Patching : A Multicast Technique for True Video-on-Demand Services*

Kien A. Hua *Ying Cai* † *Simon Sheu*

School of Computer Science
University of Central Florida
Orlando, FL 32816-2362
U. S. A.
E-mail: {kienhua, cai, sheu}@cs.ucf.edu

Abstract

Until now, true video-on-demand can only be achieved using a dedicated data flow for each service request. This brute-force approach is prohibitively expensive. Using multicast can significantly reduce the system cost. This solution, however, must delay services in order to serve many requests as a batch. In this paper, we consider a third alternative called *Patching*. In our technique, an existing multicast can expand dynamically to serve new clients. Allowing new clients to join an existing multicast improves the efficiency of the multicast. Furthermore, since all requests can be served immediately, the clients experience no service delay and true video-on-demand can be achieved. A significant contribution of this work, is making multicast work for true video-on-demand services. In fact, we are able to eliminate the service latency and improve the efficiency of multicast at the same time. To assess the benefit of this scheme, we perform simulations to compare its performance with that of standard multicast. Our simulation results indicate convincingly that Patching offers substantially better performance.

1 Introduction

Video on Demand (VOD) is a critical technology for many important multimedia applications, such as home entertainment, digital video libraries, distance learning, 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 video servers. In response to a service request, a video server delivers the video to the user in an isochronous video stream. Each *video stream* can be viewed as a concatenation of a storage-I/O stream and a communication stream. That is, sufficient storage-I/O bandwidth must be available to continuously transfer the data from the storage subsystem to the *network interface card* (NIC); and the NIC in turn must have enough free bandwidth to forward the data to the user. Obviously, the smaller of the I/O bandwidth and the interconnection bandwidth determines the maximum number of concurrent video streams the server can support simultaneously (i.e., server bandwidth). In today's server designs we find that NIC is usually the impediment. This is referred to as the *network-I/O bottleneck* [1]. As an example, the forthcoming PCI bus sends 64 bits of data at 100 MHz, thus moving more than 6 Gbps. A storage subsystem designed around this bus will be able to take full advantage of the 6-Gbps data rate. On the other hand, the performance of the NIC is constrained by the external networking environment. If an OC-12 NIC is used, the video delivery capability of this system will be limited to only 622 Mbps, a 10-times difference in performance. This impedance mismatch will continue to be a problem in the foreseeable future due to the inherent differences between networking and bus technologies - improving the performance of a "very short network" (i.e., system bus) that interconnects only a handful of expansion cards in a box is a lot easier than trying to improve the performance of a huge communication network designed for millions of boxes.

It first seems that the network-I/O bottleneck can be addressed by using a larger number of NICs for each server. Bus designs, however, must limit the number of expansion cards to maintain their good performance. As an example, the 16 IRQs (*interrupt request levels*) in today's Pentium-based servers constrain them to a very small number of expansion slots. To address this problem, new-generation high-end servers, such as nCUBE Media Cube and SGI Origin 2000, are designed with crossbar or hypercube interconnects to allow incremental growth in both I/O bandwidth and interconnection bandwidth. These modern servers can accommodate a large number of NICs. Their storage-I/O capacity, however, also increases accordingly. Taking SGI Origin 2000 as an example, it has a sustained storage-I/O bandwidth of 640 Gbps [2]. Taking full advantage of this bandwidth would require 533 OC-24 NICs. Even if this is feasible, concentrating that many NICs to one server will most likely cause severe congestions in the network. A more practical design would have to sacrifice some of the storage-I/O capability by employing a large number of servers which are geographically separated from each other and interconnected in some hierarchy. This approach, however, is expensive due to the increase in the hardware and networking costs. This is aggravated by the high cost of managing a distributed system. A good design should control the degree of distribution as much as possible to minimize the hardware and operating costs. This is achieved in this paper by reducing the demand on the service bandwidth.

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

†Also, nStor Corporation, Inc., Lake Mary, FL 32746.

ACM Multimedia'98, Bristol, UK
© 1998 ACM 1-58113-036-8/98/0008 $5.00

The most direct solution to the network-I/O bottleneck is to improve the performance of NICs. This requires advances in the networking technology. Several high-performance technologies are emerging. ATM has been shown to surpass its commercial limit of 622 Mbps, experimentally reaching 2.488 Gbps. By the year 2000, HIPPI (High Performance Parallel Interface) speed is expected to grow to 25.6 Gbps. With all this power, it seems that the bottlenecks will disappear. Unfortunately, bringing these networking technologies to the desktop is generally not feasible because of the high cost of the NICs and port costs in the switches. For instance, a HIPPI-6400 switch would cost about $20,000 per port. It is also unrealistic to assume that companies can afford to revamp their network infrastructure as frequently as they upgrade their computing systems. With these practical constraints, the network-I/O performance of most systems will still be limited by traffic congestions in the network.

