Optimizing Patching Performance*

Ying Cai *Kien A. Hua* *Khanh Vu*

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

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

ABSTRACT

Patching has been shown to be cost efficient for video-on-demand systems. Unlike conventional multicast, patching is a dynamic multicast scheme which enables a new request to join an ongoing multicast. Since a multicast can now grow dynamically to serve new users, this approach is more efficient than traditional multicast. In addition, since a new request can be serviced immediately without having to wait for the next multicast, true video-on-demand can be achieved. In this paper, we introduce the notion of *patching window*, and present a generalized patching method. We show that existing schemes are special cases with a specific patching window size. We derive a mathematical formula to help determine the optimal size for the patching window. This formula allows us to design the best patching scheme given a workload. The proposed technique is validated using simulations. They show that the analytical results are very accurate. We also provide performance results to demonstrate that the optimal technique outperforms the existing schemes by a significant margin. It is also up to two times better than the best Piggybacking method which provides data sharing by merging the services in progress into a single stream by altering their display rates.

Keywords: Multicast, patching, multimedia communication, optimization, performance analysis, simulation

1. INTRODUCTION

Video on Demand (VOD) is a critical technology for many important multimedia applications, such as home entertainment, digital video libraries, distance learning, company training, news on demand, electronic commerce, just to name a few. A typical VOD service allows remote users to playback any video from a large collection of videos stored on one or more servers. In response to a service request, a video server delivers the video to the user in an isochronous video stream. The system resource required to sustain a video stream is referred to as a *video channel*. Typically, the number of channels a video server can support is determined by its communication bandwidth. This is due to the fact that while the storage subsystem can be designed to fully exploit the very high bandwidth of the system bus, the performance of the NICs is constrained by the much lower bandwidth of the external networking environment. This phenomenon is referred to as *network-I/O bottleneck* in [1–3].

In practice, the number of channels available to a video server is very limited. To make VOD affordable, the precious channels can be shared among many users - Early requests for a video can be made to wait for more requests to arrive, and the entire group is served in one multicast. This is referred to as *batching* in [4,5]. In this approach, since most requests are forced to wait, only near VOD [6–8] can be achieved. It is desirable to keep the maximum waiting time short. The benefit of multicast, however, will diminish. On the other hand, if users making the early requests are kept waiting too long, then they are likely to renege. To address this dilemma, a technique called *Patching* was introduced in [3].

The basic idea of patching is as follows. A new service request can exploit an existing multicast by buffering the future stream from the multicast while playing the new start-up flow from the start. Once the new flow has

*This research is partially supported by the National Science Foundation grant ANI-9714591.

been played back to the skew point, the catchup flow can be terminated and the original multicast can be shared (the skew is absorbed by the new client's buffer). We note that the multicast paths are built at the application level rather than by routers. Allowing clients to join an existing multicast significantly improves the efficiency of the multicast. Besides, since requests can receive the service immediately, true VOD can be achieved. In this paper, a technique is said to provide true VOD if it can service requests with no delay. A significant contribution of patching, therefore, is allowing true VOD systems to take advantage of multicast. This is an interesting accomplishment since true VOD and multicast are seemingly two conflicting features.

We note that the name "Patching" alludes to the fact that majority of the time the channels are used to patch the missing portion of a service, rather than having to multicast the video in its entirety. In the former case, we say that the channel is used to deliver a *patching stream*. Given that there is an existing multicast of a video, when to schedule another multicast for that same video is a critical factor. The time period after a multicast, during which patching must be used, is referred to as the *patching window* in this paper. If the patching window is set too small, multicasts are scheduled very frequently. To an extreme, the patching window can be narrowed to zero, in which case all multicasts serve only one client and Patching degenerates into the conventional VOD design, i.e., one dedicated channel per service request. On the contrary, if the patching window is made too wide, the average time distance between the initiation of a patching stream and the last multicast becomes too large. Under this circumstance, since most channels are not used to multicast data, again patching offers little benefit.

In [3], we examines two simple approaches to determine the size of patching window.

- The first one uses the length of the video as the patching window. That is, no multicast is initiated as long as there is an existing multicast for the video. In case that the client buffer space is not large enough to absorb the skew as described previously, the buffer is used to cache the tail portion of the video. This approach is called *Greedy Patching* because it tries to exploit an existing multicast as much as possible.
- Overly greedy can actually result in less data sharing [3]. The other scheme, called *Grace Patching*, uses a patching stream for the new client only if it has enough buffer space to absorb the skew. A new regular stream is scheduled otherwise. Hence, under Grace Patching, the patching window is determined by the client buffer size.

