High-Performance Prefetching Protocols for VBR Prerecorded Video

Martin Reisslein, GMD FOKUS
Keith W. Ross, Institute Eurecom

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 offering video on demand 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 percent link utilizations are achieved.

Our protocols are based on the observation that there are frequent periods of time during which the network bandwidth is underutilized. 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.
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 noncollaborative prefetching policy, can have packet loss several orders of magnitude higher than that of the discussed collaborative prefetching protocols for a wide range of buffer sizes.

In this article 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 [2], and a more detailed discussion of decentralized protocols can be found in [3].

This tutorial is organized as follows. In the following subsections we briefly review the literature on transmission schemes for VBR VoD. In the second section we discuss how the collaborative prefetching protocols can provide VoD over the cable plant. In the third section we focus on the special case where all video streams emanate from one server. We introduce the JSQ prefetching protocol, discuss some important refinements, and present simulation results. Furthermore, we discuss how the JSQ protocol handles viewer interactions such as pause, fast forward, and rewind. In the fourth section we introduce the decentralized prefetching protocol. It can be employed when the video streams emanate from multiple video servers. We discuss some of the protocol refinements and evaluate its performance through extensive simulations. We give some concluding remarks in the last section.

**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 (Fig. 1). The deterministic schemes without smoothing send the original VBR traffic into the network, and admission control ensures that the delays never exceed a prespecified limit [4–7]. For highly variable VBR traffic, these deterministic schemes typically require large initial delays to achieve moderate link utilizations [8]. The deterministic schemes with smoothing do not send the original VBR traffic into the 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 piecewise constant-rate transmission schedule takes advantage of the fact that some of the prerecorded video can be prefetched into the client buffer. (Prefetching is referred to as work-ahead in [14, 15].) 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 piecewise constant-rate transmission schedules [9–15]. It has been shown formally that, given a specific client buffer size, the Optimal Smoothing algorithm [14, 15] 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 utilization (> 90 percent) and consistently high responsiveness (less than 1 s) to interactivity.

For the probabilistic approaches, [16] 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 [17] and [18] 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 [18] 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 online as a function of the buffer contents at all the clients, and are therefore referred to as collaborative. Optimal smoothing is noncollaborative; the transmission schedule is computed before the transmission begins, and thus does not take the other ongoing connections into account. Collaborative prefetching protocols achieve nearly 100 percent link utilization, immediate responsiveness to viewer interactions. It is shown in [2, 3] that prefetching protocols with collaborative prefetching have substantially less packet loss than does optimal smoothing for the same link utilization.

**Prefetching and Residential Broadband Access**

In this section we discuss how the prefetching protocols for VoD discussed in this tutorial tie into cable modem technology. We begin with a brief overview of cable modem technology; for more details see [19–21].

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 since there is only one sender, the head-end. The upstream control traffic, however, poses a problem. Carrier sensing fails for cable
plexes and transmits the packets into its link; for simplicity, we 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 transmitted in one packet. (In our numerical work we assume for the following discussion that each video frame is 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, $p(j)$; this can be done through the recursion in Eq. (1) without communicating with the client. Through a similar recursion the server keeps track of the number of bits in the client buffers.

Because the videos are prerecorded, $(x_1(j), x_2(j), \ldots, x_M(j))$ is fully known at connection establishment.

When a client requests a specific video, the server makes an admission control decision 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 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 $1/F$ s 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 bits per second) 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, $N/F$ s 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 $1/F$ s. Let $p(j)$ be the number of frames in the prefetch buffer for connection $j$ at the beginning of slot $l$. Let $a(j)$ be the number of frames of connection $j$ that arrive to the prefetch buffer during the $l$th slot. At the end of each slot, one frame is removed from each prefetch buffer that has one or more frames. Thus, $p_{l+1}(j) = [p(j) + a(j)] - 1^*$,

$$\text{(1)}$$

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

During each slot of length $1/F$ s 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, $p(j)$; this can be done through the recursion in Eq. (1) without communicating with the client. Through a similar recursion the server keeps track of the number of bits in the client buffers.

The JSQ Prefetch Policy

The 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

---

Figure 2. VoD architecture for cable residential access.
where \( o(j^*) \) is the frame of connection \( j^* \) that is being considered for transmission. If this condition holds, 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 in Eq. 2 ensures that the server never transmits more than \( R/F \) bits in a slot of length \( 1/F \) s. Thus, there is never any loss at the bottleneck link (Fig. 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 first transmits 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 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 because 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 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.

Throughout the remainder of this article we use the original frame-based JSQ policy.

