

# A Double Patching Technique for Efficient Bandwidth Sharing in Video-on-Demand Systems\*

*Ying Cai*<sup>1</sup>

*Wallapak Tavanapong*<sup>1</sup>

*Kien A. Hua*<sup>2</sup>

<sup>1</sup> Department of Computer Science

Iowa State University

Ames, IA 50011, U.S.A

E-mail: {yingcai, tavanapo}@cs.iastate.edu

<sup>2</sup> Computer Science Program, SEECs

University of Central Florida

Orlando, FL 32816, U.S.A

E-mail: kienhua@cs.ucf.edu

September 25, 2004

## Abstract

Patching is an efficient bandwidth-sharing technique for video-on-demand systems. In this environment, a client joins an on-going regular multicast to receive and cache the data in a local buffer. The server needs to send only the leading portion of the video in a patching stream. When the client finishes playing back the patching data, it continues the playback using the data already cached in the buffer. Although this strategy enables stream sharing without the service delay, the performance of Patching has limitation: as the time distance to the last regular multicast enlarges, the patching cost for new requests increases and eventually, a new regular multicast must be scheduled to balance the cost. In this paper, we address this problem by proposing a new technique called *Double Patching*. Our research is based on the observation that a patching stream can be shared by the video requests arriving in the next  $w_p$  time units if it delivers an additional  $2 \cdot w_p$  time units of video data. With these extra data, the patching cost for these requests can be dramatically reduced. In the new technique, a client uses no more than two download channels at any one time. Thus, its implementation cost is the same as that of the original Patching. As for its performance, our study shows that the improvement achieved by the proposed technique is significant. In many cases, Double Patching doubles the performance of the original Patching.

**KEYWORDS:** Multimedia communications, multicast, latency, on-demand service, performance evaluation.

---

\*This work is partially supported by National Science Foundation under Grant No. 0092914.

# 1 Introduction

Traditionally, communication bandwidth is usually not a primary concern in designing information systems. The query results are typically very small in size. This situation is changing with video data becoming more popular. Transmitting a video clip, as the result of a query, consumes substantial bandwidth. For example, sustaining an MPEG-I video stream requires an average bandwidth of 1.5 Mbps. For high-quality videos, such as MPEG-II and HD TV, the bandwidth requirement is much higher, from 4 Mbps up to 25 Mbps. As such, a typical video server can support only very limited number of concurrent video streams. Considering that the bandwidth for a video stream has to be reserved for an extended time period, the system throughput would be minimal if we simply make one video stream serve one video request.

To deploy a large scale video service, therefore, it is essential to share bandwidth among video requests. One solution is to broadcast a video repeatedly at certain time interval. Some efficient periodic broadcast techniques (e.g., Pyramid Broadcasting[18, 1], Skyscraper Broadcasting[13], Pagoda Broadcasting[14], etc.) 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. When a video is highly demanded, periodic broadcasting is very cost-effective - the worst service latency is constant, regardless to the number of video requests. Therefore, it could be used to serve a large community with a minimal bandwidth requirement. This strategy, however, can only be used for very popular videos.

Instead of broadcasting a video repeatedly, we can schedule a video for service upon the arrival of its request. Thus, the videos with diverse popularity can be supported. To allow many video requests to share one video stream, a service request can be delayed for a while. During the waiting time, if there are more requests coming for the same video, they can be served together using one multicast stream [6, 2, 12]. This strategy, called *Batching*, can significantly reduce the demand on server bandwidth. Each request, however, must wait until the end of a batching period. If we make this period short, the

benefit of multicast diminishes. On the other hand, if users are made to wait too long, many of them may renege, again affecting the multicast efficiency.

To address the aforementioned dilemma, clients can join an on-going *regular multicast*, and cache the multicast data in their local disk [16, 17, 11, 4, 9]. To serve these clients, the server initiates a *patching stream* to multicast only the leading portion of the video. When these clients finish playing back the patching stream (i.e., reaching the *skew point*), they continue to play back the remainder of the video using the data already buffered in the local disk. We called this scheme *Patching* in [11]. The benefits of patching are twofold. First, multicasts become more efficient since they can expand dynamically to accommodate future services. Second, since service requests do not have to wait for the batching process, *true* video on demand (VoD), i.e., no service delay, can be achieved.

The duration of the patching streams is the most important determinant of patching performance. They should be short since a patching stream typically serves only few, often one, clients. However, patching streams, too short, would incur many regular multicasts degrading the system performance to that of conventional batching. To determine the optimal duration for each patching stream, optimized schedulers were proposed in [3, 9]. A different way to reduce patching costs is to enable clients to share patching data. Although various solutions have been proposed, they require either complicate channel management at both server and client sides or significant client receiving bandwidth. For instance, the stream merging algorithms [7] [8] require server to dynamically create merging trees and inform clients of which channels to download data. In addition, determining the optimal merging schedule for  $n$  video streams has a complexity of  $O(n^3)$  and requires accurate prediction on the arrival time of video requests [7]. Sharing patching data is also realized in [15] and [10] by having the clients to receive data from three or more streams simultaneously. These schemes require clients with substantial communication bandwidth making them unsuitable for many applications.

In this paper, we introduce a new technique that has the same implementation cost as the standard patching scheme while at any one time, the number of video streams a client needs to download is no more than two. Our solution is based on the following observation: a patching stream can be shared by the video requests arriving within the next  $w_p$  time units if it delivers an extra  $2 \cdot w_p$  time units of data. In specific, given a client which is  $t$  time units late for a regular multicast, we can schedule a *long patching stream* to deliver  $t+2 \cdot w_p$  time units of data instead of just  $t$  time units as in standard patching.

With these extra  $2 \cdot w_p$  time units of video data, the cost to serve a request arriving in the next  $\Delta t$  time units, where  $0 \leq \Delta t \leq w_p$ , is just a *short patching stream* delivering only the first  $\Delta t$  time units data of the video. Under standard patching, the cost would be  $t + \Delta t$  time units of the video data. The client can receive the entire video as follows. It first uses its two download channels to receive the video data from the short and long patching streams, respectively. After the short patching stream is exhausted, the same download channel is then switched to receive data from the regular multicast stream. Thus, two channels are used to download three segments from three different video streams. We prove in this paper that the three video segments make up of the entire video for the client. We call this strategy *Double Patching*, alluding to the fact that the patching cost can be dramatically reduced by using two patching streams to patch up a client time distance to a regular multicast. Our performance study shows that, with the same client download bandwidth, the new technique can improve 50% in average and in some workloads, double the performance of the standard patching.

The remainder of this paper is organized as follows. To make the paper self-contained, we describe Standard Patching and discuss the optimal scheduler in Section 2. The details of Double Patching are presented in Section 3. We introduce a performance model, and use it to optimize Double Patching in Section 4. In Section 5, we compare the performance of Double Patching and that of Standard Patching. Finally, we give our concluding remarks in Section 6.

## 2 Standard Patching

In this paper, we assume that most of the server bandwidth is multiplexed into a set of logical channels, each capable of sustaining a video stream transmitting data at the playback rate. The remaining bandwidth of the server is used for control messages such as service requests and service notifications.

Under Standard Patching, when a free channel becomes available, a new video stream is scheduled to deliver a requested video according to some scheduling policy. As examples, the following policies can be used to determine the next video to provide service:

- First Come First Served (FCFS) [5]: This policy selects the video with the oldest pending request (has been waiting for the longest time) to be served next. An advantage of this scheme is its fairness - every video is treated equally regardless of its popularity. A drawback is lower system

throughput.

