A Multi-Multicast Sharing Technique for Large-Scale Video Information Systems

Duc A. Tran Kien A. Hua Mounir Tantaoui
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32826, USA

***Abstract*—Despite the amelioration in communication technologies, the server network I/O bottleneck remains a severe obstacle in video-on-demand applications. All bandwidth-sharing techniques, proposed to overcome this, assume videos to be requested in their entirety. In this paper, we extend the video-on-demand delivery problem so that it allows users to request parts of video and be able to receive them at that instant. Specifically, we propose an efficient technique called *Multicast Jumping* (MJ) that provides many desirable properties such as: (1) The demand on the server bandwidth is significantly alleviated; (2) True and scaleable video on demand is achieved; (3) Not only for streaming video segments on demand, but the technique also works for on-demand videos in their entirety.**

I. Introduction

Today's multicast protocols [1] are not optimized for video applications. Particularly, they cannot deliver video on demand (VOD). This restriction has led to a large body of recent work looking for remedies at the application level. They are *Periodic Broadcasting* (PB) ([2], [3], [4], [5], [6], [7], [8]), *Batching* ([9], [10]), *Chaining* [11], *Piggybacking* [12], and *Patching* ([13], [14]). A network-level scheme was presented in [15], which caches data at routers in the network to service subsequent requests. In a sense, cached data act as a huge "virtual server" to all users, therefore lightening the demand on the server bandwidth.

Those schemes above are based on the assumption that videos are requested and transmitted in their *entirety*. This premise might not be true in practice where users usually prefer retrieving partial contents of video. As an example, a student might want to review only the last 15 minutes of a previously recorded lecture. Such situations happen very often in VOD applications, especially in video information systems [16], [17] where queries are most of the time for special content-based segments of interest and the results should be returned in an interactive manner. We call this new environment *generalized - Video on Demand* (g-VOD).

Supporting g-VOD however is not trivial. Indeed, let us consider two users who request the same video. They may share a single multicast as the result of the existing VOD solutions. What if the requests are the first and the last 50 minutes, respectively, of a one-hour long video? Since the two parts share 80% of their content, it would be beneficial to enable sharing the overlapped data. However, it is not obvious how the users can join the same multicast because the requests are different in this case. The problem becomes more complex when we have to deal with many users. Under such a circumstance, current methods would treat requested segments as distinct videos; therefore degrading to the very inefficient dedicated-stream approach.

To conquer this, we investigate a new bandwidth-sharing technique called *Multicast Jumping* (MJ) which allows user machines to cache data in their local storage. When a user requests a video segment S, there may be an existing multicast about to carry some part of S. That common portion can be shared by the user and the multicast, and the missing *leading* data (if any) is provided by the server on a new multicast stream. According to MJ, the user plays back it (i.e., the missing) while buffering the data downloaded from the multicast partner. Once that playback is done, the user switches to render the cached data. After this task is accomplished, the user still needs to obtain the missing *ending* data, if any. The situation then mimics and is processed in the same way as the first time the user requests segment S.

In short, MJ is aimed at sharing as much as possible with ongoing multicasts in order to deal with new users. This saves a lot of server bandwidth since the server does not have to launch streams, for a single use, many times. The multicast groups can grow with new requests coming at different times, therefore relieving the number of necessary multicasts. The system is therefore scalable. Furthermore, since any request is serviced immediately, the true on-demand property is obtained. MJ is a general-purpose VOD delivery scheme in the sense that, not only for streaming video segments on demand, but it also works for on-demand videos in their entirety.

The remainder of this paper is organized as follows. In Section II, we introduce Multicast Jumping in detail. We discuss the simulation studies in Section III. Finally, we give the concluding remarks in Section IV.

II. Multicast Jumping

In this section, we introduce a novel solution called *Multicast Jumping* (MJ) that allows users requesting overlapping segments of the same video to be able to join a single service stream in an efficient way.

We assume that a g-VOD system includes a machine acting as the server that stores video files. Any user machine must

This research is partially supported by the National Science Foundation grant ANI-0088026.

be equipped with some amount of memory on the local disk for caching purposes. Thus, the price for MJ is the additional disk space required at each user node. Users wanting to watch some segment of a video send a request to the server. At any point of time, there are always a number of on-going multicast groups in the system. The server contains all information about them. If a user asks for some service, based on the content being carried on those multicasts, the server will inform the user of a schedule consisting of which multicast addresses and what time the user can tune into in order to download the data needed. The cache is used for synchronizing the tasks. When some data cannot be found on those, the server will create a new multicast to deliver this missing data to the user. This use of multicast instead of unicast enables subsequent users to join and share data. The most important point of MJ is that we try to share as much as possible overlapped segments of the same video in order to relieve the demand on the server network I/O. For simplicity, we are based on the premise that the transmission time on the network is negligible, and the playback rate and the rate of video data coming out from the server are one video data unit over one time unit.

