ELSEVIER

Computer Communications 25 (2002) 413–423

computer communications

www.elsevier.com/locate/comcom

Joint server scheduling and proxy caching for video delivery

Olivier Verscheure*, Chitra Venkatramani, Pascal Frossard, Lisa Amini

IBM T.J. Watson Research Center, New York, USA

Abstract

We consider the delivery of video assets over a best-effort network, possibly through a caching proxy located close to the clients generating the requests. We are interested in the joint server scheduling and prefix/partial caching strategy that minimizes the aggregate transmission rate over the backbone network (i.e. average output server rate) under a cache of given capacity. We present multiple schemes to address various service levels and client resources by enabling bandwidth and cache space tradeoffs. We also propose an optimization algorithm selecting the working set of asset prefixes. We detail algorithms for practical implementation of our schemes. Simulation results show that our scheme dramatically outperforms the full caching technique. © 2002 Published by Elsevier Science B.V.

Keywords: Content distribution networks; Streaming media; Server scheduling; Partial caching; Batch patching; SLA

1. Introduction

Streaming media represents a unique opportunity for Service Providers — unlike other web objects which are enhanced by edge delivery, quality video actually requires edge of network services to attain reasonable user experience. As access providers roll out faster last-mile connections, upstream congestion in the provider's backbone, peering links and best-effort Internet will limit their ability to meet customer expectations for these premium links. While streaming media brings additional complexities (very large objects, isochronous delivery and interactivity), there are clearly many advantages of edge delivery. Attributes making it especially well-suited for edge delivery include its static nature, high value to Content Providers, distribution and delivery revenue potential to Content Delivery Service Providers and the potential for content services (transcoding, ad insertion, digital rights management) best offered through decentralized techniques.

Techniques to address the lack of end-to-end bandwidth to support streaming media include (i) multicasting to groups of clients and (ii) caching at streaming proxies located closer in the network to the end user.

Multicast scheduling strategies, such as *Periodic Broascasting* and *Batching*, have been proposed to simulate on-demand access. Although multicast significantly reduces network bandwidth, it is often considered impractical due to its reliance on a fully multicast-enabled network. Additional drawbacks of multicast scheduling strategies include client requirements for receiving multiple streams, large client buffers and lack of flexibility in providing user-level quality of service (QoS). Because video distribution and delivery incurs high storage and transmission costs, and requires specialized servers at the edge, Service Providers will target valued content, for which QoS guarantees are a must and best-effort service is unacceptable.

Caching audio/video objects in streaming proxies at the network edge is another attractive solution. Besides providing improved performance to the end-user, caches save on network bandwidth between the access provider network and the origin server. Caching strategies for video objects range from caching of full video objects to caching partial video objects by segmenting the video in the temporal and/or spatial domain(s). There are at least two issues with the caching of whole videos. First, the time and bandwidth required to bring an entire video into the cache associates a very high penalty with erroneous caching decisions. Second, ongoing streams may prevent deletion at cache replacement time causing the cache to be less reactive and to drift away from the optimal operating point.

Therefore, our objective is to create a content distribution system for streaming media, as opposed to a best-effort video caching system. We achieve this by placing a streaming proxy in the path between the server and the clients. We develop a scheme, which combines stream scheduling at the origin server and caching at the proxy to minimize the aggregate transmission rate over the network while maintaining configured user-level QoS constraints. The QoS constraints are expressed in terms of maximum playback

* Corresponding author.
E-mail address: ov1@us.ibm.com (O. Verscheure).

0140-3664/02/$ - see front matter © 2002 Published by Elsevier Science B.V.
PII: S0140-3664(01)00413-3

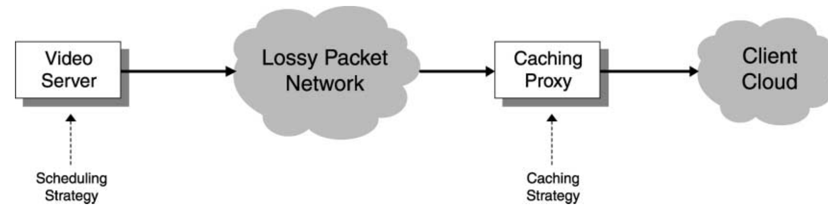

Fig. 1. Illustration of our joint scheduling and proxy caching strategy.

delay and application-level packet loss ratio (PLR). Object prefixes reflecting either popularity or contracted service levels are positioned at proxies to reduce startup latencies and enable on-demand access.

The paper is organized as follows. In Section 2, we present relevant research in the multicast and streaming media caching areas. In Section 3, we present our scheme for video stream delivery along with simulation results highlighting the bandwidth savings and cache space usage trade-offs under different scenarios. Section 4 describes a practical algorithm that is being implemented in our prototype and discusses practical issues. Finally, we present our conclusions and future work in Section 5.

2. Related work

Streaming video over multicast consumes less network bandwidth and imposes less of a load on the sender than does streaming video over multiple unicast channels. Among schemes that attempt to capitalize on the benefits of multicast for VOD are those based on the Periodic Broadcast idea [1–4] The video is divided into many portions which are continuously broadcast over multiple channels and are as such bandwidth efficient only when the request arrival rate is high. Another technique is the simple batching scheme where the server accumulates requests over a batching interval and starts a new multicast stream at the end of each interval if there were any requests in the batch. A more bandwidth efficient scheme is that of Patching [5,6]. In this scheme, each batch is served over one or two channels — either a regular channel (RC) alone or the combination of a RC and a patching channel. A RC delivers the full video from start to finish while a patching channel delivers only the missing part of the video from the start until the point at which the clients join the RC. The client receives both the patch and the ongoing stream and buffers the latter while playing back the former. Once the patch is exhausted, the client switches to the buffered regular multicast (RM). Hua et al. [5] compared the performance of patching with simple batching and found that patching was able to support true VOD at much higher request rates for a typical server configuration. Further research in this area can be found in Ref. [7], where the authors present the Optimized patching scheme which defines a Patching Window beyond which it is more efficient to start a new RM rather than generate patches. Finally, Ref. [8] extends the above technique to allow client-controlled latency/cost trade-off to provide classes of service by varying the batching interval (Fig. 1).

The other relevant body of research is that of video caching. Techniques range from caching whole videos (applying conventional memory caching techniques with modifications to account for the size) to partial objects segmented in the temporal and/or spatial domain(s). Segmentation in the temporal domain is achieved by splitting the video into constant time length (CTL) segments and segmentation in the spatial domain is achieved by encoding videos at multiple resolutions.

