Sharing Multicast Videos Using Patching Streams

Ying Cai *Kien A. Hua*

School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816-2362
U. S. A.

E-mail: {cai, kienhua}@cs.ucf.edu

August 11, 2001

Abstract

The access patterns of most information systems follow the 80/20 rules. That is, 80% of the requests are for 20% of the data. A video server can take advantage of this property by waiting for requests and serving them together in one multicast. This simple strategy, however, incurs service delay. We address this drawback in this paper by allowing clients to receive the leading portion of a video on demand, and the rest of the video from an ongoing multicast. Since clients do not have to wait for the next multicast, the service latency is essentially zero. Furthermore, since most services require the server to deliver only a small leading portion of the video, the server can serve many more clients per time unit. We analyze the performance of this approach, and determine the optimal condition for when to use this strategy. We compare its performance to a hardware solution called *Piggybacking*. The results indicate that more than 200% improvement is achievable.

KEYWORDS: Multimedia communications, multicast, patching, video on demand, service latency, performance study.

1 Introduction

Video on Demand (VOD) [1] is a basis technology for emerging multimedia applications, such as home entertainment, news on demand, digital libraries, distance learning, company training, electronic commerce, to name but a few. A typical VOD service allows remote users to play back any video from a large collection of videos stored on one or more servers. In response to a service request, a server delivers the video to the user in an isochronous data stream. The number of concurrent streams a server can support depends on its bandwidth. This fact severely limits the throughput of today's video servers. For instance, if four OC-12 NICs are used for a video server, the aggregated bandwidth of 2.4-Gbps (4×622 Mbps) constrains the video delivery capability to no more than 600 concurrent MPEG-2 streams. This number is significantly less in practice due to overhead associated with storage access delay, operating system overhead, etc.

To support large-scale video applications, we can employ many servers geographically separated from each other and interconnected in some hierarchy. Maintaining multiple data centers, however, is expensive; and the degree of distribution should be minimized to save the operational costs. This can be achieved by allowing clients to share multicast data. Several periodic broadcast techniques (e.g., [2],[3],[4],[5]) have been developed for this purpose. These schemes divide the server bandwidth into a large number of *logical channels* with equal bandwidth. To broadcast a video over K dedicated channels, the video file is partitioned into K fragments of increasing sizes, each repeatedly broadcast on its own channel. On the receiving ends, the playback and download mechanisms are carefully arranged so that before the current fragment is consumed, the next fragment is always accessible to the client. Since the size of the first fragment can be made small, low service latencies can be achieved. This approach is very cost-effective; but it can only be used for very popular videos.

To support videos of diverse popularity, we can delay service requests and wait for more to arrive for the same video during the batching interval [6, 7, 8]. All these requests can then be served in one multicast using only one server stream. This strategy substantially reduces the demand on server bandwidth. However, it can only provide near on-demand services [9]. It also has to deal with the following dilemma:

- Users making the early requests are likely to cancel their requests if they are kept waiting too

long.

- On the other hand, if we keep the waiting times short, the benefit of multicast diminishes.

To overcome the above problems, three novel ideas have been proposed:

- **Chaining**: In this scheme [10], each client machine reserves a small amount of its disk space for caching purposes. When bandwidth becomes available, the server initiates a multicast to serve all the pending requests currently waiting for some video. As these users play back the video, they cache the data in their disk buffers according to a FIFO replacement policy. Thus, the first data block of the video remains in a buffer until the cache is full. Before that happens, any new requests can be serviced from these clients creating the "second generation" of a multicast hierarchy. That is, the second-generation users are "chained" to the first. Similarly, the third generation can be chained to the second, and so forth. This process continues until the first data block of the video is dropped from the multicast hierarchy due to the cache replacement policy. After that happens, the server needs to initiate another multicast hierarchy to serve the next request. We observe that a multicast hierarchy can grow to be very big, yet uses only one server stream. This strategy leverages network bandwidth to reduce the demand on server bandwidth. Furthermore, since a multicast hierarchy can grow dynamically to accommodate new requests as soon as they arrive, true on-demand services can be achieved. A similar concept has also been used to allow clients to share their local cache in a collaborating environment [11].
- **Caching Multicast Protocol** (CMP): This scheme [12] enables nodes to receive the entire multicast although they may subscribe to the multicast at different times. This is achieved by allowing the routers, on the multicast tree, to intercept and cache a video stream passing through. These data can then be relayed to subsequent subscribers to further extend the multicast tree. This strategy significantly reduces the number of multicasts necessary, and therefore lowers the network resource requirements.
- **Piggybacking**: This approach [13, 14] alters the playback rates of the on-going services for the purpose of "merging" their respective video streams into a single stream that can serve the entire group of merged requests. As an example, let us consider a client which is being served by stream s_i. Sometime later, the server dispatches stream s_j to serve a new request for the same video. At

this time, the server slows down the data rate on s_i, and speeds up that of s_j. When s_j catches up with s_i, they are merged into a single stream freeing one of the two streams.

A drawback of Chaining is that the network is flooded with multicast data from both clients and servers. As a result, the network cost is relatively high. Although CMP addresses this problem, it requires the use of CMP-capable routers. Further research is necessary to adapt this idea for the IP environment. Piggybacking assumes that specialized hardware is available to adjust the playback and communication rates dynamically. The cost of this hardware can be significant. Furthermore, the number of data streams can be merged is severely limited by the adjusting rate which must be kept small to preserve the visual and audio quality of the playback.

In this paper, we present a different approach to sharing multicast data using patching streams. The basic idea is to allow a client node to join an existing multicast for the remainder of the video, and download the missed portion over a dedicated *patching stream*. Since Patching is an application-level solution, it does not require specialized hardware and is compatible with existing environments. Our performance results indicate that Patching outperforms Piggybacking by more than 200%.

The remainder of this paper is organized as follows. We describe Patching in details in Section 2. The optimization of patching performance is discussed in Section 3. We present our performance studies in Section 4. Finally, we give our concluding remarks in Section 5.

2 Patching

Since the bandwidth of a video server determines the number of video streams it can support simultaneously, server bandwidth can be viewed and organized as a finite set of logical *communication channels*. To provide admission control, the server maintains a count on the number of available channels. This count is decremented when a channel is dispatched for a new service, and incremented when a service terminates. As long as this count is not zero, the server can admit more pending requests.

Under Patching, most of the server bandwidth is organized into a set of logical channels, each capable of transmitting video data at the playback rate. The remaining bandwidth is used for control messages such as service requests and service notifications. Each communication channel is used to

either multicast a video in its entirety called a *regular multicast*, or to multicast only the leading portion of a video called a *patching multicast*. When a channel is used for a regular multicast, it is said to play the role of a *regular channel*; otherwise, it is referred to as a *patching channel*. If a client station tunes to a regular channel to download data, the data stream arriving at the client's communication port is called a *regular stream*. On the other hand, if the source of the data stream is a patching channel, it is called a *patching stream*. We describe the details of the Patching technique in the following subsections. We first review scheduling techniques for video multicast. We then discuss the server design, and how the client receives and plays back videos under Patching.

2.1 Scheduling Video Multicasts

Associated with each communication channel is a *client list* which contains the ID's of the client stations currently viewing the broadcast on this channel. When a channel completes its current multicast, its client list is reset. The channel becomes *free*, and is available for another multicast. Several scheduling techniques have been developed for multicasting videos. These schemes maintain a waiting queue for the pending requests. Those waiting for the same video are referred to as a *batch*. The task of a multicast scheduler is to select a batch for the next service. Some of these techniques are as follows.