A good solution to the network-I/O problem, therefore, must also include innovations in the software design. A well-known technique to reduce communication traffic is to allow the clients to share multicast data. Several such techniques have been proposed for VOD systems [3, 4]. These schemes delay requests and hope that more requests for the same video will arrive during the batching interval, and the entire group is served in one multicast. This approach significantly improves the system throughput. It, however, has to deal with the following dilemma:

- Users making the early requests are likely to renege if they are kept waiting too long.
- On the other hand, if we keep the waiting times short then the benefit of multicast diminishes.

To overcome these problems, the challenge is to achieve the following two goals:

1. The waits must be very short for all requests independent of their arriving order.
2. Each multicast must still be able to serve a large number of users.

In this paper, we present a novel multicast technique which achieves these two seemingly conflicting goals. Under the new scheme, 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 been played back to the skew point, the catch-up 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. Since clients can join an existing multicast, this approach significantly improves the efficiency of the multicast. Furthermore, since requests can receive the service immediately without delay, true VOD can be achieved. In this paper, a technique is said to be able to offer true VOD if it can service requests without a delay. A significant contribution of our work is making multicast work for true VOD services. In fact, we are able to eliminate the service latency and improve the efficiency of multicast at the same time. We will discuss this scheme in more detail in Section 3.

We note that once we have determined the portion of storage-I/O bandwidth that can be utilized for each server in the system, the video files must be carefully laid out to minimize the storage cost. Since videos are not accessed with the same frequency, how to replicate, stripe, and place the files over a minimum number of storage devices to support the access pattern is a nontrivial problem. This issue have been studied intensively; and some recent techniques are presented in [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. The interested reader is referred to those papers for many excellent ideas. In this paper, we focus on techniques to reduce the demand on the network-I/O bandwidth.

The remainder of this paper is organized as follows. We describe the conventional multicast techniques in more detail in Section 2 to make the paper self-contained. The proposed method is introduced in Section 3. In Section 4, we present our simulation study. Some related works are discussed in Section 5. Finally, we give our concluding remarks in Section 6.

2 Conventional Multicast Techniques

In conventional multicast techniques, service requests for the same video arriving within a short time duration are bunched together and served in a batch by a single data stream. Since requests are generally made for different videos, a server can have a number of pending batches at any one time. Different batching policies have been proposed. They are primarily different in the criterion used to select the next batch to receive service. We briefly discuss some of these techniques in the following.

- *First Come First Served* (FCFS) [16]:
  When sufficient bandwidth becomes free, this policy selects the batch with the oldest request (which has been waiting for the longest time) to serve next. The advantage of this scheme is its fairness - every video is treated equally regardless of its popularity. Obviously, the drawback of this strategy is the lower system throughput.
- *Maximum Queue Length First* (MQL) [16]:
  Unlike FCFS, this policy is designed to maximize the system throughput by selecting the batch with the largest number of pending requests to serve first. This strategy, however, is unfair since it favors more popular videos.
- *Maximum Factored Queue length first* (MFQ) [4]:
  This scheme improves on the FCFS and MQL policies by taking into account both the waiting times of the requests and the popularity of the videos. When sufficient bandwidth becomes available, MFQ selects the pending batch with the largest size weighted by the *best* factor, (the associated access frequency)$^{-\frac{1}{2}}$, to serve next. This scheme can achieve throughput close to that of MQL with little compromise of its fairness.

It will be clear shortly that our technique can be used to boost the performance of all batching schemes. We, however, used MFQ in our performance study since it has been shown to provide better performance than those of the other techniques [4].

The above multicast techniques are referred to as *Scheduled Multicast* because the sever selects the next batch to multicast according to some dynamic scheduling policy. Another approach is called *Periodic Broadcast* [16, 17, 18, 19]. 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 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, it can only be used for very popular videos. In this paper, we focus on the more general scheduled multicast approach.

3 Patching

In this section, we introduce a novel multicast technique called *Patching*. In this scheme, most of the communication bandwidth of the server is organized 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.

The video server maintains a waiting queue WQ; 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. Associated with each communication channel is a *client list* which contains the IDs of the clients currently viewing the broadcast on this channel. When a free channel becomes available, the server checks WQ for any entries, and admits a batch of service requests according to some scheduling policy such as those discussed in Section 2. The assignment of this batch to the free channel is done by specifying the desired video fragment, and inserting the clients in the batch into the corresponding client list. When the multicast is finally activated, the specified video is multicast on that channel to the clients in the list.

In conventional batching techniques, each channel must multicast the video in its entirety. As a result, the channel is held up for the entire duration of the video playback. This severely limits the number of batches can be served simultaneously. An important objective of the patching technique is to substantially improve the number of requests each channel can serve per time unit, thereby significantly reduce the per-customer system cost. This goal can be achieved by greatly reducing the time required to serve each batch. As an example, let us consider the following scenario. After channel C_i has multicast a video to batch B_i for three minutes, another channel, say C_j, becomes free and is used to serve a new batch B_j which also requested the same video. Let the length of the video be 60 minutes. If we use conventional batching, C_j will be busy for the next 60 minutes serving B_j. Alternatively, the clients in B_j can buffer the stream broadcast on C_i while playing the new start-up flow broadcast on channel C_j. After three minutes, when the catch-up flow has been played back to the skew point, C_j can be released and the original multicast on C_i can now be shared by both batches. This strategy is referred to as *Patching* in this paper. 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. We observe in this example that C_j is held up for only three minutes, compared to 60 minutes under batching. Therefore, patching can potentially improve the throughput of channel C_j up to 20 times. Besides improving the system throughput, since patching allows clients to start their playback immediately, true video-on-demand can be achieved. In other words, we can eliminate service latency without compromising the benefit of multicast.