Full-caching strategies for video in a cluster of caches is considered in Ref. [9]. Key conclusions are that in streaming proxies, replication or striping of objects based on explicit tracking of request frequencies achieves higher hit rates rather than doing LFU or LRU on a per-request basis. They found that the cache replacement policy (LRU or LFU) did not make any difference because most videos had ongoing streams and could not be chosen for replacement. For this reason and the fact that bringing a large video file to a cache is very expensive, partial caching (including prefix caching) were proposed.

In Ref. [10], the authors present a caching scheme for adaptive, layered video (segmented in the temporal and spatial domains) such that the *quality* of the cached stream is proportional to its popularity. They also combine it with a fine-grained cache replacement strategy that tracks statistics per layer of video and eliminates the least popular segments of the video. The scheme has been designed with the goal of being adaptive to the network but not with the explicit goal to minimize the bandwidth streamed out of the origin server. Secondly, although this method results in caching the most popular parts of a video, the quality of video playback can be variable among different viewers of the same video, which might be undesirable. Finally, the adaptive scheme works well with layered encoding of videos, which is not employed in most popular formats. Another work that considers partial caching is *MiddleMan* [11]. This scheme works over a cluster of proxies on a LAN and the combined space is managed by a central *MiddleMan* who does the cache replacement decisions. The caches store only as much of the object as is played back by the client. Other video caching schemes include Resource-based Caching *RBC* [12] which focuses on the management of resources in the cache. RBC determines which objects (partial or whole) to cache such that the space and bandwidth of the cache are uniformly utilized. Prefix caching is proposed in

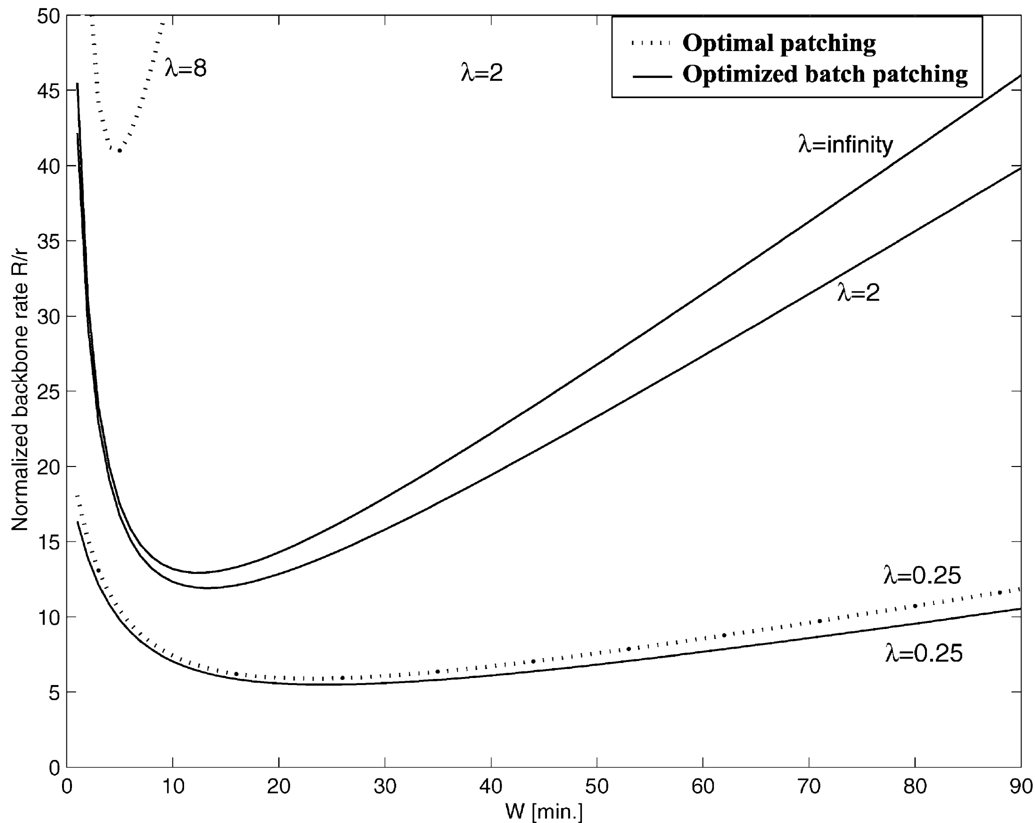

Fig. 2. Optimized batch patching versus optimal patching. This graph shows how the normalized backbone rate evolves with the size of the patching window W for different average inter-arrival rates following a Poisson distribution λ. The duration of the video asset T is 90 min. The batching interval b is set to 1 min.

Refs. [13,14]. Although caching the prefix can hide the startup latency and jitter in the network, this scheme does not reduce the aggregate transmission from the origin server.

In this paper, we build on some of the ideas in the multicast research area to minimize the aggregate bandwidth streamed out of the origin server and also present a practical scheme in which videos are cached in a proxy such that the space used by the video can be proportional to its popularity and the available bandwidth to the server.

3. Partial caching and batch patching

In this section, we develop a scheme which combines efficient stream scheduling at the server, and both prefix and partial caching in a proxy located close to the clients generating the requests.

Our main objective may be formulated as follows: given a set of video assets and their respective characteristics, minimize the average rate streamed out of the origin server under a cache of fixed capacity χ. We assume that the network is ideal (i.e. jitter- and error-free environment) and that the clients wish to be served instantaneously (i.e. null playback delay). Section 4 explains why these apparently strong assumptions do not lead to any loss of generality.

The motivation behind this problem formulation results from the following observation: minimizing the average backbone rate is equivalent to maximizing the average byte hit ratio (BHR) at the proxy under a given server scheduling strategy. The problem formulation is refined in Section 3.5 after the complete description of our joint strategy. The study is first performed on a single video asset. Then, we consider a heterogeneous set of video assets and related request patterns. We now describe the server scheduling strategy we build our scheme on.

3.1. Optimized batch patching

White and Crowcroft have recently introduced the concept of optimized batch patching [8], which aims at minimizing the average server output rate (i.e. backbone rate). Basically, client requests are batched together on an interval basis before requesting either a patch or a RM from the server. The interval is fixed and noted b. Following the reasoning from Ref. [7], there is an optimal *Patching Window*, noted W, after which it is more bandwidth efficient to start a new RM rather than send patches.