- *First Come First Served* (FCFS) [15]: When a communication channel becomes free, this policy selects the video with the oldest pending request (which has been waiting for the longest time) to be served next. The advantage of this scheme is its fairness - videos are treated equally regardless of their popularity. This scheme, however, is not optimized for system throughput.

- *Maximum Queue Length First* (MQL) [15]: Unlike FCFS, this policy is designed to maximize the system throughput. When a free channel becomes available, this technique selects the video with the most number of pending requests to be served next. Obviously, this strategy is unfair since it favors more popular videos.

- *Maximum Factored Queue length first* (MFQ) [7]: This scheme improves on the FCFS and MQL by taking into account both the waiting times of the requests and the popularity of the videos. When a channel becomes free, MFQ selects the video with the most number of pending requests

weighted by the *best* factor, (access frequency)$^{-\frac{1}{2}}$, to be served next. This scheme can achieve throughput close to that of MQL with fairness near that of FCFS.

Any of the above scheduling policies can be used with Patching. When a free channel becomes available, one of the above techniques can be used to admit a batch of pending requests for some video. This is done by inserting these clients into the client list associated with the free channel. The patching mechanism then has to decide whether to perform a regular multicast or a patching multicast of the video. When the multicast is finally activated, the desired video segment is multicast on that channel to the clients in the client list.

2.2 Server Design

The server main routine is presented in Figure 1. We explain it as follows. In this paper, we use $v[p]$ to denote the first p minutes of some video v. The client buffer size is B minutes. That is, each client has enough buffer space to cache up to B minutes of video data. To request a video, a client sends a request token *(clientID, VideoID)* to the server, where *ClientID* is the address of the client and *VideoID* is the ID of the requested video. When a channel becomes free and there is at least one pending request, the main routine is called to dispatch a free channel for the next service. This free channel is used to deliver a regular multicast if there is no existing regular multicast currently serving the requested video. On the other hand, if there is one such regular multicast, the free channel is used to deliver a patching multicast if the elapsed time, $skew$, since the beginning of that regular multicast is less than a predetermined threshold, $W(v)$, called *patching window*. We will derive the optimal value for this threshold in the next section. The patching multicast delivers the missed portion of v (i.e., $v[skew]$) if the clients have enough buffer space (i.e., $skew \leq B$) to cache the regular stream while they are playing back the patching stream; otherwise, the patching multicast delivers the entire video except the last B or $|v| - skew$ minutes whichever is smaller (i.e., $v[|v| - min(B, |v| - skew)]$). Once the service is ready, the server notifies the relevant clients with a service token, *(PID, RID)*, where PID and RID are the IDs of the patching channel and the regular channel, respectively. We note that if PID is null, the clients expect to receive the entire video from a new regular multicast on RID; otherwise, they need to receive the first part of the video from a patching multicast on PID and the remainder of the video

from an existing regular multicast on RID.

Algorithm: *Server Main Routine*

$\|v\|$:	playback duration of the video v
$W(v)$:	the size of patching window for video v
B:	client buffer size in time unit
$LastRegular$:	The channel ID of the last regular multicast of video v
$LastRegularTime$:	The start time of the last regular multicast of video v
$CurrentTime$:	current time

1. Select a video, say v, to serve according to a given scheduling policy (e.g., FCFS [15], MQL [15], MFQ [7], etc.).
2. Initialize a service token as (PID=*null*, RID=*null*) and dispatch a free channel, say *FreeChannel*.
3. If none of the existing regular multicasts currently serving v, then set RID = *FreeChannel* and go to Step 6.
4. Compute $skew = LastRegularTime - CurrentTime$.
5. If $skew > W(v)$ then set RID = *FreeChannel*; otherwise, do the following:
   - Set RID = *LastRegular* and PID = *FreeChannel*.
   - The workload for the free channel is determined as follows:
     - If $skew \leq B$, the workload is $v[skew]$;
     - Otherwise, the workload is $v[|v| - min(B, |v| - skew)]$.
6. For each request token (ClientID = *vClient*, VideoID = v) in the job queue, do the following:
   - Append *vClient* to the client list of *FreeChannel*.
   - If PID is not null, append *vClient* to the client list of channel *LatestRegular*.
   - Send the service token to notify the client *vClient*.
   - Delete the request token from the job queue.
7. Activate the multicast on *FreeChannel*.

Figure 1: Algorithm for Video Server

In conventional batching techniques, each channel multicasts the video in its entirety, and is held up for the entire duration of the video playback. This characteristic incurs long service delay and severely limits the system throughput. An important improvement of the patching technique is the substantial increase in the channel throughput. That is, each channel can serve many more batches per time unit. This is accomplished by using the channels mostly to deliver only the small leading portion of a video. The average turnaround time of the channels, therefore, is very short allowing them to serve more

clients per time unit. The small turnaround time also helps to reduce service latency. As an example, let us consider the following scenario. After channel C_i has multicast a video to batch B_i for three minutes, another channel, say C_j, becomes free and is used to serve a new batch B_j which requested the same video. Let the length of the video be 60 minutes. If we use conventional batching, C_j will be busy for the next 60 minutes serving B_j. In contrast, Patching enables the clients in B_j to buffer the stream broadcast on C_i while playing the new start-up flow broadcast on the patching channel C_j. After three minutes, when the catch-up flow has been played back to the skew point, C_j can be released and the original multicast on C_i can now be shared by both batches B_i and B_j. We observe that C_j is held up for only three minutes, compared to 60 minutes under conventional batching. The throughput of channel C_j, therefore, can be greatly improved.

2.3 Client Design

To implement patching, a client station needs to have three threads of control: two data loaders L_p and L_r, and a video player *VideoPlayer*. L_p and L_r are responsible for downloading data from the patching channel and the regular channel, respectively, and writing them to a local buffer. $VideoPlayer$ fetches data from this buffer, reassembles the video frames, and renders them onto the screen.

To request a video, a client sends a request token $(ClientID, VideoID)$ to the server, and waits for the service token (PID, RID) to arrive. When the client receives the service token, it examines the token; and two scenarios can happen:

1. If PID is null, the server is about to start a new regular multicast of the requested video on channel RID. In this case, the client needs to activate only loader L_r to receive the entire video from RID. As the data arrives at the client, the $VideoPlayer$ renders the video frames onto the screen.

2. If PID is not null, the server is about to do a patching multicast on channel PID. The client must also tune into channel RID for the remaining portion of the video. In this case, the client needs to activate both loaders L_r and L_p to simultaneously download data from RID and PID, respectively. Initially, the $VideoPlayer$ plays back the patching stream as the data arrive at the client. The regular stream arriving from RID is temporarily cached in the local buffer. When

the patching multicast ends, the *VideoPlayer* switches to play back the data in the local buffer as L_r continues to download the remainder of the video file.

We show in Figure 2 an example to illustrate the patching idea. Clients A, B and C are sharing a multicast although they are in different stages of the video playback. Client A arrived first. It has been served entirely by a regular stream. Client B arrived next. Its video player has exhausted the patching stream, and is currently playing back the regular stream cached in the local buffer. Client C arrived most recently. It is still playing back the patching stream as the regular stream is being cached in the local buffer.