In Patching, a client might have to download data on two channels simultaneously. In the above example, the clients in batch B_j must initially download data from both channels C_i and C_j. Although the patching data can be consumed as soon as they arrive, the shared data on channel C_i must be temporarily buffered to the local disk. As a result, the price for patching is the additional disk space required at each client station. This cost, however, should be minimal. As an example, a disk space of 100 Mbytes can cache about 10 minutes of MPEG-1 video. Such a disk space costs less than $10 today. The high cost of a VOD system is due mostly to the network costs. For instance, the cost of networking contributes to more than 90% of the hardware cost of the Time Warner's Full Service Network project in Orlando. Therefore, it is essential for a VOD design to take full advantage of the aggregate bandwidth of the network. If the clients are workstations, the small additional disk space is trivial. If set-top boxes are used to receive videos, the content provider can take up the cost of the additional disk space. The significant increase in the number of subscribers who can receive the services simultaneously easily makes up for this nominal cost. We note that client buffer is also used to implement VCR functions [20, 21, 22]. In this case, the buffer can be used to support patching at no additional cost.

3.1 Client Design

In the proposed technique, a communication channel is used to either multicast a video in its entirety called a *regular multicast*, or to multicast only the leading portion of a video called a *patching multicast*. 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 into 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*.

To implement patching, a client station needs to have three threads of control: two data loaders L_p and L_r, and a video player *VideoPlayer*. While L_p and L_r are responsible for downloading data from the patching channel and the regular channel, respectively, *VideoPlayer* is used to fetch the data from the local buffer, reassemble the video frames, and render them onto the screen.

To request a video, a client sends a request token $(ClientID, VideoID)$, where $ClientID$ is its own address and $VideoID$ is the ID of the requested video. When the server is ready for the service, it notifies the client with a service token, (PID, RID), where PID and RID are the IDs of the patching channel and the regular channel, respectively. The Client examines this token; and two scenarios can happen:

1. If PID is null, the server is about to start a regular multicast of the video on channel RID. In this case, the client needs to activate only loader L_r to receive

Algorithm: *Client Main Routine*

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

Algorithm: *Loader* L_p

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

Algorithm: *Loader* L_r

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

Algorithm: *VideoPlayer*

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

Figure 2: Algorithms for client stations

Algorithm: *Server Main Routine*

1. Dispatch a free channel, say *FreeChannel*.
2. Select the next video, say v, to serve according to a given scheduling policy (e.g., FCFS, MQL, MFQ, etc.)
3. Initialize the service token as (PID=*null*, RID=*null*).
4. If there is no regular multicast of video v in progress, set RID = *FreeChannel*. Otherwise,
   - Set PID=FreeChannel and RID=*LatestRegular*, where *LatestRegular* is the latest regular channel for video v.
   - Call either *GreedyPatching*(*FreeChannel*, *LatestRegular*) or *GracePatching*(*FreeChannel*, *LatestRegular*) to determine the portion of video data which should be multicast on channel *FreeChannel*.
5. For each request token (ClientID = *vClient*, VideoID = v) in *WQ*, do the following:
   - If PID is null, we append *vClient* to the client list of channel *FreeChannel*. Otherwise, it is appended to the client lists of both channels *FreeChannel* and *LatestRegular*.
   - Send the service token to notify the client *vClient*.
   - Delete the request token from *WQ*.
6. Activate the multicast on *FreeChannel*.

Figure 3: Algorithm for the video server

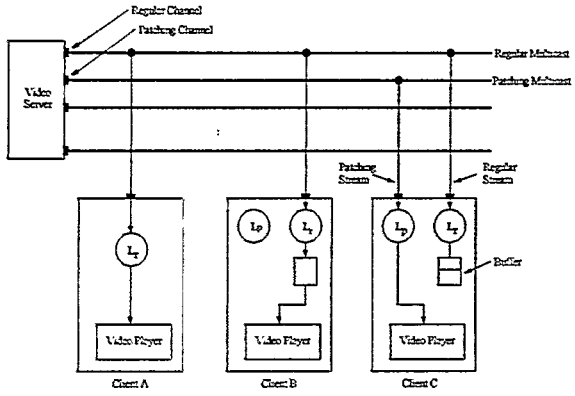


Figure 1: Patching Technique

the video data. As the data arrives at the client, the *VideoPlayer* renders the video frames onto the screen.

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

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 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.

We present the client routines in Figure 2. We note that the data from the patching channel (if any) and the regular channel are first buffered in *PatchBuffer* and *RegularBuffer*, respectively. However, the data downloaded to the *PatchBuffer* are immediately piped to *VideoPlayer*. The size of *PatchBuffer*, therefore, is negligible. We will simply refer to *RegularBuffer* as the client buffer in this paper.

