High–Performance Prefetching Protocols for VBR Prerecorded Video*

Martin Reisslein †
GMD FOKUS
Kaiserin–Augusta–Allee 31
10589 Berlin, Germany
reisslein@fokus.gmd.de
http://www.fokus.gmd.de/usr/reisslein

Keith W. Ross
Institute Eurecom
2229 Route des Cretes
06904 Sophia–Antipolis, France
ross@eurecom.fr
http://www.eurecom.fr/~ross

September 1998

Abstract

We describe several high–performance prefetching protocols for the transport of VBR prerecorded video over a shared channel. The protocols are particularly well–suited for the offering of video–on–demand (VoD) over a residential cable network, a residential ADSL network, or over a shared satellite channel. We advocate the use of VBR–encoded video instead of CBR video because, for the same image quality, VBR–encoded video can have a significantly lower average rate. However, in order to exploit the efficiency of VBR encoding, it is necessary that the VBR scheme multiplex and prefetch streams so that near 100% link utilizations are achieved. Our protocols are based on the observation that there are frequent periods of time during which the networks bandwidth is under utilized. During these periods the server can prefetch frames from any of the ongoing videos and send the frames to the buffers in the appropriate clients. Simulation results based on MPEG encoded traces show that the discussed prefetching protocols compare favorably with other VBR transport protocols.

Keywords: Client–Server Protocol, Prefetching, Traffic Management, VBR–Encoded Video, Prerecorded Video, Video on Demand.

*Supported partially by NSF grant NCR96–12781

†Corresponding Author: Martin Reisslein, GMD FOKUS, Kaiserin–Augusta–Allee 31, 10589 Berlin, Germany, reisslein@fokus.gmd.de, phone: +49–30–3463–7282, fax: +49–30–3463–8000, http://www.fokus.gmd.de/usr/reisslein

ADSL	Asynchronous Digital Subscriber Line
CBR	Constant Bit Rate
JSQ	Join–the–Shortest–Queue
MPEG	Motion Pictures Expert Group
TCP	Transmission Control Protocol
VBR	Variable Bit Rate
VoD	Video on Demand

Table 1: List of acronyms.

1 Introduction

We discuss high–performance prefetching protocols for the delivery of video on demand (VoD) from servers to clients across a shared channel. The shared channel could be a cable–TV plant, an access link to an ADSL switch, or a shared satellite channel. The protocols assume that the videos are variable–bit–rate (VBR) encoded. Our protocols achieve near 100% link utilizations. They also allow for immediate commencement of the video upon user request and near instantaneous response to viewer interactions such as pause, resume and temporal jumps. To achieve this high performance the protocols require that each client has a moderate amount of memory dedicated to the VoD application. The client could be a television with a set–top box capable of performing buffering and decoding, or it could be a household PC.

The protocols explicitly assume that the videos are VBR encoded with high peak–to–mean ratios. The focus on VBR encoded video is motivated by the fact that for the same perceived video quality, Constant Bit Rate (CBR) encoding produces an output rate significantly higher than the average rate of the corresponding VBR encoding for action movies [1]. Therefore schemes for VBR–encoded video that achieve near 100% channel utilization, while keeping losses at a negligible level, can allow for significantly more video connections than can CBR–encoded video.

Our prefetching protocols achieve the constant perceptual quality, responsiveness to user interactions, and high link utilizations by exploiting two special properties of the prerecorded video: (1) for each video, the traffic in each video frame is known before the video session begins; (2) while the video is being played, some of the video can be prefetched into the client memory. It is this second property — the ability to prefetch a portion of any video — that is particularly central to the discussed high–performance protocols. Admission control for the prefetching protocols is rather simple. Initially, we require that all link utilizations do not exceed 95 %. The prefetching protocols are based on the observation that, due to admission control and the VBR nature of the multiplexed traffic, there will be frequent periods of time during which the shared link's bandwidth is under utilized. During these periods the server can prefetch video frames from any of the ongoing videos and send the prefetched frames to the buffers in the appropriate clients. With this prefetching, many of the clients will typically

have some prefetched reserve in their buffers.

The prefetching protocols create a *buffer pooling effect* so that the system behaves as if the individual client buffers are aggregated into one large buffer which is shared by all the clients. Our empirical work with public–domain traces indicates that prefetching gives dramatic reductions in packet loss. In particular, if each client dedicates a small amount of buffer capacity to the VoD application, this scheme can multiplex a large number of connections over the shared link and have negligible playback starvation. With the outlined prefetching protocols, the connections collaborate through buffer pooling. This collaboration among the connections contributes significantly to the outstanding performance. We also present numerical results which show that *Optimal Smoothing*, a non–collaborative prefetching policy, can have packet loss that is several orders of magnitude higher than that of the discussed collaborative prefetching protocols for a wide range of buffer sizes.

In this tutorial we present an overview of two classes of prefetching protocols. To fix ideas, we discuss the two classes in the context of VoD over cable plant. The first class of protocols assumes that all the video streams emanate from a single server. We survey the Join–the–Shortest–Queue (JSQ) prefetching protocol and its variants. The second class of protocols permits the video streams to emanate from multiple independent service providers. We refer to this latter class of protocols as *decentralized protocols*. In this tutorial we only present an overview of the two classes of protocols. A more detailed discussion of the JSQ protocols can be found in [19], and a more detailed discussion of the decentralized protocols can be found in [20].

This tutorial is organized as follows. In the following subsection we briefly review the literature on transmission schemes for VBR Video on Demand. In Section 2 we discuss how the collaborative prefetching protocols can provide VoD over the cable plant. In Section 3 we focus on the special case where all video streams emanate from one server. We introduce the Join–the–Shortest–Queue (JSQ) prefetching protocol, discuss some important refinements and present simulation results. We furthermore discuss how the JSQ protocol handles viewer interactions such as pause, fast forward and rewind. In Section 4 we introduce the decentralized prefetching protocol. It can be employed when the video streams emanate from multiple video servers. We discuss some of the protocols refinements and evaluate its performance through extensive simulations. We give some concluding remarks in Section 5.

1.1 Literature Review

The traffic management schemes for VBR video in the literature fall into four main categories: deterministic without smoothing; deterministic with smoothing; probabilistic; and probabilistic with collaborative prefetching; see Figure 1. The deterministic schemes without smoothing send into the network the original VBR traffic, and admission control ensures that the delays never exceed a prespecified limit [8, 11, 12, 27]. For highly variable VBR traffic, these deterministic schemes typically require large initial delays to achieve moderate link utilizations [13]. The deterministic schemes with smoothing do not send the original VBR traffic into the

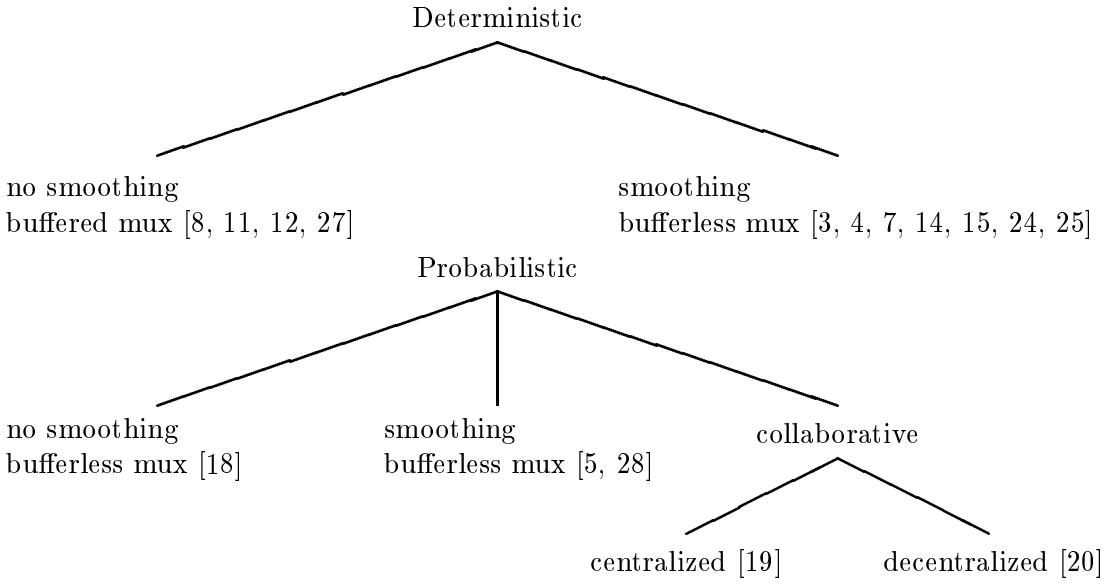

Figure 1: Transmission Schemes for VBR Video on Demand.

network, but instead send some smoothed version of it. The video traffic is transmitted at different constant rates over different intervals according to a specific transmission schedule. The piece–wise constant–rate transmission schedule takes advantage of the the fact that some of the prerecorded video can be prefetched into the client buffer. (Prefetching is referred to as work–ahead in [24, 25].) The challenge lies in computing transmission schedules that are as smooth as possible (i.e., have the smallest peak rate and rate variability) while ensuring that the client buffer neither overflows nor underflows. Several independent research teams have proposed algorithms that compute these smooth, piece–wise constant–rate transmission schedules [3, 4, 7, 14, 15, 24, 25]. It has been shown formally that given a specific client buffer size the Optimal Smoothing algorithm [24, 25] computes the smoothest transmission schedule, that is, it gives the greatest reduction in peak–rate and rate variability. The Optimal Smoothing algorithm relies on the insight that transmission at a constant rate is as smooth as possible. Hence, a smooth transmission schedule consists of constant–rate segments that are as long as possible. Furthermore, when a rate change is required to avoid overflow or underflow of the client buffer, the rate change is scheduled as early as possible. This allows for as small a rate change as possible, resulting in a transmission schedule that is as smooth as possible. However, none of the deterministic schemes (with or without smoothing) allows for both high link utilizations (>90%) and consistently high responsiveness (less than a second) to interactivity.

