Periodic Broadcast and Patching Services - Implementation, Measurement, and Analysis in an Internet Streaming Video Testbed[1]

Michael K. Bradshaw, Bing Wang, Subhabrata Sen, Lixin Gao[†], Jim Kurose, Prashant Shenoy, and Don Towsley

Dept. of Computer Science
{bradshaw,bing,sen,kurose,
shenoy,towsley}@cs.umass.edu

[†] Dept. of Electrical and Computer Engineering
lgao@ecs.umass.edu

University of Massachusetts
Amherst, MA 01003

Abstract—
Multimedia streaming applications can consume a significant amount of server and network resources. Periodic broadcast and patching are two approaches that use multicast transmission and client buffering in innovative ways to reduce server and network load, while at the same time allowing asynchronous access to multimedia steams by a large number of clients. Current research in this area has focussed primarily on the algorithmic aspects of these approaches, with evaluation performed via analysis or simulation. In this paper, we describe the design and implementation of a flexible streaming video server and client testbed that implements both periodic broadcast and patching, and explore the issues that arise when implementing these algorithms. We present measurements detailing the overheads associated with the various server components (signaling, transmission schedule computation, data retrieval and transmission), the interactions between the various components of the architecture, and the overall end-to-end performance. We also discuss the importance of an appropriate server video segment caching policy. We conclude with a discussion of the insights gained from our implementation and experimental evaluation.

I. Introduction

The emergence of the Internet as a pervasive communication medium has fueled a dramatic convergence of voice, video and data on this new digital information infrastructure. A broad range of multimedia applications, including entertainment and information services, distance learning, corporate telecasts, and narrowcasts will be enabled by the ability to stream continuous media data from servers to clients across a high-speed network.

Several challenges must still be met, however, before high quality multimedia streaming becomes a widespread reality. Many of these challenges result from the significant load that many video applications place on both server and network resources. This load results from both the high bandwidth requirements ($4-6$ Mbps for full motion MPEG-2) and the long-lived nature (tens of minutes to $1-2$ hours) of these applications. The scenario is further complicated by the fact that the client population is likely to be both large and heterogeneous, with different clients asynchronously issuing requests to control their received media streams. Because of these concerns, there has been tremendous interest in developing algorithms for efficient distribution of network video to such a client population. Periodic broadcast and patching [1–7], described in more detail in Section II, are two approaches that have received considerable recent attention. These approaches exploit the use of multiple multicast channels to reduce network and server resource use over the case of multiple unicast transmissions, while at the same time satisfying the asynchronous requests of individual clients and providing a guaranteed bound on playback startup latency. Research on periodic broadcast and patching has been primarily algorithmic in nature, with performance studied either analytically or through simulation. In either case, simplifying assumptions are necessarily made (e.g., abstracting out control/signaling overhead, operating system issues such as the interaction between disk and CPU scheduling, multicast join/leave times, and more) in order to evaluate performance. While there are a number of existing production (e.g., Darwin, RealServer, Windows Media Server, Oracle Video Server) and experimental [8–18] video server efforts, all of them use traditional unicast or multicast streaming. To date, there have been neither implementations nor empirical evaluations of either periodic broadcast or patching.

In this paper we report on the implementation, measurement, and analysis of a working video server testbed implementing both periodic broadcast and patching algorithms. Our testbed consists of a 100Mbps switched Ethernet LAN connecting a Linux-based, application-level video server with a collection of both Linux- and Windows-based clients. The goals of our work are to develop a proof-of-concept prototype, and to use this prototyping effort to expose and develop solutions to the underlying system issues that arise when putting periodic broadcast and patching algorithms into practice.

Our experimental evaluation presents measurements detailing the overheads associated with the various server components (signaling, transmission schedule computation, data retrieval and transmission), the interactions between the various components of the architecture, and the overall end-to-end performance. We find that our server is well-able to support the real-time, bandwidth-intensive data delivery requirements imposed by periodic broadcast and patching. Under periodic broadcast, our server can easily process a client request rate of 600 requests per minute (returning periodic broadcast schedule information to each client), while at the same time streaming video segments over multiple multicast groups and missing no more than a few (less than 1%) data transmission deadlines. Under patching, our server can come close to fully loading a 100 Mbps network with patched-in clients, again while missing very few data transmission deadlines. Our measurements also show that in a loaded LAN environment, an initial client startup delay of less than 1.5 seconds is sufficient to handle startup signaling and absorb data jitter induced at either the client or the server. Our evaluations also show that we can dramatically reduce the demands placed on the underlying server operating system by using a Least Frequently Used (LFU) video-data-cache replacement policy. More generally, our results highlight the importance of combining theoretical work with implementation and empirical evaluation to fully understand systems issues.

The remainder of the paper is organized as follows. Section II discusses periodic broadcast and patching algorithms and identifies the system issues we face in implementing periodic broadcast and patching. Section III describes the client and server architecture, as well as the interactions (e.g., signalling) between them. Our experimental configuration and the performance metrics of interest are discussed in Section IV, followed by our measurements, analysis and evaluation in Section V. Finally, Section VI reflects on the important lessons learned and concludes the paper.

II. Background: Algorithms and Design Principles

In this section we present background material on the multimedia transmission algorithms, and identify key design principles of our streaming media testbed.