Let us denote by $[s_1, s_2]$ the segment containing frames s_1, $s_1 + 1$, .., s_2. $[s_1, s_2)$, $(s_1, s_2]$ and (s_1, s_2) have similar meanings except that they exclude frames s_2, s_1, and both of them, respectively.

A. A simplified view

Without loss of generality, we consider only one video V, of which users request various segments. In practice, the server bandwidth can be de-multiplexed to make any n virtual servers, each serving one of the n videos. To see the idea of MJ from a simple perspective, let us suppose that a new user U with cache size W starts requesting a segment $[a, b]$. At this moment, let us assume there exists an on-going multicast G that is about to carry another segment $[c, d]$. We consider the following possibilities: (1) $0 \leq c - a \leq W$, $c \leq b$ and $b \leq d$: In this case, the server allocates a new multicast G_{new} containing only user U and delivers the segment $[a, c)$. U will start playing back as soon as the data comes. At the same time, U joins group G and receives the segment $[c, b]$ from this existing multicast. The arriving data is stored in the cache. Once U consumes up $[a, c)$, the data in the cache still contains frame c because of the condition $c - a \leq W$. Therefore, U will switch to playing back data in the cached until the whole $[c, b]$ is received. G_{new} can be released since it is of no use anymore. After U playing backs frame b, U cancels its membership with G because its request is fully satisfied. (2) $0 \leq c - a \leq W$, $c \leq b$ and $b > d$: This case is very similar to the previous case. The only difference is that the segment $(d, b]$ cannot be downloaded from multicast G. This can be treated simply. The algorithm is as follows. The server allocates a new multicast G_{new} containing only user U and delivers the segment $[a, c)$. U will start playing back as soon as the data comes. At the same time, U joins group G and receives the segment [c, d] from this existing multicast. The arriving data is stored in the cache. Once U consumes up [a, c), the data in the cache still contains frame c because of the condition $c - a \leq W$. Therefore, U will switch to playing back data in the cached until the whole $[c, d]$ is received. G_{new} can be released since it is of no use anymore. After U playing backs frame d, it still needs to get $(d, b]$. In this situation, the server again creates another multicast including just U and transmits $(d, b]$ to this user. The new multicast will expire once frame b is already received by U.

Let us consider a more complicated scenario. When a new user requests a segment, there are multiple on-going multicast groups, each being about to receive a segment overlapping with the requested one. A practical issue is to decide which multicast the user should join in order to best utilize the in-network bandwidth and minimize the server involvement. We call that selected multicast the *share stream*. Another thought pops up in the second possibility presented above. After user U receives d from G, is it mandatory to require that the server create a new multicast, which we call the *live stream*, for the user to download $(d, b]$? The answer is "no" since by that time there may have been more multicasts that are created after U starts asking for service. U would be able to jump into one of them for the remaining segment, resulting in even zero physical streams from the server. The details are thoroughly described below.

B. User design

To implement MJ, a user machine needs to have three threads of control: two data loaders L_{live} and L_{share}, and a video player VP. L_{live} is responsible for downloading data from the live stream originating from the server. L_{share} is used to get data from the share stream whereas VP is invoked to fetch data from the local buffer, reassemble the video frames and render them on the screen.

When a user likes to watch a video segment, he sends a request to the server and waits for the reply. That request contains the information about the user identifier, the user cache size, a particular video, and the two ends of the segment (i.e., the starting and ending identifier, usually represented by frame number). The reply message is of the form ($LiveStreamID$, $ShareStreamID$, $ExpireTime_{live}$, $ExpireTime_{share}$, $FinishFlag$) where $LiveStreamID$, $ShareStreamID$ are multicast group addresses, and $ExpireTime_{live}$, $ExpireTime_{share}$ integers. When the user is notified, the reply is interpreted as follows. If $ShareStreamID = NULL$, the user needs activate only loader L_{live} in order to join and receives data from multicast $LiveStreamID$ for $ExpireTime_{live}$ time units. After that, the live stream is released. As the data arrives at the user, VP renders the video frames onto the screen. In the case that $ShareStreamID \neq NULL$, the user joins multicasts $LiveStreamID$ and $ShareStreamID$, and activates both loaders L_{live} and L_{share}. These loaders are used to download data from the multicast streams at the same time. Data obtained by L_{share} resides in the user cache. The user quits the