- Maximum Queue Length First (MQL) [5]: Unlike FCFS, this policy is designed to maximize system throughput by selecting the video with the highest number of pending requests to be served first. This strategy can be considered unfair because it favors more popular videos.
- Maximum Factored Queue Length First (MFQ) [2]: This scheme improves upon FCFS and MQL by taking into account both the waiting times of requests and the popularity of the videos. It can achieve throughput close to that of MQL with fairness approaching that of FCFS.

The new video stream can be a *regular stream* or a *patching stream*. A regular stream multicasts the video in its entirety while a patching stream transmits only the first  $t$  time units of the video, where  $t$  is the temporal distance to the starting time of the last regular multicast of the same video.

To support Standard Patching, each client station needs to have three threads of control: two data loaders and a video player. Two scenarios can occur:

- If a client is served by a new regular stream, only one data loader is necessary to receive the video data. The data packets arriving at the client are piped directly to the video player for rendering onto the screen.
- In the case that a patching stream is scheduled, both of the client's loaders must be used to receive data from the patching stream and the latest regular stream simultaneously. In this case, the video player starts playing back the video data from the patching stream first while the data packets from the regular stream are temporarily cached in the client buffer. When the patching stream ends, the video player switches to play back the data in the disk buffer as the remaining data packets continue to arrive in the regular stream.

We show an example in Figure 1 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 being 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.

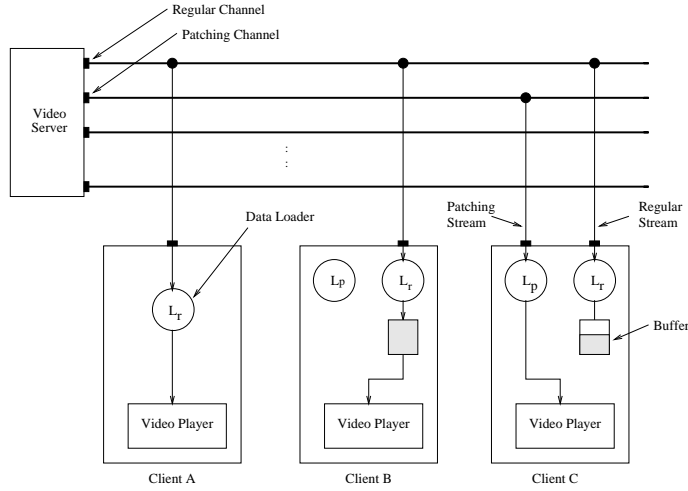


Figure 1: Three possible patching stages.

Under Standard Patching, a patching stream is scheduled for a new request if the temporal distance between its arrival time and the starting time of the last regular stream of the same video is less than a certain threshold called *patching window* [3]; otherwise, a new regular stream is initiated, and all subsequent requests arriving within the next patching window will be served by patching streams. A regular stream and its following patching streams initiated before the next regular stream are said to form a *patching group*. If we use  $|v|$  to denote the video length,  $\lambda$  the video request rate,  $B$  the client buffer size in terms of video playback duration, and  $w$  the patching window size for the video, then the mean total amount of data,  $D_{[0,w]}$ , transmitted by one patching group to support true on-demand services can be obtained as follows [3]:

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

The mean bandwidth requirement can then be computed as:

$$Bandwidth_{StandardPatching} = \frac{D_{[0,w]}}{w + \frac{1}{\lambda}} \cdot b, \quad (2)$$

where  $b$  is the playback rate of the video. Apparently, the size of patching window  $w$  has a profound influence on  $Bandwidth_{StandardPatching}$ . We say a patching window is *optimal* if it results in the smallest bandwidth requirement. A detailed analysis on how to determine the optimal patching window

is presented in the Appendix. It shows that the demand on server bandwidth is minimal when one of the following six patching windows is used:

- $w_1 = 0$
- $w_2 = \frac{\sqrt{2 \cdot |v| \cdot \lambda^2 - \lambda + 1} - 1}{\lambda}$ , consider  $w_2$  if  $w_2 < B$
- $w_3 = B$
- $w_4 = |v| - B$
- $w_5 = \frac{-1 + \sqrt{\lambda^2 \cdot (B - |v|) \cdot (|v| - B + 1) + \lambda \cdot (2 \cdot D_{[0, |v| - B]} - 1) + 1}}{2 \cdot \lambda}$ , consider  $w_5$  if  $|v| - B < w_5 < |v|$
- $w_6 = |v|$

Thus, by computing and comparing the bandwidth requirements corresponding to the above six patching window sizes, we can determine the optimal patching window.

### 3 Double Patching

Standard Patching is very economic in delivering patching data. Each patch is just long enough to cover the missing portion of the video. In this section, we present a better patching technique called Double Patching. In this new scheme, a patching stream may transmit more than just the regular patching data (i.e., *long patching stream*). At first sight, this seems to be a waste of system resources. This small “waste”, however, can significantly reduce the patching costs of the subsequent services.

As a motivating example, let us consider a regular stream of video  $v$  starting at time 0. At time  $t$ , a patching stream,  $S_a$ , is initiated for client  $A$ . One time unit later, another patching stream,  $S_b$ , is scheduled for client  $B$ . If we use  $v[t_1, t_2]$  to denote the video segment from time  $t_1$  to time  $t_2$  assuming the beginning of the video as time zero, then the patches required by clients  $A$  and  $B$  are  $v[0, t]$  and  $v[0, t + 1]$ , respectively. In total, the server delivers  $2 \cdot t + 1$  time units of data specifically for these two clients. Alternatively, if we make  $S_a$  deliver a long patching stream  $v[0, t + 2]$ , a patch with two extra time units of data, then client  $B$  can share stream  $S_a$  with client  $A$  as follows:

- Stream  $S_b$  needs to transmit only the first time unit of the video,  $v[0, 0]$ . As client  $B$  receives this segment using its loader  $L_1$ , the client immediately plays back the video. At the same time, it downloads  $v[1, t + 2]$  in stream  $S_a$  using loader  $L_2$ .

- When stream  $S_b$  ends, client  $B$  starts to play back the second video segment  $v[1, t + 2]$ , as  $L_1$  switches to download the remaining data,  $v[t + 3, |v|]$  (where  $|v|$  is the length of the video), being transmitted in the regular stream.
- Finally, when the playback of  $v[1, t + 2]$  is completed, client  $B$  continues to play back the third video segment  $v[t + 3, |v|]$ .

This strategy is referred to as Double Patching in this paper. We note that the three video segments,  $v[0, 0]$ ,  $v[1, t + 2]$ , and  $v[t + 3, |v|]$ , make up the complete video for client  $B$ . In total, the server delivers  $(t + 2) + 1$  time units of data specifically for these two clients. Therefore, the saving in comparison with Standard Patching is  $\frac{(2-t+1)-(t+3)}{2-t+1} = \frac{t-2}{2-t+1}$ . When the time skew  $t$  is large, this saving is approximately 50%. The two patching strategies are illustrated in Figure 2. The shaded areas represent the data received by client  $B$ . We note that Double Patching requires only two data loaders as in Standard Patching, and these two data loaders are used to receive data from three concurrent streams. In other words, under Double Patching, the server resource is saved by exploiting client bandwidth more efficiently. Thus, Double Patching improves the standard technique without incurring additional system costs.