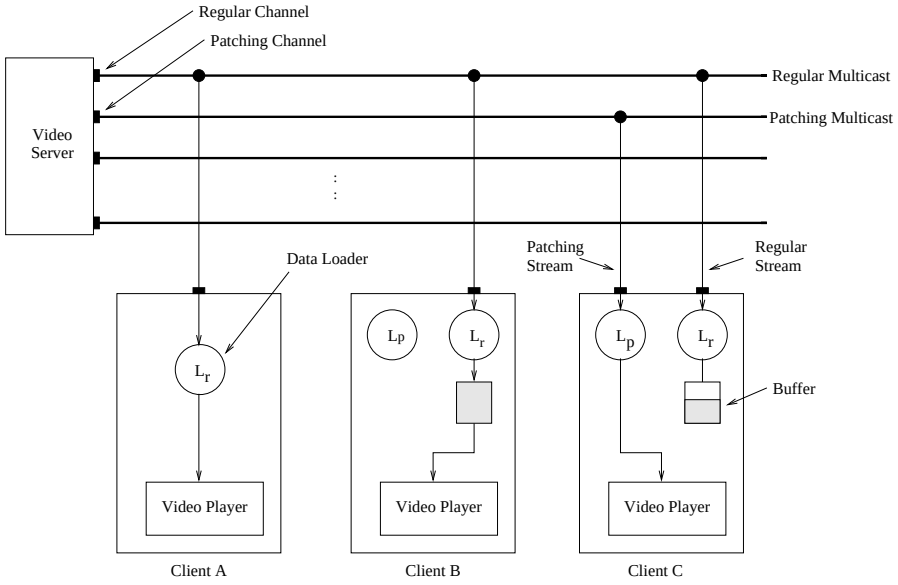

Figure 2: Patching Technique

We present the client routines in Figure 3. We note that data from the patching channel and the regular channel are first buffered in the *PatchBuffer* and *RegularBuffer*, respectively. The data downloaded to the *PatchBuffer* are immediately piped to the *VideoPlayer*. Although the *RegularBuffer* may accumulate data, the cost for this storage space is negligible. As an example, 150 Mbytes of disk space can buffer about 5 minutes of MPEG-2 video. Such a disk space costs less than $5 today, and is commonly available in today's workstations and set-top boxes. We note that client buffer is also used to implement VCR functions [16, 17, 18]. The same buffer space can be used to support patching at no additional cost.

In terms of communication costs, Patching requires each receiving end to have enough bandwidth

Algorithm: *Client Main Routine*

1. Send a request token ($ClientID$, $VideoID$) to the video server.
2. Wait until the service token ($PatchingID$, $RegularID$) from the server arrives.
3. If *PatchingID* is not null, start the data loader L_p.
4. Start the data loaders L_r.
5. Start the video player *VideoPlayer*.

Algorithm: *Loader* L_p /* for patching stream

1. Do the following until no more data arrive on the patching channel *PatchingID*:
   - Download one data packet on channel *PatchingID*;
   - Store the data packet to *PatchBuffer*.
2. Terminate L_p.

Algorithm: *Loader* L_r /* for regular stream

1. Do the following until no more data arrive on the regular channel *RegularID*:
   - Download one data packet on channel *RegularID*;
   - Store the data packet to *RegularBuffer*.
2. Terminate L_r.

Algorithm: *VideoPlayer*

1. Do the following until no more data in *PatchBuffer*
   - Fetch one playback unit from *PatchBuffer*;
   - Free the disk space for the fetched data;
   - Reassemble the fetched data into frames and render them onto the screen.
2. Do the following until no more data in *RegularBuffer*
   - Fetch one playback unit from *RegularBuffer*;
   - Free the disk space for the fetched data;
   - Reassemble the fetched data into frames and render them onto the screen.
3. Terminate *VideoPlayer*.

Figure 3: Algorithms for client stations

to download data from two channels simultaneously. This requirement is readily satisfied with today's local distribution networks such as DSL and cable modems. DSL achieves speeds currently of 8 Mbps in one direction, and eventually speeds as high as 50 Mbps. Cable modem is another technology getting popular. To ensure that each house is allocated enough bandwidth, the head-ends can be moved deeper into the neighborhoods so that each cable runs past fewer number of houses.

3 Optimizing Patching Performance

Given that a regular multicast of a video is in progress, when to schedule the next regular multicast for the same video is critical to the overall performance of a patching system. We recall that the period after a regular multicast, during which patching multicasts are used, is referred to as the *patching window*. If this window is set too small, there are too many regular multicasts reducing their efficiency. To an extreme, the patching window can be narrowed to zero, in which case all multicasts are of type regular, each serves only one client. Under this condition, Patching degenerates into the conventional VOD scheme which uses a dedicated channel for each service request. On the other hand, if the patching window is made too wide, the average time distance between a patching multicast and the last regular multicast becomes too large. Most patching streams have to carry a heavier load causing longer turnaround times for the channels. We recall that a shorter turnaround time allows the channels to be more productive and therefore more services done per time unit.

In our preliminary study [19], we considered two straightforward approaches to determine the patching window size.

- **Greedy Patching**: The patching window is the entire length of the video. That is, no regular multicast is initiated as long as there is an existing multicast for the video. The name "Greedy Patching" alludes to the fact that the server tries to share each regular multicast as much as possible. When the buffer space of B minutes is not enough to absorb the skew as described in Section 2, the client uses this storage space to buffer the remainder or the last B minutes of the regular stream, whichever is shorter.

- **Grace Patching**: Overly greedy can actually result in less data sharing [19]. Grace Patching uses a patching multicast for the new clients only if it has enough buffer space to absorb the skew

as described in Section 2. Hence, under Grace Patching, the patching window is determined by the client buffer size.

Setting the size of the patching window according to the video length or the client buffer size [19, 20] does not always result in good performance. The popularity of the video must also be taken into account. In this section, we develop a mathematical model to determine the optimal patching window for a given video.

We note that an alternative to Greedy Patching is to reuse the buffer space in a given service [21]. This scheme is best explained using an example. Let us say, the client can buffer only 5 minutes of video data; but the most recent regular stream started 6 minutes ago. In this case, a patching stream delivers to the client the first 6 minutes (from the 1st minute to 6th minute) of the video. As the client plays back these 6 minutes, it buffers the regular stream until the buffer is full. When the playback of the patching stream is finished, the client continues to play back the next five minutes (from the 7th minute to 11th minute) from the buffer. When this is done, another patching stream sends the next 6 minutes of the same video (from the 12th minute to 17th minute) to the client. As it plays back this patching stream, the client buffers the next 5 minutes from the regular stream. This pattern is repeated until the end of the video. We observe that this scheme relies heavily on patching streams. They account for one half of the total data transmission in the above example. Since patching streams are not shared in this environment, this technique is not efficient in sharing computing and network resources. This scheme is also quite complex because an index must be built for each video to allow starting a patching stream from anywhere in the video. Furthermore, the client and the server must synchronize the many patching streams with the regular stream in a given service. Due to these disadvantages, we will not consider this strategy in this paper.