They refer to an RM-epoch as one in which a RM was started and a non-RM epoch as one in which a RM did not begin. The average backbone rate, R, is calculated in terms of the mean of the aggregate number of bytes contained in

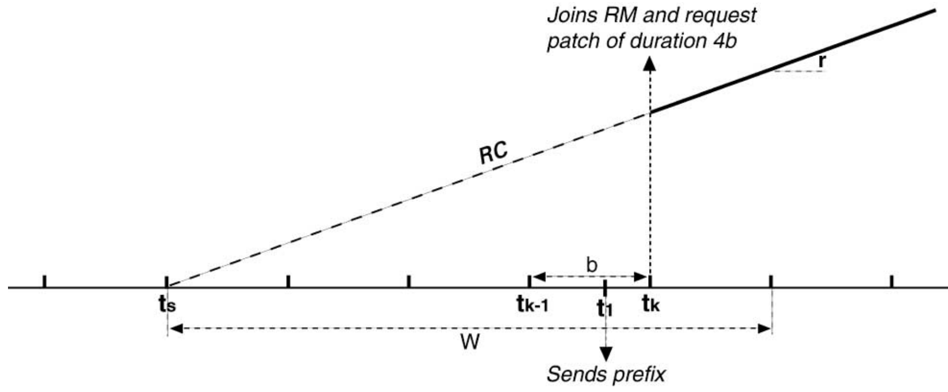

Fig. 3. Optimized batch patching with prefix caching. A Regular Channel of duration T-b starts at time t_s. A client requests an asset at time $t_1 \in [t_{k-1}, t_k)$. The interval limit t_k is such that $t_k < t_s + W$.

all patches commencing between two adjacent RM epochs chosen at random and the mean interval between RM epochs

$$R = \frac{(1 - P_b)rW^2 + (1 - P_b)brW + 2rbT}{2bW + \dfrac{2b^2}{1 - P_b}} \tag{1}$$

where $P_b = P_b(0)$ denotes the probability of gathering zero request in a batch of duration b (empty batch), T is the duration of the video and r denotes the streaming rate of the video asset.

The optimal patching window W is derived by differentiating R and setting the result equal to 0. This yields

$$W = \frac{-b + \sqrt{P_b b^2 + 2(1 - P_b)bT}}{b(1 - P_b)} \tag{2}$$

This scheme outperforms other multicast-based techniques in terms of average backbone rate over a large range of request rates. Fig. 2 compares the normalized backbone rate (that is, R/r) required by this scheme versus the optimal

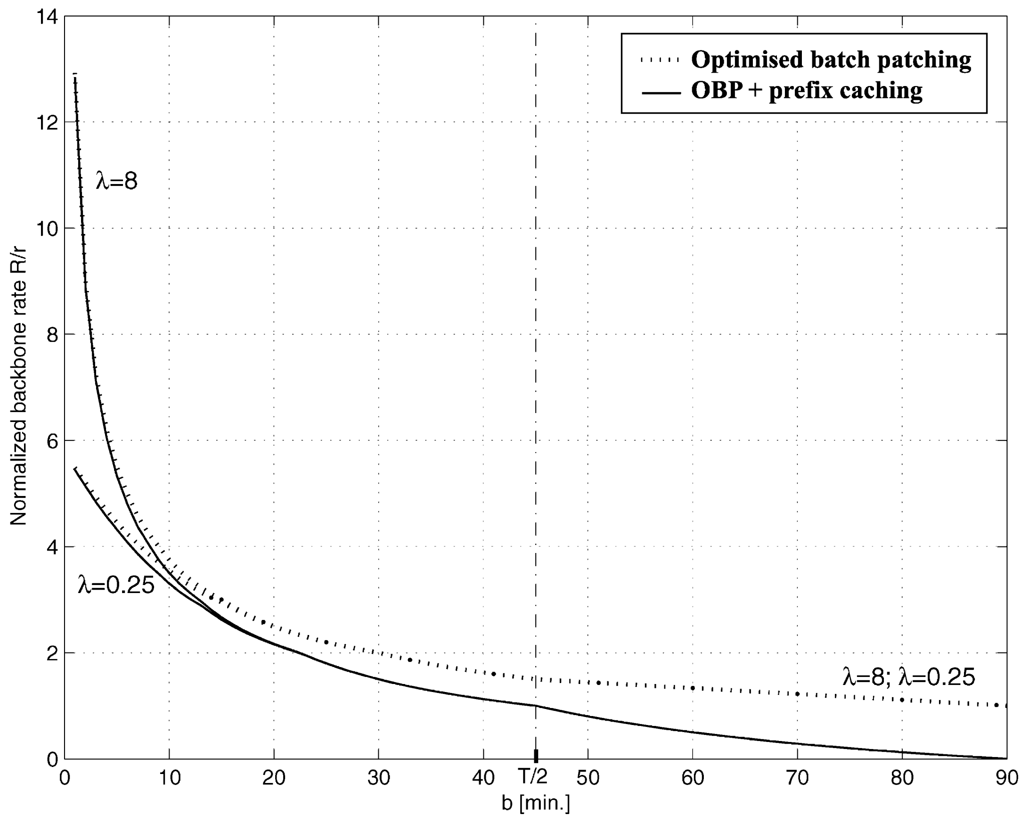

Fig. 4. Optimized batch patching with and without prefix caching. This graph shows how the normalized backbone rate evolves with the duration of the batching interval b (duration of the prefix) for different average inter-arrival rates λ following a Poisson distribution. The duration of the video asset is 90 min. The patching window is computed from Eq. (2) with T being replaced by $T - b$ when a prefix of b is cached.

patching algorithm [7] for different Poisson request rates (that is, $P_b = e^{-b\lambda}$) and a batching interval b set to 1 min. Optimized batch patching clearly outperforms optimal patching, albeit at the expense of higher latency (playback delay). Actually, the higher the interval b over which requests are batched, the better the performance is. Therefore, the authors integrated the concept of classes of service (CoS) in their scheme. In addition, these multicast-based techniques rely on multicast-enabled routers. In the remainder, we propose an extension to this scheme which alleviates the above problems and even adds some flexibility.