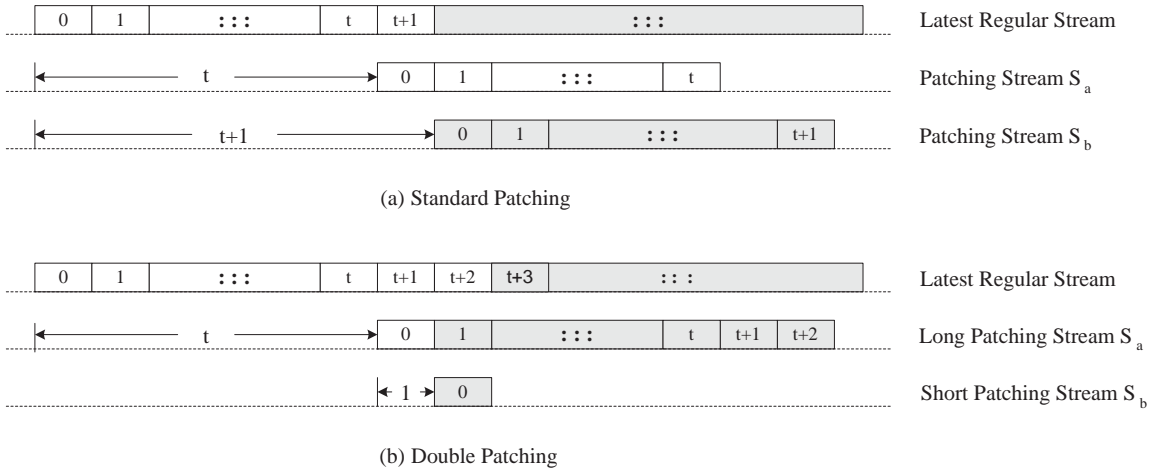


Figure 2: A comparison of Standard Patching and Double Patching

We discuss the client and server designs under Double Patching in the next two subsections. Without loss of generality, we assume that the server has only one video. In practice, the server bandwidth can be multiplexed to make  $n$  virtual servers, each serves one of the  $n$  videos.

### 3.1 Server Design

As in Standard Patching, the server bandwidth is multiplexed into a set of logical channels, each capable of sustaining a video stream transmitting data at the playback rate. These channels are used to deliver three types of video streams: regular streams, long patching streams, and short patching streams. Depending on the type of the stream currently riding on a given channel, it is referred to as a *regular channel*, an *long patching channel*, or a *short patching channel*. When a channel is dispatched, it is given a workload, specified as  $v[t]$ , where  $v$  and  $t$  denote the unique ID of the video file and the desired playback duration, respectively. For instance,  $v[3]$  indicates that the free channel should deliver only the first three time units of video  $v$ . After the transmission is completed, the channel again becomes *free*, and is available for the next service.

The stream scheduling policy is as follows. The very first request for a given video is serviced using a regular stream. For all requests arriving within the next  $w_m$  time units, either a long patching stream or a short patching stream is scheduled.  $w_m$  is called a *multicast window*, and is the minimal scheduling distance between any two consecutive regular streams. A regular stream and its following long and short patching streams initiated before the next regular stream are said to form a *multicast group*. After a regular stream, short patching streams are scheduled for requests arriving within the next  $w_p$  time units. These consecutive short patching streams form a *patching group*. The threshold  $w_p$  is called *patching window*, and is set to be no larger than  $w_m$ . We note that this patching window is different from the patching window in Standard Patching. If  $w_p = w_m$ , Double Patching becomes Standard Patching. In general,  $w_m$  is many times greater than  $w_p$ . After  $w_p$  time units, a long patching stream is scheduled for the next service request. This stream “leads” the next patching group which consists of short patching streams scheduled for requests arriving within the next  $w_p$  time units. This pattern is repeated for each of the subsequent patching groups until a new request whose time skew to the regular stream of the current multicast group is greater than  $w_m$ , in which case a regular stream is initiated to start a new multicast group. The scheduling pattern for one multicast group is illustrated in Figure 3. The benefits of Double Patching are twofold. First, the efficiency of multicast can be dramatically improved because each regular stream and its following long patching streams can now be shared by many clients. Second, a client can be admitted for service immediately by scheduling a very short patching stream, zero service latency can be achieved with a minimal cost. These two features

allow Double Patching to leverage standard multicast for true video-on-demand services. In the next section, we will determine the values of  $w_m$  and  $w_p$  such that the mean server bandwidth required to support true on-demand services is minimized.

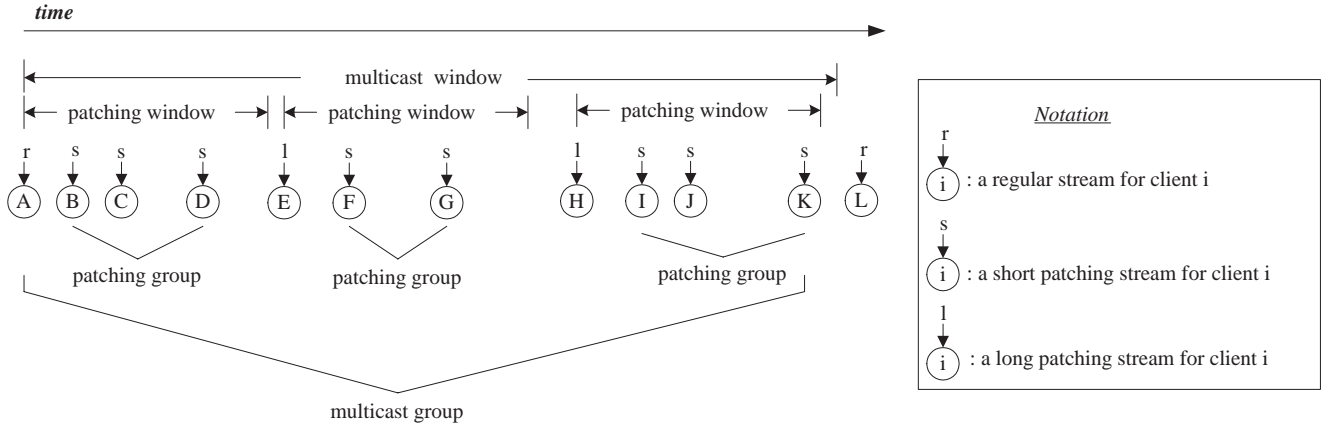


Figure 3: Pattern of stream initiation

A client makes a service request by sending a request token ( $ClientID$ ) to the server, where  $ClientID$  is its unique network address. In practice, the client also needs to specify the desired video. However, for clarity sake we assume in this paper that the server has only one video. To handle occasional bursts of requests, the server can provide a waiting queue as in typical client-server systems. Most requests, however, should be able to receive service as soon as they arrive at the job queue, unless system resources are inadequate. We present the algorithm for the server software in Figure 4. This procedure is executed when a channel becomes free. The server notifies the client by sending a service token,  $(ID_{sp}, ID_{lp}, ID_r)$ , where  $ID_{sp}$ ,  $ID_{lp}$ , and  $ID_r$  are the IDs of the short patching channel, the long patching channel, and the regular channel, respectively. This token informs the client the channels it should use to receive data for its requested video. We note in this procedure that if the free channel is to deliver a long patching stream, its workload is set to  $v[skew + 2 \cdot w_p]$ , where  $skew$  is the current time distance to the beginning of the last regular stream. That is,  $v[skew]$  is the patching data for this client. We will show in the next subsection that transmitting the extra  $2 \cdot w_p$  time units of data allows the subsequent requests arriving within the next  $w_p$  time units to share this long patching stream.