Many Internet multimedia applications have asynchronous clients that may request playback of the same video stream at different times. Economically viable high-volume video services will require effective techniques to minimize the incremental cost of serving a new client, while also limiting client start-up latency and the likelihood of rejecting requests due to resource constraints. For popular video streams, server and network resources can be significantly reduced by allowing multiple clients to receive all, or part of, a single multicast transmission [1–4, 19–21]. For example, the server could *batch* requests that arrive close together in time [19], and multicast the stream to the set of batched clients. A drawback of batching, however, is that client playback latency increases with an increasing amount of client request aggregation. Several recently proposed techniques, such as periodic broadcast and patching [1–7], overcome this drawback by exploiting client buffer space and the existence of sufficient client network bandwidth to listen to multiple simultaneous transmissions. These capabilities can be used to reduce server and network resource requirements, while still guaranteeing a bounded playback startup latency.

Periodic broadcast schemes [1–4, 19] exploit the fact that clients play back a video sequentially, allowing data for a later portion of the video to be received later than that for an earlier portion. A period broadcast server divides a video object into multiple segments, and continuously broadcasts these segments over a set of multicast channels. To limit playback startup latency, earlier portions of the video are broadcast more frequently than later ones. Clients simultaneously listen to multiple channels, storing future segments for later playback.

In patching (a closely related technique is "stream tapping") [5–7, 22], the patching server streams the entire video sequentially to the very first client. Client-side workahead buffering is used to allow a later-arriving client to receive (part of) its future playback data by listening to an *existing* ongoing transmission of the same video; the server need only additionally transmit those earlier frames that were missed by the later-arriving client. As a result, server and network resources can be saved. Unlike batching, patching allows a client to begin playback immediately by receiving the initial video frames directly from the server. Similar to periodic broadcast, patching exploits client buffer space to store future video frames. Unlike periodic broadcast, a patching server transmits video data on-demand, when a new client arrives.

Our overview of periodic broadcast and patching has been necessarily brief. We note here that our goal in this paper is not to propose a new periodic broadcast or patching algorithm, but rather to explore the issues that arise when these algorithms are put into practice. In our testbed, we implement the periodic broadcast algorithm from [2] and the patching algorithm from [22] as representative algorithms. For a detailed description of these and other periodic broadcast and patching approaches, the reader is referred to the references cited above.

Before describing the server and client architecture in the following section, it is valuable to consider several important principles embodied in the design and implementation of our server testbed:

- **Separation of control and data functionality.** Both server and client implementations separate control and data functionality. Since the control and data paths impose significantly different demands on the underlying system, this separation allows us to independently optimize each component. A clean separation of control and data paths also allows us to experiment with different server architectures. From an operational standpoint, we will see that this sepa-

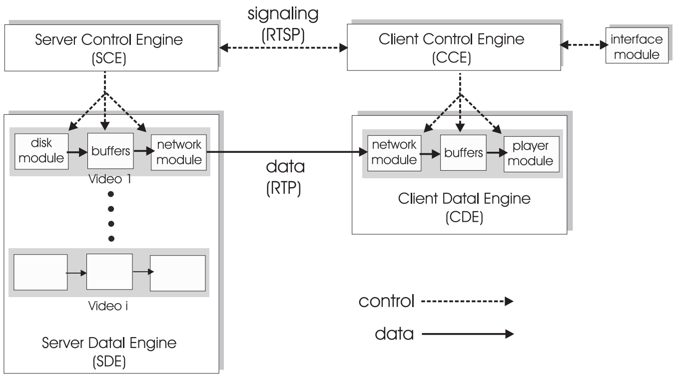


Fig. 1. Server and client architecture and interaction

ration also allows one component to be isolated from effects of a workload overload on the other.

- **Standards-based architecture.** Our server and client implementations are based on existing streaming media standards such as RTP [23, 24], RTSP [25] and SDP [26]. The advantages of a standards-based architecture are two-fold. First, it allows us explore how various streaming media techniques such as periodic broadcast and patching can be implemented in the context of these standards. Second, it helps us identify potential limitations of these standards in supporting such techniques.
- **Support for IP Multicast.** Our server and client implementations are designed to take advantage of IP multicast. The use of IP multicast facilitates more efficient use of server and network resources. Of particular interest to us in this paper are practical considerations that arise in the use of IP multicast (e.g., multiplexing a finite number of multicast channels among users, client join/leave latencies and techniques to hide such latencies). We note that while many-to-many inter-domain multicast has been slow to be deployed, one-to-many intra-domain multicast (as would be used in an enterprise or cable/DSL-based last-hop network network video server) is much simpler to deploy and manage [27].
- **Use of off-the-shelf components.** Our server and client are designed to run on vanilla operating systems such as Linux and Windows. This allows us to easily set up multiple clients and server. However,we do not benefit from the numerous special-purpose resource management techniques (e.g., rate-based scheduling) that have been proposed recently.

III. Server and Client Architecture and Interaction

In this section we describe the server and client architecture, as well as the control signaling that occurs between them. A high-level overview of the server and client architecture shown in Figure 1. We begin with the server.

A. Server Architecture

The server consists of two main modules, a **Server Control Engine (SCE)** and a **Server Data Engine (SDE)**, embodying our principle of separating control and data functionality. The primary role of the SCE is to handle control interactions between the server and its clients. The primary role of the SDE is to retrieve video data from disk (or an in-memory cache), and transmit the data into the network.