For the probabilistic approaches, [18] considers sending the original VBR encoded video into an unbuffered multiplexer. This scheme allows for responsive interactivity, but introduces

packet loss whenever the aggregate transmission rate exceeds the link rate. In [5] and [28] related ideas are explored whereby the original traffic is first smoothed before it is statistically multiplexed at an unbuffered link; the statistical multiplexing of the smoothed traffic can substantially increase link utilization at the expense of small packet loss probabilities. In particular, in [28] the authors demonstrate that statistically multiplexing the optimally smoothed video traffic can give moderately high link utilizations.

We cover probabilistic transmission schemes with collaborative prefetching in this tutorial. These protocols determine the transmission schedule of a connection on–line as a function of the buffer contents at *all* the clients and are therefore referred to as *collaborative*. Optimal Smoothing is non–collaborative; the transmission schedule is computed before the transmission begins and thus does not take the other ongoing connections into account. The collaborative prefetching protocols achieve nearly 100 % link utilization, immediate commencement of playback and instantaneous response to viewer interactions. It is shown in [19] and [20] that prefetching protocols with collaborative prefetching have substantially less packet loss than does Optimal Smoothing for the same link utilization.

2 Prefetching and Residential Broadband Access

In this section we discuss how the prefetching protocols for VoD discussed in this tutorial tie into the cable modem technology. We begin with a brief overview of the cable modem technology; for more details see [9, 10, 16].

Coaxial cable was originally installed in residential neighborhoods for the broadcast of one–way analog TV. Cable is a shared medium; all of the homes attached to the same coaxial cable must share the channel bandwidth. Medium access control for the downstream video traffic is particularly simple as there is only one sender, the headend. The upstream control traffic, however, poses a problem. Carrier sensing fails for cable plants with tree–and–branch structure, where only the headend hears every source. Remedies for this problem are currently being developed [9]. As of the writing of this paper, there are no fixed standards that specify how upstream and downstream bandwidth are allocated to homes. Typically, the upstream traffic is transmitted in the 5–40 MHz range. The downstream bandwidth from $40 - 750$ MHz is split into 6 MHz channels for analog TV. Each of these channels yields approximately 25 Mbits/sec when 64 Quadrature Amplitude Modulation (QAM) is employed and could thus carry a couple of video streams.

Figure 2 shows a possible VoD architecture with cable. Multiple video servers attach directly to the cable headend as do multiple cable trunks. Homes are attached to cable trunks via cable modems. The video servers could be owned by one video service provider or by multiple competing service providers (all of whom run their applications over the decentralized prefetching protocol described in Section 4). The request for a video is relayed from the viewers home to the headend via the upstream channels. The headend, acting as an Ethernet switch, ATM switch, or router, forwards the request to the appropriate video server. The video server

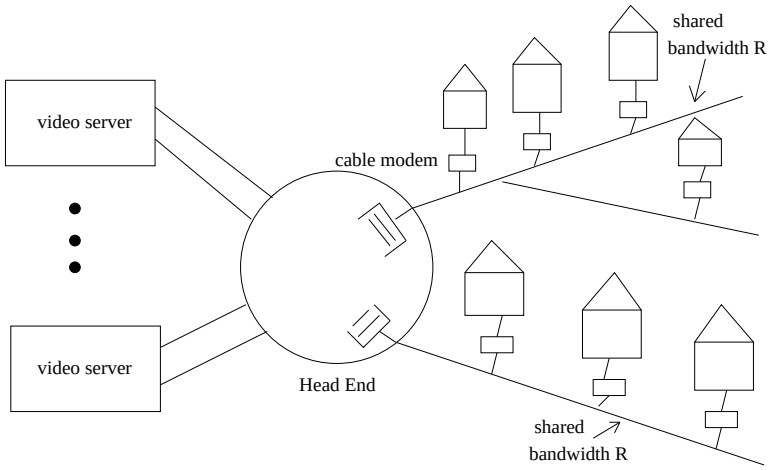

Figure 2: VoD architecture for cable residential access

.

immediately starts transmitting the video frames. The switch in the headend forwards the frames to the appropriate output queue. All the videos requested by viewers connected to the same cable trunk are multiplexed onto the shared channel of capacity, say R bits/sec. The prefetching protocols discussed next allow for the efficient use of the valuable trunk bandwidth, R. They achieve transmission with negligible losses and thus constant high video quality for average trunk bandwidth utilizations of 95%.

3 Centralized Prefetching

It is instructive to start the discussion with a VoD architecture where all video streams emanate from one video server. This basic model for VoD with one video server is illustrated in Figure 3. The video server contains a large number of videos in mass storage. For notational simplicity, assume that each video consists of N frames and has a frame rate of F frames/sec. The videos are VBR encoded using MPEG 1, MPEG 2 or some other video compression algorithm. Let J denote the number of video connections in progress. Although some of the connections might be transmitting the same video, the phases (i.e., the start times) are typically different. The server packetizes the frames of the ongoing connections and then statistically multiplexes and transmits the packets into its link; for simplicity, we assume for the following discussion that each video frame is transmitted in one packet. (In our numerical work we assume the more realistic case of fixed size packets.) Let $x_n(j)$ denote the number of bits in the nth frame of the jth connection. (The notation used in this tutorial is summarized in Table 2.) Because the videos are prerecorded, $(x_1(j), x_2(j), \ldots, x_N(j))$ is fully known at connection establishment.

When a client requests a specific video, the server makes an admission control decision by deciding whether or not to grant the request. If it grants the request, a connection is established and the server immediately begins to transmit the connection's packets into the network. The

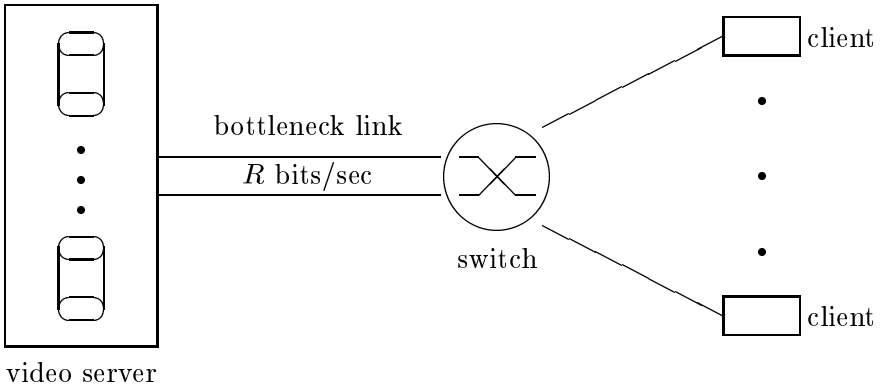

Figure 3: Centralized Video on Demand (VoD) architecture. One video server multiplexes prerecorded videos onto a link of capacity R bits/sec.

$a_l(j)$	number of frames arriving to connections j's prefetch buffer during slot l
$b(j)$	number of bits in connection j's prefetch buffer
$B(j)$	buffer capacity of client j in bits
F	frame rate in frames/sec
J	number of ongoing video connections
N	number of frames in videos
$p_l(j)$	number of prefetched frames in connection j's prefetch buffer at the beginning of slot l
R	bandwidth of shared channel in bits/sec
$x_n(j)$	number of bits in nth frame of video j
w_l	send window in slot l
Δw_l	send window increment in slot l

Table 2: Overview of notation used in this tutorial.

connection's packets are transmitted in a fixed, predetermined order. When packets arrive at the client, they are placed in the client's prefetch buffer. The video is displayed on the user's monitor as soon as a few frames have arrived at the client.

Under normal circumstances, every $1/F$ seconds the client removes a frame from the prefetch buffer, decompresses it, and displays it. If at one of these epochs there are no complete frames in its prefetch buffer, the client loses the current frame; the client will try to conceal the loss by, for instance, redisplaying the previous frame. At the subsequent epoch the client will attempt to display the next frame of the video.

We denote R (in bit/sec) for the maximum transmission rate of the server. Although the protocol allows for pause and temporal jumps, we will initially exclude these interactive features in order to simplify the discussion; thus N/F seconds elapse from when the user begins to watch the video until when the video ends. We shall also initially assume that all prefetch buffers are infinite.