3.1 An illustrative Example

To gain some insight on the effect of the patching window size on patching performance, let us consider the following example. The length of the video is 30 minutes. Each client has enough disk space to buffer up to 15 minutes of video. The service request rate is one per minute. Thus, the server must decide at each minute whether to initiate a patching or regular multicast. This decision is influenced by the patching window size. We compare three different window sizes in the following in terms of the

amount of data transmission required to serve the first 30 service requests:

- **Greedy Patching**: Assuming that a regular multicast, say S_0, is initiated at time 0, the next 29 multicasts, S_1, S_2, ..., and S_{29}, must be patching type. The total amount of data transmitted for the first 30 users can be computed as follows:
  - S_0 delivers 30 minutes of data.
  - For $1 \leq i \leq 29$, S_i delivers i minutes of data.

  Thus, the server has to deliver a total of $30 + \sum_{i=1}^{29} i = 465$ minutes of video data for the first 30 services. We note that this scheme is not very efficient since a large number of clients can share only the last portion (less than 15 minutes) of the regular multicast.
- **Grace Patching**: Since the size of the client buffers is 15 minutes, the patching window for this example is 15 minutes. S_0 and S_{16} must be regular multicasts. Together, they deliver $2 * 30 = 60$ minutes of data. The other multicasts are patching type. They deliver a total of $\sum_{i=1}^{15} i + \sum_{i=1}^{13} i = 211$ minutes of data. In total, the server delivers $60 + 211 = 271$ minutes of data for first 30 services.
- **Patching Window is Five Minutes**: Under this setting, we have five regular multicasts, S_0, S_6, S_{12}, S_{18}, and S_{24}. They deliver a total of $5 * 30 = 150$ minutes of data. The other multicasts are patching type. Together, they deliver $5 * \sum_{i=1}^{5} i = 75$ minutes of video data. Thus, the total amount of video data transmitted for the first 30 services is 225 minutes.

In the above example, Greedy and Grace use a patching window of 30 minutes and 15 minutes, respectively. Although Grace Patching is substantially better than Greedy Patching, reducing the patching window size further to 5 minutes improves the performance by another 20%. The question is, what is the optimal patching window size? To answer this question, we analyze the effect of patching window size on server bandwidth requirement in the following subsection.

3.2 Optimal Patching Window

We say that a patching window is optimal if it results in minimal requirement on server bandwidth. Using the optimal patching window ensures that the system cost is minimal. To determine this optimal value, we derive in this subsection a mathematical formula to capture the relationship between the

patching window size and the required server bandwidth. In our analysis, we will refer to the amount of data in terms of time units. For instance, we say that the amount of data is five minutes if it takes five minutes to playback that amount of data. We also use the following notations in our discussion:

- D_t: The amount of data delivered by the multicast initiated at some time t.
- $D_{(t_1,t_2]}$ The mean total amount of data delivered by the multicasts initiated during the time period $t_1 < t \leq t_2$.
- $D_{[t_1,t_2]}$: The mean total amount of data delivered by the multicasts initiated during the time period $t_1 \leq t \leq t_2$.
- $P(k,t)$: The probability of having k multicasts initiated over t time units.

In this paper, a regular multicast and all the subsequent patching multicasts for the same video initiated before the next regular multicast are said to form a *multicast group*. Our strategy for computing the server bandwidth requirement is as follows:

- We first determine the mean total amount of data, D, transmitted by a multicast group.
- We then calculate the average time duration τ of a multicast group.
- The server bandwidth requirement can then be computed as $\frac{D \cdot b}{\tau}$, where b is the video playback rate.

Without loss of generality, we reset the time to zero whenever a regular multicast is initiated. That is, a multicast group always starts at time zero. Given a video v, we compute the mean total amount of data transmitted by a multicast group under various window sizes $W(v)$ as follows.

1. $\underline{W(v) = 0}$: Under this condition, Patching becomes the traditional batching technique; and all the multicasts are regular type, each forms a group by itself. Since a regular multicast delivers the video in its entirety, the mean total amount of data delivered by a multicast group is

$$D_0 = |v|, \tag{1}$$

where $|v|$ denotes the video length.

2. $0 < W(v) \leq B$: We recall that B is the size of the client buffer. In this case, we need only focus on the interval $[0, W(v)]$ because multicasts initiated after time $W(v)$ belong to another multicast group. The amount of data delivered by a patching multicast initiated at time t, $0 < t \leq W(v)$, is t time units. If k patching multicasts are initiated between t and $t + \Delta t$, the amount of data delivered by these k multicasts can be approximated as $k \cdot t$, if Δt is small enough. Since the probability of initiating k multicasts during a time interval of Δt is $P(k, \Delta t)$, the total amount of data delivered by the multicasts initiated between t and $t + \Delta t$ is $\sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t)$. To calculate the mean total amount of video data delivered by a multicast group, we can partition $(0, W(v)]$ into $\lfloor \frac{W(v)}{\Delta t} \rfloor$ small time segments. Then we have

$$D_{[0,W(v)]} = D_0 + \sum_{t=1}^{\lfloor \frac{W(v)}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t) \tag{2}$$

We note that the first term, D_0, is due to the regular multicast which leads the multicast group.

3. $B < W(v) \leq |v| - B$: We focus on two time intervals $[0, B]$ and $(B, W(v)]$. Again, we do not need to concern with the multicasts initiated after time $W(v)$ because they belong to another multicast group. The amount of data transmitted during the first interval $[0, B]$ (i.e., $D_{[0,B]}$) can be computed using Equation 2. For a multicast initiated at time t during the second time interval, since the client buffer is not large enough to cache the next t time units of the video, the client caches the last B time units of the regular multicast instead. In this case, the patching multicast must transmit $|v| - B$ amount of data. Let k denote the number of patching multicasts initiated during the second period $(B, W(v)]$. The total amount of video data delivered by these multicasts is $k \cdot (|v| - B)$. Since the probability of initiating k patching multicasts during this time interval is $P(k, W(v) - B)$, the total amount of data transmitted during $(B, W(v)]$ is $\sum_{k=1}^{\infty} k \cdot (|v| - B) \cdot P(k, W(v) - B)$. Thus, the mean total amount of data transmitted by the multicast group is:

$$D_{[0,W(v)]} = D_{[0,B]} + \sum_{k=1}^{\infty} k \cdot (|v| - B) \cdot P(k, W(v) - B) \tag{3}$$

4. $|v| - B < W(v) \leq |v|$: As in the third case, we need only focus on two time intervals, $[0, |v| - B]$ and $(|v| - B, W(v)]$. The mean total amount of data transmitted during the first time interval, $D_{[0,|v|-B]}$, can be computed using Equation 3. The mean total amount of data transmitted during

the second time interval is computed as follows. A multicast initiated at time t in this period needs to deliver t time units of data. Let k denote the number of multicasts initiated between t and $t + \Delta t$, the amount of data delivered by these k multicasts is $k \cdot t$, if Δt is small enough. If we partition the second time period into $\lfloor \frac{W(v)-|v|+B}{\Delta t} \rfloor$ small segments, then the amount of data delivered by the multicasts initiated during the ith time segment can be approximated as $\sum_{k=1}^{\infty} k \cdot (|v| - B + i \cdot \Delta t) \cdot P(k, \Delta t)$, if Δt is small enough. Finally, we have

$$D_{[0,W(v)]} = D_{[0,|v|-B]} + \sum_{t=1}^{\lfloor \frac{W(v)-|v|+B}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot (|v| - B + t \cdot \Delta t) \cdot P(k, \Delta t) \tag{4}$$