Algorithm: *User Main Routine*

1) Send a request ($UserID$, $CacheSize$, $VideoID$, $FirstFrame$, $LastFrame$) to the server.
2) Wait until receiving a reply ($LiveStreamID$, $ShareStreamID$, $ExpireTime_{live}$, $ExpireTime_{share}$, $AdditionalExpireTime$, $FinishFlag$) from the server.
3) If $ShareStreamID = NULL$, activate loader L_{live}.
4) Else activate loaders L_{live} and L_{share}.
5) Start the video player VP.
6) If $FinishFlag = OFF$, do the following
   - Wait until loader L_{share} stops.
   - $FirstFrame$ is assigned to $FirstFrame + ExpireTime_{live} + ExpireTime_{share}$.
   - Go back to step 1.
7) If $AdditionalExpireTime = NULL$, go back to step 1.
8) Assign $ExpireTime_{live}$ to $AdditionalExpireTime$
9) Activate loader L_{live} and video player VP.

Fig. 1. The main routine at the user node

multicasts after $ExpireTime_{live}$ and $ExpireTime_{share}$ time units have expired, respectively. Once it quits the live stream, its VP continues playing back data from the cache. When it quits the share stream and $FinishFlag$ is off, the user is done with his/her request. Otherwise, he/she sends another request to the server for the yet-missing data. This task mimics the first time the user asks for service. The process is repeated until the user receives a reply whose field is turned on or until $ShareStreamID$ is null.

We present the user routines in Figure 1 (main procedure), Figure 2 (loader algorithms) and Figure 3 (video player). Note that the data from the live stream and the share stream (if any) are first buffered in $RegularBuffer$ and $CacheBuffer$, respectively. However, the data saved into $RegularBuffer$ is pipelined to VP immediately. Consequently, the size of this buffer is negligible. We will simply refer to $CacheBuffer$ as the user cache in this paper.

- Algorithm: *Loader* L_{live}
  1) Join $LiveStreamID$
  2) Download one data unit on stream $LiveStreamID$
  3) Store it to $RegularBuffer$
  4) Decrease $ExpireTime_{live}$ by 1
  5) If $ExpireTime_{live} = 0$, quit $LiveStreamID$ and exit.
  6) Else Go back to step 2.
- Algorithm: *Loader* L_{share}
  1) Join $ShareStreamID$
  2) Download one data unit on stream $ShareStreamID$
  3) Store it to $CacheBuffer$
  4) Decrease $ExpireTime_{share}$ by 1
  5) If $ExpireTime_{share} = 0$, quit $ShareStreamID$ and exit.
  6) Else Go back to step 2.

Fig. 2. Loaders at the user node

C. Server design

We now discuss the details of the server design. The server maintains a service table to monitor the information of ongoing multicasts. Each row represents a group and has five attributes. Attribute $MulticastID$ is the multicast address of the group. Attribute $MemberList$ is the identifier (address) list of all members who are listening to the multicast. Attribute $VideoID$ is the identifier of the video being multicast. Attribute $NextFrame$ is the index of the frame that is about to delivered next. Finally, attribute $LastFrame$ is the index of the frame that will be delivered at last. This table is updated as time goes by. After every one time unit, all $NextFrame$ values are reduced by one. A row is removed if its $NextFrame$ value exceeds $LastFrame$ value.

Algorithm: *Video player* VP

1) Fetch one playback unit from $RegularBuffer$
2) Free the disk space for the fetched data
3) Reassemble the fetched data into frames and render them onto the screen
4) If $RegularBuffer$ is not empty, go back to step 1.
5) Fetch one playback unit from $CacheBuffer$
6) Free the disk space for the fetched data
7) Reassemble the fetched data into frames and render them onto the screen
8) If $CacheBuffer$ is not empty, go back step 5.
9) Exit.

