# Scalable Scheduling Support for Loss and Delay Constrained Media Streams *

Richard West, Karsten Schwan and Christian Poellabauer

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

## Abstract

*Real-time media servers need to service hundreds and, possibly, thousands of clients, each with their own quality of service (QoS) requirements. To guarantee such diverse QoS requires fast and efficient scheduling support at the server. This paper describes the practical issues concerned with the implementation of a scalable real-time packet scheduler resident on a server, designed to meet service constraints on information transferred across a network to many clients. Specifically, we describe the implementation issues and performance achieved by Dynamic Window-Constrained Scheduling (DWCS), which is designed to meet the delay and loss constraints on packets from multiple streams with different performance objectives. In fact, DWCS is designed to limit the number of late packets over finite numbers of consecutive packets in loss-tolerant and/or delay-constrained, heterogeneous traffic streams. We show how DWCS can be efficiently implemented to provide service guarantees to hundreds of streams. We compare the costs of different implementations, including an approximation algorithm, which trades service quality for speed of execution.*

## 1. Introduction

**Background.** Real-time media servers need to service hundreds and, possibly, thousands of clients, each with their own quality of service (QoS) requirements. Many such clients can tolerate the loss of a certain fraction of the information requested from the server, resulting in little or no noticeable degradation in the client's perceived quality of service when the information is received and processed. Consequently, loss-rate is an important performance measure for the service quality to many clients of real-time media servers. We define the term *loss-rate*[10, 5] as the fraction of packets in a stream either discarded or serviced later than their delay constraints allow. However, from a client's point of view, loss-rate could be the fraction of packets either received late or not received at all.

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

Some clients cannot tolerate any loss of information received from a server, but such clients often require delay bounds on the information. Consequently, these types of clients require deadlines which specify the maximum amount of time packets of information from the server can be delayed until they become invalid.

To guarantee such diverse QoS requires fast and efficient scheduling support at the server. This paper describes the practical issues concerned with the implementation of a scalable real-time packet scheduler resident on a server, designed to meet service constraints

*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.

on information transferred across a network to many clients. Specifically, we describe the implementation issues and performance achieved by Dynamic Window-Constrained Scheduling (DWCS), which is designed to meet the delay and loss constraints on packets from multiple streams with different performance objectives. In fact, DWCS is designed to limit the number of late packets over finite numbers of consecutive packets in loss-tolerant and/or delay-constrained, heterogeneous traffic streams. We show how DWCS can be efficiently implemented to provide service guarantees to hundreds of streams.

**The DWCS Scheduler.** 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, as follows:

- *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, $i$. Hence, for every $y_i$ packet arrivals in stream $i$, a minimum of $y_i - x_i$ packets must be scheduled for service by their deadlines.

At any time, all packets in the same stream have the same loss-tolerance, while each successive packet in a stream has a deadline that is offset by a fixed amount from its predecessor. Using these attributes, 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.

The DWCS algorithm is described in detail in an accompanying paper[16]. We will only mention the necessary details in this paper, so that the reader understands the implementation issues on which this paper focuses. DWCS has been implemented as part of the Dionisys QoS infrastructure, designed to support end-to-end quality of service guarantees[1] on information to many clients. Both Dionisys and DWCS are implemented on the Solaris 2.5.1 operating system.

The following section describes related work, while Section 3 describes the DWCS algorithm. Implementation issues are then described in Section 4. Section 5 presents an experimental evaluation of DWCS, showing the effects of different implementations and the cost of approximating DWCS, which trades quality of service for scalability. Finally, the conclusions and future work are discussed in Section 6.

## 2. Related Work

Recent research has put substantial effort into the development of efficient scheduling algorithms for media applications. Compared to such work, given the presence of some underlying bandwidth reservation scheme, the DWCS algorithm has the ability to share bandwidth among competing clients in strict proportion to their deadlines and loss-tolerances. This is similar to (weighted) fair scheduling[4, 18, 6, 2, 9, 14], which attempts to allocate bandwidth in proportion to stream *weights*. Similar proportional share algorithms have been targeted at CPU scheduling[13, 15]. However, the idea of 'windowing' in DWCS is closer to the work of Hamdaoui and Ramanathan[7] who 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). In comparison, DWCS can also perform static priority and earliest-deadline first scheduling, supporting both deadline and non-deadline constrained traffic. Furthermore, DWCS can meet explicit delay and 'windowed' loss constraints, using only two attributes that enable a diverse range of service specifications.