In this paper, we argue that setting the size of the patching window according to the video length or the client buffer size is aimless and does not always result in good performance. We show that the request rate of the video must also be taken into account. With this observation, we generalize the patching technique by proposing a method to determine the optimal patching window for each video. This extension allows us to substantially improve on our previous work. Another major difference between this paper and [3] is the performance studies. In [3], simulation results were provided to show how much patching can improve on conventional batching. In the current paper, in addition to comparing the new approach with Grace patching, we also demonstrate that patching is up to two times better than *Piggybacking* [9,10], which merges the services in progress into a single stream by altering their display rates. We will discuss this scheme in more detail later.

The remainder of this paper is organized as follows. We describe related work in Section 2. Patching is presented in Section 3. In Section 4, we introduce a technique to optimize the patching performance. We discuss our performance study in Section 5. Finally, we give our concluding remarks in Section 6.

2. RELATED WORKS

We first exploited the idea of letting clients of the same multicast to receive the service at their own earliest possible time in [2]. The technique was called *Dynamic Multicast* or *Chaining*. Unlike conventional multicast which must first determine the multicast tree before the multicast can proceed, a multicast tree in Dynamic Multicast grows dynamically at the application level to accommodate late requests for the same service. This approach requires a small additional disk space at the client side to buffer data. Each client also acts as a mini-server to forward the cached data to other clients in the downstream. The aggregate storage space of these clients effectively forms a huge network cache temporarily holding data for future requests. As long as the first

part of the video is still in the multicast tree, i.e., in some client's buffer, the next batch of requests for the same video can join this tree as its newest generation. It was shown in [2] that latency and throughput can be vastly improved compared to batching. This scheme is very scalable because the clients using the service also contribute their resources (i.e., buffer space and forwarding bandwidth) to the community. In this way, each client can be seen as a contributor, rather than just a burden to the video server. This feature allows Dynamic Multicast to scale beyond the limitation of regular batching. Implementing this novel idea, however, is a great challenge. The control mechanism is quite complex. If a forwarding client decides to turn off its system, the receiving client must promptly switch to a sibling of the departing client. If there is no sibling left, the server must be able to send an emergency stream within a short notice to support the affected client now detached from the multicast tree. Due to these difficulties, we have designed Patching for our prototype system [†]. We note that Patching can be seen as another form of dynamic multicast in the sense that its multicast trees also expand dynamically to serve late requests.

Another technique which allows clients arriving at different times to share a data stream is called *Adaptive Piggybacking* [9]. An adaptive piggybacking procedure is defined to be a policy for altering display rates of requests in progress (for the same video), for the purpose of "merging" their respective video streams into a single stream that can serve the entire group of merged requests. Let us consider a client which is currently served by some video channel. Sometime later, another request for the same video arrives, the server dispatches another channel to serve this new request. At this time, the server slows down the data rate on the former channel, and speeds up that of the later channel. The affected clients must adapt accordingly to the new playback rates. Once the second stream catches up with the first stream, they are merged into a single multicast freeing one of the two channels. Obviously, this approach can improve the service latency as compared to simple batching. A limitation of this technique is that the variation of the playback rate must be within, say $\pm 5\%$, of the normal playback rate, or it will result in a perceivable deterioration of the playback quality. This fact limits the number of streams that can be merged, and therefore the effectiveness of Adaptive Piggybacking. As an example, let us consider a stream A which started six minutes before a stream B. If B is adjusted to a speed 5% faster than the normal playback rate, it will take B 114 minutes to catch up with A. Under this condition, if the video is 120 minutes long, stream A will likely finish before B can catch up. Later in this paper, we will give simulation results which shows that patching is up to two times better than the best piggybacking technique [10]. In terms of implementation, Adaptive Piggybacking is also quite complex. Although techniques are available to "time compress" movies, dynamically changing speed is a much harder problem. One cannot simply use several versions of each video to support the different playback rates since the display adjustment must be gradual to insure that it is not noticeable to the user. For this technique to work, more work on specialized hardware will be necessary to support on-the-fly modification [9]. This specialized hardware is likely more expensive than the general purpose off-the-shelf component, i.e., disk buffer, used in patching.