Fig. 3. User video player

Having received a request for a segment [a, b] of video V from a user whose cache size is W, the server accesses the service table to look for a multicast group being about to download the segment [c, d] of V such that $0 \leq c - a \leq W$, $c \leq b$, and $c + d - a - b$ is minimized. In other words, the user is scheduled to join an on-going multicast whose content is most shared with the requested segment provided that the leading missing portion is smaller than the user cache size. If such a multicast is not found, the server simply creates a new multicast including the user and delivers the requested data to the user on that stream. Otherwise, the server also creates a new multicast, but only transmits [a, c). The server then asks the user to join the found multicast (i.e., share stream in our terminology) to download the overlapped content [c, min(b, d)] and informs the user of the time he/she must quit the multicast.

If $d < b$, the user still has to get (d, b] somehow. A straightforward way is to create a new multicast from the server to provide this piece of data. However, we thought it is not necessary to do that. Indeed, since time has moved up, the service table may have been updated with more new multicasts. It is possible that the missing segment can be obtained from one of these groups. If this happens, the server treats (d, b] as a new request just arriving from the same user. The server processes this request in the same manner as it does with [a, b], which has been described above. Details of the server routine is shown in Figure 4.

We can think of the user as jumping from one multicast group to another several times until he/she gets the final frame. MJ utilizes the existing multicasts as much as possible in order to avoid the server having to allocate many new multicasts. As a result, the server bandwidth is less demanded in this particular instance. A problem with MJ may be that multicasts are up-

Algorithm: *Server Main Routine*

1) Wait for a new request
2) Receive a request ($UserID, CacheSize, VideoID, FirstFrame, LastFrame$).
3) Search the service table for a row ($ShareStreamID, MemberList, VideoID, Next, Last$) such that $0 \leq Next - FirstFrame \leq CacheSize$, $Next \leq LastFrame$ and $Next + Last - FirstFrame - LastFrame$ is minimized.
4) If such a row is not found, do the following:
   - Create a new multicast $NewStreamID$ containing $UserID$
   - Insert a new row ($NewStreamID, \{UserID\}, VideoID, FirstFrame, LastFrame$)
   - Send a reply ($NewStreamID, NULL, LastFrame - FirstFrame + 1, NULL, NULL, ON$)
   - Go back to step 1.
5) Change the row to ($ShareStreamID, MemberList \cup \{UserID\}, VideoID, First, Last$)
6) Create a new multicast $NewStreamID$ containing $UserID$
7) Insert a new row ($NewStreamID, \{UserID\}, VideoID, FirstFrame, Next - 1$)
8) If $LastFrame \leq Last$, do the following:
   - Send a reply ($NewStreamID, ShareStreamID, Next - FirstFrame, LastFrame - First + 1, ON$)
   - Go back to step 1.
9) Else Send a reply ($NewStreamID, ShareStreamID, Next - FirstFrame, Last - First + 1, NULL, OFF$).
10) Go back to 1.

Fig. 4. Algorithm for the video server

dated quite often due to the jumps. However, we can surmount this by limiting the number of jumps to be below a threshold ζ, or the size of remaining segment to be above a threshold η. In other words, if the number of jumps already equals ζ or the remaining data size is less than η, the remaining data must be downloaded from the server.

III. Performance Evaluation

To assess the potential of Multicast Jumping (MJ), we carried out a simulation. Since no other work excluding the dedicated stream approach (DS) has been done to address the issues of g-VOD, we compared MJ with DS in this section. For simplicity and without loss of generality, we consider only one video of which users request various segments. To best explore the performance merits of MJ, we are interested in the four measures listed as follows. (1) Server bandwidth requirement (SBR): The total bandwidth the server has to reserve in order to satisfy all requests. (2) Server load used per request served (SLRS): The average time the server spends serving a successful request. (3) Success rate (SR): The percentage of requests that are successfully served out of all requests. (4) Data over time (DOT): The average data amount transmitted per time unit, computed as the ratio between the total amount of data requested and successfully served, and the simulation running time. The SBR and SLRS demonstrate the efficiency in dealing with the server bottleneck, while the others reflect the system throughput. For requested segments are of different sizes, the DOT is more suitable to be studied than the average number of requests in a time unit. In our experiment, we assume that requests are served immediately if services are available, or they will renege. Hence SR can also be considered in a metric sense to judge the level of achieving true video-on-demand services.

