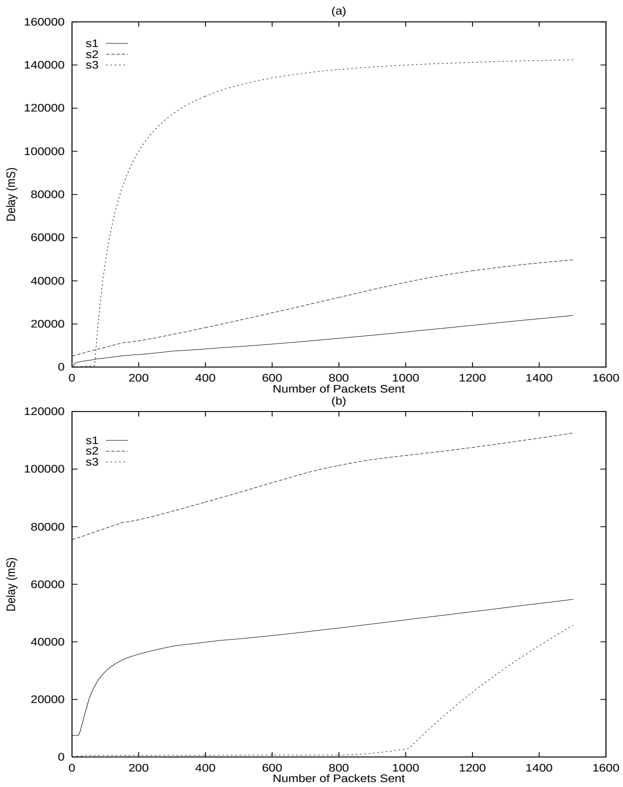

Figure 9: Mean packet delay as a function of the packets sent for three streams. (a) Loss-tolerances for s_1, s_2 and s_3 are 1/3, 2/3 and 0/100, respectively. (b) Loss-tolerances for s_1, s_2 and s_3 are 1/3, 2/3 and 0/1500, respectively.

s_1 and s_2 approach their steady-states, close to $2:1$ bandwidth shares.

Figures 9(a) and (b) show the mean packet delay as a function of the packets sent for the three streams with parameters described above (and a scheduler period of $50mS$). In Figure 9(a), about 67 packets are serviced from s_3 before s_3's delay increases. This is when the loss-tolerance is 0/100. In Figure 9(b), about 1000 packets from s_3 are serviced before a significant delay increase. This is when s_3's loss-tolerance is 0/1500. Observe that s_3's delay rises when it no longer has precedence over s_1 and s_2.

4.3 Effective Bandwidth Usage

With DWCS, packets are each stamped with deadlines, which means that late packets in lossy traffic can be discarded before transmission, thereby reducing the bandwidth wasted on useless information. Without using per-packet deadline and loss-tolerance information, fair scheduling policies can only assume that all packets must be transmitted. Therefore, some packets that are late will actually be transmitted when they are useless. Hence, we can measure the effective bandwidth usage, ξ_i per stream i, which is formulated as: $\xi_i = (\frac{sent\_ok_i(t).C_i}{t}).B$, where $sent\_ok_i(t)$ is the number of packets sent from stream i before their deadlines in the time interval t, C_i is the service time of each packet in stream i, and B is the total link capacity. However, since algorithms that do not exploit packet deadlines must ultimately transmit all packets, the ratio $\rho_i = \frac{sent\_ok_i(t)}{sent\_total_i(t)}$ may be less than 1.0, where $sent\_total_i(t)$ is the total number of packets sent from stream i in the interval t. This means, $1 - \rho_i$ is the fraction of bandwidth consumed by packets that are late and, hence, useless to the receiver. For DWCS, ρ_i can be as high as 1.0, because it exploits knowledge of packet deadlines and can drop late packets instead of transmitting them if the application permits such loss. With SFQ, bandwidth can be wasted due to the transmission of deadline-constrained packets that are late and, hence, no longer valid. This can be seen from Figure 7(a), which is an overload scenario causing many packets to miss deadlines.

5 Conclusions

DWCS has the ability to limit the number of late packets over finite numbers of consecutive packets in loss-tolerant or delay-constrained, heterogeneous traffic streams. DWCS can support a combination of static priority and dynamic priority (bandwidth-allocated) traffic. In fact, DWCS can perform fair-bandwidth allocation, static priority (SP) and earliest-deadline first (EDF) scheduling. Unlike fair-scheduling algorithms, DWCS can be unfair when necessary, as well as showing all the fairness characteristics of fair-schedulers such as SFQ.

DWCS does not require a-priori knowledge of the worst-case loading from multiple streams to establish the necessary bandwidth allocations to meet per-stream delay and loss-constraints. Since deadlines and loss-tolerances explicitly capture the service requirements of traffic streams, DWCS has the ability to manage dynamic traffic arrivals, where the maximum number of traffic streams is not known a-priori.

Finally, DWCS can safely drop late packets in loss-tolerant streams without unnecessarily transmitting them, thereby avoiding unnecessary bandwidth consumption. Hence, the effective bandwidth usage by DWCS is higher than with algorithms that do not exploit delay and loss information at the packet level.

6 Future Work

There are several issues still to be addressed in our on-going research involving DWCS. We have shown DWCS can service packets in an end-system but we believe DWCS can also support CPU scheduling. As part of our ongoing work, we shall investigate the ability of DWCS to support combined thread and packet scheduling.