For each client request, the SCE computes two schedules.

- The **transmission schedule** specifies when "segments" of each video are to be retrieved from disk (or the in-memory cache) and transmitted into the network. A "segment" of video contains a continuous portion of data from a given video; the start/stop times of a video's segments depend on the transmission algorithm (periodic broadcast or patching) used, and (for the case of patching) the requests being generated by clients. The SCE passes the transmission schedule to the SDE, which then retrieves and transmits RTP-formatted data according to this schedule. There is only one transmission schedule per video, even when multiple clients are receiving a given video.
- The **reception schedule** specifies the order in which the end-client receives this data. The reception schedule, formatted as an SDP message, is sent to the client in an RTSP response; the client then uses this schedule to receive data on the specified multicast or unicast channels.

As noted above, the two main server components are the server control engine (SCE) and the server data engine (SDE). Let us consider each of these in turn.

A.1 Server Control Engine (SCE)

The SCE is implemented as a multi-threaded single-process system. A single SCE **listener thread** listens on a well-known port for incoming client requests, and places an incoming request on a message queue. A pool of free **scheduler threads** wait to serve requests on the message queue. Once a scheduler thread receives an incoming request, it is responsible for all subsequent control interaction with the client, including the generation of the transmission and reception schedules, and the sending of the reception schedule to the client.

Because of our use of periodic broadcast and patching, scheduling requires different information than is typically used in video servers. The scheduler thread must be aware of which videos are currently be transmitted and the particular broadcast algorithm being used. If a requested video is already playing, the scheduling thread may need to augment the SDE's transmission schedule for that video to accommodate the new client. For example, if a client requests a video that is being broadcast via patching, the scheduler thread must determine whether a unicast data patch should be sent to the client (as well as the specific data that is to be sent), or whether a new multicast transmission of that video should be initiated. In either case the scheduler thread will need to make the necessary changes to the transmission schedule, and inform the SDE of the new schedule. After sending the reception schedule to

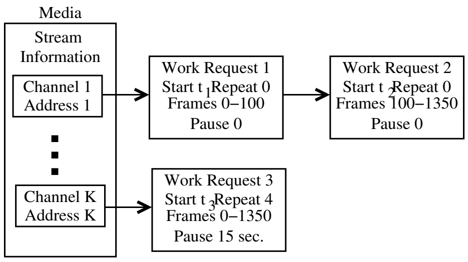


Fig. 2. Data Structure Organization

the client, a scheduler thread waits for a new client request.

The example above suggests that the transmission schedule data structure must be carefully designed in order to be sufficiently general to express a transmission schedule for different video delivery schemes (e.g., batching, patching, and periodic broadcast). As illustrated in Figure 2, a data structure *(Media)* is maintained for each media stream currently being transmitted. This data structure contains stream-specific information such as the file location, length, and type of the stream. It also contains a list of structures, with each element corresponding to a multicast or unicast channel on which some part of the video is to be transmitted. Since portions of the video can be transmitted simultaneously on multiple channels, a list of channels is needed. Each channel structure contains information identifying the corresponding channel, type of transmission (multicast or unicast), and a linked list of structures known as "work requests." The work request list contains information that determines what data will be transmitted on that channel, and when. We note that an important advantage of specifying the transmission schedule at the frame level is that it facilitates uniform handling of different video file formats, e.g., MPEG, AVI, etc., at the SCE.

Let us illustrate the representation of a transmission schedule via a simple example. Suppose the server must deliver a 45 sec (1350 frame) video according to the following transmission schedule : (i) initiate transmission of the frames $0 - 100$ on some address at time t_1, (ii) initiate transmission of frames $100 - 1350$ on the same address at some later time t_2. The work request list associated with channel 1 in Figure 2 shows the abstract representation for this schedule. Channel 1 in the video structure is initialized with the outgoing address of this stream. The linked list of work requests indicates that at time t_1 frames $0 - 100$ will be played and frames $100 - 1350$ will be played at time t_2.

Some algorithms, such as periodic broadcast, require the repeated transmission of a sequence of frames. If the server must transmit frames 0 through 1350 on a second connection, once every minute starting at time t_3, for a total of five transmissions, it then allocates a new channel (Channel K in Figure 2). A single work request is then associated with this channel, with Start Frame=0, End Frame=1350. Assuming a 15 sec gap after each complete transmission of the video, the Pause field is set to 15 sec. The Repeat field is set to 4, indicating the video transmission will be repeated four additional times after the first run.

The final important piece of the SCE is the multicast channel pool. In addition to determining what video segment to send and when, the SCE must also select a multicast address to be used. Rather than searching over all available time slots on all available multicast channels, the SCE uses horizon scheduling [28] to efficiently make this assignment.

A.2 Server Data Engine (SDE)

Recall that the SDE is responsible for retrieving video data from memory (either disk or an in-memory cache) and then transmitting this data into the network.

The SDE is a multi-threaded, single-process entity. It maintains two threads for each video that is currently being transmitted. A **disk thread (DT)** handles retrieval of the video's data from disk into main memory; a separate **network thread (NT)** transmits the data from main memory to the network according to the server transmission schedule. A global buffer cache manager is responsible for allocating equal-sized free memory blocks to each DT. Each individual DT, in turn, is responsible for managing its cache of video data. Currently, each DT is allocated an equal-sized portion of the in-memory cache. We'll see later that the global buffer's size and the DT's cache management policy play an important role in determining system performance.