3.2. Partial caching applied to batch patching

We build on the optimized batch patching idea by introducing a proxy cache in the path from the origin server to the clients cloud. We adopt the intuitive approach consisting of storing the first b units of time in the proxy cache (i.e. batching period). That is, the proxy *permanently* caches a *prefix* of b units of time. Moreover, we impose the proxy cache to play the role of a client for the origin server. That is, all the patches and RMs streamed out of the server are requested by the proxy and are thereby streamed through it. This design approach has several advantages among which, (i) it eliminates the need for network-level multicast, (ii) it allows for client-based stream adaptation (heterogeneous client capabilities) and (iii) it has the potential to decrease the number of streams concurrently streamed to a given client. The former leads to a change of terminology. In the remainder, we use *regular channel* and *patch channel* instead of regular and patch multicasts.

Our scenario is illustrated by Fig. 3. The proxy divides the time axis into intervals $[t_{i-1}, t_i]$ of duration b units of time. Assume a request arrives at the proxy at time $t_1 \in [t_{k-1}, t_k)$. The proxy immediately starts streaming the requested asset to the client. Assume the most recent RC was started at time t_s, with $t_s < t_1$ is an integral number of b units of time. *If* t_k is such that $t_k < t_s + W$, the proxy joins the RC at time t_k and streams it through to the client, which buffers the stream while playing back the prefix. Also at time t_k, the proxy requests a patch of duration $t_k - t_s$ and passes it on to the client. However *if* $t_k \geq t_s + W$, a new RC of the asset of duration $T - b$ (the prefix of duration b is sitting in the proxy) is requested from the server at time t_k. In practice, the streams requested from the server are unicast to the proxy, which implements multicast at the application level to provide the services mentioned above.

The average backbone rate is computed from Eqs. (1) and (2) by replacing T with $T - b$. Fig. 4 compares the optimized batch patching technique with and without prefix caching in terms of the required normalized backbone rate versus the duration of the prefix b in minutes. We again assume a Poisson arrival process such that $P_b = e^{-b\lambda}$. The two techniques provide approximately the same normalized backbone rate for small values of b. The difference becomes noticeable for $b > 10$ min. Also, each technique provides the same performance independent of the request rate for batching intervals $b \geq 15$ min. Note that increasing b is equivalent to increasing either the client playback delay (without prefix) or the cache occupancy (with prefix). Note also that for $b \geq T/2$, the optimal patching window is zero ($W^{\text{opt}} = 0$) when a prefix is cached. Therefore, R/r reduces to

$$\frac{R}{r} = \frac{(T - b)(1 - P_b)}{b} \leq (1 - P_b).$$

The drawbacks of this straightforward extension are twofold. *First*, the link connecting the requesting client to the proxy must accommodate up to three concurrent streams. Indeed the first client of a batching period will have already triggered the streaming of the RC and of the patch, which will be needed by the late arrivals in the same batching period, which are still playing back the prefix. *Second*, the client buffer must accommodate up to $W + b$ units of time at the streaming rate r. Indeed the client must buffer the on-going RC while receiving the patch of maximum size W. In addition, the last client of the batching interval must store up to b extra units of time. Thus, $B = (W + b)r$, which may not be practical.

Therefore, we extend this first approach by considering the *temporary* partial caching of either (i) the patch only or (ii) the patch and the RC. In the first case, the proxy eliminates the need to stream the patch to the clients by temporarily caching the *right* portions of it (the client manages only up to two concurrent streams). In the second case, the proxy caches whatever it takes to allow for sequential streaming of the asset from beginning to end to the clients (the client manages only a single stream).

We now examine these two extensions separately and derive the equations leading to the estimation of the average backbone rate R, the average cache occupancy X and the client buffering requirements B. The derivations of all the equations are not presented here due to space constraints and can be found in Ref. [15].

3.3. Partial caching of patch only

The proxy eliminates the second stream to the client by caching the patch. Let $[t_{k-1}, t_k)$ denote a batch which requires a patch of k buffers in the interval b to kb. At t_k, the proxy determines which patch intervals are not cached[1] and starts fetching the first required interval and completes it at t_{k+1}. At this time, it is aware of whether or not there are requests in $[t_k, t_{k+1})$. If there are requests, it does not free the buffer when all the requests in $[t_{k-1}, t_k)$ are serviced but retains it to service requests in $[t_k, t_{k+1})$ and subsequent non-zero batches. Every time there is a batch with zero requests, the buffer is released once it has been streamed to all the clients in the previous batch.

By caching the patch, we clearly save on bandwidth from

[1] It needs to fetch at least one patch interval which is between $(k - 1)b$ to kb.

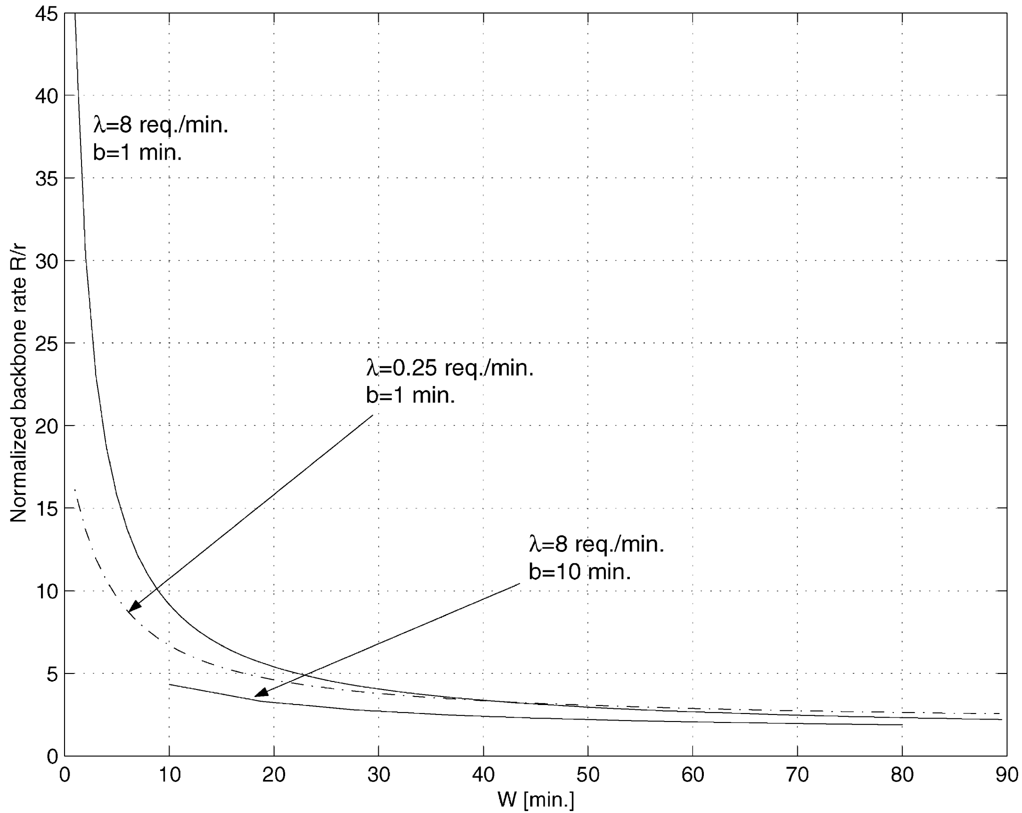

Fig. 5. Optimal batch patching with prefix and patch caching. This graph shows how the normalized backbone rate evolves with the duration of the patching window W for different average inter-arrival rates $\lambda = \{8, 0.25\}$ req/min following a Poisson distribution, and prefix durations $b = \{1, 10\}$ min.