**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 the condition of Eq. 2 is violated; once it 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 Eq. 2 and finding a new \( j^* \) that minimizes \( p(j) \). If the Eq. 2 holds with the frame from the new connection \( j^* \), we transmit the frame, update \( p(j^*) \), and continue the procedure of transmitting by the connections that minimize the \( p(j) \)'s. Whenever a frame violates Eq. 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 that stops when Eq. 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 \( o(j^*) \) 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 Eq. 2 and the condition

\[
b(j^*) + x_{(j^*)} o(j^*) < B(j)
\]

are satisfied. Equation 3 ensures that the server

---

\(^1\) In defining this bit-based policy, we assume fixed-size packets.

---

**Table 1. An overview of the notation used in this article.**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a(j) )</td>
<td>Number of frames arriving to connections ( j )’s prefetch buffer during slot ( I )</td>
</tr>
<tr>
<td>( b(j) )</td>
<td>Number of bits in connection ( j )’s prefetch buffer</td>
</tr>
<tr>
<td>( B(j) )</td>
<td>Buffer capacity of client ( j ) in bits</td>
</tr>
<tr>
<td>( F )</td>
<td>Frame rate in frames per second</td>
</tr>
<tr>
<td>( j )</td>
<td>Number of ongoing video connections</td>
</tr>
<tr>
<td>( N )</td>
<td>Number of frames in videos</td>
</tr>
<tr>
<td>( p(j) )</td>
<td>Number of prefetched frames in connection ( j )’s prefetch buffer at the beginning of slot ( I )</td>
</tr>
<tr>
<td>( R )</td>
<td>Bandwidth of shared channel in bits per second</td>
</tr>
<tr>
<td>( x_{(j)} )</td>
<td>Number of bits in the ( n )th frame of video ( j )</td>
</tr>
<tr>
<td>( w_l )</td>
<td>Send window in slot ( I )</td>
</tr>
<tr>
<td>( \Delta w_l )</td>
<td>Send window increment in slot ( I )</td>
</tr>
</tbody>
</table>
number of bits in each frame. (We are aware that these are packets per second, summed over all connections in progress. We expect VoD systems in the future to have videos with 40 bytes of overhead; therefore, 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/s.

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 four movies that achieves 95 percent 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 \( N \)th 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 percent confidence interval is less than 10 percent 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 Fig. 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 percent utilization. By increasing the capacity only to 256 kbytes, the loss probability is reduced to \( 4.7 \times 10^{-6} \). By further increasing the buffer to 512 kbytes, no loss was observed for any of 8000 replications, corresponding to 3.2 \( \times \) 10^6 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 [23]).

Figure 4 demonstrates that, with a small prefetch buffer at each client, the JSQ prefetching policy will allow for almost 100 percent 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.

**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 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.

**Experimental Results**

In this subsection we present the results of a simulation study for the JSQ prefetch policy introduced in the previous subsection. Throughout we use the refined stopping rule discussed in the previous subsection. Our simulation study makes use of MPEG-1 encodings of the four movies in Table 2. 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/s. Each of the traces has \( N = 40,000 \) frames, corresponding to about 28 min. The mean number of bits per frame and the peak-to-mean ratio are given in Table 2, which also gives the average and peak bit rates. It can be argued that the average rate in bits per second 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 Mb/s. 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 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 percent confidence interval is less than 10 percent 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 Fig. 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 percent utilization. By increasing the capacity only to 256 kbytes, the loss probability is reduced to \( 4.7 \times 10^{-6} \). By further increasing the buffer to 512 kbytes, no loss was observed for any of 8000 replications, corresponding to 3.2 \( \times \) 10^6 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 [23]).

Figure 4 demonstrates that, with a small prefetch buffer at each client, the JSQ prefetching policy will allow for almost 100 percent 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.

**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 an 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 \( \delta \) frames into the future. If \( \delta < p(j) \), we discard \( \delta \) frames from the head of the prefetch buffer and set \( p(j) = p(j) - \delta \). If \( \delta < p(j) \), 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 \( p(j) = 0 \) 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 frequently be 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, since pauses can only improve performance; we furthermore assume that \( \delta > p(j) \) for all forward temporal jumps. Therefore, the presented results give 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 95 percent link utilization, with a delay assumption that the propagation and processing delays are negligible.

Decentralized Prefetching

We now return to the VoD architecture with multiple video servers shown in Fig. 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 \( I \) 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 \( x_j \) denote the number of bits in the \( j \)th frame. As in the previous section, 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 (i.e., every \( 1/F \) s), 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 \( R \) b/s. The maximal delay incurred in the multiplexer is therefore \( 1/F \) s. 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 \( \leq 1/F \) s. An alternative implementation with a larger buffer (which may be built in by the multiplexer manufacturer) is as follows. The server timestamps each frame 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 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 utilization is \( \leq 95 \) percent. The average link utilization is \( util = \frac{\sum_{j=1}^{J} x_{avg}(j) / R}{\sum_{j=1}^{J} x_{avg}(j)} \), where \( x_{avg}(j) \) is the average frame size in bits of the \( j \)th 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) [24, 25] 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 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 make efficient use of the buffers at the client and 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 \( 1/F \) s. 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 syn-

---

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.
buffer. Suppose that the client has a finite buffer capacity of $B$ frames.