Both the DT and NT operate in rounds. Let the disk round-length be denoted by δ and the network round-length be denoted by τ. In each δ-round, the disk thread wakes up, uses the server transmission schedule to determine which parts of the video are to be retrieved in that round, issues asynchronous read requests for retrieving that data into main memory, and then sleeps until the next round. In each τ-round, the network thread wakes up, determines the data that are to be transmitted on each channel in that round, transmits that data from the main memory buffer cache, and goes to sleep. Coarse-grained locks on the NT and DT threads (as opposed to finer-grained locks on the in-memory data blocks) are used to ensure that threads for a given video do not access the cache concurrently.

The separation of disk retrieval and network transmission activities is motivated by the very different nature of the disk and network subsystems. To prevent starvation due to long or variable disk access times, data is prefetched from disk and temporarily staged in main memory. To reduce the impact of disk overheads, the DT issues asynchronous read requests for large chunks of data at a time. Therefore the disk round, δ, is relatively large, and is currently set to 1 sec. Note that δ is a lower bound on the startup

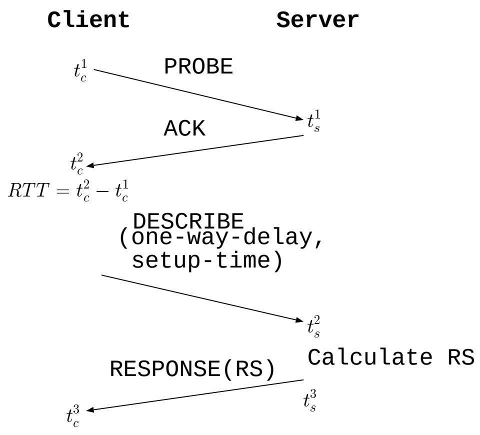


Fig. 3. Client-server signaling

delay a client must wait before receiving a new stream. For the network, it is desirable to avoid injecting bursts of traffic. The network round length, τ, is thus typically much smaller than δ to allow the NT to transmit data more "smoothly" into the network. Because Linux is a non-real-time operating system, it is possible that a sleeping thread is executed significantly later than its scheduled invocation time. Thus, when a thread completes its activities for a round, it checks to if its start time for the next round has passed, and if so immediately starts its assigned activity for the next round rather than going to sleep. In our experimental results, we'll see that although the SDE runs on top of Linux, which offers no real-time scheduling support, the SDE seldom suffers from timing irregularities that result in missed data transmission deadlines.

B. Client Architecture

The client consists of the client control engine (CCE) and data engine (CDE). The CCE obtains user requests using a GUI interface and communicates them to the server in RTSP messages. The client data engine receives data from multiple video segments according to the reception schedule, and reorders received out-of-playback-order data, thus presenting the abstraction of a logically sequential stream to the decoder software. A clean separation of functionality between the end-client (which is responsible for signaling and receipt of data) and the video player (which is responsible for decoding and display) allows a great deal of flexibility, and enables our client software to inter-operate with several widely used players, including mpeg2decode [29], Real and Windows media players.

C. Signaling Protocol

Communication between the client and the server is based on the Real Time Streaming Protocol (RTSP) [25] and occurs over a separate TCP connection. Figure 3 illustrates the signaling steps involved in initiating video playback:

1. At time t_c^1, the client sends a PROBE message to the server. The server receives the PROBE message at time t_s^1 and immediately returns an ACK message. The ACK message reaches the client at time t_c^2.
2. The client uses t_c^1 and t_c^2 to compute the round trip time (RTT) and to estimate the server-to-client one way delay. Exact estimation of this one-way-delay is difficult, because the server-to-client and client-to-server routes may differ; the routes followed by multicast and unicast sessions may differ as well.
3. The client composes and send an RTSP DESCRIBE message, piggybacking additional information (using the SDP description language), including its estimates of the one-way-delay and initial setup time in the DESCRIBE message. The setup time includes time for creating sockets and setting initial parameters; its value is obtained from off-line profiling of our client code.
4. The server receives the RTSP DESCRIBE message at time t_s^2. It calculates the reception schedule for this client and piggybacks it (in SDP format) in an RTSP RESPONSE message to the client.

IV. Experimental Configuration and Performance Metrics

In this section, we discuss our experimental configuration in more detail, describing the videos used in our testbed, the client/server machines and the network connecting them, and the periodic broadcast and patching algorithms that are used in transmitting data. We also define the performance metrics of interest.

The videos used in the experiments are listed in Table I. Each video is RTP packetized with additional headers added as specified in [24]. In all experiments, the server transmits each stream at the playback rate on separate channels. That is, if a video is to be played at 30 frames per second, the server transmits the video at 30 frames per second. Due to lack of space, we do not report experiments performed to tune the values of δ and τ. The experiments show that improper setting of δ and τ lowers the server performance and that settings of $\delta = 1$ sec and $\tau = 33$ msec lead to good server performance. We use these values throughout the evaluation reported here.

For the experimental measurements described in this paper, we use three separate 400 MHz machines, each with a Pentium II processor, 400MB RAM, and running a Linux OS. Each machine is connected to a switch via 100Mbit Ethernet. Each of these three machines serves a specific role: server, a workload generator, and a client.