the server compared to the previous approach. The number of patch buffers streamed from the server, in units of time, is given by μ (refer to Ref. [15] for details)

$$\mu = b \sum_{i=1}^{\left\lfloor \frac{W}{b} \right\rfloor} \left[i(1-P_b)P_b^{\left\lfloor \frac{i}{2} \right\rfloor} + \sum_{j=1}^{\left\lfloor \frac{i}{2} \right\rfloor} (2j-1)(1-P_b)^2 P_b^{j-1} \right] \tag{3}$$

The normalized average backbone rate R/r is thus obtained from

$$\frac{R}{r} = \frac{\mu r + (T-b)r}{b(1+n)},$$

where n is the mean number of batches between two RCs and is derived in Ref. [8] as

$$n = \frac{W}{b} + P_b(1-P_b)$$

Note that the reason why we average over $(n+1)$ intervals instead of over the entire duration of the stream is that the patch buffers obtained for one period of $n+1$ cannot be used in the next $n+1$ interval. This is because of the one intervening batching interval that triggers a RC. Requests in this batching interval do not require any patch and will consequently release the buffer $[b, 2b]$, which will be needed by the next batching interval. This triggers a new cycle of patch byte requests to the server as described earlier.

The cache buffer occupancy, which includes the prefix of duration b, is given by Eq. (4) below Ref. [15].

$$X = b + b\left[\left(\left\lfloor \frac{W}{b} \right\rfloor - 1 \right)(1-P) \sum_{i=1}^{\left\lfloor \frac{W}{b} \right\rfloor} (1-P)^i P^{\left\lfloor \frac{W}{b} \right\rfloor - i} \right] \tag{4}$$

The client still needs to buffer the on-going RC while playing back the patch. Late clients in a batching period buffer an additional b units of time. Thus, $B = (W + b)r$.

Fig. 5 shows the evolution of the normalized backbone rate R/r versus the duration of the patching window W under different average inter-arrival request times λ following a Poisson distribution and batching intervals b. Clearly, the longer the patching window is, the lower the backbone rate is. That is, the backbone rate may no longer exhibit a minimum value for a patching window duration within $[0, T - b]$. Indeed the longer the duration W, the higher the temporary patch buffer size required at the proxy.

Fig. 6 highlights this remark. Note that for $\lambda = 8$, P_b tends to zero and therefore, $W = T - b$ leads to $X = T$ (b permanently stored and $T - b$ units of time temporarily buffered). The tradeoff between permanent (prefix) and

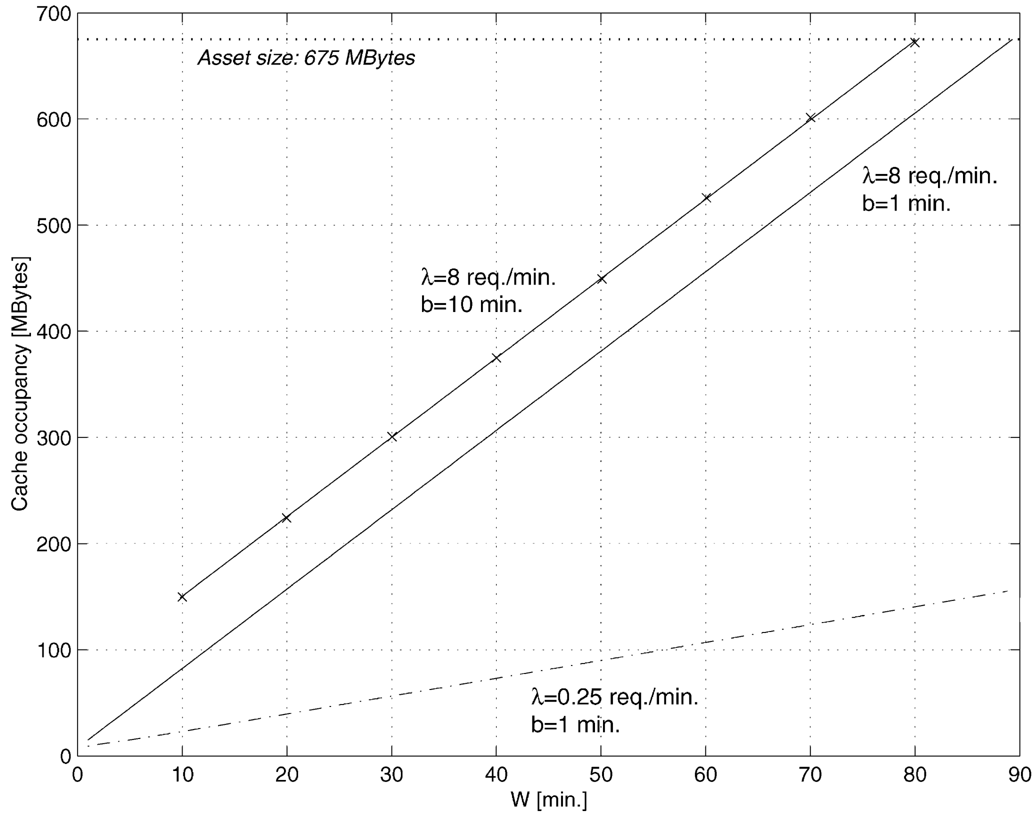

Fig. 6. Optimal batch patching with prefix and patch caching. This graph shows the cache occupancy versus the duration of the patching window W for different average inter-arrival rates $\lambda = \{8, 0.25\}$ req./min following a Poisson distribution, and prefix durations $b = \{1, 10\}$ min.

temporary buffers is also shown. Increasing the prefix duration leads to a gain in normalized rate (see Fig. 5) at the expense of higher buffer occupancy (see Fig. 6). The set of equations clearly indicates that the optimal solution to our optimization problem is full asset caching if the proxy cache can accommodate for it. We elaborate on this in Section 3.5. Finally, note that the slope of these straight lines is dictated by the factor $b\lambda$.