Another related technique, called *Bridging*, is presented in [11]. Bridging is a buffer management method. In this scheme, data read for a leading stream are held in the server buffer, and trailing requests are serviced from this buffer instead of issuing another storage-I/O stream. This technique allows multiple requests to share a storage-I/O stream. However, it does not reduce the demand on the network-I/O bandwidth.

Note that for highly populous video, *Periodic Broadcast* can be used to improve the system performance. The latest periodic broadcast techniques can be found in [12–15]. These schemes divide the server bandwidth into a large number of logical channels with equal bandwidth. To broadcast a video over its, say K, dedicated channels, the video file is partitioned into K fragments of increasing sizes, each of which is repeatedly broadcast on its own channel. To play back a desired video, a client tunes into the appropriate channel to download the first data fragment of this video at the first occurrence. As these data are arriving at the client, they are rendered onto the screen. For the subsequent fragments, the client downloads the next fragment at the earliest possible time after beginning to play back the current fragment. Thus, at any point, the client downloads from at most two channels and consumes the data fragment from one of them in parallel. To ensure the continuous playback, the size of the data fragments must be chosen such that the playback duration of any fragment is longer than the worst latency in downloading the next fragment. To achieve low service latencies, the size of the first fragments can be made very small to allow them to be broadcast more frequently. Although this approach is very efficient,

[†]We are implementing a multicast-based VOD system using one server and ten workstations interconnected through an ATM switch.

it can only be used for very popular videos. In this paper, we focus on the more general multicast approach.

3. PATCHING

In this section, we generalize the patching technique using the notion of *patching window*. We first present the server design, and then discuss the receiving and playback strategies at the client end.

3.1. Server Design

Most of the server bandwidth is multiplexed into a set of logical channels, each is capable of transmitting a video at the playback rate. The remaining bandwidth of the server is used for control messages such as service requests and service notifications. Each 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*. In the former case, the channel is said to play the role of a *regular channel*. In the latter, it is referred to as a *patching channel*. If a client station tunes to a regular channel to download its 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, then we refer to this data stream as a *patching stream*. These terminologies are illustrated in Figure 1.

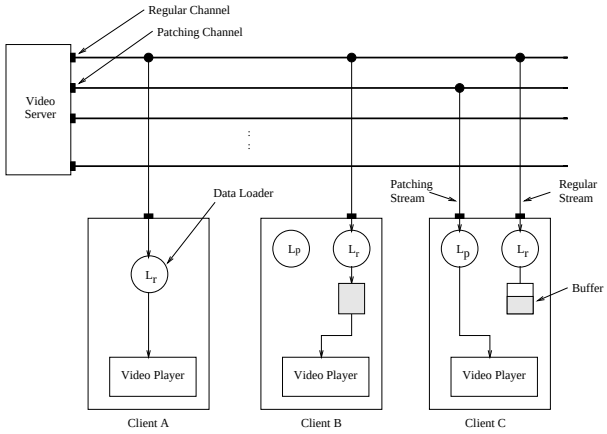

Figure 1. Three possible patching stages.

The server maintains a job queue, and all arriving requests are first appended to this queue for dispatch at the next occasion. The next occasion arrives when a channel becomes free. When a channel is available, say at time t, the server checks the job queue for any entries, and admits all the requests for some video according to some scheduling policy. To determine whether to start a new regular multicast, the server computes the *skew* which is defined as the time difference between t and the last regular multicast of the same video. If the skew is greater than the patching window, a new regular multicast is initiated; otherwise, a patching multicast is used to deliver only the patching data. Let $|v|$ denote the length of the requested video v in time unit. We use $v[t]$ to represent the first t seconds of the video v. Let B be the size of the client buffer in playback time unit. The patching clip is determined as follows:

- If $0 < skew \leq B$, the client buffer is able to bridge the skew. The patching clip is $v[skew]$.
- If $B < skew \leq |v| - B$, the client buffer is not large enough to bridge the skew, but can be used to buffer the last B seconds of the last regular multicast. In this case, the patching clip is $v[|v| - B]$.
- if $|v| - B < skew < |v|$, the client can buffer only the last $|v| - skew$ time units of the last regular multicast. The patching clip, under this condition, is $v[skew]$.

We note that a job queue is used to handle occasional bursts of requests as in any other client-server systems. Most requests, however, should be able to receive the service immediately upon arrival at the job queue, unless the system resources are inadequate. We note that although batching can also be used here to reduce the number of patching multicasts, we are interested in true VOD and will not consider this option in this paper.