To make these ideas a little more precise, we divide time into slots of length $1/F$. This slotted time structure is based on the periodicity of the frame deadlines at the clients; the clients remove a frame from their buffers every $1/F$ seconds. Let $p_l(j)$ be the number of frames in the prefetch buffer for connection j at the beginning of slot l. Let $a_l(j)$ be the number of frames of connection j that arrive to the prefetch buffer during the lth slot. At the end of each slot, one frame is removed from each prefetch buffer that has one or more frames. Thus

$$p_{l+1}(j) = [p_l(j) + a_l(j) - 1]^+, \tag{1}$$

where $[x]^+ = \max(x, 0)$.

During each slot of length $1/F$ seconds the server must decide which frames to transmit from the J ongoing videos. The *prefetch policy* is the rule that determines which frames are transmitted in each slot. The maximum number of bits that can be transmitted in a slot is R/F.

For each ongoing connection j the server keeps track of the number of prefetched frames, $p_l(j)$; this can be done through the recursion (1) without communicating with the client. Through a similar recursion the server keeps track of the number of bits in the client buffers.

3.1 The JSQ Prefetch Policy

The Join–the–Shortest–Queue (JSQ) prefetch policy attempts to balance the number of frames across all of the prefetch buffers. In describing this policy, we drop the subscript l from all notations. At the beginning of each slot the server determines the j^* with the smallest $p(j)$, transmits one frame from connection j^* and increments $p(j^*)$. Within this same slot the server repeats this procedure over and over again, at each iteration finding a new j^* that minimizes $p(j)$, transmitting a frame from connection j^* and incrementing $p(j^*)$.

Due to the finite transmission rate of the server, at some point the server must stop transmitting frames within the slot. To this end, let z be a variable that keeps track of the total number of bits sent within the slot; z is reinitialized to zero at the beginning of every slot.

The stopping rule works as follows. Before transmitting a frame from connection j^* we check to see if

$$z + x_{\sigma(j^*)}(j^*) \leq R/F, \tag{2}$$

where $\sigma(j^*)$ is the frame of connection j^* that is being considered for transmission. If this condition holds, then we transmit the frame and update z; otherwise, we do not transmit the frame, set $p(j) = [p(j) - 1]^+$ for $j = 1, \ldots, J$ and recommence the procedure for the subsequent slot. This is our *basic stopping rule*; later we shall discuss a slightly more complicated stopping rule.

Note that the stopping rule (2) ensures that the server never transmits more than R/F bits in a slot of length $1/F$ seconds. Thus there is never any loss at the bottleneck link (see Figure 3). We define the loss probability of the JSQ prefetch policy as the long–run fraction of frame periods during which one or more clients experience playback starvation. Playback starvation at one or more clients occurs if the stopping rule forces the server to stop transmitting for the slot before all of the frames for the clients that had no prefetched frames at the beginning of the slot ($p = 0$) have been transmitted. Recall that the server in each slot transmits first the frames to the clients with the fewest prefetched frames and thus gives priority to the clients without any prefetched frames. Hence, playback starvation occurs when the frame sizes of all the frames that the server is supposed to send to the clients with $p = 0$ add up to more than R/F bits. If this is the case, one or more clients have no prefetched frames and do not receive any frames during the slot. These clients suffer playback starvation as they are unable to decode and display a frame at the end of the slot. When playback starvation occurs the server skips the frames that would not meet their deadlines and sends the next frames to those clients in the next slot.

Up to now we have measured the length of the queue in a prefetch buffer by the number of frames present in the buffer. An alternative measure for length is the number of bits in the buffer. With this measure, the JSQ policy strives to equalize the number of bits in each of the prefetch buffers. At the beginning of each slot the server determines the connection with the smallest number of bits in the client buffer, transmits one packet[1] from this connection and increments the counter that keeps track of the number of bits in the client buffer. Within this same slot the server repeats this procedure over and over again, at each iteration finding a new connection that minimizes the number of bits in the client buffer, transmitting a fixed size packet from this connection and incrementing the bit counter. For infinite prefetch buffers, the procedure stops when z (the total number of bits transmitted within the slot) equals R/F. For finite prefetch buffers, the procedure stops when $z = R/F$ or when all the prefetch buffers are full.

The drawback of this policy is that it can produce frame levels in the prefetch buffers which are highly unbalanced; this can occur when one set of connections has a large number of bits per frame and a second set has a small number of bits per frame. An advantage of this policy is that it can fill all the prefetch buffers to the brim when drain rate is far below the link rate.

[1] In defining this bit–based policy, we assume fixed size packets

Throughout the remainder of this paper we use the original frame–based JSQ policy.

3.2 Refinements of the JSQ policy

We now discuss a few important refinements of the JSQ policy. First, we introduce a refined stopping rule. Recall that during each slot the server transmits a sequence of frames until condition (2) is violated; once (2) is violated, the server does not transmit any more frames in the slot. An alternative stopping rule is to try to transmit more frames in the slot by removing from consideration the connection that violates (2) and finding a new j^* that minimizes $p(j)$. If the condition (2) holds with the frame from the new connection j^*, we transmit the frame, update $p(j^*)$, and continue the procedure of transmitting frames from the connections that minimize the $p(j)$'s. Whenever a frame violates condition (2), we skip the corresponding connection and find a new j^*. When we have skipped over all of the connections, we set $p(j) = [p(j) - 1]^+$ for $j = 1, \ldots, J$ and move on to the next slot. This is our *refined stopping rule*. To reduce the online computational effort we can also, of course, consider rules which fall between the basic and refined stopping rules. For example we could use a rule which stops when condition (2) has been violated K times, where $1 < K < J$.

The next refinement of the JSQ policy limits the number of bits an ongoing connection may have in its client's prefetch buffer. This important refinement is useful when the client for connection j, $j = 1, \ldots, J$, has a finite buffer capacity of $B(j)$ bits. This refinement works as follows. Suppose that the server is considering transmitting frame $\sigma(j^*)$ from connection j^*. Let $b(j^*)$ be the current number of bits in the prefetch buffer for connection j^*. It transmits this frame in the current slot only if condition (2) and the condition

$$b(j^*) + x_{\sigma(j^*)}(j^*) \leq B(j) \tag{3}$$

are satisfied. Condition (3) ensures that the server does not overflow the prefetch buffer for connection j^*. With this additional condition, we extend the definitions of the stopping rules in the obvious way.

3.3 Experimental Results

In this subsection we present the results of a simulation study for the JSQ prefetch policy introduced in the previous subsections. Throughout we use the refined stopping rule discussed in Section 3.2. Our simulation study makes use of MPEG 1 encodings of the four movies in Table 3. The associated traces, obtained from the public domain [22], give the number of bits in each frame. (We are aware that these are low resolution traces and some critical frames are dropped; however, the traces are extremely bursty. We have obtained similar results, not reported here, for *Star Wars* and *Oz*.) Each of the movies was compressed with the GOP pattern IBBPBBPBBPBB at a frame rate of $F = 24$ frames/sec. Each of the traces has $N =$ 40,000 frames, corresponding to about 28 minutes. The mean number of bits per frame and the peak–to–mean ratio are given in Table 3. Table 3 also gives the average and peak bitrates. It

Trace	Frame size		Bitrate	
	mean bits	peak/ mean	average kbits/sec	peak Mbits/sec
lambs	7,312	18.4	175	3.2
bond	24,308	10.1	583	5.8
terminator	10,904	7.3	262	1.9
mr.bean	17,647	13.0	424	5.5

Table 3: Statistics of MPEG–1 traces.

can be argued that the average rate in bits/sec is lower than what we would expect for digital compressed video (e.g., MPEG–2 video); for this reason, we have chosen a relatively small server transmission rate of 45 Mbits/sec. We expect VoD systems in the future to have videos with larger average rates and a proportionally larger server transmission rate. In this scaling, the number of videos that can be multiplexed will be approximately constant. We furthermore assume that each packet consists of 512 bytes of payload and 40 bytes of overhead; therefore, $R = 81,521$ packets/sec.

We define the link utilization as the average number of packets per second, summed over all connections in progress, divided by R. In our experiments we use a mix of the the four movies that achieves 95 % link utilization. Specifically, our mix consists of 55 lambs connections, 17 bond connections, 37 terminator connections, and 23 mr.bean connections. With these numbers, each of the four movies accounts for roughly one fourth of the link load.

In each realization of our simulation, we generate a random starting frame for each of the J ongoing connections. The starting frames are independent and uniformly distributed over $[1, N]$. All connections start with empty prefetch buffers at the beginning of slot 1. When the Nth frame of a video is removed from a prefetch buffer, we assume that the corresponding user immediately requests to see the entire movie again. Thus, there are always J connections in progress. We run each simulation until the 90% confidence interval is less than 10% of the estimated loss probability. Recall that the loss probability is defined as the long–run fraction of frame periods during which at least one client experiences starvation.

In Figure 4 we plot the loss probability for client buffer sizes ranging from 2 KBytes to 256 KBytes. Figure 4 shows the dramatic improvement in performance that comes from prefetching and the JSQ policy. Without any prefetching we have a loss probability of 0.29 for 95 % utilization. By increasing the capacity only to 256 KBytes, the loss probability is reduced to $\approx 4.7 \times 10^{-6}$. By further increasing the buffer to 512 KBytes, no loss was observed for any of 8000 replications, corresponding to 3.2×10^8 frame periods! Whenever loss occurred, the buffer contents at each of the clients was nearly zero, confirming the existence of a strong pooling effect. (The existence of pooling is further justified by the analytical work of Reiman [17]).