The server keeps track of the number of frames in its buffer, let $a$ be the number of frames that are received and acknowledged by the client during slot $l$. Clearly, $0 \leq a \leq |w_j|$ if all frames sent are received by the client. If frames (or acknowledgments) are lost we have $a < |w_j|$. Figure 7 illustrates the timing of the prefetching protocol. There is no synchronization of slots between different servers. A new connection starts with a send window of $w_0 = 1$. The send window is increased by a small increment $\Delta w$, say 0.1, at the beginning of each slot (i.e., $w_j = w_{j-1} + \Delta w$). After computing the send window the server transmits $|w_j|$ frames (Fig. 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 received and acknowledged by the client during slot $l$. Clearly, $0 \leq a_l \leq |w_l|$. If all frames sent are received by the client, if frames (or acknowledgments) are lost we have $a_l < |w_l|$. Figure 7 illustrates the timing of the prefetching protocol. There is no synchronization of slots between different servers $j$ and $k$. 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:

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

Let $b_l$ be the number of bits in the client buffer at the beginning of slot $l$. The server keeps track of $b_l$ through a recursion similar to Eq. 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 $w_l = 1$. The loss of frames is indicative of acute link overloading, 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 (i.e., $w_0 = 1$). The window is increased by a small increment $\Delta w$ (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.

**Refinements of the Decentralized Prefetching Protocol**

**The Client Buffer Constraint** — 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

$$b_l + s_k \leq B$$ (5)

is satisfied. Equation 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:

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

Figure 8 illustrates this refined send window policy. When the client buffer is empty at the beginning of slot $l$ (i.e., when $b_l = 0$), the send window is incremented by $\Delta w_{\max}$. When the client buffer is full (i.e., 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 $\Delta w_l$ according to Eq. 6, calculates the new send window, $w_l = w_{l-1} + \Delta w_l$, and sends $|w_l|$ 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 [3]. Because of space limitation we give here only a brief qualitative discussion. When $\Delta w_{\max}$ is large, $\Delta w$ gives large increments and thus allows the server to send more frames. The parameter $\Delta w_{\max}$ has to be large enough to allow 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 still has 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, $\Delta w_{\text{max}}$ values between 2 and 8 and $e$ values between 4 and 10 give good performance [3]. We choose $\Delta w_{\text{max}} = 5$ and $e = 6$ for the numerical work in this article.

**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 $w_1$ frames of the video immediately after the request of the client has been processed. Subsequent transmissions are scheduled $l/F$ s, $l = 1, \ldots, N - 1$, 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 $t_1 \gg t_2$ depicted in Fig. 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 $R_l$ 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 $R_l$ bits into the multiplexer buffer. With the depicted phase alignment ($t_1 \gg t_2$), connection $k$ has clearly a disadvantage since it can transmit only $R_l$ 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 $w_1$ frames of the video immediately after the request of the client has been processed. The server draws a random phase $b_t$, $t = 1, \ldots, N - 1$ from a uniform distribution over $[-1/2F, 1/2F]$ in each frame period. The subsequent transmissions are scheduled $l/F + b_t s$, $s = 1, \ldots, N - 1$ after the initial transmission. With this transmission rule, the slot phases are constantly reshuffled. Unfair phase alignments therefore cannot persist for extended periods of time.

**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 earlier. Again 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 $R = 45$ Mb/s; the corresponding multiplexer buffer holds 234,375 bytes ($= R/F$).

In each realization of the simulation we generate random starting frames as described earlier. For each replication of the simulation we also draw random (nonsynchronized) slot phases $t(j)$ for each of the $J$ connections. The $t(j)$s are independent and are drawn from a uniform distribution over $[0, l/F]$. 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 (Fig. 7), that is, server $j$ puts its traffic into the queue at instants $t(j) + (l - 1)F$, $l = 1, \ldots, N$ (Fig. 11). $b_t = 1, \ldots, N$ with randomized transmission).

In all our simulations we assume that all clients have the same buffering capacity of $B$ bits.