3.2. Client Design

To support patching, each client station needs to have three threads of control: two data loaders and a video player. The two data loaders are responsible for downloading data from the patching channel and the regular channel, respectively. Initially, the video player plays back the patching stream as the data arrive. The regular stream is temporarily cached in the client buffer. When the patching stream ends, the video layer switches to play back the data in the buffer as the remaining data continue to arrive in the regular stream.

We show in Figure 1 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 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.

4. OPTIMIZING PATCHING PERFORMANCE

From the description of patching presented in the last section, one can observe that the size of the patching window has a profound influence on the performance of the overall system. In this section, we first give some motivation examples. We then derive a mathemetical model for determining the optimal window size. This model is validated using simulation.

4.1. Motivation Examples

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 arrival rate is one request per minute. Thus, the server must decide at each minute whether to initiate a patching or regular multicast. This decision is influenced by the size of the patching window. Using this example, let us examine the performance of patching under various patching window sizes in the following:

- **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. We note that this scheme is not very efficient since a large number of clients can share only the tail 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 others are patching multicasts. They deliver a total of $\sum_{i=1}^{15} i + \sum_{i=1}^{13} i = 211$ minutes of data. In total, the server delivers only $60 + 211 = 271$ minutes of data.
- **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 others are patching multicasts. Together, they deliver $5 * \sum_{i=1}^{5} i = 75$ minutes of video data. Thus, the total amount of video data transmitted is 225 minutes.

We note that Greedy and Grace use a patching window of 30 minutes and 15 minutes, respectively. Although Grace Patching is substantially better than Greedy Patching in this example, using a patching window size of 5 minutes improves the performance by another 20%. The question then is, what is the optimal patching window size ? To answer this question, we derive a mathematical model to investigate the effect of patching window size on the demand of the server bandwidth.

4.2. Optimal Patching Window

We say that a patching window size is optimal if it results in the minimal requirement on the server bandwidth. To determine the optimal patching window size, we derive a mathematical formula to capture the relationship

between the patching window size and the required server bandwidth. In this paper, a regular multicast and all the following 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 $\frac{D}{\tau}$.

In our analysis, we will refer to the amounts 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 use the following notations in our analysis. D_t denotes the amount of data delivered by the multicast initiated at some time t. $D_{(t_1,t_2]}$ and $D_{[t_1,t_2]}$ represent the mean total amount of data delivered by the multicasts initiated during the time period $t_1 < t \leq t_2$ and $t_1 \leq t \leq t_2$, respectively. $P(k,t)$ the probability of having k multicasts initiated in t time units.

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. We compute the mean total amount of data transmitted by a multicast group under various window sizes as follows.

1. $W(v) = 0$: Under this condition, Patching becomes a 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

$$D_0 = |v|,$$

   where $|v|$ denotes the video length.

2. $0 < W(v) \leq B$: 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 multicast initiated at time t, $0 < t \leq W(v)$, is t time units. If k 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

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

   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 1. For a multicast initiated at time t in the second time interval, since the client buffer size is not large enough to cache the next t time units of the video, patching uses the buffer space to cache the last B time units of the regular multicast instead. This patching multicast, therefore, 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 $k \cdot (|v| - B)$. As 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 a multicast group is:

$$D_{[0,W(v)]} = D_{[0,B]} + \sum_{k=1}^{\infty} k \cdot (|v| - B) \cdot P(k, W(v) - B) \tag{2}$$

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 2. 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

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

If we make Δt equal to 1 second and keep the precision of $W(v)$ to second, then we have

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

In this paper, we assume that the multicast initiation process is Poisson with rate λ. 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.

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

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.

$$Server\_Bandwidth = \frac{D_{[0,W(v)]}}{W(v) + \frac{1}{\lambda}} \cdot b, \tag{3}$$

where b is the video playback rate. Using the above formula, we can determine the optimal patching window size. As an example, the plots in Figure 2 indicate that the optimal patching window size is 5 minutes for the given workload because the demand on the server bandwidth is minimum at this setting. We will discuss Figure 2 in more detail shortly. For the moment, let us apply Equation 3 to compute the server bandwidth requirement for the various patching schemes.