Figure 4 demonstrates that with a small prefetch buffer at each client the JSQ prefetching

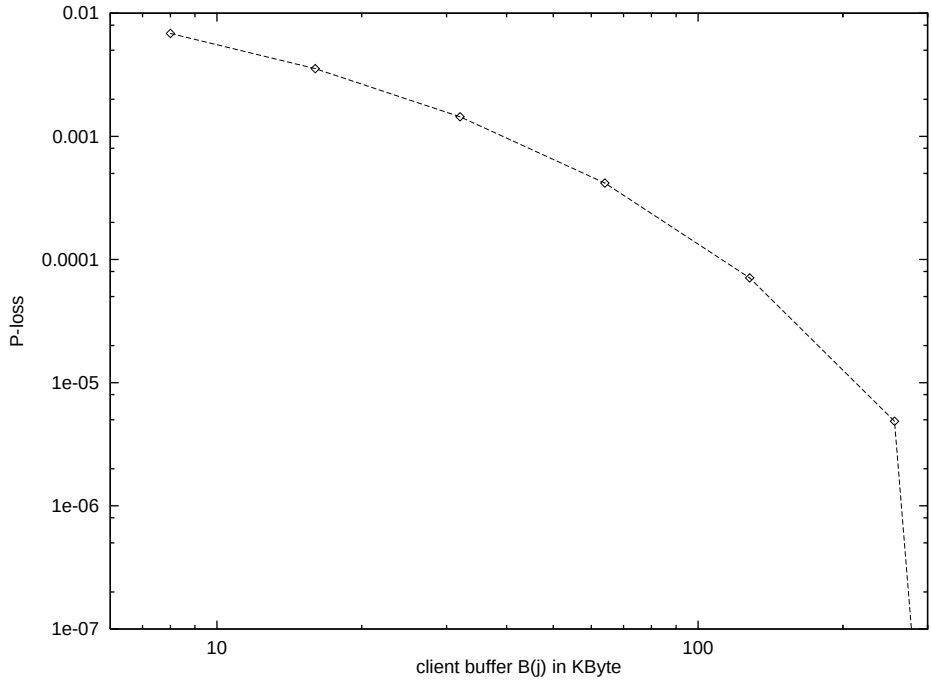

Figure 4: Loss probability as a function of client buffer size for 95 % link utilization.

policy will allow for almost 100% utilization with negligible packet loss. Also note that the scheme allows for near immediate playback of the video upon user request. It can be argued that for MPEG–2 traces with an order of magnitude larger average rate, the prefetch buffer will have to be an order of magnitude larger to achieve the same loss probabilities. But even with this order of magnitude increase, only 5 Mbytes of prefetch buffer is required to give negligible packet loss.

3.4 Interactivity

In this section we describe how the JSQ prefetch policy can be adapted to account for pauses as well as forward and backward temporal jumps. The JSQ protocol allows these interactive actions to be performed with minimal delay. We assume that whenever a user invokes a interactive action, the client sends a message indicating the interactive action to the server.

Suppose that the user for connection j pauses the movie. Upon receiving notification of the action, the server can simply remove connection j from consideration until it receives a resume message from the client; while the connection is in the paused state, its prefetch buffer remains at a constant level. A slightly more complex policy would be to fill the corresponding buffer with frames once all the other prefetch buffers are full or reach a prespecified level.

Suppose that the user for connection j makes a temporal jump of δ frames into the future. If $\delta < p(j)$, we discard δ frames from the head of the prefetch buffer and set $p(j) = p(j) - \delta$. If $\delta \geq p(j)$ we set $p(j) = 0$ and discard all the frames in the prefetch buffer. Finally, suppose that the user for connection j makes a backward temporal jump. In this case we set $p(j) = 0$

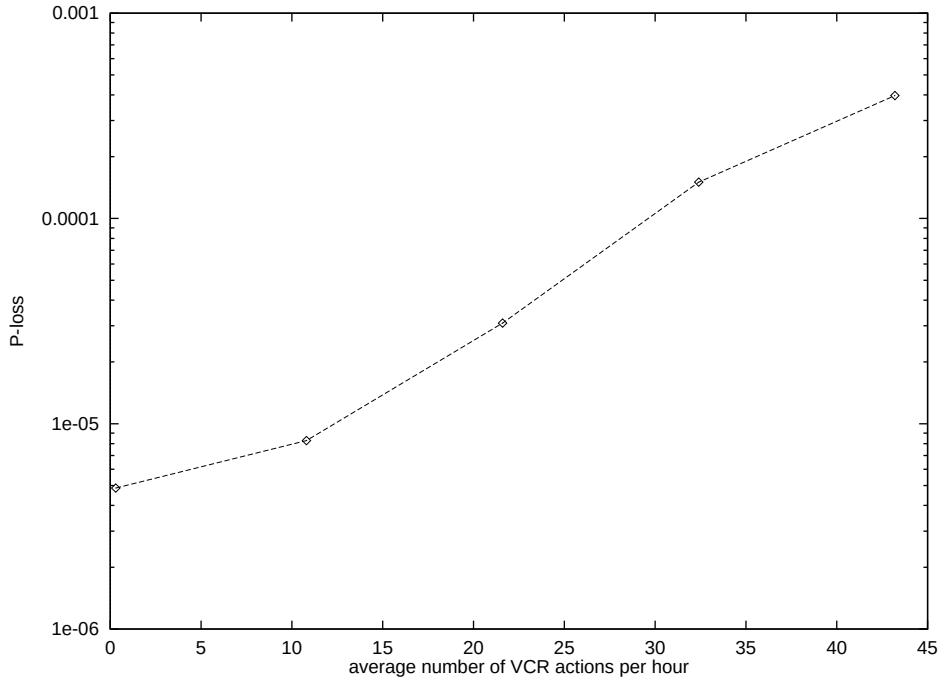

Figure 5: Loss probability as a function of the average number of temporal jumps per hour for 256 KByte client buffers and 95 % link utilization.

and discard all frames in the prefetch buffer.

In terms of frame loss, pauses actually improve performance because the movies which remain active have more bandwidth to share. Frequent temporal jumps, however, can degrade performance since prefetch buffers would be frequently set to zero. Whenever we set a prefetch buffer to zero, the pool loses some of its total savings, thereby increasing the likelihood of loss.

We now present some numerical results for interactive actions. We consider only forward and backward temporal jumps and ignore pauses as pauses can only improve performance; we furthermore assume that $\delta > p(j)$ for all forward temporal jumps. The presented results give therefore a conservative estimate of the actual performance. In our simulation, we assume that each user performs temporal jumps repeatedly, with the time between two successive jumps being exponentially distributed with constant rate. When a user performs such an action, her prefetch buffer is set to zero. Figure 5 shows the loss probability for 11, 22, 32 and 43 temporal jumps per hour (on average). This experiment was conducted with a client buffer of 256 KBytes and a 95 % link utilization. As we would expect, loss probabilities increase as the rate of temporal jumps increase; however, the increase is not significant for a sensible number of temporal jumps.

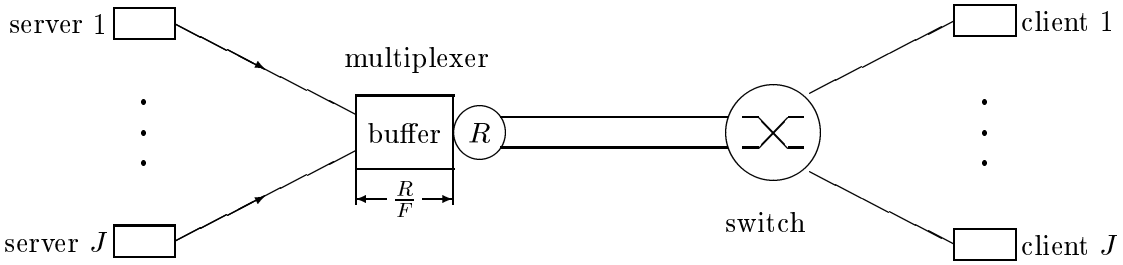

Figure 6: Decentralized Video on Demand Architecture.

4 Decentralized Prefetching

We now return to the VoD architecture with multiple video servers shown in Figure 2. Figure 6 depicts a schematic of this model for VoD[2]. We assume for the purpose of this study that each video server feeds one client; thus there are J video servers feeding J clients. In explaining the client–server interaction, we focus on a particular client–server pair. For simplicity, we again assume that each video frame is transmitted in one packet. Let x_n denote the number of bits in the nth frame. As in Section 3 we assume that the videos are prerecorded. The sequence $(x_1, x_2, \ldots, x_N)$ is therefore fully known before the transmission of the video. At the beginning of each frame period, that is, every $1/F$ seconds, the server decides according to a prefetching policy, outlined in the next section, which and how many frames to transmit. The server sends the frames to the multiplexer buffer. Frames that do not fit into the multiplexer buffer are lost. The multiplexer buffer of size R/F bits is served at rate R bits/sec. The maximal delay incurred in the multiplexer is therefore $1/F$ seconds. For simplicity we assume that the propagation and processing delays are negligible. The client instantaneously sends a positive acknowledgment to the server for each frame received.