Almost all fair scheduling algorithms use a single *weight* for each stream, to compute weighted-fair bandwidth-allocations. DWCS complements this work by not only being able to perform fair scheduling but also being able to meet explicit delay and 'windowed' loss constraints. Toward this end, DWCS uses two attributes per stream, which enables a diverse range of service specifications. Unlike the above work on scheduling, this paper is focused on the practical issues of implementing a specific algorithm (in our case, DWCS) to meet the service constraints of large numbers of clients.

There has also been a significant amount of research on the construction of scalable media servers[3]. For

example, recent work by Jones et al. concerns the construction and evaluation of a reservation-based CPU scheduler for media applications[8] such as the audio/video player used in their experiments. These results demonstrate the importance of explicit scheduling when attempting to meet the demands of media applications.

We now describe DWCS in more detail.

## 3. Dynamic Window-Constrained Scheduling

**Algorithm Outline.** 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. Packets in the same stream all have the same original and current loss-tolerances, and 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.

The loss-tolerance of a packet (and, hence, the corresponding stream) 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 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. If it is paramount that a stream never loses more packets than its loss-tolerance permits, then admission control must be used, to avoid accepting connections whose QoS constraints cannot be met due to existing connections' service constraints.

| **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**

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.

**Loss-Tolerance Adjustment.** 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:

**(A) Loss-tolerance adjustment for a stream whose head packet is serviced before its deadline:**

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

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: $y'_i = y_i; x'_i = x_i$.

**(B) Loss-tolerance adjustment for a stream whose head packet misses its deadline:** 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 $x'_j = x_j; y'_j = y_j;$
else if ($x'_j = 0$) then (see (C) below)
$x'_j = 2x_j - 1; y'_j = 2y_j + (y'_j - 1);$ (method 1)
or
$x'_j = x_j; y'_j = y_j;$ (method 2)

The pseudo-code for DWCS is as follows:

```
while (TRUE) {
  /* Using precedence rules in Table 1          */
  find stream i with highest priority;
  service packet at head of stream i;
  adjust loss-tolerance of stream i according to
    rules in (A);
  /* Set deadline of new head packet in stream i */
  /* by adding an offset, interpacket_gap(i),    */
  /* to the old deadline.                        */
  deadline(i) = deadline(i) + interpacket_gap(i);
  for (each stream j missing its deadline) {
    while (deadline missed) {
      adjust loss-tolerance for j according to
        rules in (B);
      if (packet is droppable) {
        drop head packet in stream j;
        /* Make next packet in stream j the      */
        /* head packet.                          */
      }
      deadline(j) = deadline(j) +
        interpacket_gap(j);
    }
  }
}
```

**(C) Loss-tolerance adjustment for a stream that violates its original loss-tolerance:** The problem with adjusting loss-tolerances as shown above is what to do if a packet misses its deadline when its current loss-tolerance is $0/y'$. In this situation, DWCS can be configured to favor the adversely affected packet stream, bringing the amortized loss for packets in that stream back to the original loss-tolerant value, over a larger window of packets. To achieve the original loss-tolerance over, say, $3y$ packets, the current loss-tolerance of $0/y'$ is changed to $x'/y'$, where $x' = 2x - 1$ and $y' = 2y + y' - 1$. A more simple approach, which is used throughout this paper, is to reset the loss-tolerances of streams that violate their original loss-tolerances. Further details about loss-tolerance adjustments for streams that violate their original loss-tolerances is beyond the scope of this paper, but can be found in an accompanying paper[17]. We now describe the implementation issues regarding DWCS.

## 4. DWCS Implementation Issues

The efficient implementation of the DWCS packet scheduler for scalable media servers successfully copes with several important issues any such scheduler must address. These issues concern: (1) *concurrency*, (2) efficient *packet schedule representation*, and (3) *scalability*, which are now described in more detail.

**Concurrency.** Given that multiple threads are involved in generating media streams, packetizing them, and submitting packets, it is critical to avoid unnecessary synchronization overheads between packet generators, schedulers, and dispatchers. In our implementation of DWCS, we combine packet scheduling and dispatching in a single *scheduler* thread that selects the next packet for service, while each stream also has its own *server* thread that queues a packet to be scheduled. Synchronization between server and scheduler threads is necessary because without it, the scheduler can have an inconsistent view of the number of queued packets for each stream. The current DWCS implementation eliminates such synchronization by using a circular queue for each of the $n$ streams, at the cost of at most one wasted slot in each queue. Specifically, let $fp_i$ be the (front) pointer to the head packet in stream $i$, and $bp_i$ be the (back) pointer to the next free slot in the queue for stream $i$. The server (arrival) process performs the following pseudo-code to enter a new packet into the queue for stream $i$:

```
while (fp == bp->next); /* full queue */
add new packet to queue position
  pointed to by bp;
bp = bp->next;
```

The scheduler performs the following when attempting to remove a packet from a queue:

```
if (fp == bp); /* empty queue */
else {
  remove head packet from queue position
    pointed to by fp;
  fp = fp->next;
}
```

It is important to note that the entries in the circular queues are simply pointers to the packets themselves; the packet bodies are buffered in a shared memory area accessed by the DWCS scheduler.