If we make Δt equal to 1 second and keep the precision of $W(v)$ to second, Equations 1, 2, 3, and 4 become:

$$D_{[0,W(v)]} = \begin{cases} |v| & \text{if } W(v) = 0, \\ D_0 + \frac{W(v) \cdot (W(v)+1)}{2} \cdot \sum_{k=1}^{\infty} k \cdot P(k,1) & \text{if } 0 < W(v) \leq B, \\ D_{[0,B]} + (|v| - B) \cdot \sum_{k=1}^{\infty} k \cdot P(k, W(v) - B) & \text{if } B < W(v) \leq |v| - B, \\ D_{[0,|v|-B]} + \frac{(W(v)-|v|+B)(W(v)+|v|-B+1)}{2} \cdot \sum_{k=1}^{\infty} k \cdot P(k,1) & \text{if } |v| - B < W(v) \leq |v|. \end{cases}$$

In this paper, we assume that the arrival of the service requests follows a Poisson process with rate λ [1]. Thus, the multicast initiation process also follows the same Poisson process to ensure no service delay. The probability density function is $f_t = \lambda e^{-\lambda x}$, for $x \geq 0$, where t is the random variable representing the time interval of two successive multicast initiations. Under this assumption, $P(k,t) = \frac{(\lambda t)^k e^{-\lambda t}}{k!}$ and $\sum_{k=1}^{\infty} k \cdot P(k,t) = t \cdot \lambda$. Thus, the formula for $D_{[0,W(v)]}$ can be simplified as follows.

$$D_{[0,W(v)]} = \begin{cases} |v| & \text{if } W(v) = 0, \\ D_0 + \frac{W(v) \cdot (W(v)+1)}{2} \cdot \lambda & \text{if } 0 < W(v) \leq B, \\ D_{[0,B]} + (|v| - B) \cdot (W(v) - B) \cdot \lambda & \text{if } B < W(v) \leq |v| - B, \\ D_{[0,|v|-B]} + \frac{(W(v)-|v|+B)(W(v)+|v|-B+1)}{2} \cdot \lambda & \text{if } |v| - B < W(v) \leq |v|. \end{cases} \tag{5}$$

Since the multicast initiation rate is λ, the mean duration of a multicast group is $W(v) + \frac{1}{\lambda}$. Thus, the required server bandwidth can be computed as follows.

$$Server\_Bandwidth = \frac{D_{[0,W(v)]}}{W(v) + \frac{1}{\lambda}} \cdot b, \tag{6}$$

where b is the video playback rate. We note that the term "$1/\lambda$" is in the denominator since the time interval between two consecutive regular multicasts varies although the size of the patching window is

fixed.

To determine the optimal value for $W(v)$ (when *Server_Bandwidth* is minimized), we compute the derivative of the *Server_Bandwidth* with respect to $W(v)$. Since $D_{[0,W(v)]}$, in Equation 6, is defined differently for four different subdomains of $W(v)$ (see Equation 5), we need to consider four different cases in deriving the derivative as follows:

- $W(v) = 0$: Since this subdomain has only one value, the optimal patching window is

$$W_{opt} = 0 \tag{7}$$

  regardless of λ.

- $0 < W(v) \leq B$: In this subdomain, $Server\_Bandwidth = \frac{|v| + \frac{W(v)\cdot(W(v)+1)}{2}\cdot\lambda}{W(v)+\frac{1}{\lambda}} \cdot b$, according to Equations 5 and 6. By analyzing the derivative of the above function with respect to $W(v)$, we can find that:
  - if $0 < W(v) \leq \frac{\sqrt{2\cdot|v|\cdot\lambda^2-\lambda+1}-1}{\lambda}$, *Server_Bandwidth* decreases as $W(v)$ increases.
  - if $W(v) > \frac{\sqrt{2\cdot|v|\cdot\lambda^2-\lambda+1}-1}{\lambda}$, *Server_Bandwidth* increases as $W(v)$ increases.

  Thus, the optimal value for the patching windows is as follows:

$$W_{opt} = \begin{cases} \frac{\sqrt{2\cdot|v|\cdot\lambda^2-\lambda+1}-1}{\lambda} & \text{if } W_{opt}\frac{\sqrt{2\cdot|v|\cdot\lambda^2-\lambda+1}-1}{\lambda} \leq B, \\ B & \text{otherwise.} \end{cases} \tag{8}$$

- $B < W(v) \leq |v| - B$: In this subdomain, $Server\_Bandwidth = \frac{D_{[0,B]}+(|v|-B)\cdot(W(v)-B)\cdot\lambda}{W(v)+\frac{1}{\lambda}} \cdot b$, according to Equations 5 and 6. The derivative of *Server_Bandwidth* with respect to $W(v)$ is equal to $\frac{(|v|-B)(1+\lambda\cdot B)-D_{[0,B]}}{(W(v)+\frac{1}{\lambda})^2}\cdot b$. Since this derivative is either strictly increasing or decreasing over this range, the equation for *Server_Bandwidth* is a monotonic function with respect to $W(v)$. As a result, the minimal bandwidth is achieved either when $W(v) \to B^+$ or $W(v) = |v| - B$. When $W(v) \to B^+$, the corresponding bandwidth requirement is the same as when $W(v) = B$. Thus, we are left with:

$$W_{opt} = |v| - B \tag{9}$$

  as another potential optimal patching window.

- $\underline{|v| - B < W(v) \leq |v|}$: In this subdomain, we have:

$$Server\_Bandwidth = \frac{D_{[0,|v|-B]} + \frac{(W(v)-|v|+B)(W(v)+|v|-B+1)}{2} \cdot \lambda}{W(v) + \frac{1}{\lambda}} \cdot b$$

according to Equations 5 and 6. By analyzing the derivative of the above function with respect to $W(v)$, we can find that the bandwidth requirement is minimized when $W(v)$ has the following value:

$$W(v) = \frac{-1 + \sqrt{\lambda^2 \cdot (B - |v|) \cdot (|v| - B + 1) + \lambda \cdot (2 \cdot D_{[0,|v|-B]} - 1) + 1}}{2 \cdot \lambda}.$$

Let us denote the above value as W_{min}. Since W_{min} can be within or outside the range $(|v|-B, |v|]$ as illustrated in Figure 4, we discuss the optimal patching window for each of the three cases as follows.