With these delay assumptions, the server receives acknowledgments for all frames successfully received by the client within one frame period. The server therefore knows whether the frames sent in the previous frame period were received before deciding which frames to send in the current frame period.

The multiplexer design just described uses a finite buffer of size R/F to ensure that the multiplexer delay is $\leq 1/F$ seconds. An alternative implementation with a larger buffer (that may be built–in by the multiplexer manufacturer) is as follows. The server timestamps each of the frames it sends. When a frame reaches the front of the multiplexer buffer, the multiplexer checks to see if the delay of the frame is $\leq 1/F$. If the delay exceeds $1/F$, the multiplexer

[2] Although we discuss the decentralized prefetching protocol in the context of a single shared link, the protocol applies to arbitrary networks with multiple shared links.

discards the frame. The multiplexer can also periodically check all the frames in the queue and purge those that have a delay exceeding $1/F$.

When a client requests a specific video, the network accepts the new connection as long as the average link utilizations are $\leq 95\%$. The average link utilization is $util = F \sum_{j=1}^{J} x_{\text{avg}}(j)/R$, where $x_{\text{avg}}(j)$ is the average frame size in bits of the jth connection, which is calculated by averaging the corresponding sequence $(x_1, \ldots, x_N)$.

The decentralized prefetching protocol employs window flow control; it is inspired by the Transmission Control Protocol (TCP) [6, 26] widely used in the Internet. The basic decentralized prefetching protocol works roughly as follows. The server maintains a send window, limiting the number of frames the server is allowed to send in a frame period. The send window is increased by a small increment when all acknowledgments arrive in time. Due to admission control and the VBR nature of the traffic, there are periods of time during which the network is underutilized. The send window grows larger than one during these periods, allowing the server to prefetch future frames into the client memory. In times of network congestion, frames are lost or delayed and the corresponding acknowledgments do not arrive at the server before their timeouts. In this case, the send window is reduced to throttle the server and alleviate the congestion. The reservoir of prefetched frames in the client buffer allows the client to continue playback during these periods of congestion. Starvation at the client occurs only if the reserve of prefetched frames at the client is completely depleted and the current frame is lost or delayed due to network congestion.

4.1 Basic Decentralized Prefetching Protocol

In this section we give a detailed description of the basic decentralized prefetching protocol that allows the server to determine how many frames to send in each frame period. This protocol strives to (1) make efficient use of the buffers at the client and (2) avoid bandwidth "hogging" by a particular connection and thus give each connection a fair share of the bandwidth. The protocol attempts to allow each client to build up a reservoir of prefetched frames.

When discussing the server policy we again focus on a particular connection. We divide time into slots of length $1/F$. Let l denote the current slot; l is a local variable maintained by the server. In the course of the transmission of a video with N frames, l runs from 1 through N. *We do not assume any synchronization of time slots among the client–server pairs.*

Of central importance to our policy is the *send window*, denoted w_l, which limits the amount of traffic the connection can transmit in slot l. Specifically, the server is allowed to transmit $\lfloor w_l \rfloor$ frames during slot l. (We assume for simplicity that only complete frames are transmitted.) A new connection starts with a send window of $w_0 = 1$. The send window is increased by a small increment Δw, say 0.1, at the beginning of each slot, i.e. $w_l = w_{l-1} + \Delta w$. After computing the send window the server transmits $\lfloor w_l \rfloor$ frames; see Figure 7. Note that $w \geq 2$ allows for prefetching of future frames. To keep track of the number of prefetched frames in the client buffer, let p_l be the number of frames in the client buffer at the beginning of slot l. This variable is initialized to $p_1 = 0$. Let a_l denote the number of frames that are

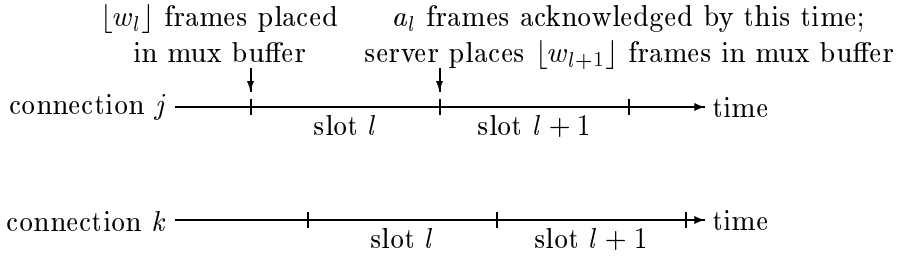

Figure 7: Timing diagram of prefetching policy. Server j places $\lfloor w_l \rfloor$ frames in the multiplexer buffer at the beginning of slot l. The acknowledgments for a_l frames arrive from the client by the end of slot l. The server processes the acknowledgments and puts $\lfloor w_{l+1} \rfloor$ frames in the multiplexer buffer at the beginning of slot $l+1$. There is no synchronization of slots between any distinct servers j and k.

received and acknowledged by the client during slot l. Clearly, $0 \leq a_l \leq \lfloor w_l \rfloor$; a_l is equal to $\lfloor w_l \rfloor$ if all frames sent are received by the client. If frames (or acknowledgments) are lost we have $a_l < \lfloor w_l \rfloor$. Figure 7 illustrates the timing of the prefetching protocol. Frame l is removed from the client buffer at the end of slot l if the client buffer contains one or more frames. The server keeps track of p_l through the following recursion:

$$p_{l+1} = [p_l + a_l - 1]^+. \tag{4}$$

Let b_l be the number of bits in the client buffer at the beginning of slot l. The server keeps track of b_l trough a recursion similar to (4).

If the server does not receive a positive acknowledgment for a frame sent at the beginning of the previous slot within one frame period, it assumes that the frame is lost. If a connection without any prefetched frames in the client buffer ($p_l = 0$) suffers loss, the client experiences starvation and may apply error concealment techniques to conceal the loss of video information. If the client has some prefetched frames in its buffer ($p_l > 0$), the server retransmits the lost frames. Whenever loss occurs, the server resets its send window to $w = 1$. The loss of frames is indicative of acute link overload and by reducing the send window we can throttle the server and thus alleviate the congestion. We refer to the send window policy described in this section as the *basic window policy*. It can be summarized as follows. A connection starts with a send window of one, that is, $w_0 = 1$. The window is increased by a small increment Δw (we use $\Delta w = 0.1$) at the beginning of each frame period. The number of frames a connection is allowed to send is limited by the integral part of the send window. If loss occurs, the window is reset to one.

4.2 Refinements of the Decentralized Prefetching Protocol

Client Buffer Constraint

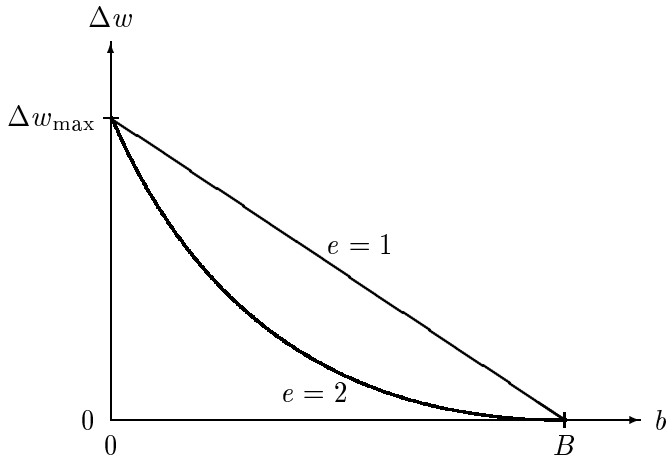

Figure 8: Illustration of the dynamic window policy.

We first introduce a modification of the decentralized prefetching protocol that limits the number of bits an ongoing connection may have in its client buffer. Suppose that the client has a finite buffer capacity of B bits. Now, suppose that the server is considering transmitting frame k. It transmits this frame in the current slot only if the send window allows the transmission of the frame and the *client buffer constraint*

$$b_l + x_k \leq B \tag{5}$$

is satisfied. Condition (5) ensures that the server does not overflow the client buffer.

Dynamic Send Window

We now introduce a refinement of the send window policy. The idea behind this refinement is to increase the send window by a large increment when the client buffer holds only a small reserve of prefetched frames and throttle the server when the client buffer contains a large reserve of prefetched frames. To this end, we compute the window increment as a function of the amount of prefetched data in the client buffer:

$$\Delta w_l = \Delta w_{\max}(1 - \frac{b_l}{B})^e, \quad \Delta w_{\max} > 0, \ e > 0. \tag{6}$$

Figure 8 illustrates this refined send window policy. When the client buffer is empty at the beginning of slot l, that is, when $b_l = 0$, the send window is incremented by $\Delta w_{\max}$. When the client buffer is full, that is, when $b_l = B$, the send window is not increased at all. We refer to this send window policy as the *dynamic window policy*. The dynamic window policy can be summarized as follows. At the beginning of slot l, the server computes Δw_l according to (6), calculates the new send window, $w_l = w_{l-1} + \Delta w_l$, and sends $\lfloor w_l \rfloor$ frames. As with the basic

window policy, a new connection starts with a send window of $w_0 = 1$ and resets the window to $w_l = 1$ if the acknowledgments do not arrive by the end of slot l.