**Packet Schedule Representation**. To attain high performance, DWCS does not separate the data structures used for packet scheduling from those used for packet dispatching, in contrast to what is often done for multi- or uni-processor CPU task schedulers[12]. Moreover, rather than evaluating the large number of packets associated with streams, DWCS manages the streams with which these packets are associated. There are two heap data structures used by DWCS for stream management, as shown in Figure 1(a).

The first heap concerns streams' timing attributes; it orders streams in earliest deadline first order. Using this heap, DWCS can efficiently determine which

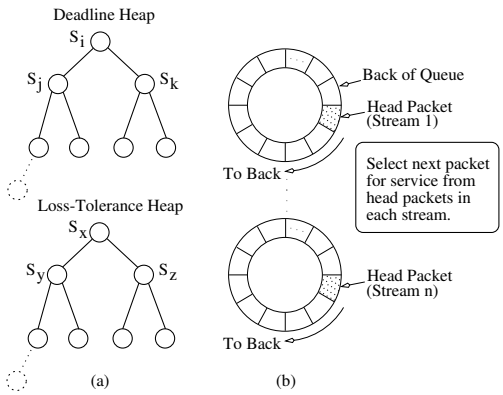


**Figure 1. (a) A fast implementation of DWCS uses two heaps: one for deadlines and another for loss-tolerances. (b) Using a circular queue for each stream eliminates the need for synchronization between the scheduler and the server queueing packets.**

streams need to have their packet loss-tolerances adjusted when the packets at the head of one or more streams miss their deadlines. In fact, the scheduler need only check those streams with packet deadlines less than the current time. As soon as a stream is reached that has an earliest packet deadline greater than the current time, the scheduler can stop checking any more streams. All those streams with missed packet deadlines have their identifiers removed from the loss-tolerance heap, the corresponding stream loss-tolerances are adjusted, and the stream identifiers are then reinserted back into the loss-tolerance heap. New deadlines are also calculated for the head packets in these streams, and the deadline heap is modified accordingly. DWCS now services the head packet of the stream at the top of the loss-tolerance heap. Streams with equal packet loss-tolerances are ordered according to the rules in Table 1.

**Scalability by Use of Flexibility in Scheduling.** Packet streams will differ in terms of the scheduling rates and latencies they require, ranging from high rate scheduling for small packets containing audio samples to scheduling larger packets at somewhat lower rates for raw or compressed video streams. Clearly, scheduling at high rates can result in significant scheduling overheads. We address this issue by associating with each DWCS scheduler a policy that governs its rate of execution relative to the packet rates of the streams it services or even relative to the fidelity of scheduling currently being experienced. For instance, this policy may choose to run an approximation of DWCS when scheduling latency must be improved, by reducing the frequency with which DWCS checks streams for missed deadlines and therefore, also reducing the quality of stream scheduling.

Ordinarily, DWCS checks streams for packets that have missed their deadlines every time a packet has been serviced. If a stream has one or more late packets, the stream's corresponding packet loss-tolerance must be adjusted. In the worst case, every stream can have late packets every time the scheduler completes the service of one packet, requiring $O(n)$ time to find the next packet for service from $n$ streams. If the scheduler only checks missed deadlines and, hence, adjusts loss-tolerances at most once after $p$ packets have been serviced, the execution time of the algorithm can be reduced.

The next section describes the overheads and performance of different implementations of DWCS.

## 5. Experimental Evaluation

**DWCS Scalability.** All experiments were performed on SparcStation Ultra II Model 2148s machines, running at 170MHz.

In the first experiment, we ran the scheduler with increasing numbers of streams and measured both the number of deadlines missed and the number of loss-tolerance violations. A stream's original loss-tolerance, $x/y$, is violated when more than $x$ packet deadlines have been missed for every $y$ consecutive packets in the same stream. Figure 2(a) shows the number of deadlines that are missed as the number of streams is increased from 80 to 760. There are eight traffic classes, with equal numbers of streams in each class. The classes have loss-tolerances that range from 1/80 to 1/150 (so that all streams in the same class have the same loss-tolerance) and the packet deadlines for consecutive packets in each stream are all 500 time units apart. The traffic model is such that there are always backlogged packets in each stream, and the scheduler takes one time unit to service a packet, with at most one packet serviced when the scheduler executes. The results show the effects of servicing a total of 5 million packets.