**Algorithm:** *Server Main Routine*

$|v|$ : playback duration of the video  
 $v[t]$ : the first  $t$  time units data of the video  
 $w_m$ : multicast window size  
 $w_p$ : patching window size  
 $LastRegular$ : the ID of the last regular channel, initialized to *null*  
 $LastLPatching$ : the ID of the last long patching channel, initialized to *null*  
 $skew(c)$ : the time skew to the start time of the current stream on channel  $c$

- Initialize service token as  $(ID_{sp} = null, ID_{lp} = null, ID_r = null)$ ;
- Dispatch a free channel, say *FreeChannel*;
- If  $LastRegular$  is *null* or  $skew>LastRegular) > w_m$ , do the following:
  - Set  $ID_r = FreeChannel$  and its workload to be  $v[|v|]$ ;
  - Set  $LastRegular = FreeChannel$ .
- Otherwise, do the following:
  - If  $skew>LastLPatching) > w_p$ , do the following:
    - \* Set  $ID_r = LastRegular$ ;
    - \* Set  $ID_{lp} = FreeChannel$  and its workload to be  $v[skew>LastLPatching) + 2 \cdot w_p]$ ;
    - \* Set  $LastLPatching = FreeChannel$ .
  - Otherwise, do the following:
    - \* Set  $ID_r = LastRegular$ ;
    - \* Set  $ID_{lp} = LastLPatching$ ;
    - \* Set  $ID_{sp} = FreeChannel$  and its workload to be  $v[skew>LastLPatching)]$ .
- For each request token ( $ClientID$ ) in the waiting queue, do the following:
  - Send the service token  $(ID_{sp}, ID_{lp}, ID_r)$  to notify the client  $ClientID$ ;
  - Delete the request token from the waiting queue.
- Activate *FreeChannel*.

Figure 4: Algorithm for Video Server

## 3.2 Client Design

A client uses two data loaders,  $L_1$  and  $L_2$ , to receive data and write them to a local buffer. Another thread, *VideoPlayer*, fetches the corresponding data packets from the buffer, reassembles the video frames, and renders them onto the screen. As discussed in the server design, a client sends its own network address as part of the token for a service request. It then waits for the service token ( $ID_{sp}$ ,  $ID_{lp}$ ,  $ID_r$ ). When it arrives, the client examines the values of  $ID_{sp}$  and  $ID_{lp}$  and acts accordingly as follows:

1.  $ID_{sp} = null$  and  $ID_{lp} = null$ : This combination indicates that the server is about to start a new regular stream on channel  $ID_r$ . The client needs only use one loader,  $L_1$ , to receive the entire video.
2.  $ID_{sp} \neq null$  and  $ID_{lp} = null$ : This combination indicates that the server is about to start a short patching stream on channel  $ID_{sp}$ . This scenario happens when the client time skew to the last regular stream is within  $w_p$  time units. In this case, the client must activate both loaders  $L_1$  and  $L_2$  in order to download data from the short patching channel  $ID_{sp}$  and the regular channel  $ID_r$  simultaneously.
3.  $ID_{sp} = null$  and  $ID_{lp} \neq null$ : This combination indicates that the server is about to start a new long patching stream. In this case, the client needs to activate both loaders  $L_1$  and  $L_2$  to download data from the long patching channel  $ID_{lp}$  and the regular channel  $ID_r$  simultaneously.
4.  $ID_{sp} \neq null$  and  $ID_{lp} \neq null$ : This combination indicates that the server is about to start a short patching stream on channel  $ID_{sp}$ . This case happens when the client's time skew to the last regular stream is greater than  $w_p$  time units, while to the last long patching stream is less than  $w_p$  time units. Under this circumstance, the client uses loader  $L_1$  to download data first from the short patching channel  $ID_{sp}$ , and then from the regular channel  $ID_r$ . In parallel with these activities,  $L_2$  is used to receive data on the long patching channel  $ID_{lp}$ . The data stream arriving on channel  $ID_{lp}$  is used to bridge the gap between the two streams coming on channels  $ID_{sp}$  and  $ID_r$ , respectively. These three data segments make up the entire video.

The above four cases are illustrated in Figure 5. The client's *VideoPlayer* can start as soon as  $L_1$  begins downloading data. For the first three cases, it is trivial to see that the playback is continuous

and complete since the client is served by at most two data streams, and it has two loaders to keep up with the incoming data. The fourth case is less obvious; and we discuss it as follows.

Let us say a long patching stream for a video  $v$  started  $t$  time units after the last initiation of a regular stream. According to the server routine (Figure 4), this long patching stream transmits the video segment  $v[0, t + 2 \cdot w_p]$ . If a client arrives  $p$  time units after this stream begins, then the short patching stream initiated for this client, according to the server routine, delivers the video segment  $v[0, p]$ . This  $p$  time units of data is received by loader  $L_1$  and directly piped to the *VideoPlayer* for display. At the same time,  $L_2$  receives the video segment  $v[p + 1, t + 2 \cdot w_p]$  from the long patching stream. The corresponding data packets are cached in the client's local disk. After  $p$  time units, *VideoPlayer* starts consuming data in this disk buffer while  $L_1$  switches to download the video segment  $v[t + 2 \cdot p + 1, |v|]$  on the regular channel. Since the server routine ensures that  $p \leq w_p$ , the three downloaded segments,  $v[0, p]$ ,  $v[p + 1, t + 2 \cdot w_p]$ , and  $v[t + 2 \cdot p + 1, |v|]$ , are sufficient to make up the entire video.

We present the client routines in Figure 6. We note that the data from the short patching channel (if any), the long patching channel (if any), and the regular channel are first stored in *Buffer*<sub>1</sub>, *Buffer*<sub>2</sub>, and *Buffer*<sub>3</sub>, respectively. As the data loaders fill these buffers, the *VideoPlayer* consumes the data in these buffers one at a time.