3.2 Server Design

We now discuss the details of the server design. After a channel has completed its current batch, its client list is reset. The channel is now said to be *free*, and is available for the next multicast. The server main routine given in Figure 3 is repeated if the server has at least one free channel and its waiting queue WQ is not empty. When a free channel is dispatched for a service, it is given a *workload* specified as $v[t_p]$, where v and t_p denote the unique ID of the video file and the desired playback duration, respectively. For instance, *vid*[3] indicates that the free channel should multicast only the first 3 minutes of the video *vid*. We note in the server main routine that if there is no regular channel currently multicasting the video being scheduled, the workload for the free channel is the entire video (i.e., regular multicast). Otherwise, either *GreedyPatching* or *GracePatching* is called to determine the patching workload for the free channel according to the status of the latest regular channel. These functions decide how much of the video data should be delivered on the free channel, which is the actual cost of serving the current batch. Greedy Patching tries to have the current batch share the data with an existing regular multicast whenever possible. Grace Patching, on the other hand, will schedule a new regular multicast for the current batch if the client buffer is not large enough to cover the missing portion of the video. In this case, the free channel becomes the latest regular channel for this video. We discuss these two strategies in more detail in the following two subsections.

3.2.1 Greedy Patching

Greedy Patching schedules a new regular multicast for a batch only if there is no regular multicast currently serving the same video. The algorithm for this technique is given in Figure 4. It says that if a client arrives, say m, minutes late and does not have enough storage space to buffer the next m minutes of the last regular multicast of the same video, the buffer space is used to cache the last m minutes of that multicast.

Algorithm: *GreedyPatching(FreeID, RegularID)*

- t: current time
- t_s: start time of the regular multicast on channel *RegularID*
- V: the video currently multicast on channel *RegularID*
- $|V|$: playback duration of the video V
- B: size of the client buffer in playback time unit

1. If $t - t_s \leq B$, we set the workload for channel *FreeID* to $V[t - t_s]$.
2. Otherwise, the workload is set to $V[|V| - Min(B, |V| - (t - t_s))]$.

Figure 4: Greedy Patching

3.2.2 Grace Patching

Overly greedy, Greedy Patching can result in less data sharing. As an example, let us consider a regular multicast of a 60-minute video started 10 minutes ago. Assuming that each client has only enough buffer space for up to five minutes of video, the clients in the current batch have to buffer the last five minutes of the regular multicast resulting in only five minutes of data sharing. Furthermore, batches arriving within the next 50 minutes will benefit from the same regular multicast for no more than five minutes. As an alternative, if we schedule a new regular multicast for the current batch, then the batches arriving within the next five minutes for the same video will be able to share this new regular multicast for at least 55 minutes. This is the approach taken by Grace Patching presented in Figure 5. This scheme schedules a new regular multicast if the client buffer is not large enough for the patching clip. A potential drawback of this scheme is that it results in many more regular multicasts

Algorithm: *GracePatching(FreeID, RegularID)*

- t: current time
- t_s: start time of the regular multicast on channel *RegularID*
- V: the video currently multicast on channel *RegularID*
- $|V|$: playback duration of the video V
- B: size of the client buffer in playback time unit

1. If $t - t_s \leq B$, then the workload for channel $FreeID$ is $V[t - t_s]$.
2. Otherwise, *FreeID* is designated to start a new regular multicast as follows:
   - Modify the service token as (PID= *null*, RID = *FreeID*).
   - Set the workload for the new regular channel $FreeID$ as $V[|V|]$.

Figure 5: Grace Patching

compared to Greedy Patching. In the next section we will show simulation results to compare the performance of these two strategies.

4 Performance Study

In this section, we show simulation results to demonstrate the benefits of Patching. We chose MFQ as our experimental environment since it has been shown to perform better than FCFS and MQL [4]. Nevertheless, since Patching can be used to boost the performance of any batching schemes, the performance results presented herein can be generalized for all batching techniques. Our study includes both patching methods: *Greedy* and *Grace*. We use MFQ as a reference to assess the performance of these two patching strategies.

4.1 Simulation Environment

We first describe our simulation environment. Each client station is equipped with a disk buffer. The default size of this disk space is five minutes of video data. Each simulation run consists of 200,000 service requests. Each request is modeled by an interarrival time, a client ID, and a video choice. The interarrival time is assumed to follow a Poisson distribution. For each request, it is generated by a Poisson process which is exponentially distributed with a mean of $\frac{1}{\lambda}$, where λ is the request rate. The selection of the videos is modeled using a *Zipf*-like distribution [23]. That is, the probability of choosing the ith video is $\frac{1}{i^z \sum_{j=1}^{N} \frac{1}{j^z}}$, where N is the total number of videos in the system, and z is called the *skew factor*. A larger z corresponds to a more severe skew condition indicating that some videos are requested more frequently than the others. We set this value at 0.7 which is typical for VOD applications [3]. We assume that the system contains 100 videos, all of them are 90 minutes long. The server is capable of supporting 1,200 channels which is about the same as the bandwidth of the system used in the Time Warner trial in Orlando, which can deliver 1,000 MPEG-1 streams simultaneously. Our workload and system parameters are summarized in Table 1. The default values are listed under the "Default" column. We also vary some of these parameters to do sensitivity analysis. The range of values used for such studies are given in the third column under the heading "Range."