In Figs. 10, 11, and 12 we show typical plots of the client buffer contents, $b_t$, the window increment, $\Delta w$, and the send window, $w_{2t}$, 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 $B = 1$ Mb. We employ the dynamic window policy without randomized transmission with $\Delta w_{\text{max}} = 5$ and $e = 2$. The plots illustrate how the client buffer contents control the window increment. The left part of Fig. 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 Fig. 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 $\Delta w_{\text{max}} = 5$. This allows connection 2 to transmit very aggressively. We observe from Fig. 12 that the send window becomes as large as 17.9 in time slot 41,363. Figure 13 shows that the first 14 of the 17 frames sent in time slot 41,363 can be accommodated by the multiplexer buffer; the remaining three frames are lost. The send window is therefore reset to $w_{2t} = 1$ at the end of slot 41,363. At the beginning of slot 41,364 the new send window is computed as $w_{2t} = w_{2t} + \Delta w_{t+1} = 1 + 2.9 = 3.9$, allowing connection 2 to send three frames. We see from Fig. 13 that all three frames fit into the multiplexer buffer, and the send window increases to $w_{2t} = 3.9 + 3.1 = 7.0$ in slot 41,365. The right part of Fig. 10 shows that the large send windows enable connection 2 to fill its client buffer again. By time slot 41,380 the client buffer is filled to 90 percent 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 percent 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 $\Delta w_{\text{max}} = 5$ and $e = 6$. The figure shows that the basic window policy has unacceptably high losses. The loss probability is about $8 \times 10^{-3}$ 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 buffer is about $1.5 \times 10^{-5}$. By allowing a short startup latency of two
frame periods we can further reduce the loss probability significantly (see [3] for details).

In Fig. 15 we compare the decentralized prefetching protocols with JSQ prefetching and optimal smoothing [14, 15, 18]. The plot gives the loss probability as a function of the client buffer size for 95 percent link utilization. The optimal smoothing curves are obtained by applying the optimal smoothing algorithm [14, 15, 18] 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 Mbit/s link with the large deviation approximation [16, 18, 26, 27]. We do this for two versions of optimal smoothing: no initiation delay and a 10-frame initiation delay [14, 15, 18]. The decentralized prefetching results are for the dynamic window policy with randomized transmission and two frames startup latency. The JSQ prefetching results are from Fig. 4. The collaborative prefetching protocols clearly outperform optimal smoothing, which has unacceptable high loss probabilities for all buffer sizes shown. For 1 Mbyte of client buffer the loss probability for decentralized prefetching is over two 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 [28].

**Conclusion**

Prerecorded video has two special properties:
- For each video, the traffic in each video frame is known before the video session begins.
- While the video is being played, some of the video can be prefetched into the client memory.

In this article 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 LAN. The presented client–server protocols can be part of a larger

---

**Figure 11.** Window increment \( \Delta W \) versus slot time; \( \Delta W \) is computed according to the dynamic window policy with \( \Delta W_{\text{max}} = 5 \) and \( c = 2 \), that is, \( \Delta W_t = 5(1 - b_t/1 \text{ Mbit})^2 \).

**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.
Internet solution to VoD, whereby the prerecorded videos are multicasted to local servers at off-peak hours with best-effort service.

We conclude by mentioning that an alternative approach for the delivery of the video traffic is to use TCP [24, 25]. 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 data transfer is limited by TCP's congestion and receive windows. The client removes one frame from its receive buffer every 1/F s 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. Second, 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 compare the prefetching protocols to TCP, and adapt the prefetching protocols so that they can run on top of TCP. These issues are the subject of ongoing research.

References


Biographies

MARTIN REISSLEIN (reisslein@fokus.gmd.de) is a scientist with the German National Research Center for Information Technology (GMD FOKUS), Berlin. He received the Dipl.Ing (FH) degree from the Fachhochschule Darmstadt, Germany, in 1994, and the M.S.E. degree from the University of Pennsylvania, Philadelphia, in 1996, both in electrical engineering. He received his Ph.D. in systems engineering from the University of Pennsylvania in August 1998. During academic year 1997-98 he visited the University of Pennsylvania as a Fulbright scholar. His research interests include multimedia networking and stochastic modeling. He is particularly interested in traffic management schemes for the efficient transport of multimedia traffic with statistical QoS.

KEITH W. ROSS (ross@eurecom.fr) is a professor and chair of the Multimedia Communications Department at Institute Eurecom, Sophia Antipolis, France. From 1985 to December 1997 he was with the University of Pennsylvania as assistant, associate, and full professor. He received his B.S. from Tufts University (1979) in Electrical Engineering, his M.S. from Columbia University (1981) in electrical engineering, and his Ph.D. from the University of Michigan (1983) in computer information, and control engineering. He has published over 40 papers in leading journals and has published a book on multiservice loss models for broadband telecommunication networks. He is currently writing a multimedia textbook on Internet protocols and data networks.