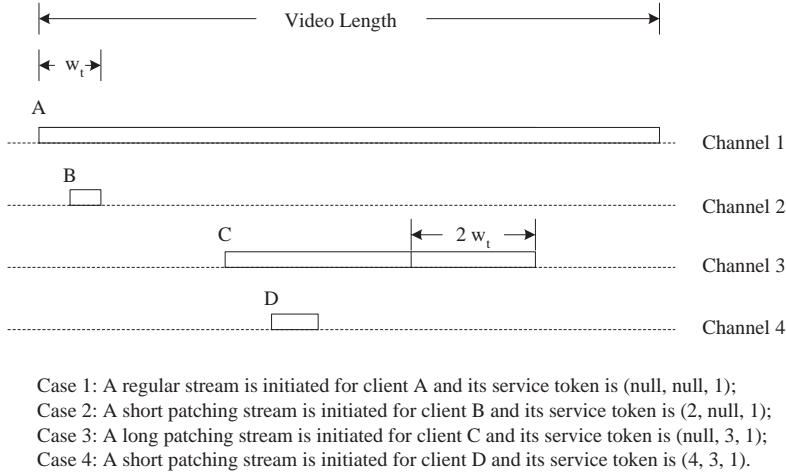


Figure 5: Different settings of service token

**Algorithm:** *Client Main Routine*

1. Send a request token ( $ClientID$ ) to the video server;
2. Wait until the service token ( $ID_{sp}, ID_{lp}, ID_r$ ) from the server arrives;
3. Examine the values of  $ID_{sp}$  and  $ID_{lp}$  and do one of the following:
  - If  $ID_{sp} = null$  and  $ID_{lp} = null$ , start loader  $L_1(ID_r, null)$ ;
  - If  $ID_{sp} = null$  and  $ID_{lp} \neq null$ , start loader  $L_1(ID_{lp}, null)$  and loader  $L_2(ID_r)$ ;
  - If  $ID_{sp} \neq null$  and  $ID_{lp} = null$ , start loader  $L_1(ID_{sp}, null)$  and loader  $L_2(ID_r)$ ;
  - If  $ID_{sp} \neq null$  and  $ID_{lp} \neq null$ , start loader  $L_1(ID_{sp}, ID_r)$  and loader  $L_2(ID_{lp})$ ;
4. Start the video player *VideoPlayer*.

**Algorithm:** *Loader  $L_1(c_1, c_2)$*

1. Do the following until no more data arrive on channel  $c_1$ :
  - Download one data packet on channel  $c_1$ ;
  - Store the data packet to *Buffer<sub>1</sub>*;
2. If  $c_2 \neq null$ , do the following until no more data arrive on channel  $c_2$ :
  - Download one data packet on channel  $c_2$ ;
  - Store the data packet to *Buffer<sub>2</sub>*;
3. Terminate  $L_1$ .

**Algorithm:** *Loader  $L_2(c)$*

1. Download one data packet on channel  $c$  and let it be *CurPacket*;
2. Do the following until no more data from channel  $c$  or *CurPacket* is the same as the first data packet stored in *Buffer<sub>3</sub>*:
  - Store *CurPacket* to *Buffer<sub>2</sub>*;
  - Download one data packet on channel  $c$  and let it be *CurPacket*;
3. Terminate  $L_2$ .

**Algorithm:** *VideoPlayer*

1. Playback video data cached in *Buffer<sub>1</sub>* until it is empty;
2. If there are data in *Buffer<sub>2</sub>*, playback until it is empty;
3. If there are data in *Buffer<sub>3</sub>*, playback until it is empty;
4. Terminate *VideoPlayer*.

Figure 6: Algorithms for client stations

## 4 Optimization of Double Patching

In the previous section, we presented the algorithms for both the server and client software. An important issue remains unsolved: what should the sizes of the multicast window  $w_m$  and the patching window  $w_p$  be? Their sizes dictate the overall performance of the system. We examine this issue in this section. We develop a performance model for Double Patching, and use it to derive the optimal values for  $w_m$  and  $w_p$ .

We say the values of  $w_m$  and  $w_p$  are optimal if they result in minimal requirement on the server bandwidth in providing true on-demand services, i.e., no service delay. To determine their optimal values, we derive a mathematical formula to capture the relationship between their sizes and the required server bandwidth. We recall that a regular stream and the following streams initiated before the next regular stream are said to form a *multicast group*. The following strategy can be used to compute the mean server bandwidth requirement:

- We first determine the mean total amount of data,  $D$ , transmitted by a multicast group.
- We then calculate the average time duration  $\tau$  between two consecutive multicast groups.
- The mean server bandwidth requirement can then be computed as  $\frac{D}{\tau}$ .

In our analysis, we will refer to the amount of data in terms of their playback duration. For instance, we say that the amount of data is five minutes if it takes five minutes to playback that amount of data. We note that the mean total amount of data delivered by one multicast group,  $D$ , is equal to the summation of  $D_r$ ,  $D_{lp}$ , and  $D_{sp}$ , where  $D_r$ ,  $D_{lp}$ , and  $D_{sp}$  denote the mean total amount of data delivered by the regular stream, the long patching streams, and the short patching streams, respectively, in one multicast group. They can be calculated as follows. Without loss of generality, we assume that the time clock is reset to 0 when a new multicast stream starts.

- $D_r$ , the amount delivered by the regular stream:

Since there is only one regular stream in one multicast group, we have

$$D_r = |v|,$$

where  $|v|$  is the playback length of the video.

- $D_{lp}$ , the amount delivered by the long patching streams:

We note that after the initiation of the regular stream, the mean interval of initiating a new long patching stream is  $w_p + \frac{1}{\lambda}$  time units, where  $\lambda$  is the request arrival rate. This is due to the fact that a long patching stream is always scheduled for the first request arriving  $w_p$  time units after the initiation of the regular stream or the previous long patching stream in the same multicast group. Therefore, on average there are  $\lfloor \frac{w_m}{w_p + \frac{1}{\lambda}} \rfloor$  long patching streams in one multicast group. Since a long patching stream delivers the first  $t + 2 \cdot w_p$  time units of the video data if its time skew to the regular stream is  $t$  time units, we have

$$D_{lp} = \sum_{n=1}^{\lfloor \frac{w_m}{w_p + \frac{1}{\lambda}} \rfloor} [n \cdot (w_p + \frac{1}{\lambda}) + 2 \cdot w_p].$$

- $D_{sp}$ , the amount delivered by the short patching streams:

We recall that the short patching streams in one multicast group can be organized into patching groups. On average, there are  $\lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor + 1$  patching groups inside a multicast group. We note that except the last one, each patching group delivers the same mean amount of video data as the patching window  $w_p$  is fixed. The amount of data delivered by such a patching group can be analyzed as follows. When a short patching stream is scheduled for a skew of  $t$  time units, it delivers a patch of  $t$  time units. Thus, if  $k$  short patching streams are initiated between  $t$  and  $t + \Delta t$ , the amount of data delivered by these streams can be approximated as  $k \cdot t$ , if  $\Delta t$  is small enough. If we use  $P(k, T)$  to denote the probability of initiating exactly  $k$  short patching streams during a time interval of  $T$ , then the total amount of data delivered by the short patching streams initiated between  $t$  and  $t + \Delta t$  can be computed as  $\sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t)$ . As a result, if we partition the patching window  $w_p$  into  $\lfloor \frac{w_p}{\Delta t} \rfloor$  small time slices, then the mean amount of video data delivered by this group of short patching streams can be calculated as  $\sum_{t=1}^{\lfloor \frac{w_p}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t)$ . For the short patching streams in the last group, as they are started for the requests arriving in the last  $w_m - \lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor \cdot (w_p + 1/\lambda)$  time units, the amount of data they deliver is equal to  $\sum_{t=1}^{\lfloor \frac{w_m - \lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor \cdot (w_p + 1/\lambda)}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t)$  time units. Therefore, the total

amount of data delivered by the short patching streams in one multicast group can be calculated as

$$D_{sp} = \lfloor \frac{w_m}{w_p + \frac{1}{\lambda}} \rfloor \cdot \sum_{t=1}^{\lfloor \frac{w_p}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t) + \sum_{t=1}^{\lfloor \frac{w_m - \lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor \cdot (w_p + \frac{1}{\lambda})}{\Delta t} \rfloor} \sum_{k=1}^{\infty} k \cdot t \cdot P(k, \Delta t).$$