- For the traditional baseline algorithm which does not take advantage of the client buffer, its patching window size can be regarded as 0. The mean server bandwidth required is, therefore, as follows.

$$Server\_Bandwidth_{baseline} = D_0 \cdot \lambda \cdot b = |v| \cdot \lambda \cdot b$$

- For Greedy Patching, the patching window size is the length of the video, i.e., $W(v)_{greedy} = |v|$. Thus, the required server bandwidth is

$$Server\_Bandwidth_{greedy} = \frac{D_{[0,|v|]}}{|v| + \frac{1}{\lambda}} \cdot b$$

- For Grace Patching, the size of patching window is determined by the client buffer size B, i.e., $W(v)_{grace} = B$. Thus, the required server bandwidth is

$$Server\_Bandwidth_{grace} = \frac{D_{[0,B]}}{B + \frac{1}{\lambda}} \cdot b$$

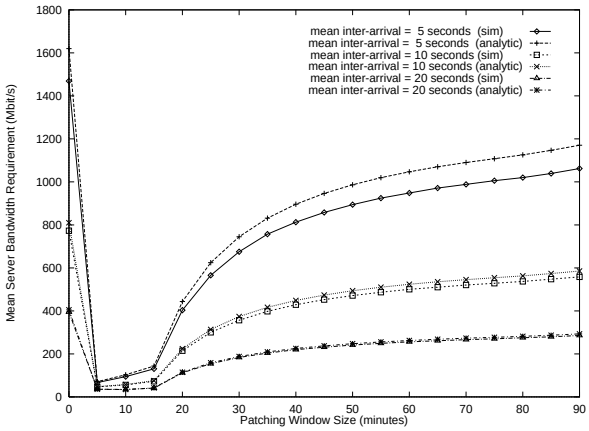

Figure 2. Validation of Analysis of Patching Algorithm.

4.3. Validation of the Analythical Model

We validated Equation 3 using simulation. In this study, we fixed the video length at 90 minutes and the client buffer size at 15 minutes of video data. We created three sequences of client requests according to a Poisson distribution with a mean inter-arrival time of 5, 10, and 20 seconds, respectively. Each sequence contains 200,000 requests. The size of patching window was varied between 0 and 90 minutes for each sequence. For each patching window size, the simulator counts the data transmitted by the server under the true VOD requirement. This amount is then divided by the time interval between the first and the last request to compute the mean server bandwidth requirement.

Both the simulation results and the results computed using Equation 3 are plotted in Figure 2. We observe that the formula is quite accurate in predicting the server bandwidth requirement. The slight error comes from the fact that the request arrival times are rounded to the next minute in our simulation. For example, two different requests arriving at the 2.5 second and 2.8 second theoretically require two separate multicasts in Equation 3. They are regarded as arriving at the same time, at the third second, in the simulation, and are served together using a single multicast. This explains the larger error under the smaller average inter-arrival times. Under this condition, more requests arrive within one second increasing the chance that multiple requests are served by one multicast. For a given mean inter-arrival time, we observe that the worst error occurs when the patching window size is 0. This is due to the fact that all requests arriving within a one-second interval must share a multicast in the simulation. The fact that this multicast must transmit the video in its entirety causing the error to be most severe.

5. PERFORMANCE STUDY

For convenience, we refer to the proposed technique as *Optimal Patching*. In this section, we compare the performance of Optimal Patching with that of Piggybacking [9] which is most relevant to our work. We focus on *Equal-Split* strategy as the merging algorithm for piggybacking because it has been shown to be the most efficient [10]. To demonstrate the significance of using an appropriate patching window, we also compare the new technique to Grace Patching, the best patching scheme so far. We decide not to include Greedy Patching in our study because it rarely performs better than Grace Patching [3]. We select the mean server bandwidth required to support true VOD as our performance metric. Thus, a better scheme should require less server bandwidth in order to ensure true VOD as the QoS. We assume that the server has only one video. If a system has n videos, the corresponding mean system bandwidth requirement is simply the summation of the bandwidth required to support the true VOD services 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 patching to support true VOD can be computed using Equation 3. Similarly, the same can be computed for Piggybacking using the formulas derived and validated in [10]. To make our paper self-contained, we brief describe the best Piggybacking technique, Equal-Split Merging algorithm, in the following. We will also provide the formula for its bandwidth requirement. However, we will not repeat the derivation details for brevity's sake. The interested reader is referred to [10] for more detail.