The parameters $\Delta w_{\max}$ and e are used to tune the policy. We provide a detailed numerical study of the impact of these parameters on the performance of the decentralized prefetching protocol in [20]. Because of page limitation we give here only a brief qualitative discussion of these parameters. A large $\Delta w_{\max}$ gives large increments Δw and thus allows the server to send more frames. The parameter $\Delta w_{\max}$ has to be large enough to allow for prefetching of future frames. If $\Delta w_{\max}$ is too large, however, a few connections can "swamp" the multiplexer and degrade the protocols' performance.

The parameter e can be set to give a connection with a nearly empty client buffer an increased chance of filling its client buffer. To see this, note that for $e = 1$, the window increment decreases linearly as the client buffer contents increase. For $e > 1$, connections with fairly large buffer contents are allowed substantially smaller increments (compared to when $e = 1$), while a connection with small client buffer contents has still a large window increment. This gives a connection with a small reserve of prefetched frames a better chance of filling its client buffer.

We found that the decentralized prefetching protocol works well for a wide range of parameters. In particular, $\Delta w_{\max}$ values between 2 and 8 and e values between 4 and 10 give good performance [20]. We choose $\Delta w_{\max} = 5$ and $e = 6$ for the numerical work in this paper.

Randomized Transmission

We now add a refinement that helps to ensure fair bandwidth distribution among the ongoing connections. In the protocol described so far, the server transmits the first $\lfloor w_1 \rfloor$ frames of the video immediately after the request of the client has been processed. Subsequent transmissions are scheduled l/F seconds, $l = 1, \ldots, N-1$, after the initial transmission. The relative slot phases remain fixed for the entire duration of a video. To see how this can lead to unfair bandwidth distribution consider the phase alignment with $t_j \gg t_k$ depicted in Figure 9. Suppose connections j and k are the only connections in progress. Now consider a scenario where connection j fills the multiplexer buffer completely at the beginning of its slot l. Connection k is then able to fit Rt_k bits into the multiplexer buffer at the beginning of its slot l. When connection j is up for transmission again, at the beginning of its slot $l+1$, it can fit Rt_j bits into the multiplexer buffer. With the depicted phase alignment ($t_j \gg t_k$), connection k has clearly a disadvantage since it can transmit only Rt_k bits in a frame period as long as connection j keeps on filling the multiplexer buffer to capacity.

To avoid this unfair bandwidth distribution we introduce *randomized transmission*: The server transmits the first $\lfloor w_1 \rfloor$ frames of the video immediately after the request of the client has been processed. The server draws a random phase δ_l, $l = 1, \ldots, N-1$ from a uniform distribution over $[-1/2F,\ 1/2F]$ in each frame period. The subsequent transmissions are scheduled $l/F + \delta_l$ seconds, $l = 1, \ldots, N-1$ after the initial transmission. With this transmission rule, the slot phases are constantly re–shuffled. Unfair phase alignments can therefore not persist for extended periods of time.

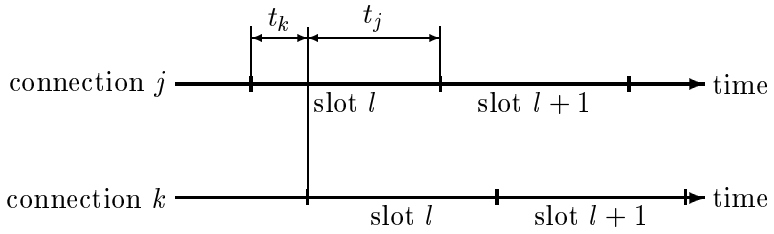

Figure 9: Phase alignment favoring connection j. If both connections fill the multiplexer buffer to capacity whenever they transmit, connection j can transmit Rt_j bits in a frame period, while connection k can transmit only Rt_k bits.

4.3 Experimental Results

In this section we present the results of a simulation study of the decentralized prefetching protocol. The study is based on the same MPEG 1 traces used in Section 3.3. As in Section 3.3 we assume that the video frames are transported in packets consisting of 512 bytes of payload and 40 bytes of overhead. Again, we fix the link rate at $R = 45$ Mbits/sec; the corresponding multiplexer buffer holds 234,375 bytes ($= R/F$).

In each realization of the simulation we generate random starting frames as described in Section 3.3. For each replication of the simulation we also draw random (non–synchronized) slot phases $t(j)$ for each of the J connections. The $t(j)$'s are independent and are drawn from a uniform distribution over $[0,\ 1/F]$. The $t(j)$'s determine the relative starting times of the slots for the J connections. Note that the frames of connection j scheduled for transmission in slot l are placed in the multiplexer buffer at the beginning of the slot (see Figure 7), that is, server j puts its traffic into the queue at instants $t(j) + (l-1)/F,\ l = 1, \ldots, N$ ($t(j) + (l-1)/F + \delta_{l-1},\ l = 1, \ldots, N$ with randomized transmission). In all our simulations we assume that all clients have the same buffering capacity of B bits.

In Figures 10, 11 and 12 we show typical plots of the client buffer contents, b_l, the window increment, Δw_l, and the send window, w_l, versus slot time, l, for four arbitrarily chosen connections. Figure 13 gives the number of frames that are successfully placed in the multiplexer buffer by each of the four connections. For this simulation run we have set the client buffer capacity to $B = 1$ MBit. We employ the dynamic window policy without randomized transmission with $\Delta w_{\max} = 5$ and $e = 2$. The plots illustrate how the client buffer contents control the window increment. The left part of Figure 10 shows that the client buffer of connection 2 is drained. This is due to a high action scene in the video. We see from Figure 11 that the window increment of connection 2 increases as the client buffer is depleted. By the end of time slot 41,274 the client buffer is empty and the window increment has risen to $\Delta w_{\max} = 5$. This allows connection 2 to transmit very aggressively. We observe from Figure 12 that the

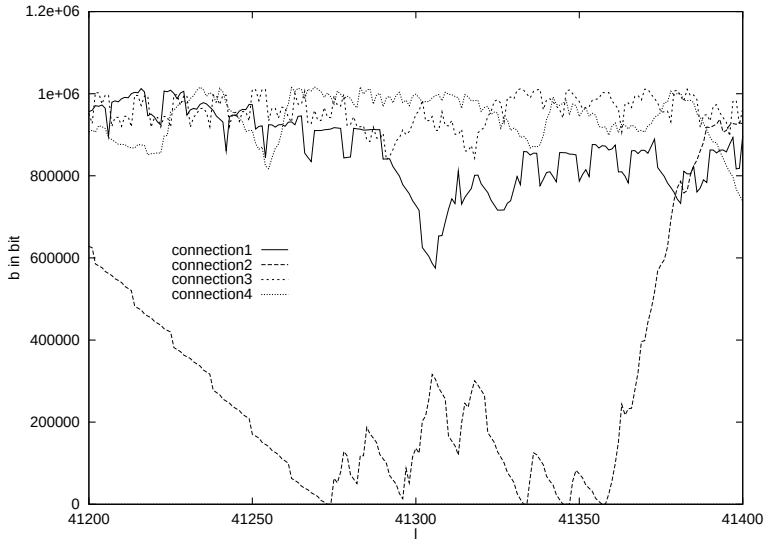

Figure 10: Client buffer contents in bits versus slot time of four arbitrarily chosen connections with 1 MBit of client buffer.

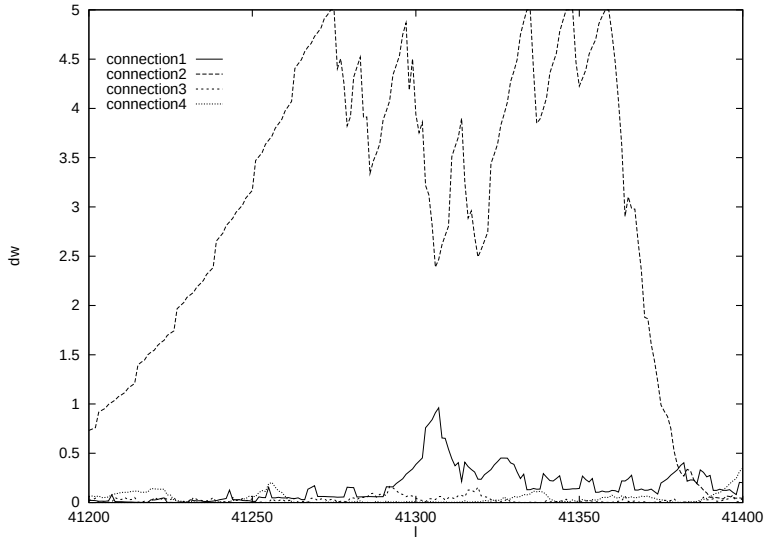

Figure 11: Window increment Δw versus slot time; Δw is computed according to the dynamic window policy with $\Delta w_{\max} = 5$ and $e = 2$, that is, $\Delta w_l = 5(1 - b_l/1\text{MBit})^2$.