In this paper, we assume that the request arrival process is Poisson with rate  $\lambda$ . 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 requests. Under this assumption,  $P(k, t) = \frac{(\lambda t)^k e^{-\lambda t}}{k!}$  and  $\sum_{k=1}^{\infty} k \cdot P(k, t) = t \cdot \lambda$ . If we make  $\Delta t$  equal to 1 second and keep the precision of  $w_m$  and  $w_p$  to second, then we have

$$D_{sp} = \lfloor \frac{w_m}{w_p + \frac{1}{\lambda}} \rfloor \cdot \sum_{t=1}^{w_p} t \cdot \lambda + \sum_{t=1}^{w_m - \lfloor \frac{w_m}{w_p + 1/\lambda} \rfloor \cdot (w_p + \frac{1}{\lambda})} t \cdot \lambda.$$

Since the mean interval of initiating two consecutive regular streams is equal to  $w_m + \frac{1}{\lambda}$ , the mean server bandwidth requirement is

$$Bandwidth_{DoublePatching} = \frac{D_r + D_{lp} + D_{sp}}{w_m + \frac{1}{\lambda}} \cdot b, \quad (3)$$

where  $b$  is the playback rate of the video. The above formula can be used to determine the optimal sizes for the multicast window  $w_m$  and the patching window  $w_p$ .

## 5 Performance Study

We study the performance advantage of Double Patching in this section. Since Standard Patching has been shown to outperform conventional multicast techniques by a significant margin [11], we focus on comparing our technique with Standard Patching in this study. The mean bandwidth required to achieve true on-demand services is used as the performance metric. We assume that the arrival of the service requests follows a Poisson distribution with a mean arrival rate  $\lambda$ . The mean server bandwidth required by Double Patching can be computed using the formulas presented in Section 4. Similarly, the same can be computed for Standard Patching using the formulas we derived in [3]. We note that these formulas were repeated in Section 2 (i.e., Equation 2) to make the paper self-contained. To be fair, we used optimal patching windows for both patching techniques in this study.

The performance parameters are given in Table 1. We note that the video is assumed to be encoded using MPEG-1 with an average playback rate of 1.5 Mbits/sec. We are interested in the effect of the *inter-request arrival time*, *client buffer size*, and *video length* on the mean server bandwidth requirement.

Parameter	default	variation
Number of videos	1	N/A
Normal Playback rate $b$ (Mbps)	1.5	N/A
Client buffer size $B$ (minutes)	15	2 - 30
Mean request interval $\frac{1}{\lambda}$ (seconds)	50	5 - 95
Video length $ v $ (minutes)	90	30 - 150

Table 1: Parameters

## 5.1 Effect of Request Inter-Arrival Time

In this study, we performed sensitive analysis with respect to the 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 7. It shows that Standard Patching is consistently more demanding on server bandwidth. In particular, when the inter-arrival time is very short, i.e., 5 seconds, Standard Patching requires about 130% more bandwidth compared to the new scheme. We can explain the plot in Figure 7 as follows. Under Standard Patching, patching streams have to deliver patches to bridge the time skew to the last regular multicast. As this time skew increases, the patching costs become more expensive. This condition is aggravated if the request arrival time is shorter (see Figure 7) because more patching streams would have to be issued. Double Patching addresses this drawback by using long patching streams. Although a long patching stream is slightly more expensive than a standard patching stream, the former allows many succeeding requests to use a short patching stream. The significant savings due to these short patching streams outweigh the cost of sending some extra data on the long patching stream.

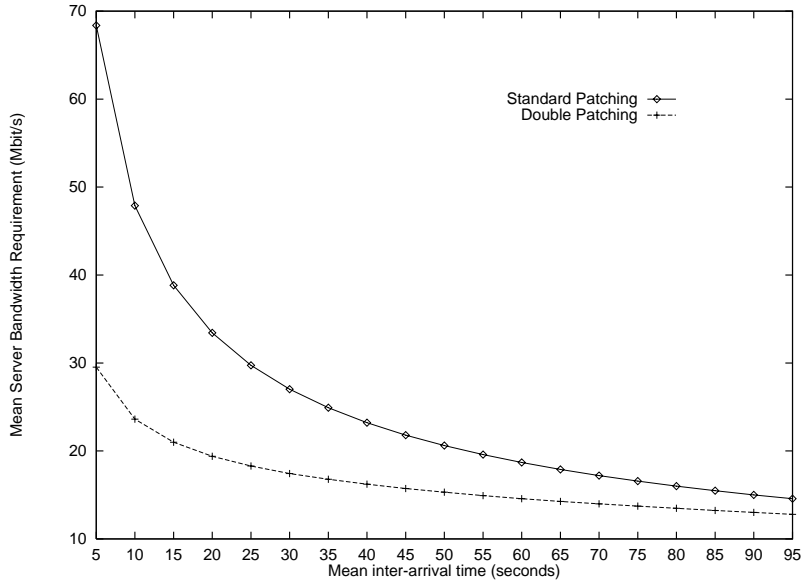


Figure 7: Effect of request inter-arrival time

## 5.2 Effect of Client Buffer Size

The purpose of this study is to investigate the tradeoff between client buffer size and server bandwidth. We fixed the mean request inter-arrival time at 50 seconds, and the video length at 90 minutes. We varied the client buffer size from 2 to 30 minutes of video data, and observed the effect on mean server bandwidth requirement under Standard Patching and Double Patching. The results are plotted in Figure 8. It shows that the two schemes perform similarly when the buffer size is small (i.e., less than 6 minutes). However, as the buffer size increases, the advantage of Double Patching becomes apparent. This can be explained as follows. Sharing the regular streams is key to reducing the demand on server bandwidth. Since Standard Patching issues regular streams relatively more frequently to keep the patching cost low, increasing the client buffer size does not help to allow more clients to share a given regular stream. In contrast, Double Patching enables many more requests to share a regular stream through long patching streams. Increasing the client buffer size, therefore, allow more service requests arriving very late to share a given regular stream. The plot in Figure 8 shows that Double Patching is more able to leverage client buffer to reduce the demand on server and therefore network bandwidth. This is a desirable property since disk space is generally much less expensive than communication bandwidth. One can hardly find a hard disk less than 10GB on today’s market. In fact, commercial video set-top boxes have already equipped with large-volume disk drive that can buffer many hours of

high quality videos. As an example, the standard SONY SVR-3000 and TiVo TCD240080 each comes with 80 hours of recording capacity from the factory.

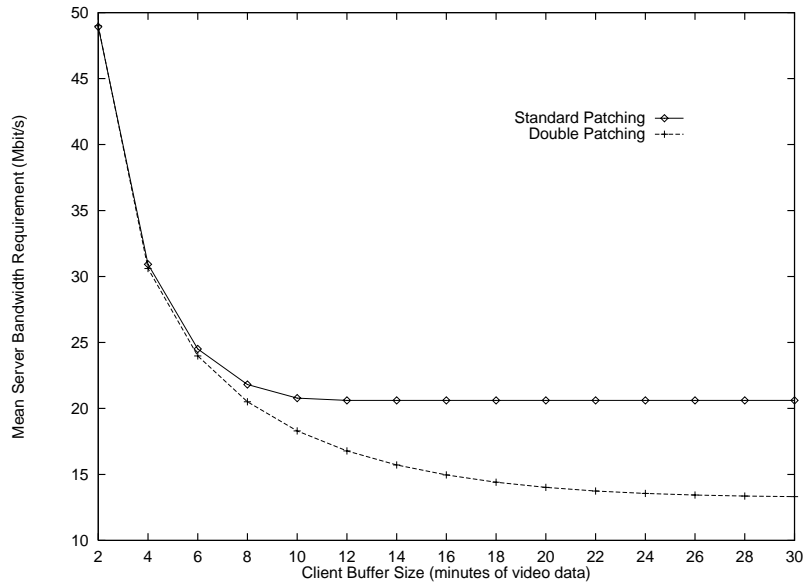


Figure 8: Effect of client buffer size

### 5.3 Effect of Video Length

In this study, we fixed the client buffer at 15 minutes, and the mean inter-arrival time at 50 seconds. We varied the video length from 30 to 150 minutes to study its effect on the performance of Standard Patching and Double Patching. The results are plotted in Figure 9. Again, we observe that Standard Patching consistently requires more bandwidth in order to provide on-demand services.