3.4. Partial caching of patch and regular channel

In this scheme, the client receives a single unicast stream from the proxy. The proxy caches data from the RC and forwards it to the clients. The client buffer requirement is zero in this case. Since the proxy serves all the requests in an interval of $W + b$ from a single RC, it has to maintain a circular buffer of up to $(W + b)$ units, continuously saving data from the RC, for the entire duration of the video. This buffer is required for each instance of the RC, which is triggered every $(n + 1)$ intervals. The size of this buffer for each RC depends on the batching intervals that have non-zero requests. If there are requests in an interval $[t_{k-1}, t_k)$ that require a patch of $(k - 1)$ buffers, then kb needs to be buffered from the ongoing stream while these requests are playing back the patch and/or the prefix. This is irrespective of whether or not the previous batching interval had any requests.

The scheme is better explained with an example. Suppose there are five batching intervals ($[t_{i-1} - t_i)$, $i = 1, 5$) in $W + b$, that is, $W = 4b$. The interval $[t_0, t_1)$ triggers the RC at t_1. The proxy buffers b from the RC to accommodate late requests in this interval that are playing back the prefix. Suppose $[t_1 - t_2)$ had requests, then at t_2 the proxy adds another buffer to circular buffer and does not request the server for any patch bytes. Suppose the next two intervals do not have any requests and the fifth interval $[t_4 - t_5)$ has a request. At t_5, the $2b$-long circular buffer allocated by the first two intervals would have advanced with the RC and will contain the interval $[3b - 5b]$. The fifth interval requires the buffer to be $5b$ long and so it adds three more buffers of size br to the circular buffer. The proxy then fetches the missing patch bytes in the intervals $[b - 2b]$ and $[2b - 3b]$ from the server while storing the interval $[5b - 6b]$ from the RC.

Here again, we reuse buffers that are allocated by each interval of W for subsequent intervals. Each RC results in the caching of the stream equivalent to the circular buffer (X) allocated to it, thereby reducing the length of the stream to be transmitted by the server by X. Taking this into account, the average backbone rate R/r over the total asset duration is given by Ref. [15]

$$\frac{R}{r} = \sum_{i=1}^{\frac{T-b}{n+1}} (\mu + (T-b) - (i-1)X^*)$$

where μ is obtained from

$$\mu = b(1-P)P \sum_{i=1}^{\left\lfloor \frac{W}{b} \right\rfloor} \sum_{j=1}^{i} jP^{j-1}$$

and the cache occupancy X^* computed over $(n+1)$ batching intervals is

$$X^* = b \sum_{i=1}^{\left\lfloor \frac{W}{b} \right\rfloor} i(1-P)P^{\left\lfloor \frac{W}{b} \right\rfloor - i},$$

To obtain the cache occupancy X, over the total asset duration, we simply multiply X^* by $T - b/n + 1$, to account for the multiple RCs that are active simultaneously, each of which uses a storage of X^*. Therefore, if we have a request in every single interval of b (i.e. $P_b = 0$), we end up caching the whole stream in average.

The same conclusions about R/r versus W for different λs and bs as in the previous scheme hold in this scheme as well. However, this scheme clearly uses more temporary buffer at the proxy (higher likelihood to fully cache the assets), while decreasing the backbone rate significantly. Moreover, this scheme results in a single stream to the clients and requires no storage from the client device (i.e. $B = 0$), which makes this approach well suited to streaming multimedia assets to handheld devices. This scheme is very similar to Interval Caching [16] where $W + b$ is the length of an interval. The difference is that this interval is cached in the proxy and is stored for use by future intervals.

3.5. Multiple assets

The multiple assets problem may be formulated as follows: *Given* a cache of capacity χ, and a set of N videos assets characterized by their streaming rate r_i, duration T_i and interval-based request probability P_{b_i} *find* the tuple $\{b_i, W_i\} \forall 1 \le i \le N$ that *minimizes* the aggregate backbone rate $R = \sum_{i=1}^{N} R_i$, *under* the constraint $\sum_i^N X_i \le \chi$.

Clearly, this problem can be solved via standard optimization techniques. However, its multi-dimensional nature makes it computationally expensive. Therefore, we propose a simplification which relies on the previous observation that both an increase of the prefix duration b_i or the patching window W_i for asset i always result in a lower backbone rate R_i and a corresponding increase of the temporary buffer X_i. Also, we have shown that W_i must be null for $b_i \ge T_i/2$. These observations lead to the following simplification: we impose $W_i = b_i$ for all $b_i < T_i/2$. This simplification slightly degrades the performance of our system but dramatically decreases the complexity.

We now briefly describe the algorithm based on the above simplification: First, we impose a video unit (e.g. a group-of-pictures) by which the prefixes b_i will either be decreased or increased. Then we calculate for every asset, the product of its popularity by its respective size (similar to the SLRU technique). Assume that all the b_i are first set to T_i. We eliminate a number of video units from the prefix duration b_i. This number is inversely proportional to the product value for asset i. Finally, we iterate until the sum $\sum_i^N X_i \le \chi$.

We compared the full caching technique with the optimal batch patching with prefix and patch caching scenario. We assumed a Poisson distribution of the request inter-arrival times, a Zipf distribution of the asset popularity with various parameters (from which the λ_i are derived), a set of 100 videos with constant streaming rates and durations uniformly distributed in, respectively $[56, 1500]$ kbps and $[15, 90]$ minutes, and a cache size χ three times smaller than the sum of all the asset sizes. Preliminary results show that the gain in backbone rate is tremendous (from approximately 4 to 8 times lower depending on the parameter of the Zipf distribution). Moreover, our scheme rapidly adapts to changes in request statistics, while this is a known drawback of a full caching scenario.

In the remainder, we detail algorithms for practical implementations of the schemes proposed in this section.

4. Practical issues

In all of the scenarios, the proxy receives the client requests, immediately starts streaming the prefix and also batches them, on a per asset basis. At the end of each batch, the proxy determines if a RC needs to be started or a patch needs to be requested from the origin server, based on the value of W computed using the expression in Section 3.