PARAMETER	DEFAULT	RANGE
Number of videos	100	N/A
Video length (minutes)	90	N/A
Server Bandwidth (streams)	1,200	400-1,800
Client buffer (min of data)	5	0-10
Request rate (requests/min)	50	10-90
Skew factor	0.7	N/A

Table 1: Parameters used for the simulation studies.

We choose *average latency*, *defection rate*, and *unfairness* as the performance metrics. We explain these terms in the following:

- *Defection Rate*: This is the percentage of service requests which are canceled because the waiting time exceeds the client's tolerance. We note that reducing the defection rate improves the system throughput.
- *Unfairness*: Let d_i denote the defection rate for video i and $\overline{d}$ be the mean defection rate. We define the unfairness as $\sqrt{\frac{\sum_{i=1}^{N}(d_i - \overline{d})^2}{N-1}}$, where N is the number of videos from which clients may make requests.
- *Average Latency*: It is defined as $\frac{\sum_{i=1}^{n} Latency_i}{n}$, where n is the total number of client requests; and $Latency_i$ is the service latency or the duration between the arrival time of request i and the time i is admitted for service.

We want to investigate the effect of request rate, server communication bandwidth, and client buffer size on the above metrics. In the following subsections, we report our simulation results under two different environments: one allows defection and the other one does not.

4.2 Study I: No Defection Allowed

In this study, we assume that users do not renege once they have submitted a service request. Defection rate and unfairness are, therefore, irrelevant in this case. We will discuss only the average latencies of the various schemes under different workload and system parameters.

4.2.1 Effect of Server Communication Bandwidth

In this study, the client buffer size was fixed at 5 minutes of data; and the average request rate was 50 requests per minute. The simulation results are shown in Figure 6. We observe that Greedy Patching offers little performance improvement over MFQ under this workload. This is due to the fact that the client buffer size is too small for Greedy Patching to exploit the data sharing feature. Most of the batches miss the last multicast of the same video by more than five minutes. As a result, majority of the data sharing are limited to the last five minutes of the video. To improve this condition, clients need to have more buffer space. We will investigate this option in the next subsection.

Unlike Greedy Patching, we observe that Grace Patching performs very well. By starting a new regular multicast whenever a batch arrives more than five minutes late, Grace Patching gives the subsequent batches a better chance to share essentially the entire multicast with the current batch. This possibility improves with the increases in the server communication bandwidth. With more bandwidth, the batches can be admitted with less waiting time increasing their chances of joining a previous multicast. This fact can be seen in Figure 6. It shows that the performance of Grace Patching improves with the increases in the server communication bandwidth. We note that it requires Grace Patching only 1,400 channels to provide true VOD services, i.e., service latency is zero. With this bandwidth, Greedy Patching and MFQ still suffer a rather long average service latency of more than 2 minutes.

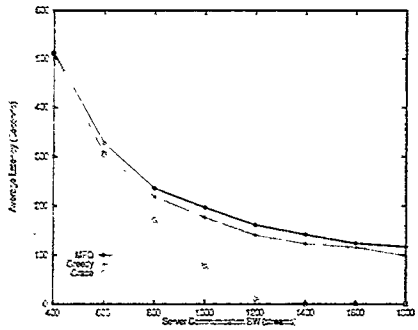


Figure 6: Effect of server bandwidth

We note that Grace Patching performs worse than Greedy Patching when the server communication bandwidth is very limited. This can be explained as follows. When the server has less than 600 channels, the average service latencies of all three schemes are more than five minutes. Since the client buffer can cache only five minutes of data, a large number of batches must be served by a regular multicast under Grace Patching rendering patching essentially useless. The situation is marginally better for Greedy Patching because a batch can still share the last five minutes with the last regular multicast. Winning under these situations, however, are uninteresting since one should not operate in this inadequate range.

4.2.2 Effect of Client Buffer Size

In this study, we want to see how the client buffer size affects the average latency. The server capacity was fixed at 1,200 channels and the arrival rate at 50 requests per minute. The simulation results are plotted in Figure 7. The curve for MFQ is flat as it does not take advantage of the client buffers. The latency curve for Grace Patching drops very rapidly as the client storage size increases. Under this workload, the plot indicates that Grace Patching requires only 6 minutes of client buffer space to achieve true VOD. We also observe that although Greedy Patching can benefit from more buffer space, the performance curve drops at a very slow pace. This indicates that Greedy Patching is not a very effective technique.

4.2.3 Effect of Request Rate

In this simulation study, we fixed the server capacity at 1,200 channels and the client buffer size at 5 minutes of video data.