For experiments involving multiple videos, we use multiple

Video	Format	Length(min)	Frame rate	Bandwidth(Mbps)	File size(MB)	# of RTP pkts
Blade1	MPEG-1	12	30	1.99	180.1	155146
Blade2	MPEG-1	15	30	3	337	296706
Demo	MPEG-2	2.7	30	2	40.6	35138

TABLE I
SAMPLE VIDEOS FOR THE EXPERIMENTS.

copies of the same video (chosen from Table I), with each copy placed in a separate file on disk. The server and the underlying operating system treat each copy as a *distinct video*, and the server transmits each video using a selected broadcast scheme. This approach allows us to explore the system overheads under a homogeneous video population.

The *workload generator* is a separate machine that generates a background load of client requests in a Poisson manner, choosing one of the multiple videos being transmitted from the server with equal probability. The server sends the requested video to the workload generator. The latter does not play out the video data from the server. Instead, it logs the timing information for the request to be serviced. Once the background load reaches a steady state, we use a client, running on the third machine, to request the full stream and monitor the statistics on the received video data.

As noted in Section II, we implemented the representative algorithms described below from the families of periodic broadcast and patching algorithms. In both cases, we assume that the client has enough buffer to store the length of the video.

- **Periodic Broadcast:** For periodic broadcast, we use the GDB segmentation scheme [2]. Throughout the experimental section, we use l-*GDB* to indicate a GDB scheme where the initial segment is l seconds long. The interior segments are of size $2^{i-1}l$ where $1 < i < \lfloor \log_2 L \rfloor$. The length of the last segment is denoted as $L - \sum_{i=1}^{\lfloor \log_2 L \rfloor} 2^{i-1}l$. It should be noted that the length of the first segment determines the maximum client startup delay under ideal system and network conditions. A smaller value of l reduces this delay, but may increase the number of segments and hence the transmission bandwidth requirements. For the results reported, we use three values of l: 3 seconds, 10 seconds, and 30 seconds. The segment lengths of the resulting segmentation schemes for the 900 sec video *Blade2* are reported in Table II. In each case the actual length of the last segment is less than the length specified by GDB (shown in brackets) for that segment, due to the finite video length. Note that segment i will be transmitted once every $2^{i-1}l$ seconds. For example in 30-GDB the first 30 seconds of the video are sent out every 30 seconds, the next 60 seconds of the video are sent out every 60 seconds and so on until the last 450.9 seconds of the video, which are sent out every 480 seconds.
- **Patching.** For patching, we consider the threshold-based Controlled Multicast scheme proposed in [22]. The first request for the video is served with the initiation of a complete transmission using multicast. Subsequent requests that arrive within a threshold T time units of the last initiated multicast transmission of the video will share that stream and obtain only a prefix of the video from the server using unicast. A request arriving after T time units have elapsed is served by initiating a new complete transmission for the video . When the client arrival rate for a video is Poisson with parameter λ and the length of the video is L seconds, the threshold is chosen to be $(\sqrt{2L\lambda+1}-1)/\lambda$ seconds to minimize the average transmission required to serve a client [22].

Scheme	# of segments	Segment Lengths(sec)
3-GDB	9	$3, 6, 12, 24, 48, 96, 192, 384, 134.5(768)$
10-GDB	7	$10, 20, 40, 80, 160, 320, 270.9(640)$
30-GDB	5	$30, 60, 120, 240, 450.9(480)$

TABLE II
ATTRIBUTES FOR 3 DIFFERENT GDB SEGMENTATIONS FOR THE 3 MBPS, 15 MIN MPEG-1 *Blade2* USING FIRST SEGMENT LENGTHS OF 3 AND 10 SEC RESPECTIVELY.

In our evaluations in the following section, we will focus on a number of different performance measures. On the server side, we consider the following metrics:

- **System Read Load (SRL).** This is the volume of video data requested per unit time by the server from the underlying operating system. A read request is initiated only if a required data block is not present in the application-level cache. SRL therefore presents a measure of the workload associated with the data path that is imposed on the underlying system by the video server. The system will satisfy the request from the kernel buffer cache if possible, and otherwise fetch the block from disk.
- **Server Network Throughput (SNT).** The SNT is the volume of video data transmitted per unit time by the application, and measures the load imposed on the network protocol stack, network interface card and the outgoing network connection. This is equal to the SRL in the absence of any application-level buffered video data.
- **Deadline Conformance Percentage (DCP).** Given a transmission schedule, the DCP is the percentage of frames that the server was able to transmit to the network by their scheduled deadlines created by the server control engine.

On the client side, we consider the following performance metrics:

- **Client Frame Interarrival Time (CFIT).** Suppose r_i is the

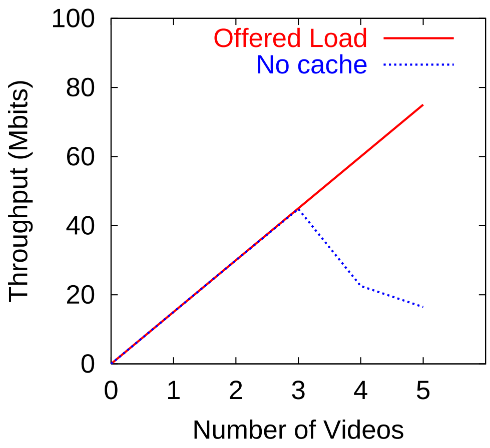


Fig. 4. System throughput under Periodic Broadcast

time that the last packet of frame i reaches the client. The difference between r_{i+1} and r_i is the client frame interarrival time. For a smooth transmission, the frame interarrival time should be constant. The variability of CFITs reflects the delay jitter caused by both the server and the network.