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 algorithm 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 notations used in the above formula are explained 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% to keep the variation in the playback rate within 5%. A higher percentage would cause noticeable sudden distortion and degradation in the video and audio quality. $Cost(w,n)$ is the amount of data delivered by the n streams initiated within the time interval w before they are finally merged into a single stream. The formula for $Cost(w,n)$ is

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

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. 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 [10]. 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.

5.1. 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 size of the client buffer was varied from 0 to 90 minutes of video data. The effect of client buffer size on the mean server bandwidth requirement is plotted in Figure 3. The curve for Piggybacking is flat because it does not require a buffer at the receiving end. This, however, is achieved at the costs of specialized hardware. The simulation results shown in Figure 3 indicate that patching can match the performance of piggybacking using a buffer space as small as two minutes. If the video is encoded using MPEG-2, two minutes of the video require a disk space of only 60 MBytes which costs about U.S. $4 today. Since any specialized hardware will likely be a lot more expensive than $4, patching is a much less expensive approach to achieve data sharing. We also observe

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

that the benefit of patching is not limited to its low cost. If we can afford $10 to provide the client station with 150 MBytes of disk space, a 200% improvement over piggybacking is possible.

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 3, 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 for each service 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, the new technique is able to recognize the appropriate amount of buffer space to utilize. It is shown in Figure 3 that Optimal Patching ignores any buffer space beyond 10 minutes in order to maintain the good performance, i.e., two times better than piggybacking. In practice, our formula can be used to determine the amount of disk space required for patching.

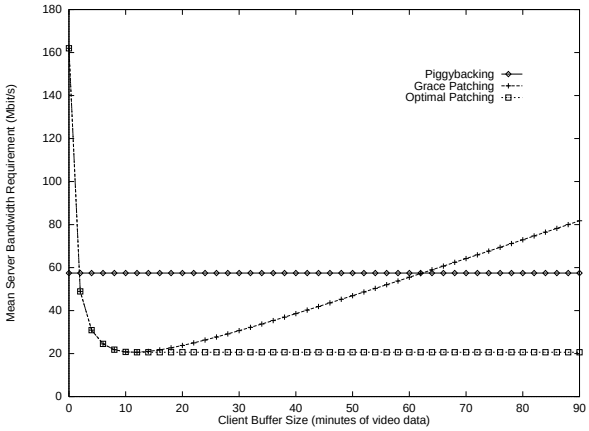

Figure 3. Effect of client buffer size.

5.2. 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 4. In general, the benefit of any multicast technique decreases with the increases in the inter-arrival time. Optimal Patching, however, can consistently outperform Piggybacking by a wide margin (i.e., between 100% and 150% under this workload). The performance gap decreases with the increases in the inter-arrival time. This is due to insufficient buffer space. When the inter-arrival time is large, most 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.

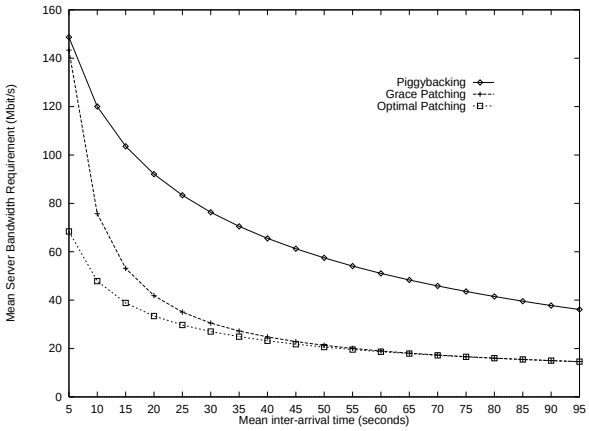

Figure 4. Effect of inter-arrival time.

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. 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 [10] that this setting results 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 5.

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 deteriorating the quality of the playback. This observation suggests that piggybacking is not a good approach for applications which involve long videos such as home entertainment, distance learning, etc.

On the contrary, Figure 5 shows that patching is much less sensitive to the length of the video because it allows a new request to join an earlier multicast immediately, without the catch-up delay. Comparing the two patching techniques, 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. Again, this study indicates that an appropriate choice of the patching window size is essential to the good performance of the patching approach.