Our workload consists of 100,000 service requests arriving according to a Poisson process with the default request rate λ = 1.0. Each segment is generated randomly, that is, the first frame and the segment length are randomly chosen, provided that they do not conflict with the video length. It implies that as the video length increases, so does the average segment length. The system server, which may be a collection of media servers, is capable of supporting by default 1000 concurrent streams imitating the bandwidth of the system used in the Time Warner trial in Orlando, which can deliver 1000 MPEG-1 streams simultaneously. Each user can cache up to two and a half minutes of video, which is reasonably small. We use one second of video data as the unit for caching. The simulation time unit is also a second. We investigated the impact of video length (varying between 30 and 150 minutes), request rate (varying between 0.2 reqs/sec and 1.8 reqs/sec), and server bandwidth (varying between 500 concurrent streams and 1500 concurrent streams) on the four measures described above. In what follows, we report the detailed numerical results and discuss their cause and effect.

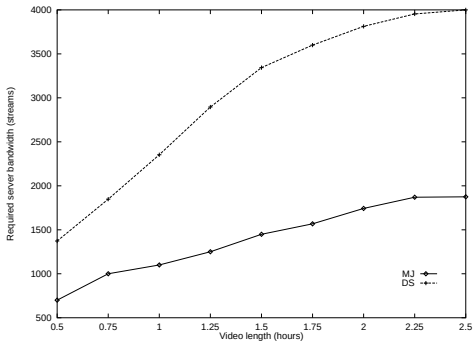

Fig. 5. Effect of video length on the bandwidth requirement

A. Bandwidth requirement

In this section we evaluate the minimum bandwidth the server has to reserve in order to serve all requests. For this purpose, we consider the impacts of video length and request rate. The simulation results are presented below.

1) Effect of video length: Figure 5 illustrates a vast difference between MJ and DS. In order to guarantee service for all requests, the bandwidths of the approaches are more required

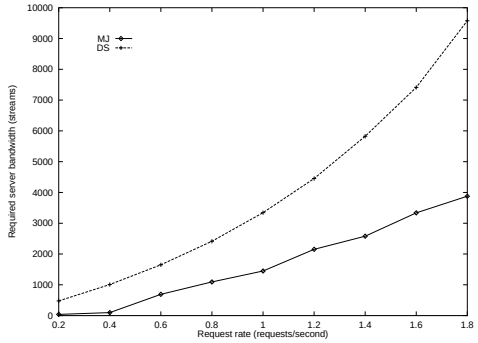

Fig. 6. Effect of request rate on the bandwidth requirement

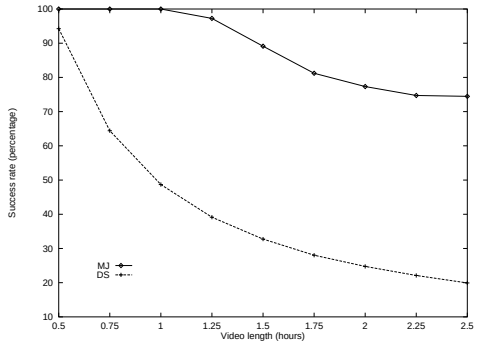

Fig. 7. Effect of video length on the success rate

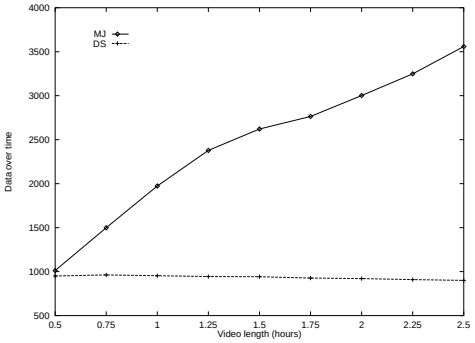

Fig. 8. Effect of video length on DOT

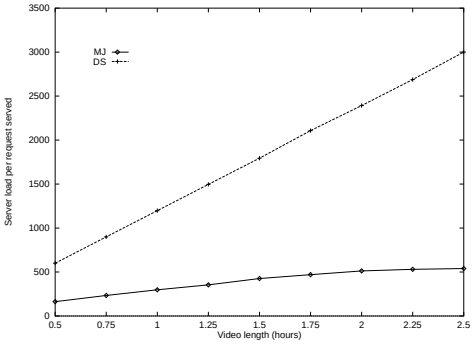

Fig. 9. Effect of video length on SLRS

as videos are longer. During this increase, however, the acceleration of MJ-curve is less than that of DS. At any point, MJ is recognizably more advanced than the other by at least two times. As an example, when videos are 1.5 hours long (usually the length of popular movies), MJ only needs a server bandwidth 43% of that required by the conventional dedicated stream paradigm.