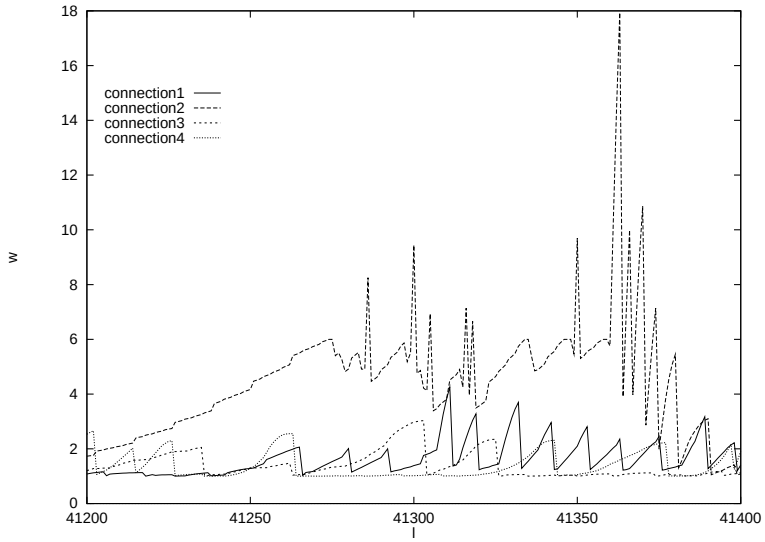

Figure 12: Send window versus slot time.

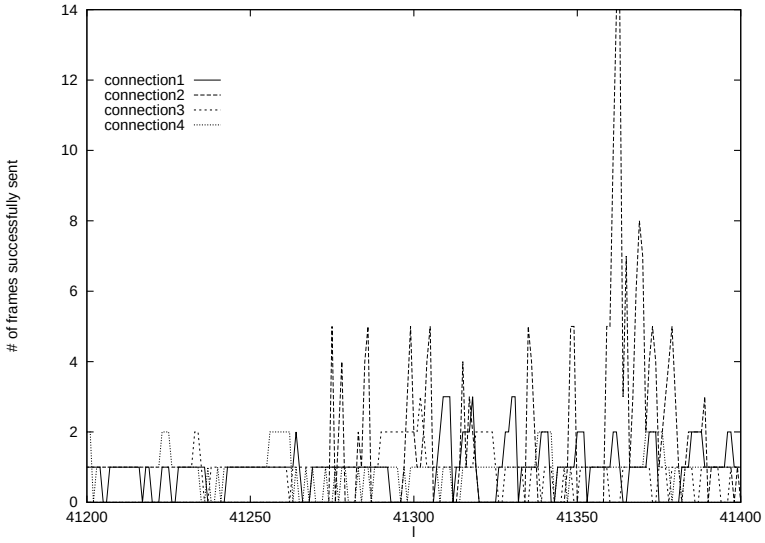

Figure 13: Number of frames successfully placed in the multiplexer buffer versus slot time.

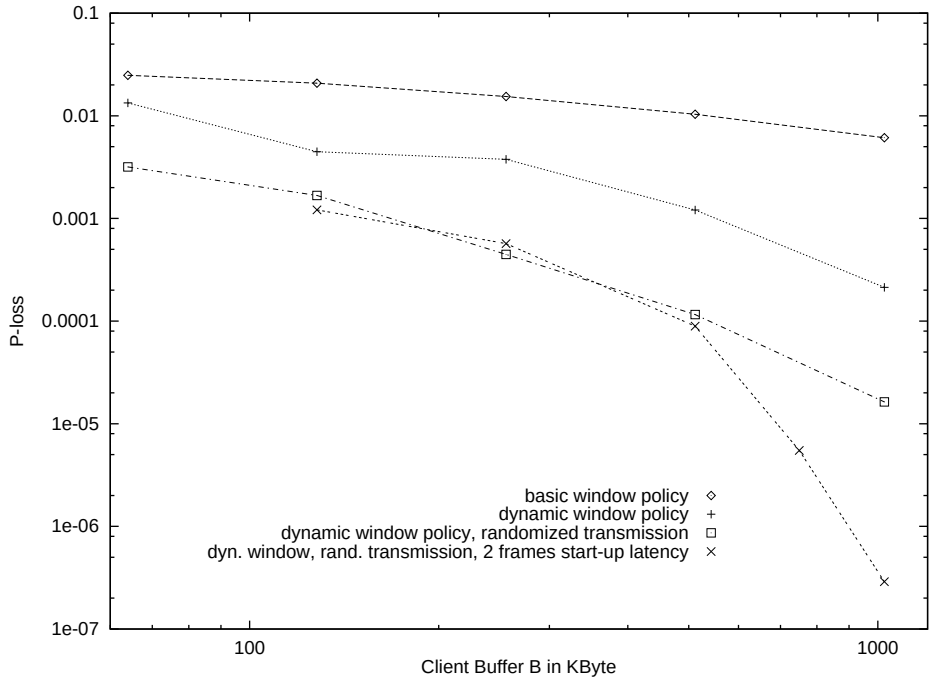

Figure 14: Loss probability as a function of client buffer size for the basic decentralized prefetching protocol and its refinements.

send window becomes as large as 17.9 in time slot 41,363. Figure 13 shows that the first 14 frames of the 17 frames sent in time slot 41,363 can be accommodated by the multiplexer buffer. The remaining 3 frames are lost. The send window is therefore reset to $w_{41,363} = 1$ at the end of slot 41,363. At the beginning of slot 41,364 the new send window is computed as $w_{41,364} = w_{41,363} + \Delta w_{41,364} = 1 + 2.9 = 3.9$, allowing connection 2 to send 3 frames. We see from Figure 13 that all 3 frames fit into the multiplexer buffer and the send window increases to $w_{41,365} = 3.9 + 3.1 = 7.0$ in slot 41,365. The right part of Figure 10 shows that the large send windows enable connection 2 to fill its client buffer again. By time slot 41380 the client buffer is filled to 90% of its capacity.

Figure 14 shows the performance of the basic decentralized prefetching protocol, and its various refinements. We plot the loss probability as a function of the client buffer size for 95% link utilization. The loss probability is again defined as the long–run fraction of frame periods during which one or more clients suffer playback starvation. For the basic window policy we use a fixed window increment of $\Delta w = 0.1$. The parameters of the dynamic window policy are set to $\Delta w_{\max} = 5$ and $e = 6$. The figure shows that the basic window policy has unacceptably high losses. The loss probability is about 8×10^{-3} for 1 MByte of client buffer. We also see that the dynamic window policy brings significant improvement over the basic window policy. The loss probability for the dynamic window policy is almost one order of magnitude smaller. Adding randomized transmission further reduces the loss probability significantly. The loss probability for the dynamic window policy with randomized transmission for 1 MByte of client

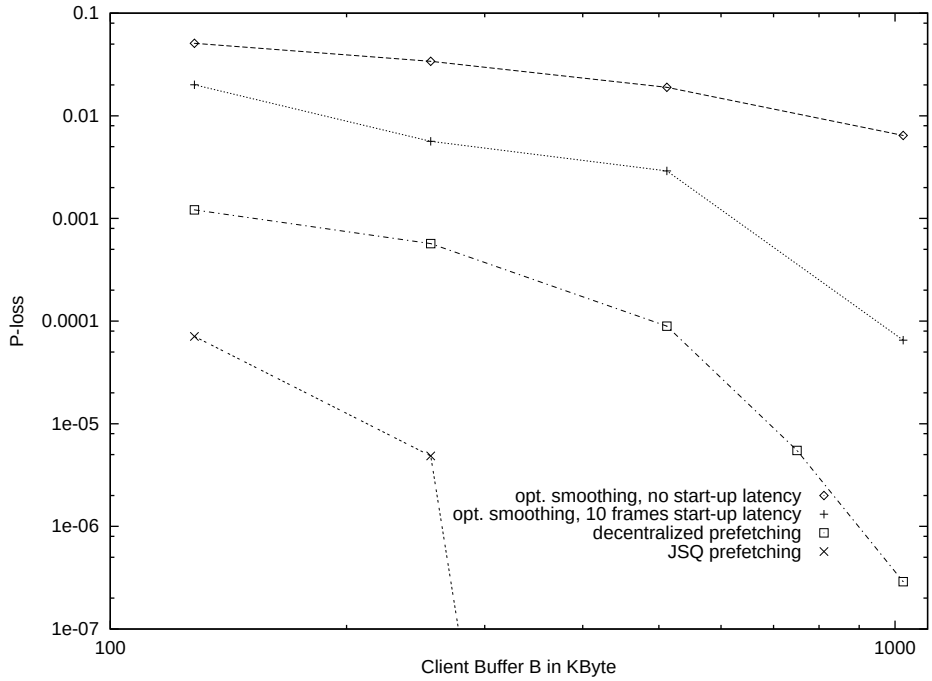

Figure 15: Loss probability as a function of client buffer size for optimal smoothing, decentralized prefetching and JSQ prefetching.

buffer is about 1.5×10^{-5}. By allowing a short start–up latency of 2 frame periods we can further reduce the loss probability significantly (see [20] for details).