Note that in Figure 2(a), a packet can miss its deadline more than once. In fact, the y-axis really shows the number of times each packet in a traffic class misses its deadline (in this case, the number of times a packet is delayed by more than 500 time units). This is actually a measure of the relative *lateness* of packets in each traffic class. In any case, Figure 2(a) shows that for less than 500 streams, each traffic class has fewer than (a cumulative total of) 5000 missed deadlines, and the

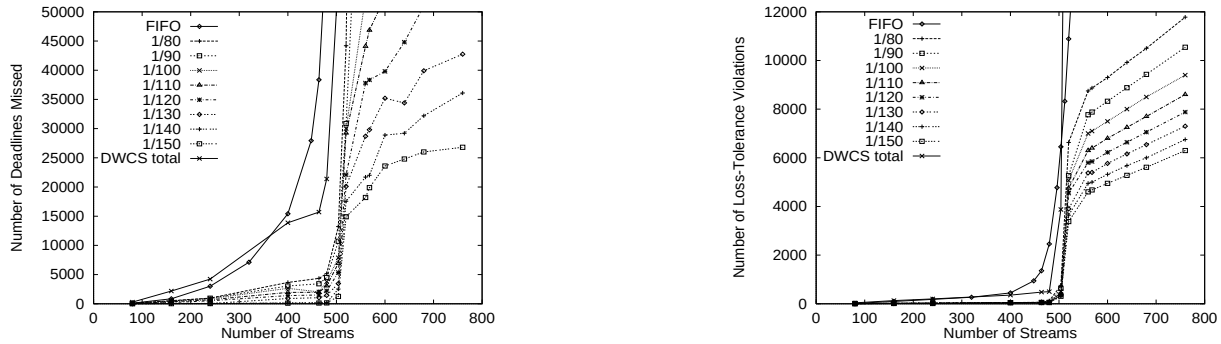


**Figure 2. (a) The number of deadlines missed, and (b) the number of loss-tolerance violations, versus number of streams.**

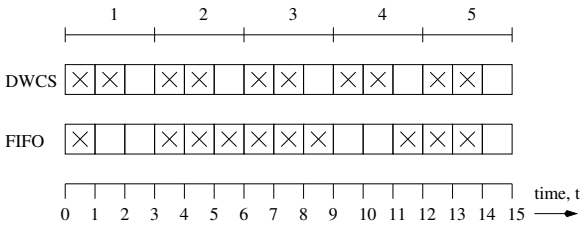


**Figure 3. An example showing possible loss-tolerance violations for a single stream using FIFO and DWCS in overload conditions, when the original loss-tolerance is 1/3. (Note that `X' denotes a late packet.)**

actual number of missed deadlines is proportional to the loss-tolerance of the corresponding class. When the number of streams rises above 500, there is no possible way to meet all packet deadlines. For example, suppose there are 560 streams and each packet in a stream has a deadline 500 time units later than its predecessor; if the scheduler services each stream equally, then after 500 time units 60 streams will have packets that have missed their deadlines. Figure 2(a) shows that the number of missed deadlines rapidly increases above 500 streams but streams in each class still miss deadlines in proportion to their loss-tolerances. Hence, no matter what the load, the number of deadlines missed is proportional to the loss-tolerances of the corresponding classes.

Observe that when the number of streams is less than 350, the total number of missed deadlines across all 8 traffic classes is greater with DWCS than FIFO. FIFO is servicing each class equally, irrespective of the loss-tolerances and deadlines. It so happens, in these experiments, similar sized bursts of packets in each stream arrive for service at the same time. The average burst size is 20 packets and bursts from each stream (and each class) are arriving close together. The average arrival time between bursts is 100 microseconds in these experiments. If the burstiness was more pronounced (with larger average burst sizes and larger variations in the time between bursts), the performance of FIFO would be far worse, since FIFO would service whole bursts of arrivals from one stream without respecting the deadlines of packets in other streams. In contrast, the performance of DWCS is less sensitive to burstiness. Moreover, DWCS maintains service to streams in proportion to their delay and loss requirements.