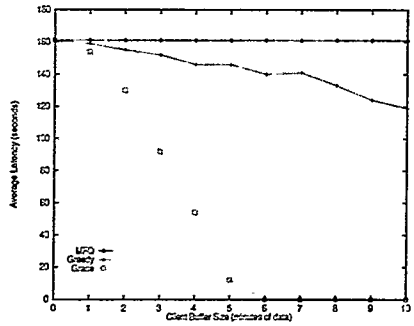


Figure 7: Effect of client buffer size

The effect of the request rate on average latency is plotted in Figure 8. Again, we see that Greedy Patching offers only nominal improvement due to insufficient client buffer space. Grace continues to perform very well. It outperforms the other schemes by very significant margins. If true VOD is required, the plot shows that MFQ and Greedy Patching must limit the request rate to about 10 requests per minutes. Grace Patching nonetheless enjoys a request rate of 40 requests per minutes, which is a 300% improvement. Since it requires substantially less bandwidth, Grace Patching offers an excellent technology for true VOD systems.

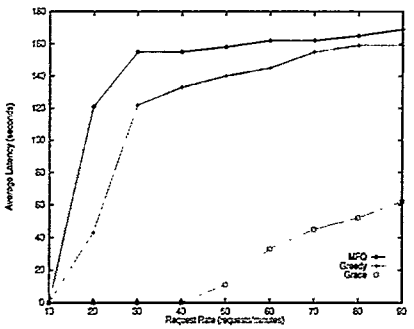
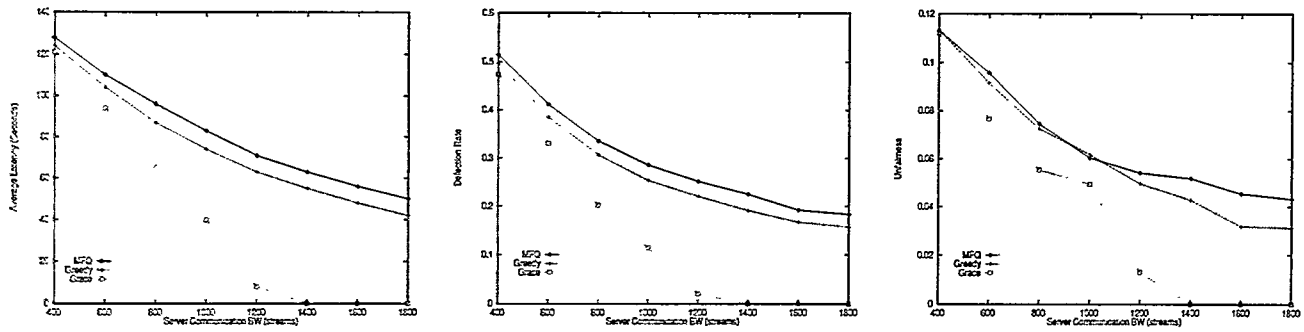


Figure 8: Effect of Request Rate

We note that Grace Batching offers no advantage when the request rate is very low (i.e., 10 requests per second). This can be explained as follows. Since the multicasts are highly efficient under Grace Patching, this scheme cannot utilize all the channels under such a low request rate. The same server bandwidth, however, is just enough for the other schemes to serve each new request as soon as it arrives. Under this circumstance, MFQ achieves true VOD by serving each client request using a dedicated channel. The system, however, totally gives up the benefit of multicast. This result demonstrates the inherent incompatibility between multicast and true video on demand. From this perspective, a significant contribution of our work is making multicast work for true VOD systems.

4.3 Study II: With Defection

In practice, if the wait is too long, the client is likely to cancel the service request. In this second study, we model the user defection behavior using a normal distribution with a mean of $\mu = 5$ minutes and a standard deviation of $\sigma = \frac{\mu}{3}$. We truncate the distribution on the left, which is negative. We present the simulation results in the following.



(a) Average Latency (b) Defection Rate (c) Unfairness

Figure 9: Effect of server communication bandwidth

4.3.1 Effect of Server Communication Bandwidth

In this study, we fixed the client buffer size at 5 minutes of data, and the request rate at 50 arrivals per minute. The simulation results are plotted in Figure 9. They show that none of the techniques can offer acceptable performance when the server bandwidth is inadequate. Grace Patching, however, is much better at taking advantage of the additional resources. As the server bandwidth increases from 400 to 1,800 channels, only the curves of Grace Patching drop rapidly to reach the zero level at 1,400 channels. With 1,400 channels, Grace Patching allows a perfectly fair system offering true VOD services with no defection. With the same hardware, MFQ and Greedy Patching can achieve only an average latency of about 65 seconds and 55 seconds, respectively. Their performance are actually much worse since the defection rates are very high. They are 20% for Greedy Patching and 23% for MFQ under this condition. Again, the simulation results confirms that Grace Patching uses the channels much more efficiently. It requires much less bandwidth to provide true-VOD services.