2) Effect of request rate: The effect of request rate on the two schemes is shown in Figure 6. As the rate increases, they require more server bandwidth in order to provide services to all requests. We can however see an outstanding improvement of MJ over DS here, regardless of how small or large the rate is. In particular, the server bandwidth required by the former is always about 300% less than that required by the latter. This gap would have been larger had we continued to increase the request rate. This improvement is due to MJ's ability to lessen bandwidth demands on the server by allowing new users to get partial data from multiple existing multicast groups. Consequently, the active periods of server streams are considerably short. In contrary, DS must occupy a long-live dedicated stream from the server whenever a request arrives. Therefore, the quicker requests arrive, the more bandwidth the server has to use.

B. Impacts on SR, DOT and SLRS measures

1) Effect of video length: We investigate the impact of video length on the success rate, DOT and SLRS in this section. The results plotted in Figure 7, Figure 8 and Figure 9 are derived from simulation runs with the video length varied from 0.5 to 2.5 hours. Other parameters are assigned with default values. Figure 7 shows that both techniques resulted in lower success rates, however the performance gap between them becomes more significant as the video length increases. In the worst case, a 75% of requests were successfully served by MJ in comparison with only 20% of that served by the other. In all other cases, MJ is superior to DS by an at least 160% better success rate.

In terms of DOT and SLRS, the advantage of MJ still holds. The video length changes do not affect the DOT value very much under DS (Figure 8). In contrast, MJ has higher DOT values as the effect of increasing the video length. The difference between them becomes recognizable when a full video is 0.75 hours long. The enhancement of MJ gets more substantial with longer videos. For instance, with 2.5 hour long videos, the average amount of data provided by the server to users in a time unit is 350% larger than that provided by the DS approach.

Figure 9 exhibits the two techniques in the context of average load used by the server for each successful request. Despite the increase of video length, MJ needs very little server load for each request where DS has to use precious server bandwidth for a longer time. As shown in the figure, 600% of SLRS is improved by MJ compared to the latter when videos are long, and 300% improved when videos are short.

2) Effect of request rate: The effect of request rate on the performance metrics is illustrated in Figure 10, Figure 11 and Figure 12 where the rate varies from 0.2 to 1.8 requests per sec-