Figure 2(b) shows the number of loss-tolerance violations for increasing numbers of streams. Again, packet deadlines, for consecutive packets in the same stream, are 500 time units apart. Few violations occur using DWCS until the number of streams reaches 500. Above 500 streams, it is impossible for any scheduler to meet all deadlines (assuming the service time per packet is one time unit and only one packet can be serviced at a time), so loss-tolerance violations start to occur. However, DWCS still manages to control the numbers of violations in proportion to the loss-tolerances of each class. Observe that the total number of violations for all 8 classes is greater with DWCS than FIFO when we reach about 500 streams. Figure 3 shows an example of what is happening. DWCS is servicing streams in proportion to their loss-tolerances but, for every window of $y$ packets in a stream, the loss-tolerance is being violated more times than with FIFO. However, DWCS spreads out where the missed deadlines occur, so it still minimizes the number of consecutive late packets for a given finite window of packets that require service. FIFO, however, can result in many consecutive deadlines being missed. This is bad for media streams, such

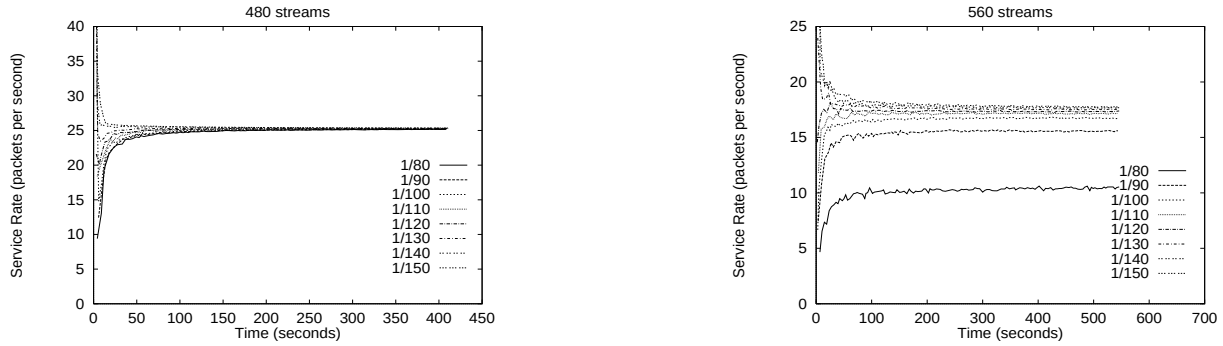


**Figure 4. Number of packets serviced per second for each stream when (a) there are 480 streams, and (b) 560 streams.**

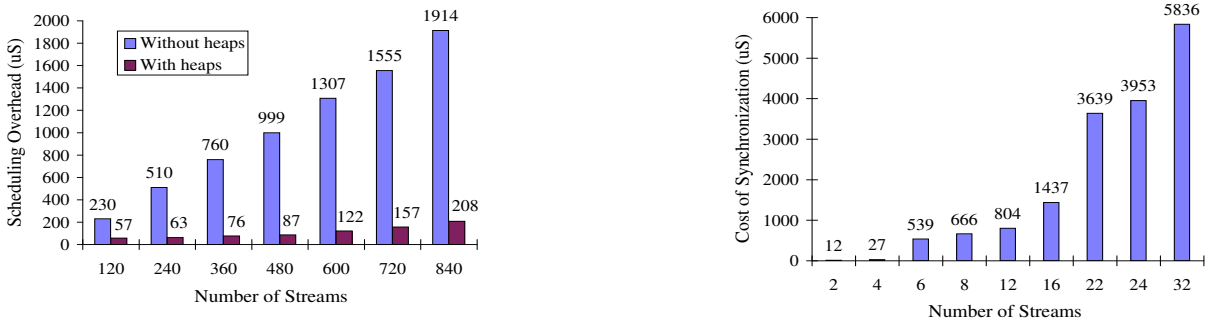


**Figure 5. (a) Service overheads and (b) synchronization costs of DWCS for increasing numbers of streams.**

as audio and video, where too many consecutive late packets result in loss of complete sequences of audio samples or video frames, rather than just a drop in signal-to-noise ratio.

To see how fast we could execute DWCS on a host processor, we measured the numbers of packets serviced in real-time for increasing numbers of streams with the same traffic classes (having the same loss-tolerances and deadlines) as before. The DWCS scheduler was implemented in the Dionisys QoS infrastructure, on Solaris 2.5.1, using two heaps, and circular queues for each traffic stream to eliminate the need for synchronization primitives, such as semaphores or locks. The results show the raw scheduler speed discounting the effects of transmission delay. In our experiments, all packets are ultimately serviced, but dropping late packets avoids unnecessary transmission delays in a real system.