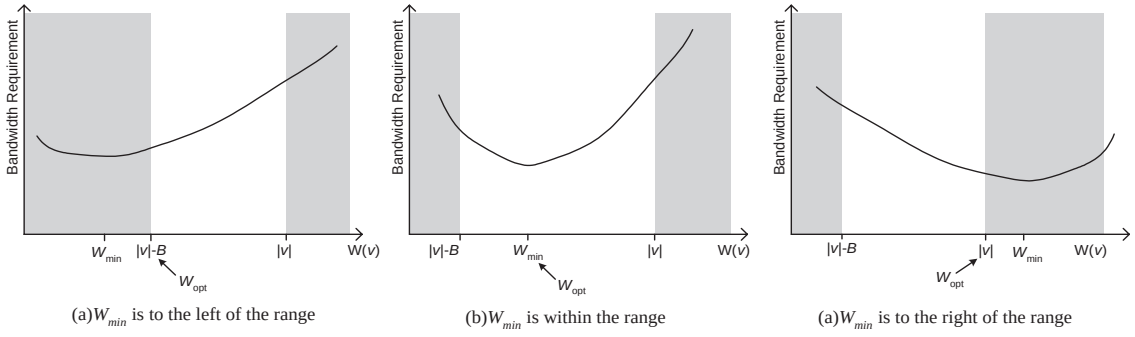

Figure 4: Three possible optimal patching windows

  - $\underline{W_{min} \leq |v| - B}$: Since the bandwidth curve is increasing for $W(v) \geq W_{min}$, the smallest bandwidth requirement for the range $(|v| - B, |v|]$ occurs when $W(v) \rightarrow (|v| - B)^+$ as illustrated in Figure 4(a). This minimal value is the same as the $Server\_Bandwidth$ value when $W(v) = |v| - B$. We have considered this case previously.
  - $\underline{|v| - B < W_{min} \leq |v|}$: As illustrated in Figure 4(b), the optimal window size for the range $(|v| - B, |v|)$ is:

$$W_{opt} = W_{min} = \frac{-1 + \sqrt{\lambda^2 \cdot (B - |v|) \cdot (|v| - B + 1) + \lambda \cdot (2 \cdot D_{[0,|v|-B]} - 1) + 1}}{2 \cdot \lambda}. \quad (10)$$

  - $\underline{W_{min} > |v|}$: This case is illustrated in Figure 4(c). Since the bandwidth curve is decreasing for $W(v) \leq W_{min}$, and W_{min} is outside the legal range (i.e., larger than the video length),

the optimal patching window size for the range $(|v| - B, |v|)$ is

$$W_{opt} = |v|. \tag{11}$$

In summary, from equations 7 through 11, the optimal patching window can be determined from the following six candidates:

- $W_1(v) = 0$
- $W_2(v) = \frac{\sqrt{2 \cdot |v| \cdot \lambda^2 - \lambda + 1} - 1}{\lambda}$
- $W_3(v) = B$
- $W_4(v) = |v| - B$
- $W_5(v) = \frac{-1 + \sqrt{\lambda^2 \cdot (B - |v|) \cdot (|v| - B + 1) + \lambda \cdot (2 \cdot D_{[0, |v| - B]} - 1) + 1}}{2 \cdot \lambda}$
- $W_6(v) = |v|$.

We note that $W_2(v)$ is a valid candidate only if $0 < W_2(v) \leq B$. Similarly, $W_5(v)$ is valid only if $|v| - B < W_5(v) \leq |v|$. For a given buffer capacity B and a video v with a request rate of λ, we first compute these six candidates of patching window. Then, for each valid patching window, we calculate the corresponding *Server_Bandwidth* using Equation 6. Finally, we select the patching window corresponding to the minimal *Server_Bandwidth* as the optimal patching window for v.

4 Performance Study

To assess the benefit of Patching, we compare its performance to that of Piggybacking [13] which is most relevant to our work. We use *Equal-Split* strategy as the merging algorithm for Piggybacking because it has been shown to be the most efficient [14]. To examine the significance of the patching window, we include both Optimal Patching and Grace Patching in this study. We decide not to include Greedy Patching because it rarely performs better than Grace Patching according to our initial study [19].

We select the server bandwidth required to support true VOD (i.e., no service delay) as our performance metric. Thus, a better scheme should require less server bandwidth. Without loss of generality,

we assume that the server has only one video. If a system has n videos, the corresponding bandwidth requirement is simply the summation of the bandwidth required for each individual video. Thus, the results reported in this section are also valid for systems with many videos.

We assume that the arrival of the requests follows a Poisson distribution with a mean arrival rate λ. The probability of having exactly k arrivals during a time period t is, therefore, $P(k,t) = \frac{(\lambda t)^k e^{-\lambda t}}{k!}$. The mean server bandwidth required by the Patching technique to support true VOD can be computed using Equation 6. Similarly, the same can be computed for Piggybacking using the formulas derived and validated in [14].

To make the paper self-contained, we brief describe the best Piggybacking technique, Equal-Split Merging algorithm, as follows. Under Equal-Split Merging, video streams are put into groups according to a predetermined time interval called *catch-up window*. The streams initiated within a catch-up window is said to form a group. The stream initiated first in a given group is called the *leading stream*. When a stream is initiated, if its temporal distance to the leading stream of the current group is within the catch-up window, the new stream is assigned to the current group; otherwise, a new group is created and the new stream is made the leading stream of the new group. The playback speed of each stream in a group is set to be either fastest or slowest so that they can be merged into a final stream as soon as possible. After the merge, the playback of the final stream is set to the normal playback rate. The bandwidth requirement for this scheme is as follows:

$$Server\_bandwidth_{es} = \frac{\sum_{n=1}^{\infty} D_{es}(n) \cdot P(n, W_{es})}{W_{es} + \frac{1}{\lambda}},$$

where W_{es} is the size of the catch-up window and $D_{es}(n)$ is the total amount of data delivered when there are exactly n streams initiated within the time period W_{es}. $D_{es}(n)$ can be calculated as

$$D_{es}(n) = \begin{cases} |v| \cdot b & \text{if } n = 1, \\ Cost(W_{es}, n) + [|v| - \frac{(1-\Delta_-) \cdot d(W_{es}, n)}{\Delta_- + \Delta_+}] \cdot b & \text{if } n >= 2. \end{cases}$$

The interested reader is referred to [14] for the derivation details of the above equations. We explain the notations used in the formula as follows. $d(w,k)$ is the mean temporal distance between the initiations of two consecutive streams, and can be approximated as $\frac{k-1}{k} \cdot w$, where $k > 1$. Δ_- and Δ_+ are the minimum and maximum percentage a stream can be slowed down and speeded up, respectively. Since the playback rate is allowed to be adjusted instantly from the slowest speed to the fastest speed, and

vice versa, we set Δ_+ and Δ_- at 2.5

$$Cost(w,k) = \begin{cases} d(w,2)(U_1+U_2) & \text{if } k=2 \\ \frac{1}{2^{k-2}}\{\sum_{i=1}^{k-3} C_i^{k-2}[Cost(\frac{d(w,k)}{2}, i+1) + Cost(\frac{d(w,k)}{2}, k-i-1) + \\ \frac{(k-1)(i+2)}{2k(i+1)}U_1 + \frac{(k-1)(k-i)}{2k(k-i-1)}U_2] + \frac{3k-2}{2k}(U_1+U_2) + 2Cost(\frac{d(w,k)}{2}, k-1)\} & \text{if } k>2, \end{cases}$$

where $U_1 = \frac{1-\Delta_+^2}{\Delta_++\Delta_-} \cdot b$, $U_2 = \frac{1-\Delta_-^2}{\Delta_++\Delta_-} \cdot b$, and $C_i^{k-2} = 2^{-(k-2)}\frac{(k-2)!}{(k-2-i)!i!}$.

The settings of the performance parameters are given in Table 1. We note that the catch-up window size for PiggyBacking is limited by the length of the video and the acceptable range for altering the playback rate. This constraint ensures that all streams in one group can be merged into a single stream before the leading stream finishes its playback [14]. In the following subsections, we will investigate the effect of client buffer size, request inter-arrival time, and video length on the server bandwidth requirement.