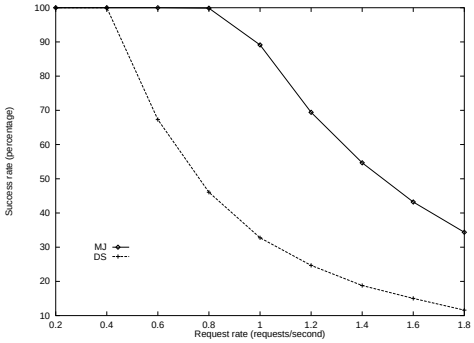

Fig. 10. Effect of request rate on the success rate

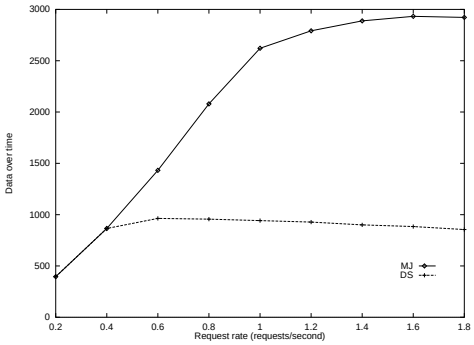

Fig. 11. Effect of request rate on DOT

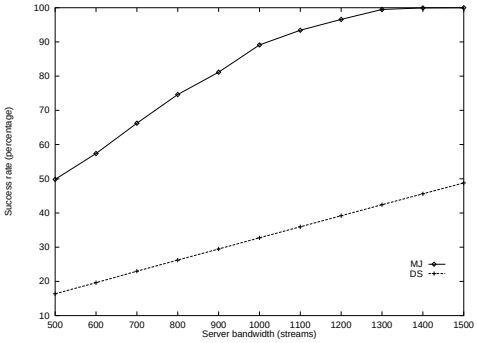

Fig. 13. Effect of server bandwidth on the success rate

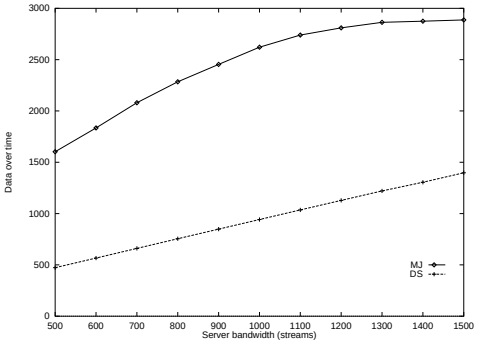

Fig. 14. Effect of server bandwidth on DOT

ond. In terms of success rate, both techniques work very well when requests arrive in a sparse manner (less than 0.4 requests per second). In contrast, as requests arrive more frequently, the performance of DS degrades dramatically. For example, when the request rate reaches 0.8 requests per second where MJ is still able to provide service to all requests, only 45% of requests are served by DS. On the other hand, DS almost has no capability of serving users when the request rate becomes very high (e.g., 1.8 requests per second). In this case, MJ still offers service to more than 30% of requests made. As stated above, since DS can satisfy most requests like MJ if the arrival rate is very small, they behave similar in terms of the DOT measure in this case. However, the larger the rate becomes, the better throughput MJ results in. As we can see in Figure 11, the DOT values of DS stop being improved as the rate exceeds 0.6 requests/second. On the contrary, the mean amount of data served by MJ increases quickly. We have noted that a higher request rate reduces the number of requests served as shown in Figure 10 but, the simulation times in various runs of this study are especially shorter as the rate gets higher. The DOT values thus are not necessarily decreased. Actually, the DOT curve of MJ has a positive slope. This exhibits a merit of MJ over DS in dealing with many requests arriving in a short period. The numerical results show that DJ offers a system throughput (in terms of DOT) about 270% better than the other when the rate is moderate, and as much as 300% when it increases.

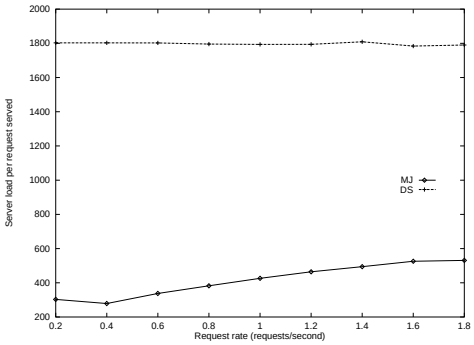

Fig. 12. Effect of request rate on SLRS

In the context of mean server load used to support a request, we experienced a sharp distinction between the two techniques (Figure 12). In the best case of the study (with a 0.4 requests/second rate), the MJ server needs use only 1/6 of the load required by DS for a successful request. In the worst scenario where the request rate reaches the highest value (1.8 requests/second), DS still defeats the other by a 300% improvement. We note the MJ curve goes down in the beginning and moves up due to the following. When the rate is smaller than 0.4 requests/second, the server can serve all the requests. As the rate increases before reaching that threshold, requests possess a denser behavior. Hence, multicasts are shared by more requests causing less bandwidth load to be used at the server.

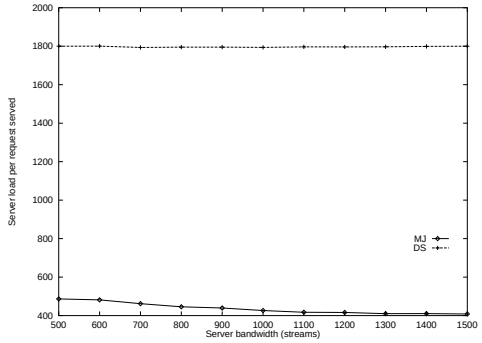

Fig. 15. Effect of server bandwidth on SLRS

3) Effect of server bandwidth: As we mentioned earlier, DS requires vastly more bandwidth of the server than MJ in order to provide service to all requests. As a complement, we investigated how the server bandwidth limitation affects the performance of the two techniques. In this study, we varied the server bandwidth from 500 to 1500 concurrent streams and calculated the corresponding SR, DOT and DPR values. The results are plotted in Figure 13, Figure 14 and Figure 15 where in any simulation run, MJ outperforms DS by a substantial margin.