- **Reception Schedule Latency (RSL).** The Reception Schedule Latency is the time from when the client requests the video at t_c^2 to when it receives the reception schedule at t_c^3 (see Figure 3).

V. Performance Measurement and Evaluation

This section presents our experimental measurements and our analysis of these results. We first examine the server's need for application-level caching and explore the effects of different caching policies on performance. Next, we present benchmarking results for the signaling and data throughput in the absence of disk and network constraints. Performance results regarding the end-to-end (server-client) data path are then reported. Finally, we discuss how our threading structure provides isolation of server control and data engines (SCE and SDE), and how naive scheduling of videos using periodic broadcast schemes can lead to bursty traffic.

A. Caching Implications for Periodic Broadcast and Patching

We begin our study by considering the load imposed on the underlying system (disk subsystem and OS level-caching) by periodic broadcast and patching. We note that the underlying disk/file access and caching policies are those implemented in the standard Linux release and thus are not optimized for video access. From an architectural standpoint, our application-level cache sits above these standard OS components and can be thought of as tailoring such standard services for video access.

Figure 4 plots the offered load (data rate required to transmit the requested number of videos) and the achieved system throughput (labeled "no cache"), as a function of the number of videos the server attempts to transmit using 30-GDB segmentation. Distinct file copies of the *Blade2* video were used. We find that the system throughput with no application-level caching matches the offered load up to 3 videos, suggesting that the underlying operating system is able to keep up with imposed load. As the number of videos (and therefore the offered load) increases further, the achieved throughput without application-level caching decreases, indicating that the underlying operating system is in an overloaded regime. We will see shortly that our application-level cache can delay the onset of such overload behavior.

Today's servers typically possess significant amounts of high speed memory. We next investigate the use of *application-level* caching and *application-specific* caching policies for reducing the demand on the underlying server operating system and its disks. While existing work has considered video caching [30–32], none of them examine the effects of caching for periodic broadcast and patching. Our application locks a tunable amount of main memory for application-level caching for each video being transmitted. A per-video caching policy is used, and the server makes read requests to the underlying systems only if a block is not present in the application-level cache. We consider Least Recently Used (LRU) cache replacement as a baseline policy. LRU is widely quoted in literature and many conventional operating systems implement this policy in their underlying kernel buffer caches. We also explore a Least Frequently Used (LFU) cache replacement policy :

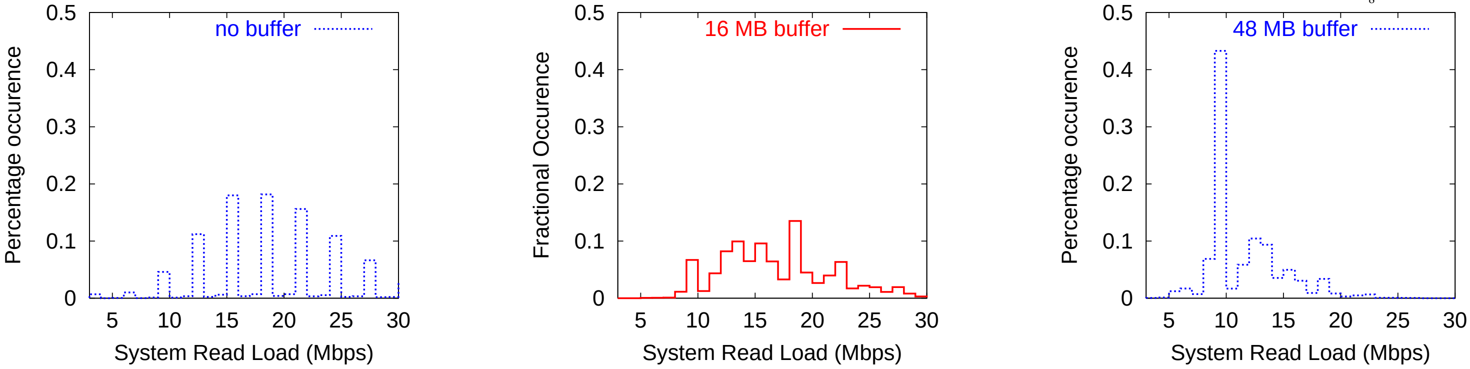
Definition. *Among the set of cached blocks, LFU replaces the block that contains data that has been least frequently used since the start of the video's playout. If there are multiple blocks with the same frequency, remove the block that is furthest from the beginning of the video.*

LFU caching possesses the following interesting property for periodic broadcast and patching (the proof of which is in Appendix A):

Theorem 1: LFU is optimal for threshold-based controlled multicast patching, under a Possion arrival process, and periodic broadcast, with any arrival process, in the sense that it minimizes the average server read load into the underlying operating system.