## 6 Concluding Remarks

Using a dedicated stream for each service request is a very expensive approach to provide video-on-demand services. The peculiar of video data requires a shift in the conventional thinking about how data should be transmitted. We recently proposed a new multicast technique, called *Patching* [11], to address this issue. Unlike conventional multicast, which necessarily entails service delays in order to serve multiple service requests together, a patching stream allow a client node to join an ongoing

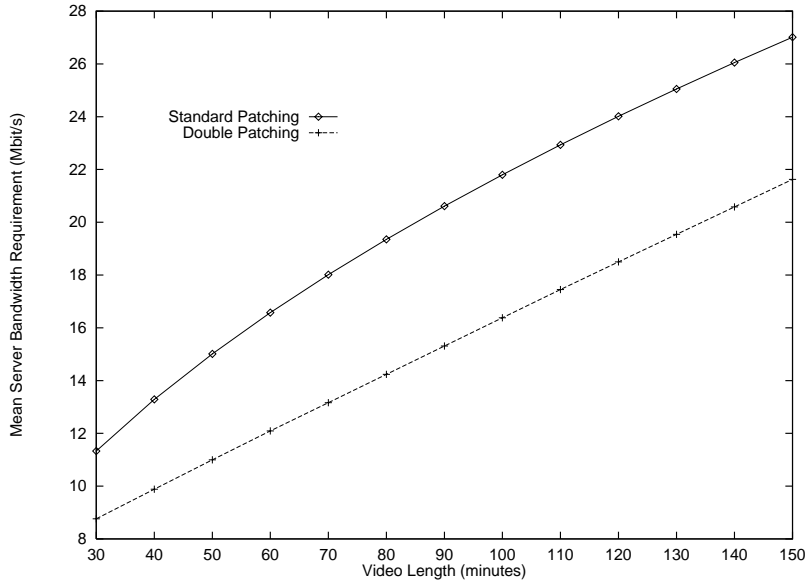


Figure 9: Effect of video length

multicast without missing the first part of the video. This novel idea of using multicast enables video-on-demand systems to leverage bandwidth sharing.

In this paper, we introduced a technique, called *Double Patching*, to address the performance limitation of our initial patching idea. Under Standard Patching, requests arriving within a patching window are able to share the same multicast using a patching stream. As time elapses, these patching streams need to “patch” more data, and therefore incur a higher communication cost. To optimize system performance, Standard Patching needs to multicast fairly frequently to keep the patching cost low. This limits data and bandwidth sharing to a small group of service requests arriving within a relatively small patching window. Double Patching overcomes this constraint by using long patching streams. Although they are slightly more expensive than a standard patching stream, they allow subsequent requests including those arriving very late to share a multicast using a short patching stream. The advantages of this strategy are twofold. First, short patching streams are significantly less expensive than the standard patching stream. Second, many more service requests can share a multicast. We note that these advantages are achieved without additional implementation cost in term of client download bandwidth requirement - the same as in Standard Patching, the new technique requires a client to download data from no more than two channels simultaneously.

To maximize the performance potential of Double Patching, we developed a probabilistic model to

analyze its performance. This analysis allowed us to design an optimal scheduler for the Double Patching approach. As we have shown in [11] that Standard Patching significantly outperforms conventional multicast (i.e., batching), we focus on comparing Double Patching with Standard Patching in this paper. Our simulation results indicate that the new method is significantly better. The performance improvement is more than double for some of the workloads.

## References

- [1] Aggarwal, C. C., J. L. Wolf, and P. S. Yu: 1996a, ‘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.
- [2] Aggarwal, C. C., J. L. Wolf, and P. S. Yu: 1996b, ‘On Optimal Batching Policies for Video-on-Demand Storage Servers’. In: *Proc. of the IEEE Int’l Conf. on Multimedia Systems’96*. Hiroshima, Japan.
- [3] Cai, Y., K. A. Hua, and K. Vu: 1999, ‘Optimizing Patching Performance’. In: *Proc. of SPIE’s Conf. on Multimedia Computing and Networking (MMCN’99)*. San Jose, CA, USA, pp. 204–216.
- [4] Carter, S. W. and D. D. E. Long: 1999, ‘Improving Bandwidth Efficiency of Video-on-Demand Servers’. *Computer Networks and ISDN Systems* **31**(1), 99–111.
- [5] Dan, A., D. Sitaram, and P. Shahabuddin: 1994, ‘Scheduling Policies for an On-Demand Video Server with Batching’. In: *Proc. of ACM Multimedia*. San Francisco, California, pp. 15–23.
- [6] Dan, A., D. Sitaram, and P. Shahabuddin: 1996, ‘Dynamic Batching Policies for an On-Demand Video Server’. *Multimedia Systems* **4**(3), 112–121.
- [7] Eager, D., M. Vernon, and J. Zahorjan: 1999, ‘Optimal and Efficient Merging Schedules for Video-on-Demand Servers’. In: *Proc. ACM Multimedia’99*. Orlando, FL, pp. 199–202.
- [8] Eager, D. L., M. K. Vernon, and J. Zahorjan: 2001, ‘Minimizing Bandwidth Requirements for On-Demand Data Delivery’. *IEEE Tras. on Knowledge and Data Engineering* **13**(5), 742–757.

- [9] Gao, L. and D. Towsley: 1999, ‘Supplying Instantaneous Video-on-Demand Services Using Controlled Multicast’. In: *Proc. IEEE International Conference on Multimedia Computing and Systems*. Florence, Italy, pp. 117–121.
- [10] Griwodz, C., M. Liepert, M. Zink, and R. Steinmetz: 1999, ‘Tune to Lambda Patching’. In: *2nd Workshop on Internet Server Performance (WISP’99)*. Atlanta, GA, U.S.A.
- [11] Hua, K. A., Y. Cai, and S. Sheu: 1998, ‘Patching: A Multicast Technique for True Video-on-Demand Services’. In: *Proc. of ACM Multimedia*. Bristol, U.K., pp. 191–200.
- [12] Hua, K. A., J.-H. Oh, and K. Vu: 2002, ‘An Adaptive Video Multicast Scheme for Varying Workloads’. *ACM Multimedia Systems* **8**(4), 258–269.
- [13] Hua, K. A. and S. Sheu: 1997, ‘Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan Video-On-Demand Systems’. In: *Proc. of the ACM SIGCOMM’97*. Cannes, France.
- [14] Paris, J. F., S. W. Carter, and D. D. E. Long: 1999, ‘Efficient Broadcasting Protocols for Video on Demand’. In: *Proc. of SPIE’s Conf. on Multimedia Computing and Networking (MMCN’99)*. San Jose, CA, USA, pp. 317–326.
- [15] Sen, S., L. Gao, J. Rexford, and D. Towsley: 1999, ‘Optimal Patching Schemes for Efficient Multimedia Streaming’. In: *Proc. IEEE NOSSDAV’99*. Basking Ridge, NJ, U.S.A.
- [16] Sheu, S., K. A. Hua, and W. Tavanapong: 1997a, ‘Chaining: A Generalized Batching Technique for Video-On-Demand’. In: *Proc. of the Int’l Conf. On Multimedia Computing and System*. Ottawa, Ontario, Canada, pp. 110–117.
- [17] Sheu, S., K. A. Hua, and W. Tavanapong: 1997b, ‘Dynamic Grouping: An Efficient Buffer Management Scheme for Video-on-Demand Servers’. In: *Proc. of 2nd Int’l Conference on Multimedia Information Systems*.
- [18] Viswanathan, S. and T. Imielinski: 1996, ‘Metropolitan Area Video-on-Demand Service Using Pyramid Broadcasting’. *Multimedia systems* **4**(4), 179–208.