In Figure 15 we compare the decentralized prefetching protocols with Join–the–Shortest–Queue (JSQ) Prefetching and Optimal Smoothing [24, 25, 28]. The plot gives the loss probability as a function of the client buffer size for 95% link utilization. The optimal smoothing curves are obtained by applying the optimal smoothing algorithm [24, 25, 28] to the traces used for the simulation of the prefetching protocols. We then compute the loss probability for statistically multiplexing the smoothed traces on a bufferless 45 Mbits/sec link with the Large Deviation approximation [2, 18, 21, 28]. We do this for two versions of optimal smoothing: no initiation delay and a 10 frame initiation delay [24, 25, 28]. The decentralized prefetching results are for the dynamic window policy with randomized transmission and 2 frames start–up latency. The JSQ prefetching results are from Section 3.3. The collaborative prefetching protocols clearly outperform Optimal Smoothing. Optimal Smoothing has unacceptably high loss probabilities for all buffer sizes shown. For 1 MByte of client buffer the loss probability for decentralized prefetching is over 2 orders of magnitude smaller than the loss probability for Optimal Smoothing. The performance of decentralized prefetching is remarkable given the fact that each server faces the classical congestion control problem of having to decide on a transmission schedule without any direct knowledge of the other ongoing connections. The JSQ protocol assumes that the server has perfect knowledge of all ongoing connections and gives outstanding performance for this centralized VoD architecture. We note that JSQ prefetching

is inspired by the least–loaded routing algorithm for circuit–switched loss networks, which is known to give excellent performance and to be extremely robust over a wide range of traffic conditions [23].

5 Conclusion

Prerecorded video has two special properties: (1) for each video, the traffic in each video frame is known before the video session begins; (2) while the video is being played, some of the video can be prefetched into the client memory. In this paper we have shown how these two properties can be exploited to achieve high performance when one or more servers transmit VBR video across a packet switched network to clients. The results should be useful for designing VoD systems that connect servers to residential broadband networks using cable or ADSL, or for VoD systems that connect the server to its clients through a satellite channel or a Local Area Network (LAN). The presented client–server protocols can be part of a larger Internet solution to VoD, whereby the prerecorded videos are multicast to local servers at off–peak hours with the best effort service.

We conclude by mentioning that an alternative approach for the delivery of the video traffic is to use the Transmission Control Protocol (TCP) [6, 26]. This TCP approach would roughly work as follows. The entire video file is passed to a TCP socket and TCP moves the video data as quickly as possible to the client. The rate of the data transfer is limited by TCP's congestion and receive windows. The client removes one frame from its receive buffer every $1/F$ seconds and decodes and displays it. There are, however, a number of drawbacks to this TCP approach. First, the TCP receive window is relatively small, allowing TCP to build up only a small reserve of prefetched frames. Secondly, TCP is a reliable transmission protocol. It retransmits every lost segment, even though the retransmitted segment may have already missed its deadline at the client. Furthermore, TCP has no implicit JSQ mechanism, that is, it does not strive to keep the client buffers full by transmitting more aggressively to the buffers that are nearly empty. For these reasons we expect the TCP approach to perform poorly. Nevertheless, it would be of interest to (1) compare the prefetching protocols to TCP; and (2) adapt the prefetching protocols so that they can run on top of TCP. These issues are the subject of ongoing research.

References

[1] I. Dalgic and F. A. Tobagi. Characterization of quality and traffic for various video encoding schemes and various encoder control schemes. Technical Report CSL–TR–96–701, Stanford University, Departments of Electrical Engineering and Computer Science, August 1996.

[2] A. Elwalid, D. Mitra, and R. H. Wentworth. A new approach for allocating buffers and bandwidth to heterogeneous regulated traffic in an ATM node. *IEEE Journal on Selected Areas in Communications*, 13(6):1115–1127, August 1995.

[3] W. Feng and J. Rexford. A comparison of bandwidth smoothing techiniques for the transmission of prerecorded compressed video. In *Proceedings of IEEE Infocom*, pages 58–67, Kobe, Japan, April 1997.

[4] W. Feng and S. Sechrest. Smoothing and buffering for delivery of prerecorded compressed video. In *Proceedings of IS&T/SPIE Multimedia Computing and Networking*, February 1995.

[5] M. Grossglauser, S. Keshav, and D. Tse. RCBR: A simple and efficient service for multiple time–scale traffic. In *Proceedings of ACM SIGCOMM*, pages 219–230, August 1995.

[6] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88 Symposium*, pages 314–329, August 1988.

[7] Z. Jiang and L. Kleinrock. A general optimal video smoothing algorithm. In *Proceedings of IEEE Infocom '98*, pages 676–684, San Francisco, CA, April 1998.

[8] E. W. Knightly and H. Zhang. D-BIND: an accurate traffic model for providing QoS guarantees to VBR traffic. *IEEE/ACM Transactions on Networking*, 5(2):219–231, April 1997.

[9] M. Laubach. Cable modem basics: An introduction to cable modem technology, October 1996. Presentation at the University of Pennsylvania
Slides available at ftp.com21.com /pub/laubach/.

[10] M. Laubach. To foster residential area broadband internet technology: IP datagrams keep going and going and going.... *ConneXions*, 10(2), February 1996.

[11] J. Liebeherr and D. Wrege. Video characterization for multimedia networks with a deterministic service. In *Proceedings of IEEE Infocom '96*, San Francisco, CA, March 1996.

[12] J. Liebeherr, D. Wrege, and D. Ferrari. Exact admission control for networks with a bounded delay service. *IEEE/ACM Transactions on Networking*, 4(6):885–901, December 1996.

[13] J. M. McManus and K. W. Ross. A comparison of traffic management schemes for prerecorded video with constant quality service. Technical report, University of Pennsylvania, Department of Systems Engineering, 1996. Available at http://www.eurecom.fr/~ross.

[14] J. M. McManus and K. W. Ross. Video on demand over ATM: Constant–rate transmission and transport. *IEEE Journal on Selected Areas in Communications*, 14(6):1087–1098, August 1996.

[15] J.M. McManus and K.W. Ross. A dynamic programming methodology for managing prerecorded VBR sources in packet–switched networks. In *Proceedings of SPIE, Performance and Control of Network Systems*, pages 140–154, Dallas, TX, November 1997. Available at http://www.eurecom.fr/~ross.

[16] P. Mockapetris. @HOME network overview, October 1996. Presentation at the University of Pennsylvania.

[17] M.I. Reiman. Some diffusion approximations with state space collapse. In F. Baccelli and G. Fayolle, editors, *Lecture Notes in Control and Informational Sciences 60*, pages 209–240. Springer-Verlag, 1983.

[18] M. Reisslein and K. W. Ross. Call admission for prerecorded sources with packet loss. *IEEE Journal on Selected Areas in Communications*, 15(6):1167–1180, August 1997.

[19] M. Reisslein and K. W. Ross. A join–the–shortest–queue prefetching protocol for VBR video on demand. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, pages 63–72, Atlanta, GA, October 1997. extended version available at http://www.fokus.gmd.de/usr/reisslein or http://www.eurecom.fr/~ross.

[20] M. Reisslein, K. W. Ross, and V. Verillotte. A decentralized prefetching protocol for VBR video on demand. In D. Hutchison and R. Schafer, editors, *Multimedia Applications, Services and Techniques — ECMAST '98 (Lecture Notes in Computer Science, Vol. 1425)*, pages 388–401, Berlin, Germany, May 1998. Springer Verlag. extended version available at http://www.fokus.gmd.de/usr/reisslein or http://www.eurecom.fr/~ross.

[21] J. Roberts, U. Mocci, and J. Virtamo (Eds.). *Broadband Network Traffic: Performance Evaluation and Design of Broadband Multiservice Networks, Final Report of Action COST 242, (Lecture Notes in Computer Science, Vol. 1155).* Springer Verlag, 1996.

[22] O. Rose. Statistical properties of MPEG video traffic and their impact on traffic modelling in ATM systems. Technical Report 101, University of Wuerzburg, Insitute of Computer Science, Am Hubland, 97074 Wuerzburg, Germany, February 1995.
ftp address and directory of the used video traces:
ftp-info3.informatik.uni-wuerzburg.de /pub/MPEG/.

[23] K. W. Ross. *Multiservice Loss Models for Broadband Telecommunication Networks.* Springer–Verlag, 1995.

[24] J. Salehi, Z. Zhang, J. Kurose, and D. Towsley. Optimal smoothing of stored video and the impact on network resource requirements. *to appear in IEEE/ACM Transactions on Networking.* Available at ftp://gaia.cs.umass.edu/pub/salehi/optimal.ton–ps.Z or http://www-users.cs.umn.edu/~zhzhang/papers.html.

[25] J. Salehi, Z. Zhang, J. Kurose, and D. Towsley. Supporting stored video: Reducing rate variability and end–to–end resource requirements through optimal smoothing. In *Proceedings of ACM SIGMETRICS '96*, pages 222–231, May 1996. Philadelphia, PA.

[26] R. Stevens. *TCP/IP Illustrated, Volume 1, The Protocols.* Addison–Wesley, 1994.

[27] D. Wrege, E. Knightly, H. Zhang, and J. Liebeherr. Deterministic delay bounds for VBR video in packet–switching networks: Fundamental limits and tradeoffs. *IEEE/ACM Transactions on Networking*, 4(3):352–362, June 1996.

[28] Z. Zhang, J. Kurose, J. Salehi, and D. Towsley. Smoothing, statistical multiplexing and call admission control for stored video. *IEEE Journal on Selected Areas in Communications*, 13(6):1148–1166, August 1997.