A.1 Periodic Broadcast

Let us now explore the impact of application-level caching on performance. We first consider two GDB segmentation schemes: 3-GDB and 10-GDB. Figure 6 plots the read load for a single video as a function of the application-level buffer cache size available for that video. We consider both actual measurements from our testbed and analytical computations of LRU and LFU performance for the same cache size. The small deviations between the



(a) no cache

(b) 16 MB cache

(c) 48 MB cache

Fig. 5. Distribution of server read loads for patching for a range of per video cache sizes

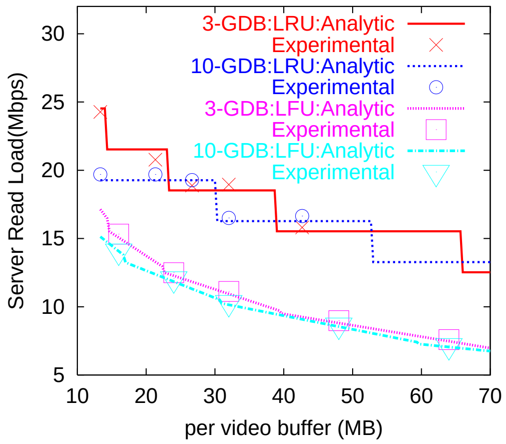


Fig. 6. Caching effects on Periodic Broadcast: plots the experimental and analytic values of the read overhead under LRU and LFU replacement policies.

analytic and experimental values are due to the large application-level memory blocks (100 KB) used for these experiments. This graph demonstrates the important role of caching in reducing the server read load (SRL). As expected the SRL is a monotonically decreasing function of increasing buffer size. In addition, increases in buffer size produce diminishing returns in SRL.

The SRL for periodic broadcast exhibits interesting characteristics under LRU caching. In order for caching gains to be realized, the buffer must be large enough for an entire segment to be cached. If the allocated buffer is less than the length of a segment, LRU will result in the replacement of blocks in increasing order of the nearest time in the future that a block is required next - effectively, a block will be requested from the underlying system each time it has to be transmitted. This explains the step-like behavior. A step change corresponds to a point where LRU has sufficient buffer to cache an additional segment. Hence the steps are 3 Mbps (equal to the bandwidth for any segment) in height. This is followed by a horizontal portion where the additional buffer is not sufficient to fully cache the next segment.

We next consider the impact of an increase in the length of the first segment of a periodic broadcast scheme on the SRL. By increasing the length of the first segment, one can trade off an increase in client playback startup latency for a decrease in server network throughput (SNT). For instance, 10-GDB has a longer first segment but a smaller SNT than 3-GDB. When SRL is the performance metric of interest, however, we find that a decreased first segment length can sometimes result in a lower SRL (both of which are desirable!) than segmentation with a longer first segment. For example, with 27 MB of buffer, under application-level LRU caching, 10-GDB induces a read load of 19.27 Mbps, while 3-GDB results in a SRL of 18.88 Mbps (Figure 6). On the other hand, for other buffer sizes, 10-GDB results in a *larger* SRL than 3-GDB. These results suggest that the use of additional caching (rather than adjusting the initial segment length) is the most "surefire" approach for reducing the read load.

Figure 6 also shows that LFU results in a significant reduction in the SRL over LRU, across the range of buffer sizes. For example, for 10-GDB, with a 32 MB buffer, the SRL drops from 16.27 Mbps under LRU to 10.14 Mbps under LFU, a reduction of 38%. Also, under LFU, even a partially cached segment contributes to caching gains. This explains why the SRL for LFU decreases more smoothly than for LRU.

The above study illustrates that application-level caching can be very effective in reducing the read load imposed on the system. For example, with just 8 MB of per video cache, and in the presence of LFU caching for each video, we find that the offered load and achieved throughput remain the same, as the number of videos increases from one to five in the setup for Figure 4. In the remainder of the paper we shall therefore report results using LFU caching, implemented using techniques from [33].

A.2 Patching

We next explore the impact of application-level caching on patching. We consider an aggregate arrival rate of .0833 clients per second, requesting one of 3 distinct file copies of *Blade2* with

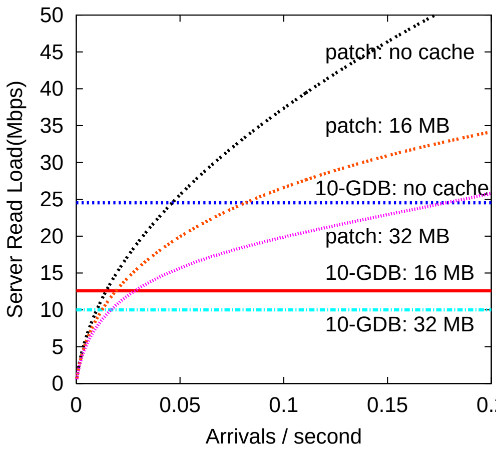


Fig. 7. Server Read Load for patching and 10-GDB for video *Blade2* with LFU caching

equal probability. Figure 5 shows the measured distribution of SRL for one instance of the *Blade2* video, over a 5 hour run with 3 different buffer sizes using LFU caching.

With no caching, the expected SRL is 18.7 Mbps. The resultant SRL reduces to 17 Mbps in the presence of a 16 MB per video cache and to 11 Mbps for a 48 MB cache. The graphs illustrate that for a given buffer size, the instantaneous read load can be much higher than the mean, and that LFU caching with even a modest amount of buffer can substantially reduce both the expected SRL, as well as lower the peak instantaneous SRL.