4.3.2 Effect of Client Buffer Size

In this study, the server capacity was fixed at 1,200 channels and the request rate at 50 arrivals per minute. The performance of the three multicast techniques under various client buffer sizes are plotted in Figure 10. We observe that the performance of Greedy Patching improves very slowly with the increases in the client buffer size. Grace Patching is much more effective in taking advantage of the additional buffer space; its curves drop rapidly. The plots indicate that it achieves true-VOD performance when the client buffer size is 6 minutes of data. Under this condition, the average latency of MFQ is more than 1 minutes with a defection rate higher than 20%. The bad defection rate makes MFQ and Greedy Patching not as fair as Grace Patching.

4.3.3 Effect of Request Rate

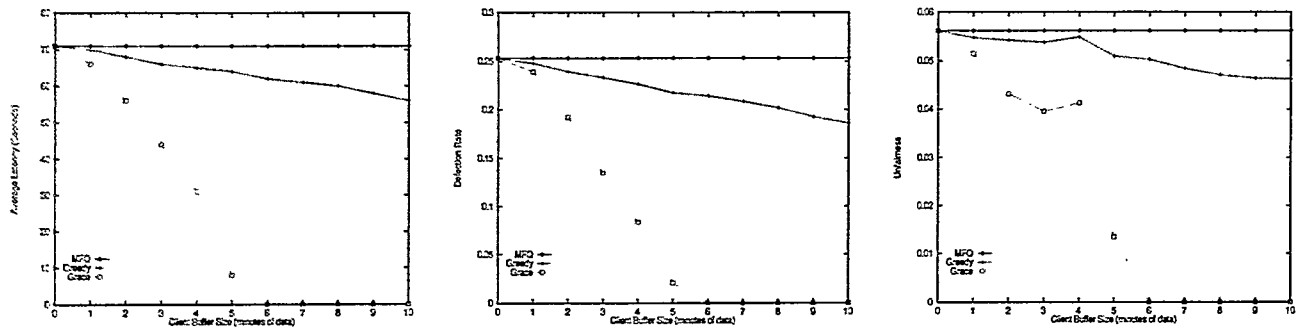
In this study, the server capacity was fixed at 1,200 channels and the client buffer size at 5 minutes of video data. We varied the request rate from 10 to 90 arrivals per minutes. The simulation results are shown in Figure 11. They are very similar to the non-defection case. Grace Patching can achieve true-VOD performance at a much higher request rate of 40 requests per minutes compared to only 10 requests per minute for MFQ. At 40 requests per minute, MFQ suffers a rather long average latency which is more than one minute. Its defection rate is also very high under this condition, almost 25%.

5 Related Works

We also exploited the idea of letting clients of the same multicast to receive the service at their own earliest possible time in [1]. The techniques were 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 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 [1] 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.

Compared with Dynamic Multicast, Patching offers a simpler form of dynamic multicast. We note that the multicast trees in Patching also grow dynamically to accommodate late requests. However, unlike Dynamic Multicast which uses client buffers to cache data for downstream clients, Patching uses client buffers to enable the clients to join an existing multicast. Since the data source is always the server, the dynamic multicast mechanism in Patching is much simpler.

Another technique which allows clients arriving at different times to share a data stream is called *Adaptive Piggy-*



(a) Average Latency (b) Defection Rate (c) Unfairness

Figure 10: Effect of client buffer size

backing [24]. An adaptive piggybacking procedure is defined to be a policy for altering display rates of services in progress (for the same object), for the purpose of "merging" their respective I/O streams into a single stream that can serve the entire group of merged services. Let us consider a client which is currently served by some communication 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 quality of service. 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. Under the same scenario, Patching would allow immediate data sharing without the merging delay as long as the client buffer can hold six minutes of the video. 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 [24].

Another related technique, called *Bridging*, is presented in [25]. 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.

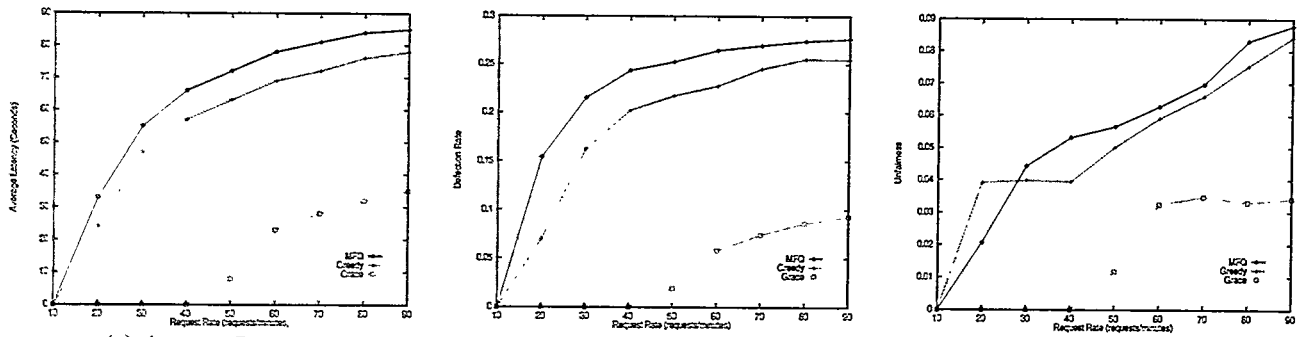
6 Concluding Remarks

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. In this paper, we considered a novel idea, called *Patching*, which extends the capability of standard multicast to support true VOD. The proposed technique has many advantages:

- Unlike conventional multicast, requests can be serviced immediately under patching. We are able to eliminate the service latency without compromising the benefit of multicast.
- In fact, patching can be seen as a better multicast technique since a multicast can now expand dynamically to service new client requests. Each multicast, therefore, can potentially serve many more clients making the multicast more efficient.
- Another desirable feature of patching is that channels are usually used only briefly to broadcast the first few minutes of the video, instead of being held up for the entire duration of the playback. This characteristic makes each channel more productive in the sense that it can service many more batches per time unit than it could under batching.
- Patching is very simple. It requires no specialized hardward. The only new requirement is to enable a client to join an existing multicast. Implementing this feature is trivial.

To evaluate the performance of Patching, we implemented a detailed simulator. The simulation results indicate that true VOD can indeed be achieved, with Patching outperforms conventional true VOD method by 300% under our workload. The performance results also show convincingly that Patching offers substantially better service latency and system throughput compared to conventional batching.

We are currently setting up our laboratory environment to build a video-on-demand prototype using Patching. Our system will have one dual-processor NT server and eight NT workstations. They are interconnected through an ATM switch. We will run a large number of logical clients on each workstation, and provide a mechanism to display any four of the current playbacks at a time. The video-on-demand system developed for our VideoCenter [26, 27] project will be used to provide the underlying functions. We will need to build on top the following components: a channel manager, a patching scheduler, and the multicast mechanism.



(a) Average Latency (b) Defection Rate (c) Unfairness

Figure 11: Effect of Request Rate

References

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

[2] Origin2000: The perfect system for evolving compute, memorey, and i/o requirements. Web page at http://www.sgi.com/origin/2000/desk.html.

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

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

[5] M. Chen, D. Kandlur, and P. Yu. Storage and retrieval methods to support fully interactive playout in a disk-array-based video server. *Multimedia Systems*, 3(3):126–135, July 1995.

[6] K. Keeton and R. H. Katz. Evaluating video layout strategies for a high-performance storage server. *Multimedia Systems*, 3:43–52, 1995.

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

[8] T. S. Chua, J. Li, B. C. Ooi, and K. L. Tan. Disk striping strategies for large video-on-demand servers. In *The 4th ACM International Multimedia Conference*, pages 297–306, Boston, MA, USA, November 1996.

[9] B. Ozden, R. Rastogi, and A. Silberschatz. On the design of a low-cost video-on-demand storage system. *Multimedia Systems*, 4(1):40–54, February 1996.

[10] J. Korst. Random duplicated assignment: An alternative to striping in video servers. In *Proc. of ACM Int'l Multimedia Conference*, pages 219–226, Seattle, U.S.A., November 1997.

[11] S.-W. Lau and J. Lui. Scheduling and data layout policies for a near-line multimedia storage architecture. *Multimedia Systems*, 5(5):310–323, September 1997.

[12] G.K. Ma, C.S. Wu, M.C. Liu, and B.S.P. Lin. Efficient real-time data retrieval through scalable multimedia storage. In *Proc. of ACM Int'l Multimedia Conference*, pages 165–172, Seattle, U.S.A., November 1997.

[13] Y. Wang, J.C.L. Liu, D.H.C. Du, and J. Hsieh. Efficient video file allocation schemes for video-on-demand services. *Multimedia Systems*, 5(5):283–296, 1997.

[14] Y. Wang and D. Du. Weighted striping in multimedia servers. In *Proc. of IEEE Int'l Conf. on Multimedia Comp. and Sys.*, pages 102–109, Ottawa, Canada, June 1997.

[15] R. Zimmerman and S. Ghandeharizadeh. Continuous display using heterogeneous disk subsystems. In *Proc. of ACM Multimedia Conf.*, pages 219–226, Seattle, U.S.A., November 1997.

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

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

[18] 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.

[19] 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.

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

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

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

[23] K. A. Hua, C. Lee, and C. M. Hua. Dynamic load balancing in multicomputer database systems using partition tuning. *IEEE Trns. on Knowledge and Data Engineering*, 7(6):968–983, December 1995.

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

[25] M. Kamath, K. Ramaritham, and D. Towsley. Continuous media sharing in multimedia database systems. In *Proceedings of the Fourth Int'l Conf. on Database Sysetms for Advanced Applications (DASFAA'95)*, Singapore, April 1995.

[26] W. Tavanapong, Kien A. Hua, and J. Wang. A framework for supporting previewing and VCR operations in a low bandwidth environment. In *ACM Conference on Multimedia Systems*, pages 303–312, Seattle, U.S.A., November 1997.

[27] Kien A. Hua, W. Tavanapong, and J. Wang. 2PSM: An efficient framwork for searching video information in a limited-bandwidth environment. *ACM Multimedia Systems*, to appear.