Figure 4 shows the numbers of packets serviced per second when there are (a) 480 streams, and (b) 560 streams. Interestingly, when there are 480 streams, the service rate approaches 25 packets per second for each and every stream and, hence, each and every class. Initially, service is granted to streams in each class in proportion to the corresponding loss-tolerances. However, as time goes on, there is more slack time to service packets before they miss their deadlines. Eventually, all the streams reach a state in which their loss-tolerances converge and the average packet service rate is the same for all streams.

In Figure 4(b), there are 560 streams. There is no possible way the scheduler can meet all deadlines. Since deadline-misses keep affecting the current loss-tolerances, in this example, the loss-tolerances of streams in each class never converge to the same value. Consequently, the service granted to streams in each class is in direct proportion to their original loss-tolerances.

Figure 5(a) shows the actual scheduling overheads for increasing numbers of streams. The performance of DWCS actually depends upon the time between the deadlines of consecutive packets in a given stream. This is because loss-tolerances will be adjusted more frequently if deadlines are missed more frequently. The

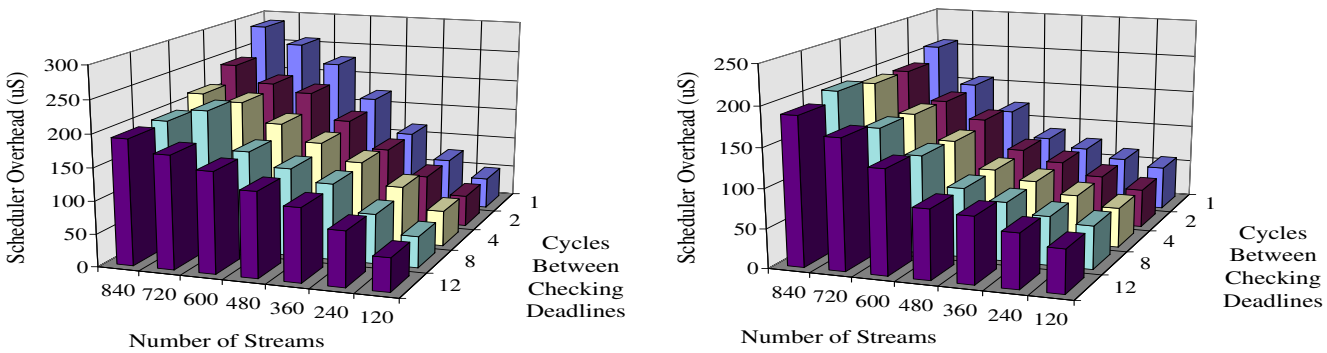


**Figure 6. Real-time scheduling overheads (in microseconds) as the number of scheduler cycles between checking deadlines is increased. All packet deadlines are 200 time units apart in the left-hand graph (a), and 500 time units apart in the right-hand graph (b).**

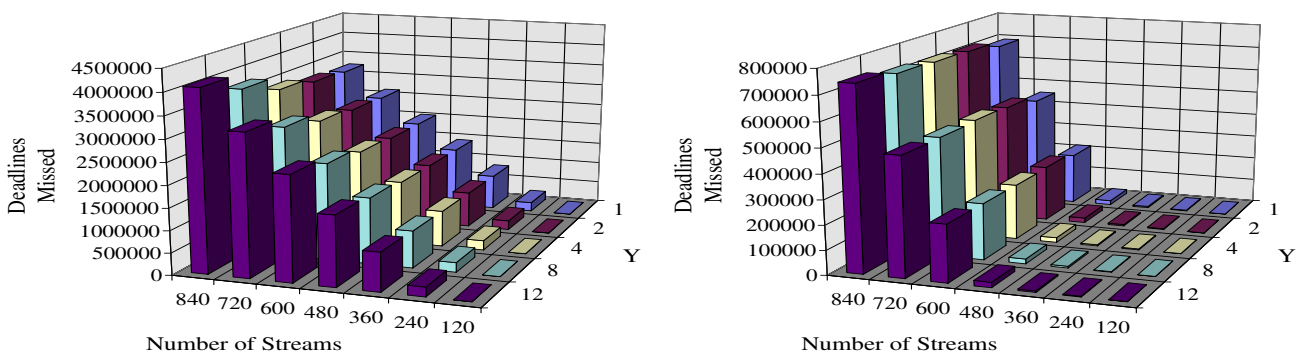


**Figure 7. Number of deadlines missed as the number of scheduler cycles between checking deadlines is increased. The `Y' axis is the number of scheduler cycles between checking deadlines.**

figure shows the case when deadlines are 500 time units apart and the scheduler increments a logical clock 1 time unit every time it services a packet. Thus, all deadlines are measured in logical time, as in the above experiments. The overheads of a two-heap implementation (as described in Section 4) of DWCS are compared with a linear-list based implementation. Clearly, implementing deadlines and loss-tolerances in a priority queue structure, such as a heap, greatly improves the performance of DWCS. This shows that the average-case performance of DWCS can be far better than the worst-case $O(n)$ time would suggest.