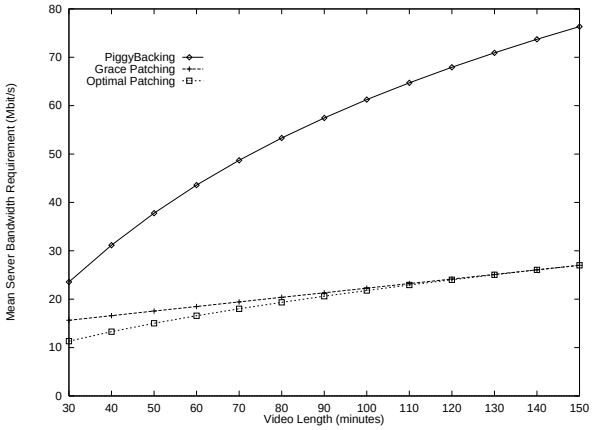

Figure 5. Effect of video length.

6. CONCLUDING REMARKS

Multicast has been shown to be an excellent technique for reducing the demand on the server bandwidth. Unfortunately, due to its inherent limitation, multicast can only be used to provide near VOD services. We have presented a novel technique, called *Patching*, to extend the capability of standard multicast to support true VOD. This approach allows requests to be serviced immediately without having to wait for the next multicast. Allowing new users to join an existing multicast also makes patching more efficient than conventional multicast.

The contribution of this paper is in optimizing the performance of patching. We generalized this approach by introducing the concept of patching window. All existing patching schemes can be seen as differing only in the way the patching window is determined. We showed that the best patching strategy so far, Grace Patching, is not optimal. To design the best patching technique, we develop a formula to help determine the optimal patching window size. This formula can be used in practice to reserve the right amount of disk space for patching.

We also compared patching with the best piggybacking technique which also allows data sharing. This approach alters the playback rates of existing services in order to merge them into a single stream. Inherently, piggybacking can never match the performance of patching due to the catch-up delay. Our performance results show that more than 200% improvement is achievable using patching. Our study also indicates that the demand on server bandwidth, under piggybacking, increases sharply as the length of the video increases. In terms of hardware costs, we observed in our study that patching required only U.S. $4 of disk space (i.e., 60 MBytes) to achieve the same performance of piggybacking which requires specialized hardware to alter the playback rate on the fly. Since any specialized hardware is likely to cost substantially more than $4, patching offers a more cost efficient solution.

REFERENCES

1. R. Rooholamini and V. Cherkassky, "Atm-based multimedia servers," *IEEE Multimedia* **2**, pp. 39–52, Spring 1995.
2. S. Sheu, K. 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*, pp. 110–117, (Ottawa, Ontario, Canada), June 1997.
3. K. A. Hua, Y. Cai, and S. Sheu, "Patching: A multicast technique for true video-on-demand services," in *Proc. of ACM Multimedia*, pp. 191–200, (Bristol, U.K.), September 1998.
4. D. P. Anderson, "Metascheduling for continuous media," *ACM Trans. on Computer Systems* **11**, pp. 226–252, August 1993.
5. A. Dan, D. Sitaram, , and P. Shahabuddin, "Scheduling policies for an on-demand video server with batching," in *Proc. of ACM Multimedia*, pp. 15–23, (San Francisco, California), October 1994.
6. T. Little and D. Venkatesh, "Popularity-based assignment of movies to storage devices in a video-on-demand system," *Multimedia Systems* **2**, pp. 280–287, January 1995.
7. 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), pp. 1110–1122, 1996.
8. E. L. Abram-Profeta and K. G. Shin, "Scheduling video programs in near video-on-demand systems," in *Proc. of ACM Multimedia*, (Seatlle, Washington), November 1997.
9. 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), pp. 140–155, 1996.
10. S. Lau, J. Lui, and L. Golubchik, "Merging video streams in a multimedia storage server: complexity and heuristics," *ACM Multimedia Systems* **6**, pp. 29–42, 1998.
11. M. Kamath, K. Ramaritham, and D. Towsley, "Continuous media sharing in multimedia database systems," in *Proc. of the 4th DASFAA'95*, (Singapore), April 1995.
12. S. Viswanathan and T. Imielinski, "Metropolitan area video-on-demand service using pyramid broadcasting," *Multimedia systems* **4**, pp. 179–208, August 1996.
13. 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.
14. 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.
15. K. A. Hua, Y. Cai, and S. Sheu, "Exploiting client bandwidth for more efficient video broadcast," in *7th Int'l Conference on Computer Communication and Networking*, pp. 848–856, (Louisiana, U.S.A), October 1998.