The proxy also runs the optimization algorithm described in Section 3.5 periodically to determine the prefixes that need to be stored in the cache. Initially, the cache starts out with CTL prefixes of the most popular videos. The optimization algorithm may be triggered either periodically, when the network utilization at the proxy falls below a certain threshold or when the access probabilities of assets change significantly. In all cases, the algorithm requires an estimate of P_b for each of the assets, to determine the length of the prefix to cache and thereby maximize the BHR. Note that the value of P_b changes with the value of b. The bigger the chosen b is, the smaller is the probability of having zero requests in the batch. Various methods can again be adopted to determine the value of P_b. As a simple approach, we could assume the inter-arrival request rate follows a Poisson distribution and track the inter-arrival time of requests over a certain time window. A more accurate but complex method would be to track the inter-arrival times, fit it to a well-known distribution and use the characteristics of the distribution to estimate P_b.

The optimization algorithm determines the value of the prefix b for each asset, given the access probabilities. The

```
ProcessRequest(r)
{
  CreateThread(StartStreamToClient(r));
  // add this request to the current batch
  addRequestToCurrBatch(r);
  // Incr.  the rqst count in each relevant
  // interval
  for (i=1; i<=currBatch.numPatchIntervals; i++)
    IntervalTable[i].RequestCount++;
}

StartStreamToClient(r)
{
  StreamPrefix();
  //b time units have elapsed and the
  //next reqd.  patch has been fetched
  for (i=1; i≤currBatch.numPatchIntervals; i++)
    StreamInterval(IntervalTable[i].buffer);
}

// This runs on a separate thread.
CloseCurrentBatch()
{
  for(;;) {
    Sleep(b) // wait for batch to end
    currBatch = startNewBatch();
    currBatch.startTime = now;
    currBatch.endTime =
      currBatch.startTime + b;
  If (newRCReqd())
    currRC = StartNewRC();
  // Add clients to regular channel
  currRC.AddBatchedClients();
  for (i=1;i<=currBatch.numPatchIntervals;i++) {
    if (IntervalTable[i].RequestCount == 0)
      // If there are no requests, free it
      // This happens if currBatch has zero
      // requests
      free(IntervalTable[i].buffer);
    else if (IntervalTable[i].buffer == NULL) {
      // get patch in new segment buffer
      IntervalTable[i].buffer = newBuffer();
      //Fetch patch interval so that it is
      //available when needed by
      // StartStreamToClient
      FetchFromOriginServer((i-1)*b, i*b);
      // once the segment is fully obtained,
      // update the table and get next segment
    }
  }
}
```

Fig. 7. Algorithm at the proxy for scenario 2.

proxy then updates the cache to the new state computed by the algorithm. With the prefixes stored in the cache, it streams videos to clients using one of the three scenarios. The following subsections describe the algorithm in the proxy for the three different schemes described in Section 3.

4.1. Cache prefix only

The proxy only stores prefixes of videos. When a request is received, the proxy may send up to three concurrent streams to the client — the prefix, the patch stream and the RC. In our scheme, an application level multicast is used to stream the latter two. The client listens on three ports for streams from the proxy — the prefix, the patch and the RM. While playing back the prefix, if data is received from the patch stream, it is cached and so is the data from the ongoing multicast. The buffer used by the client is at most $(W + b)r$.

4.2. Cache prefix and patch

In this scenario, the proxy caches data from the patch, besides the prefix, and streams these to the client thus reducing the number of streams to the client to two. For each asset that has ongoing streams, the proxy maintains an interval table, which holds information about which intervals of the video are currently cached. Each interval buffer is b time units long and is retained as long as there are requests being served from it. Since a patch can be at most W time units, we have W/b intervals in the interval table. The pseudocode for the proxy is presented in Fig. 7.

Each time a request is received by the proxy, it increments the request count against each of the patch intervals that this request needs. Moreover, at the end of each batch, the proxy checks to see if the batch triggers a RC or a patch. If it triggers a RC, the proxy requests a unicast stream from the origin server and application-level multicasts it to all the clients in the batch. If not, it determines which of the required patch intervals are locally cached, and fetches the remaining intervals from the server over a unicast patch channel. The patches are not multicast, but are streamed to the client individually since they may be at different points in the playback of the prefix. The proxy also joins all the requests in this batch to the ongoing RC.

The client receives two streams in this case — the patch and the RC. It buffers the RC while playing back the patch. The client may buffer a maximum interval of $W + b$ from the RC while it is playing back the prefix and the patch.

4.3. Cache prefix, patch and regular channel

In the third scenario, we save significantly in the required client storage and also in the number of simultaneous

```
CloseCurrentBatch()
{
   for(;;) {
    Sleep(b) // wait for batch to end
    currBatch = startNewBatch();
    currBatch.startTime = now;
    currBatch.endTime =
      currBatch.startTime + b;
    If (newRCReqd()) {
      currRC = StartNewRC();
      CircularBuffer = new CircBuffer(b);
    }
    if (currBatch.numRequests > 0) {
       n = currBatch.numPatchIntervals+1 -
         numAllocatedIntervals;
      CircularBuffer.Grow(n*b);
      numAllocatedIntervals++;
    }
    CreateThread (
    FetchMissingPatchIntervals())
    BufferRegularChannel()
  }
}
```

Fig. 8. Algorithm at the proxy for scenario 3

streams to the client. The client needs to buffer nothing and receives only one unicast stream from the proxy. The proxy stores data from the RC and serves patches from this buffer instead of requesting it from the origin server. Although this reduces the bandwidth streamed from the server, it requires a larger buffer in the cache on average. For a discussion in the bandwidth/buffer tradeoff, see Section 3.

The difference between this scenario and the previous one is that, in the previous case, in addition to the prefix, only patch buffers were being allocated at the proxy. At any time, there can be a maximum of W/b buffers active. However, when data is stored from the ongoing stream, the cache has to hold on to the buffer allocated in each window of $W + b$, for the duration of the video, continuously caching ahead from the ongoing stream. The algorithm; whose pseudo-code is given in Fig. 8; is applied by proxy separately for each RC.