**Effects of Synchronization.** Figure 5(b) shows the synchronization overheads (in microseconds) as the number of streams is increased. In this example, DWCS is applied to a real video application and each stream-generating process has to acquire a System-V UNIX semaphore before the packets in the stream can be placed into the scheduler queue. Eliminating synchronization primitives, as explained in Section 4, greatly improves the speed of execution of DWCS.

**Flexible Scheduling by Approximation.** DWCS can also be approximated in an attempt to increase its speed of execution. That is, the scheduler can reduce the frequency with which it checks streams for missed deadlines. Recall that DWCS checks streams for packets that have missed their deadlines every time a packet has been serviced. If a stream has one or more late packets, the stream's corresponding packet loss-tolerance must be adjusted to account for every deadline missed.

Figure 6 shows the scheduling overheads for DWCS

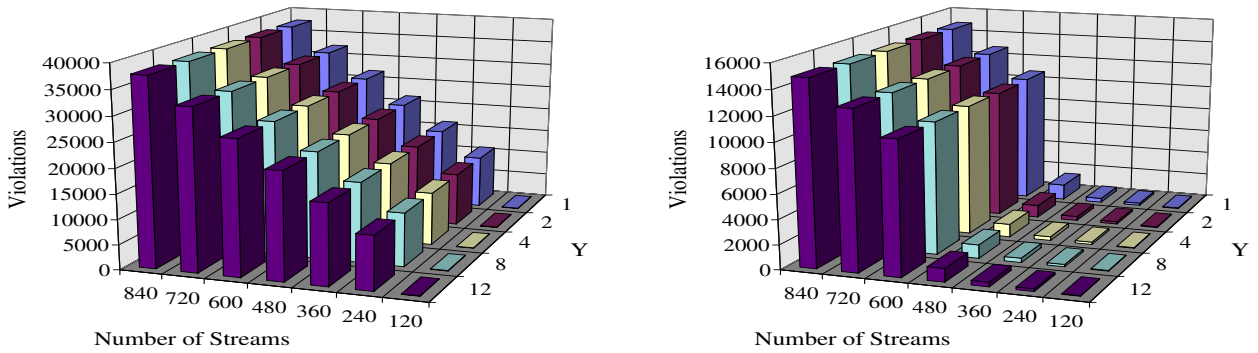


**Figure 8. Number of loss-tolerance violations as the number of scheduler cycles between checking deadlines is increased. The `Y' axis is the number of scheduler cycles between checking deadlines.**

as the number of scheduler cycles between checking deadlines is increased. The deadlines are in logical time and for each scheduler cycle, a logical clock is increased by one time unit. Thus, all packets are serviced in one logical time unit. As the number of streams is increased, the scheduling overhead increases. Scheduling latency is reduced by minimizing the frequency of checking deadlines for the case when there is a large number of streams and consecutive packet deadlines (in a given stream) are closer together. This is because fewer insertions, deletions and reordering operations are required on the corresponding stream data structures (in this case deadline and loss-tolerance heaps), when the algorithm reduces its frequency of checking deadlines.

Figures 7 and 8 show the number of deadlines missed, and the number of loss-tolerance violations, respectively, as the number of scheduler cycles between checking deadlines is increased. The results are shown for different numbers of streams when (a) the packet deadlines are 200 time units apart (left-hand graph), and (b) 500 time units apart (right-hand graph). More deadlines only start to be missed when DWCS checks deadlines less frequently, and when consecutive packet deadlines, for packets in the same stream, are closer together. However, there is little difference between the number of loss-tolerance violations when deadlines are checked only once every 12 cycles and when they are checked every cycle. Figure 6(a) shows that scheduling times are reduced for large numbers of streams as the frequency of checking deadlines is reduced. This causes more deadlines to be missed but does not affect the loss-tolerance violations too much. Since loss-tolerance is arguably a more important service objective that just a delay objective for applications that can benefit from DWCS scheduling, it is probably better to approximate DWCS (by increasing the number of scheduler cycles between checking deadlines) when the system is heavily loaded, there are a large number of streams, and deadlines are close together between consecutive packets in the same stream.