Parameter	default	variation
Number of videos	1	N/A
Normal Playback rate b (Mbps)	1.5	N/A
Fastest Playback Alteration Δ_+	2.5%	N/A
Slowest Playback Alteration Δ_-	2.5%	N/A
Size of Catch-up Window W_{es} (minutes)	9	$\frac{\Delta_++\Delta_-}{(1+\Delta_+)(1-\Delta_-)} \cdot \lvert v\rvert$
Client storage size B (minutes of data)	10	0 - 90
Mean Request Inter-arrival Time $\frac{1}{\lambda}$ (seconds)	50	5 - 95
Video length $\lvert v\rvert$ (minutes)	90	30 - 150 minutes

Table 1: Parameters used for the performance evaluation

4.1 Effect of Client Buffer Size

In this study, we fixed the mean request inter-arrival time at 50 seconds and the video length at 90 minutes. The client buffer size was varied from 0 to 90 minutes of video data. Its effect on the mean server bandwidth requirement is plotted in Figure 5. The curve for Piggybacking is flat because it does not use client buffers. The plots indicate that Patching using local buffers as small as two minutes is sufficient to match the performance of Piggybacking. This costs only USD $3 today if

the video is encoded using MPEG-2 (i.e., 60 Mbytes). If we increase the client buffer size to five minutes (150 Mbytes, about USD $8), a 200% improvement over piggybacking is achievable. Thus, we can conclude that Patching is significantly better than Piggybacking without relying on expensive specialized hardware.

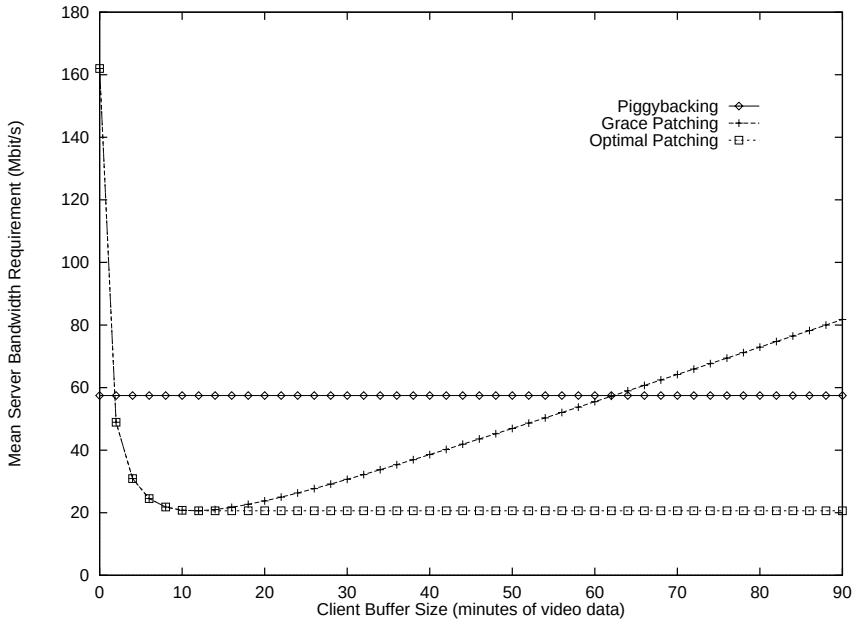

Figure 5: Effect of client buffer size on mean bandwidth requirement.

Comparing the two patching techniques, we see that the performance of Grace Patching degrades considerably when the size of the client buffer is greater than 10 minutes. This confirms our initial observation that the patching window should not be determined based on the client buffer size alone. As we can see in Figure 5, when the client buffer becomes very big, the patching window used by Grace Patching is too large. As a result, the number of patching multicasts increases significantly. Most of them must deliver a large portion of the video causing a greater demand on the server bandwidth. Optimal Patching avoids this problem by taking into account the arrival rate of the service requests. With the added intelligence, Optimal Patching is able to determine the optimal amount of buffer space to utilize. It is shown in Figure 5 that Optimal Patching ignores any buffer space beyond 10 minutes in order to maintain the optimal performance. In practice, our formula can be used to determine the amount of disk space required for patching.

4.2 Effect of Inter-Arrival Time

In this study, we perform sensitivity analysis with respect to inter-arrival time. The client buffer size was fixed at 15 minutes, and the video was assumed to be 90 minutes long. The results are plotted in Figure 6. In general, the benefit of any multicast technique decreases with the increases in the inter-arrival time. Optimal Patching, however, consistently outperforms Piggybacking by a wide margin (i.e., from 100% to 150% under our workload). The performance gap decreases with the increases in the inter-arrival time. This is due to insufficient buffer space to fully leverage Patching. When the inter-arrival time is large, many patching streams need to deliver more data. As a result, the clients need to have more storage space in order to buffer the regular stream. In practice, since disk space is inexpensive, it is worthwhile to invest in more storage space in order to fully exploit the benefit of patching.

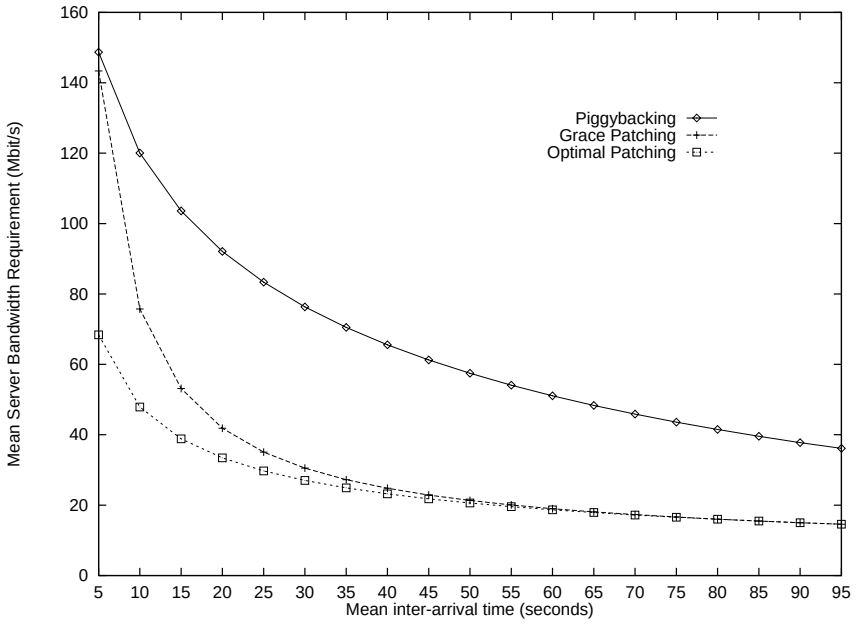

Figure 6: Effect of inter-arrival time on the mean bandwidth requirement.

4.3 Effect of Video Length

In this study, we fixed the client buffer at 15 minutes and the mean inter-arrival time at 50 seconds. The catch-up window used for Piggybacking was maximized. For instance, if the allowable alternation rate is $\pm 2.5\%$, then the maximum catch-up window is about 5% of the video length. It was shown in

[14] that this setting resulted in the best performance. To investigate the effect of video length on the performance of the three techniques, we varied the video length from 30 to 150 minutes.

The performance results are plotted in Figure 7. We see that the performance of Piggybacking degrades quickly as the video length increases. In general, the size of a good catch-up window is proportional to the length of the video. A longer video implies a larger catch-up window, and therefore a bigger catch-up delay. This delay increases very quickly with the increases in the video length because the catch-up speed must be limited to about 5% of the normal playback rate to avoid deterioration in the playback quality. Due to this property, Piggybacking is not suitable for applications, which involve long videos, such as home entertainment, distance learning, etc.