At the end of each batch, the proxy determines how large the interval cache is. It would consist of as many intervals as was required by the last non-zero batch in this patching window. It then increases the circular buffer to be as big as the patch required for this batch plus the extra b. This can be better explained with an example. For ease of explanation, batch b_i refers to a batch which requires a patch of bi segments. For instance, if we are at the end of batch b_4 at t_{5b} and the last non-zero batch was b_2, then the interval cache for this RC would be $2b + b$ long and since it is caching the ongoing stream, the buffer would contain intervals $2b - 3b$, $3b - 4b$ and $4b - 5b$. Batch b_4 needs $b - 2b$ as a patch and buffer $5b - 6b$ from the RC. So, it allocates two buffers and starts storing the ongoing stream in one while simultaneously filling the other with the patch $b - 2b$ from the origin server, while the clients are playing back the prefix. Once the requests complete the prefix, the interval $b - 2b$ is available in the buffer and they continue to play back the stream and continue through the circular buffer until the end of the stream. Note that in this scheme, each client gets an individual stream and true application-level multicast is not done.

The number of active intervals in each $W + b$ is as large as the number of non-zero batches. Intuitively, if all batches during the duration of the stream have non-zero requests, this video is really popular and we much cache the entire video and not have to request the origin server. This is the result the optimization algorithm yields. If it determines that the space used by the prefix and the interval cache is as large as the entire video, it instructs the proxy to do a full-caching of the video.

4.4. Discussion

When discussing streaming video using batching and multicasting, it is important to address issues such as network delays, jitter and random seeking (VCR functions), that most multicasting schemes do not address. The fact that our design uses application level multicasting addresses the network delay and jitter issues.

First, since all the data flows through the proxy and the proxy is aware of the network bandwidth to the server and to all the clients, it can perform QoS-related adaptations to the stream. Our implementation considers network adaptation on two planes. The optimization algorithm determines the prefix to cache for each asset depending not only on the popularity of the asset, but also on the bandwidth available to the server. The prefix cached is enough to mask the network latency and jitter to stream from the server. Also, when the proxy determines that it has to request a stream from the server (patch or RC), it determines the available bandwidth on the link and requests the stream from the server a δt time earlier. The time estimate could also be influenced by contracted service levels for given objects. Additionally, the proxy has the ability to also perform stream adaptation services to cater to heterogeneous clients.

As part of ongoing work, we are investigating schemes to support VCR functions in our framework. Various schemes for supporting VCR functions in a multicast-based VOD system are presented in Refs. [17–19].

5. Conclusions

In this paper, we present a joint server scheduling and proxy caching scheme aimed at minimizing the bandwidth streamed from the origin server. The scheme combines the bandwidth-saving merits of multicast streaming with QoS and content adaptation service capabilities of a proxy. We present multiple schemes with different bandwidth and cache-space tradeoffs that are applicable in different scenarios with different service requirements. Our schemes enable the honoring of service levels (SLAs) at the network-edge streaming proxies by adopting different tradeoffs for assets with different SLAs. From our simulations, it is evident that our scheme far outperforms full-caching schemes where an asset is cached fully or not at all. We are working on various aspects of this scheme currently, one of which is the support for VCR functions. We are also in the process of implementing a prototype version of this technique in IBMs VideoCharger Server [20].

References

[1] C.C. Aggarwal, J.L. Wolf, P.S. Yu, A permutation based pyramid broadcasting scheme for Metropolitan VOD systems. Proceedings of the IEEE International Conference on Multimedia Systems, June 1996.

[2] A. Dan, D. Sitaram, P. Shahabuddin, Scheduling policies for an on-demand video server with batching. Proceedings of ACM Multimedia, October 1994.

[3] K.A. Hua, S. Sheu, Skyscraper broadcasting: a new broadcasting scheme for metropolitan VOD systems. Proceedings of the ACM SIGCOMM, 1997.

[4] S. Viswanathan, T. Imielinski, Metropolitan Area Video-on-Demand

Service using Pyramid Broadcasting, Multimedia Systems, vol. 4 1996.

[5] K.A. Hua, Y. Cai, S. Sheu, Patching: a multicast technique for true on-demand services. Proceedings of ACM Multimedia, September 1998.

[6] S. Sen, L. Gao, J. Rexford, D. Towsley, Optimal patching scheme for efficient multimedia streaming. Proceedings of the IEEE International Conference on Multimedia Computing and Systems, June 1996.

[7] Y. Cai, K. Hua, K. Vu, Optimizing patching performance. Proceedings of ACM/SPIE Multimedia Computing and Networking, January 1999.

[8] Paul P. White, Jon Crowcroft, Optimized batch patching with classes of service, ACM Computer Communications Review (2000) Vol 30 no. 4.

[9] M. Reisslein, F. Hartanto, K. W. Ross, Interactive video streaming with proxy servers, in Proceedings of First International Workshop on Intelligent Multimedia Computing and Networking (IMMCN), pp (II) 588–591, Atlantic City, NJ, February 2000.

[10] Reza Rejaie, Mark Handley, Haobo Yu, Deborah Estrin, Proxy caching mechanism for multimedia playback streams in the internet. Proceedings of the Fourth International Web Caching Workshop, March 1999.

[11] Soam Acharya, Brian Smith, Middleman: a video caching proxy server. Proceedings of NOSSDAV, June 2000.

[12] Renu Tewari, Harrick Vin, Asit Dan, Dinkar Sitaram, Resource-based caching for web servers. Proceedings of MMCN, 1998.

[13] S. Sen, J. Rexford, D. Towsley, Proxy prefix caching for multimedia streams, IEEE Infocomm (1999) Vol 3 pp 1310–1319.

[14] Y. Wang, Z-L. Zhang, D. Du, D. Su, A network conscious approach to end-to-end video delivery over wide area networks using proxy servers, IEEE Infocomm (1998) Vol 2 pp 660–667.

[15] O. Verscheure, C. Venkatramani, P. Frossard, L. Amini, Joint server scheduling and proxy caching for video delivery. IBM Technical Report Number RC21981, 2001.

[16] A. Dan, D. Sitaram, A generalized interval caching policy for mixed interactive and long video environments. SPIE Multimedia Computing and Networking Conference, January 1996.

[17] E.L. Abram-Profeta, K.G. Shin, Providing unrestricted VCR functions in multicast video-on-demand servers. IEEE International Conference on Multimedia Computing and Systems, June 1998.

[18] K.C. Almeroth, 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 (1996) Vol 14 (6) pp 1110–1122.

[19] Z. Fei, I. Kamal, S. Mukherjee, M. Ammar, Providing interactive functions for staggered multicast near video-on-demand systems (extended abstract). Proceedings of the IEEE International Conference on Multimedia Computing and Systems (Poster paper), June 1999.

[20] IBM, The IBM VideoCharger Server, details available at http://www.ibm.com/software/data/videocharger/.