## 6. Conclusions and Future Work

This paper describes the practical issues concerned with the implementation of a scalable real-time packet scheduler, called Dynamic Window-Constrained Scheduling (DWCS). DWCS is designed to meet delay and loss constraints on information transferred across a network to many clients. In fact, DWCS has the ability to limit the number of late packets over finite numbers of consecutive packets in loss-tolerant and/or delay-constrained, heterogeneous traffic streams.

By using one heap for deadlines and another heap for loss-tolerances, the scheduling overhead of DWCS can be minimized (compared to an implementation using linear lists). Furthermore, by using a separate circular queue to buffer packets for each stream, expensive synchronization costs can be eliminated.

DWCS can also be approximated by reducing the frequency with which streams are checked for missed deadlines. The results show that approximating DWCS only reduces the scheduling overhead, thereby increasing scalability, when there are large numbers of streams, and deadlines are frequently missed. This is because the frequency with which DWCS adjusts deadlines and loss-tolerances and, hence, the number of heap insertions and deletions DWCS has to do, is reduced. Even though more deadlines are missed when the algorithm checks streams less frequently for missed

deadlines, the number of loss-tolerance violations can often be close to that achieved when DWCS checks for missed deadlines every cycle. This is an important observation, because applications that can benefit most from DWCS are those which require minimal loss-tolerance violations, or at least minimal consecutive late packets, as opposed to minimal numbers of missed deadlines overall.

When DWCS is under-loaded, few packets miss their deadlines, and any deadlines that are missed are missed in proportion to the original loss-tolerances of the corresponding streams. When DWCS cannot service all packets by their deadlines, it still provides service to streams in proportion to their deadlines and original loss-tolerances. This characteristic is similar to fair scheduling algorithms, such as WFQ[18], which are designed to allocate bandwidth in proportion to stream weights. The proportional-share property of DWCS is maintained, even when the load on the scheduler is so high that loss-tolerances are violated. Furthermore, when loss-tolerances are violated, DWCS minimizes the number of consecutive late packets over any finite window of packets in a given stream.

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. We shall investigate the ability of DWCS to support combined thread and packet scheduling. In fact, we are currently adjusting DWCS to guarantee a least upper-bound on resource (CPU or bandwidth) utilization of 100%. Suppose $l_i$ is the loss-tolerance, $C_i$ is the packet or thread service time, and $T_i$ is the time between deadlines for a packet stream or periodic thread, $i$, where $1 \leq i \leq n$. If $\sum_{i=1}^{n} \frac{(1-l_i)C_i}{T_i} \leq 1.0$, all loss-tolerances are guaranteed, with minor modifications to DWCS as presented in this paper. Finally, we shall investigate when and how DWCS can adapt[11] to changes in service demands from multiple clients with dynamically changing service requests.

## References


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

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

[3] W. J. Bolosky, R. P. Fitzgerald, and J. R. Douceur. Distributed schedule management in the tiger video fileserver. In *Sixteenth ACM Symposium on Operating System Principles*, volume 31, pages 212–223. ACM, December 1997.

[4] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair-queueing algorithm. *Journal of Internetworking Research and Experience*, pages 3–26, October 1990.

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

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

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

[8] M. B. Jones, D. Rosu, and M.-C. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Sixteenth ACM Symposium on Operating System Principles*, volume 31, pages 198–211. ACM, December 1997.

[9] H. M. V. Pawan Goyal and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *IEEE SIGCOMM'96*. IEEE, 1996.

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

[11] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. *18th IEEE Real-Time Systems Symposium*, Dec., 1997.

[12] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Trans. on Software Engineering*, 18(8):736–748, August 1992.

[13] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Real-Time Systems Symposium*. IEEE, December 1996.

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

[15] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Report MIT/LCS/TM-528, MIT, June 1995.

[16] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. Technical Report GIT-CC-98-18, Georgia Institute of Technology, 1998. To appear in the 6th International Conference on Multimedia Computing and Systems, ICMCS'99.

[17] R. West, K. Schwan, and C. Poellabauer. Scalable scheduling support for loss and delay constrained media streams. Technical Report GIT-CC-98-29, Georgia Institute of Technology, 1998.

[18] H. Zhang and S. Keshav. Comparison of rate-based service disciplines. In *Proceedings of ACM SIGCOMM*, pages 113–121. ACM, August 1991.