In the context of Dionisys, the scheduler currently runs as a periodic, real-time daemon process on Solaris 2.5.1. We are building a system, in which the combined thread and packet schedulers can run as kernel-loadable modules in both Solaris and Linux. We have a demonstration video-server application running on our system, available for public use. We shall extend this system to support other types of multimedia applications, including a novel video game application that requires temporal and spatial consistency of graphical objects. The video game and corresponding Semantic-Distributed Shared Object (S-DSO) library[24] is also available for downloading.

While DWCS has worst-case time complexity linear in the number of streams, in practice its performance can be far better, since not all streams always need to have their loss-tolerances adjusted after a packet transmission. We shall investigate the average-case behavior of DWCS and look at ways to improve worst-case performance at the cost of scheduling accuracy. For example, the loss-tolerances of packets in streams that have missed their deadlines could be adjusted periodically, where a larger period causes the tolerances to change less frequently, thereby reducing the scheduler's ability to meet packet-level service constraints.

New algorithms, such as Hierarchical Fair Service Curve (H-FSC) scheduling[21], aim to support both link-sharing and guaranteed real-time services with decoupled delay and bandwidth allocation. H-FSC dynamically assigns priorities to packets based on their service requirements. We shall compare and contrast DWCS with H-FSC in future work.

Finally, as part of ongoing research in the area of adaptive systems[19] at the Georgia Institute of Technology, we shall investigate when and how DWCS can adapt to changes in service demands from application-level service requests.

References

[1] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of qos architectures. *Multimedia Systems Journal, Special Issue on QoS Architecture, (to appear)*, 1997.

[2] Jon C.R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. In *ACM SIGCOMM'96*, pages 143–156. ACM, August 1996.

[3] Jon C.R. Bennett and Hui Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *IEEE INFOCOMM'96*, pages 120–128. IEEE, March 1996.

[4] Partha P. Bhattacharya and Anthony Ephremides. Optimal scheduling with strict deadlines. *IEEE Transactions on Automatic Control*, 34(7):721–728, July 1989.

[5] Richard J. Black. *Explicit Network Scheduling.* PhD thesis, University of Cambridge, December 1994.

[6] Andrew Campbell. A quality of service architecture. In *ACM SIGCOMM Computer Communication Review.* ACM, April 1994.

[7] Alan Demers, Srinivasan Keshav, and Scott Schenker. Analysis and simulation of a fair-queueing algorithm. *Journal of Internetworking Research and Experience*, pages 3–26, October 1990.

[8] Domenico Ferrari. Client requirements for real-time communication services. *IEEE Communications Magazine*, 28(11):76–90, November 1990.

[9] S.J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOMM'94*, pages 636–646. IEEE, April 1994.

[10] G. Gopalakrishna and G. Parulkar. Efficient quality of service in multimedia computer operating systems. Technical Report WUCS-TM-94-04, Department of Computer Science, Washington University, August 1994.

[11] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, April 1995.

[12] J. Michael Harrison. Dynamic scheduling of a multiclass queue: Discount optimality. *Operations Research*, 23(2):370–382, March-April 1975.

[13] Shivenra S. Panwar, Don Towsley, and Jack K. Wolf. Optimal scheduling policies for a class of queues with customer deadlines to the beginning of service. *Journal of the ACM*, 35(4):832–844, October 1988.

[14] A. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks.* PhD thesis, Massachusetts Institute of Technology, February 1992.

[15] Harrick M. Vin Pawan Goyal and Haichen Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet scwitching networks. In *IEEE SIGCOMM'96*. IEEE, 1996.

[16] Xingang Guo Pawan Goyal and Harrick M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *2nd Symposium on Operating Systems Design and Implementation*, pages 107–121. USENIX, 1996.

[17] Jon M. Peha and Fouad A. Tobagi. A cost-based scheduling algorithm to support integrated services. In *IEEE INFOCOMM'91*, pages 741–753. IEEE, 1991.

[18] Jennifer L. Rexford, Albert G. Greenberg, and Flavio G. Bonomi. Hardware-efficient fair queueing architectures for high-speed networks. In *INFOCOMM'96*, pages 638–646. IEEE, March 1996.

[19] Daniela Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. *18th IEEE Real-Time Systems Symposium*, Dec., 1997.

[20] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Real-Time Systems Symposium*. IEEE, December 1996.

[21] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *SIGCOMM'97*, pages 249–262. ACM, October 1997.

[22] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource mangement. Technical Report MIT/LCS/TM-528, MIT, June 1995.

[23] Richard West and Karsten Schwan. Interactors: Capturing tradeoffs in bandwidth versus cpu usage for quality of service constrained objects. Technical Report GIT-CC-98-03, Georgia Institute of Technology, January 1998.

[24] Richard West, Karsten Schwan, Ivan Tacic, and Mustaque Ahamad. Exploiting temporal and spatial constraints on distributed shared objects. In *Proc. 17th IEEE ICDCS, To appear*. IEEE, 1997.

[25] Hui Zhang and Srinivasav Keshav. Comparison of rate-based service disciplines. In *Proceeedings of ACM SIGCOMM*, pages 113–121. ACM, August 1991.

[26] Lixia Zhang. *A New Architecture for Packet Switched Network Protocols.* PhD thesis, Massachusetts Institute of Technology, July 1989.