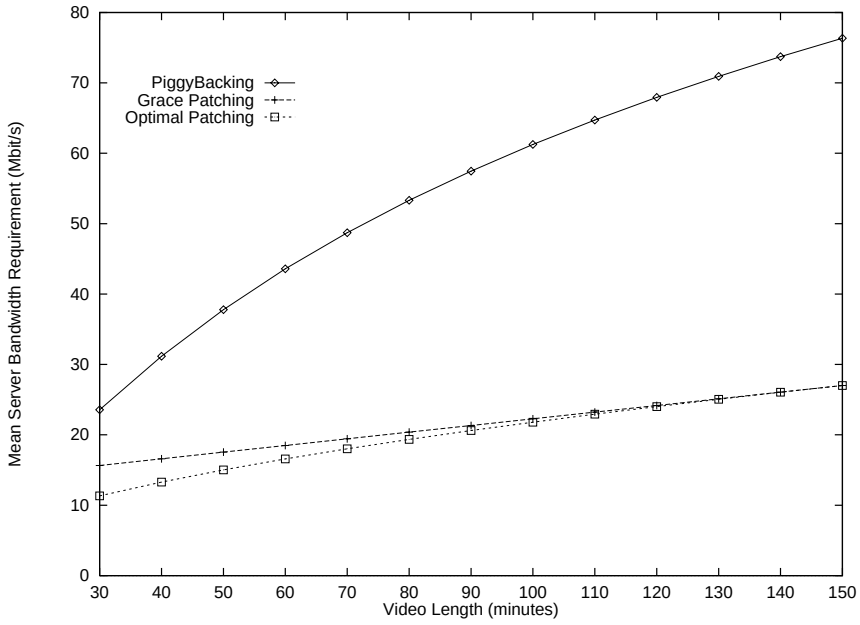

Figure 7: Effect of video length on the mean bandwidth requirement.

On the contrary, Figure 7 shows that Patching is much less sensitive to the length of the video. This is due to the fact that Patching does not have the catch-up problem. Comparing the two patching schemes, Grace does not perform as well under short videos. Its patching window is too large under these conditions resulting in too many long-duration patching multicasts.

5 Concluding Remarks

Multicast has been shown to be very effective in reducing the demand on server bandwidth. It, however, can only be used to provide near VOD services. In this paper, we presented a technique, called *Patching*, to address this issue. Patching has the following benefits:

- Since requests can be served immediately, true VOD can be achieved.
- Each multicast is more efficient since it can expand over time to serve more clients.
- The service throughput of each communication channel is better since its average turnaround time is very short allowing it to serve many more clients per time unit.

To fully exploit the potential of this technique, we developed an analytical method to optimize the patching performance. Our design technique ensures that minimum server bandwidth is used for a given performance level. To assess the benefit of Patching, we compared it with Piggybacking. Our performance results indicate that more than 200% improvement can be achieved.

In summary, the contribution of this paper is twofold. First, we introduce a novel technique to leverage multicast technology for VoD applications. Second, we present a design methodology to ensure the optimality of the proposed technique.

References

[1] H. Vin, PV Rangan, and S. Ramanathan. Designing an on-demand multimedia service. *IEEE Commun. Mag.*, 30(7):56–64, 1992.

[2] S. Viswanathan and T. Imielinski. Metropolitan area video-on-demand service using pyramid broadcasting. *Multimedia systems*, 4(4):179–208, August 1996.

[3] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. A permutation-based pyramid broadcasting scheme for video-on-demand systems. In *Proc. of the IEEE Int'l Conf. on Multimedia Systems'96*, Hiroshima, Japan, June 1996.

[4] K. A. Hua and S. Sheu. Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems. In *Proc. of the ACM SIGCOMM'97*, Cannes, France, Sepetember 1997.

[5] J. F. Paris, S. W. Carter, and D. D. E. Long. Efficient broadcasting protocols for video on demand. In *Proc. of SPIE's Conf. on Multimedia Computing and Networking (MMCN'99)*, pages 317–326, San Jose, CA, USA, January 1999.

[6] A. Dan, D. Sitaram, and P. Shahabuddin. Dynamic batching policies for an on-demand video server. *Multimedia Systems*, 4(3):112–121, June 1996.

[7] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. On optimal batching policies for video-on-demand storage servers. In *Proc. of the IEEE Int'l Conf. on Multimedia Systems'96*, Hiroshima, Japan, June 1996.

[8] Kien A. Hua, J-H Oh, and Khanh Vu. An adaptive video multicast scheme for varying workloads. *ACM Multimedia Systems*, to appear.

[9] T.D.C. Little and D. Venkatesh. Popularity-based assignment of movies to storage devices in a video-on-demand system. *Multimedia Systems*, 2(6):280–287, January 1995.

[10] S. Sheu, Kien A. Hua, and W. Tavanapong. Chaining: A generalized batching technique for video-on-demand. In *Proc. of the Int'l Conf. On Multimedia Computing and System*, pages 110–117, Ottawa, Ontario, Canada, June 1997.

[11] L. K. Wright, S. McCanne, and J. Lepreau. A reliable multicast webcast protocol for multimedia collaboration and caching. In *Proc. of ACM Multimedia*, Los Angles, CA., October 2000.

[12] Kien A. Hua, D. Tran, and R. Villafane. Caching multicast protocol for on-demand video delivery. In *Proc. of SPIE's Conf. on Multimedia Computing and Networking (MMCN'99)*, pages 2–13, San Jose, CA, USA, Janaury 2000.

[13] L. Golubchik, J. Lui, and R. Muntz. Adaptive piggybacking: a noval technique for data sharing in video-on-demand storage servers. *ACM Multimedia Systems*, 4(3):140–155, 1996.

[14] S. Lau, J. Lui, and L. Golubchik. Merging video streams in a multimedia storage server: complexity and heuristics. *ACM Multimedia Systems*, 6:29–42, 1998.

[15] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. In *Proc. of ACM Multimedia*, pages 15–23, San Francisco, California, October 1994.

[16] K. Almeroth and M. H. Ammar. The use of multicast delivery to provide a scalable and interactive video-on-demand service. *IEEE Journal on Selected Areas in Communications*, 14(6):1110–1122, 1996.

[17] M. S. Chen and D. D. Kandlur. Stream conversion to support interactive video playout. *IEEE Multimedia magazine*, 3(2):51–58, Summer 1996.

[18] W. Feng, F. Jahanian, and S. Sechrest. Providing vcr functionality in a constant quality video-on-demand transportation service. In *Proc. of the IEEE Int'l Conf. on Multimedia Systems'96*, Hiroshima, Japan, June 1996.

[19] Kien A. Hua, Ying Cai, and Simon Sheu. Patching: A multicast technique for true video-on-demand services. In *Proc. of ACM Multimedia*, pages 191–200, Bristol, U.K., September 1998.

[20] S. W. Carter and D. D. E. Long. Improving bandwidth efficiency of video-on-demand servers. *Computer Networks and ISDN Systems*, 31(1):99–111, March 1999.

[21] S. Sen, L. Gao, J. Rexford, and D. Towsley. Optimal patching schemes for efficient multimedia streaming. In *Proc. IEEE NOSSDAV'99*, Basking Ridge, NJ, U.S.A, June 1999.