In terms of success rate and DOT, MJ improves with a steeper level than the other technique as more server bandwidth is added. The most significant difference between them happens when the server has very limited bandwidth. For instance (Figure 13), when the server can only support 500 streams simultaneously, the number of requests that are denied due to the DS server bottleneck is at least three times greater than that of MJ. This shows the efficiency of MJ even in the case of the server having little bandwidth. Additionally, MJ satisfies all requests when the server bandwidth is 1500 streams whereas the other can provide service to only half of all requests. The phenomenon is similar in terms of DOT (Figure 14). Not only are more requests satisfied by MJ than DS, but also the average amount of data served by the former in a time unit is considerably higher than the latter.

From the server perspective, Figure 15 illustrates a tremendous advance of MJ in utilizing the server bandwidth. With more bandwidth added, the average load used per request changes very little since DS allocates a full stream for each requested segment. On the other hand, greater server bandwidth increases the number of successfully served requests making multicast groups to be shared more frequently. As a result, the server uses server bandwidth for a much shorter time to treat requests. As we can see in the figure, the SLRS of MJ is about 400% less than that of DS, on the average.

IV. Conclusions

We have concentrated on g-VOD, a realistic environment, which allows users to request portions of a video. Specifically, we have presented a technique called *Multicast Jumping* (MJ) in which the server responds to a request by sending to the user a *service schedule*. This schedule informs the user when and from which multicast addresses to download the requested data. When the schedule expires and the user still needs to play a missing portion, he/she invokes the server for that particular data. The server again ends up with another schedule for the user. This process is repeated until the user receives the entire requested segment. MJ is a multi-multicast sharing technique in the sense that users can share more than one multicast at proper times without asking for plenty of bandwidth at the server. Thus, MJ saves network bandwidth at the server side. Additionally, it has advantages such as scalability to support large-scale applications and ability to provide true VOD services.

References

[1] S. Deering, "Host extension for ip multicasting," *RFC-1112*, August 1989.

[2] 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.

[3] 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.

[4] J. F. Paris, , D. D. E. Long, and P. E. Mantey, "Zero-delay broadcasting protocols for video on demand," in *Proc. ACM MULTIMEDIA'99*, Orlando, USA, October-November 1999, pp. 189–198.

[5] D. Saparilla, K. W. Ross, and M. Reisslein, "Periodic broadcasting with vbr-encoded video," in *Proc. of IEEE INFOCOM '99*, 1999.

[6] S. Viswanathan and T. Imielinski, "Metropolitan area video-on-demand service using pyramid broadcasting," *ACM Multimedia systems Journal*, vol. 4, no. 4, pp. 179–208, August 1996.

[7] Y. C. Tseng, C. M. Hsieh, M. H. Yang, W. H. Liao, and J. P. Sheu, "Data broadcasting and seamless channel transition for highly-demanded videos," in *Proc. of IEEE INFOCOM '00*, 2000.

[8] S. Sen, L. Gao, and D. Towsley, "Frame-based periodic broadcast and fundamental resource tradeoffs," in *IEEE Performance, Computing and Communications Conference*, April 2001.

[9] 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.

[10] A. Dan and P. Shahabuddin, "Scheduling policies for an on-demand video server with batching," in *ACM MULTIMEDIA'98*, San Francisco, October 1998.

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

[12] L. Golubchik, J. Lui, and R. Muntz, "Adaptive piggybacking: a noval technique for data sharing in video-on-demand storage servers," *ACM Multimedia Systems*, vol. 4, no. 3, pp. 140–155, 1996.

[13] Kien A. Hua, Ying Cai, and Simon Sheu, "Patching: A multicast technique for true video-on-demand services," in *Proc. of ACM MULTIMEDIA*, Bristol, U.K., September 1998, pp. 191–200.

[14] S. Sen, L. Gao, J. Rexford, and D. Towsley, "Optimal patching schemes for efficient multimedia streaming," in *Proc. of IEEE NOSSDAV*, NJ, USA, June 1999.

[15] K. A. Hua, D. A. Tran, and R. Villafane, "Caching multicast protocol for on-demand video delivery," in *Proc. of the ACM/SPIE Conference on Multimedia Computing and Networking*, San Jose, USA, January 2000, pp. 2–13.

[16] D. A. Tran, K. A. Hua, and K. Vu, "Semantic reasoning based video database systems," in *Proc. of 11th International Conference on Databases and Expert Systems Applications*, London, U.K, September 2000.

[17] D. A. Tran, K. A. Hua, and K. Vu, "Videograph: A graphical object-based model for representing and querying video data," in *Proc. of 7th International Conference on Conceptual Modeling (ER2000)*, Salt Lake city, USA, October 2000.