## Appendix

To determine the optimal value for  $w$  under standard patching (when  $Bandwidth_{StandardPatching}$  is minimized), we compute the derivative of the  $Bandwidth_{StandardPatching}$  with respect to  $w$ . Since  $D_{[0,w]}$ , in Equation 2, is defined differently for four different subdomains of  $w$  (see Equation 1), we need to consider four different cases in deriving the derivative as follows:

- $w = 0$ : Since this subdomain has only one value, the optimal patching window is

$$w_{opt} = 0 \tag{4}$$

regardless of  $\lambda$ .

- $0 < w \leq B$ : In this subdomain,  $Bandwidth_{StandardPatching} = \frac{|v| + \frac{w \cdot (w+1)}{2} \cdot \lambda}{w + \frac{1}{\lambda}} \cdot b$ , according to Equations 1 and 2. By analyzing the derivative of the above function with respect to  $w$ , we can find that:

- if  $0 < w \leq \frac{\sqrt{2 \cdot |v| \cdot \lambda^2 - \lambda + 1} - 1}{\lambda}$ ,  $Bandwidth_{StandardPatching}$  decreases as  $w$  increases.
- if  $w > \frac{\sqrt{2 \cdot |v| \cdot \lambda^2 - \lambda + 1} - 1}{\lambda}$ ,  $Bandwidth_{StandardPatching}$  increases as  $w$  increases.

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

$$w_{opt} = \begin{cases} \frac{\sqrt{2 \cdot |v| \cdot \lambda^2 - \lambda + 1} - 1}{\lambda} & \text{if } 0 \leq \frac{\sqrt{2 \cdot |v| \cdot \lambda^2 - \lambda + 1} - 1}{\lambda} \leq B, \\ B & \text{otherwise.} \end{cases} \tag{5}$$

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

$$w_{opt} = |v| - B \tag{6}$$

as another potential optimal patching window.

- $|v| - B < w \leq |v|$ : In this subdomain, we have:

$$Bandwidth_{StandardPatching} = \frac{D_{[0,|v|-B]} + \frac{(w-|v|+B)(w+|v|-B+1)}{2} \cdot \lambda}{w + \frac{1}{\lambda}} \cdot b$$

according to Equations 1 and 2. By analyzing the derivative of the above function with respect to  $w$ , we can find that the bandwidth requirement is minimized when  $w$  has the following value:

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

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 10, we discuss the optimal patching window for each of the three cases as follows.

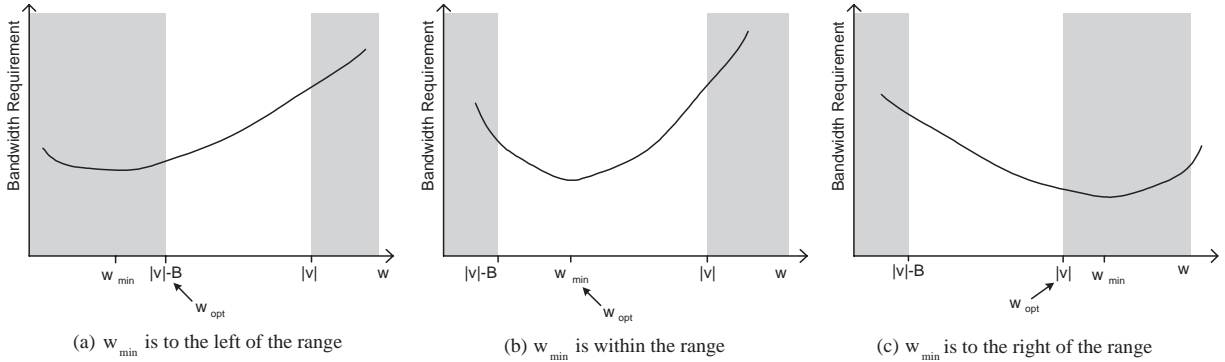


Figure 10: Three possible optimal patching windows

- $w_{min} \leq |v| - B$ : Since the bandwidth curve is increasing for  $w \geq w_{min}$ , the smallest bandwidth requirement for the range  $(|v| - B, |v|]$  occurs when  $w \rightarrow (|v| - B)^+$  as illustrated in Figure 10(a). This minimal value is the same as the  $Bandwidth_{StandardPatching}$  value when  $w = |v| - B$ . We have considered this case previously.
- $|v| - B < w_{min} \leq |v|$ : As illustrated in Figure 10(b), the optimal window size for the range  $(|v| - B, |v|)$  is:

$$w_{opt} = w_{min} = \frac{-1 + \sqrt{\lambda^2 \cdot (B - |v|) \cdot (|v| - B + 1) + \lambda \cdot (2 \cdot D_{[0,|v|-B]} - 1) + 1}}{2 \cdot \lambda}. \quad (7)$$

- $w_{min} > |v|$ : This case is illustrated in Figure 10(c). Since the bandwidth curve is decreasing for  $w \leq w_{min}$ , 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

$$w_{opt} = |v|. \quad (8)$$

In summary, from equations 4 through 8, the optimal patching window can be determined from the following six candidates:

- $w_1 = 0$
- $w_2 = \frac{\sqrt{2 \cdot |v| \cdot \lambda^2 - \lambda + 1} - 1}{\lambda}$
- $w_3 = B$
- $w_4 = |v| - B$
- $w_5 = \frac{-1 + \sqrt{\lambda^2 \cdot (B - |v|) \cdot (|v| - B + 1) + \lambda \cdot (2 \cdot D_{[0, |v| - B]} - 1) + 1}}{2 \cdot \lambda}$
- $w_6 = |v|$ .

We note that  $w_2$  is a valid candidate only if  $0 < w_2 \leq B$ . Similarly,  $w_5$  is valid only if  $|v| - B < w_5 \leq |v|$ . For a given buffer capacity  $B$  and a video  $v$  with a request rate of  $\lambda$ , we first compute these six candidates of patching window. Then, for each valid patching window, we calculate the corresponding  $Bandwidth_{StandardPatching}$  using Equation 2. Finally, we select the patching window corresponding to the minimal  $Bandwidth_{StandardPatching}$  as the optimal patching window for  $v$ .