A.3 Choosing between Periodic Broadcast and Patching

It has been shown in [34] that there exists a request arrival rate above which periodic broadcast results in lower network bandwidth usage and below which patching results in lower network bandwidth usage. This result also holds when SRL is the performance metric of interest, in the absence of application-level caching. Here we examine the effect that LFU caching has on this crossover point, in the context of the SRL. Figure 7 plots the *analytically* computed values of the SRL for patching and 10-GDB periodic broadcast across a range of request arrival rates, for different per-video LFU cache sizes. When no caching is used, the crossover point between patching and periodic broadcast occurs at an arrival rate of 0.046 requests per second (2.8 per minute). At this point the SRL imposed by both schemes is equal to 25 Mbps. As the buffer cache increases, we find that the SRL decreases as expected for both schemes, but 10-GDB sees lower SRL. With increasing buffer size, the crossover point for the two schemes shifts to lower arrival rates. These studies demonstrate that the caching scheme and cache buffer size both impact the the crossover point and need to be factored in its computation.

B. Component Benchmarks

We now turn our attention to the performance of individual components in the server. In particular, we consider the time needed to complete a signaling operation (serve a client's request to view a video) at the server, and the times needed for a network thread and a disk thread to execute during a τ-round and δ-round, respectively (see section 3.1).

In order to make these measurements, the server is configured as follows. As before, there are 5 scheduler threads that handle client requests to view a video. The same algorithm as with periodic broadcast is used to schedule the scheduler threads, the network threads, and the disk threads. In order to remove the (unknown) effects of an unknown amount of OS-internal disk block caching, all video data is pre-loaded and locked into our application-level video cache. In configuration 1 (see Table III), three copies of the *demo* video (see Table I) are placed into memory, with each video divided into eight equal-length segments. Each video segment is then transmitted over a separate multicast group, resulting in a total of 24 segments being transmitted, each over its own multicast channel. In configuration 2 (see Table III), one copy of the *demo* video is divided into 24 segments and locked into memory. Again, each video segment in then transmitted over a separate multicast group. In both configurations, the server is transmitting a total of 48Mbits per second.

B.1 Signaling Costs

Let us first consider the average time between the receipt of DESCRIBE message by a server scheduler thread until its generation and transmission of a RESPONSE message to the client (see Figure 3 for an overview of signaling activity). This is roughly the delay at the server between the receipt of a client signaling request and the server's reply to this request. It includes the time needed to compute the client's reception schedule, update the disk fetch and network transmission schedules, and allocate and initialize a disk and network thread for the video (if needed). In configuration I, the workload generator varied client request rates between 1 and 1670 clients per minute. We observed that the time needed to complete the signaling remained nearly constant at 8 ms with negligible queuing delay (where "queueing delay" refers to the time between the receipt of the DESCRIBE message from the client and the initiation of handling of that message by a scheduler thread. Beyond 1670 clients per minute, the server was no longer able to accept client TCP signaling connections within TCP's timeout period, and the connections were consequently refused by the OS.

We next measure the average signaling delay and queueing delay under configuration II. Figure 8 shows these results. As the client request rate increases, the average signaling delay needed to process the request increases. We also see that the average queue-

Configuration	# Channels	# Videos	Bandwidth per Video	NT completion time	DT completion time
I	8	3	16 Mbits	1.60 ms/30 ms	6.16 ms/1 sec
II	24	1	48 Mbits	5.08 ms/30 ms	8.39 ms/1 sec

TABLE III
TIMING MEASUREMENTS IN THE DATA ENGINE

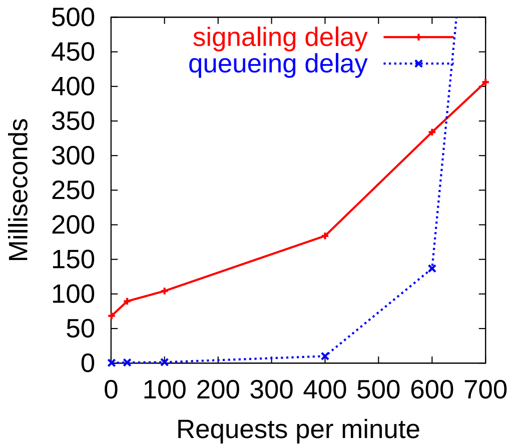


Fig. 8. Signaling delay and Queueing delay for Clients in Configuration II

ing delay remains negligible until 600 requests per minute. Beyond this rate, the server is no longer able to respond to requests and connection timeouts occur. It is interesting to note that the signaling delay (the time needed to process the client request *once the client request has begun being processed*) increases as the client request rate increases. We conjecture that this results from an increased chance of the scheduler thread being interrupted during processing.

B.2 Cost of Delivering Data

We now discuss the amount of time needed by the disk thread (DT) and the network thread (NT). The disk thread runs once per second. It performs three actions: determine what data the NT will send out in the next second, check the cache for the data needed, and request any missing data from disk. Since all of the data is in memory, our benchmarks do not include any time needed to make requests to the disk. The network thread executes 30 times a second. During each execution it determines what data needs to be sent on the network and sends the data into the network. For the two configurations, the amount of time needed by the DT and NT is shown in Table III. In general, we find that the amount of time needed for each of these threads depends on the amount of data being sent out. Perhaps most interesting is that the deadline conformance percentage (DCP) for these experiments was over 99%, an observation that we will return to in Section V-D.

C. End-End performance

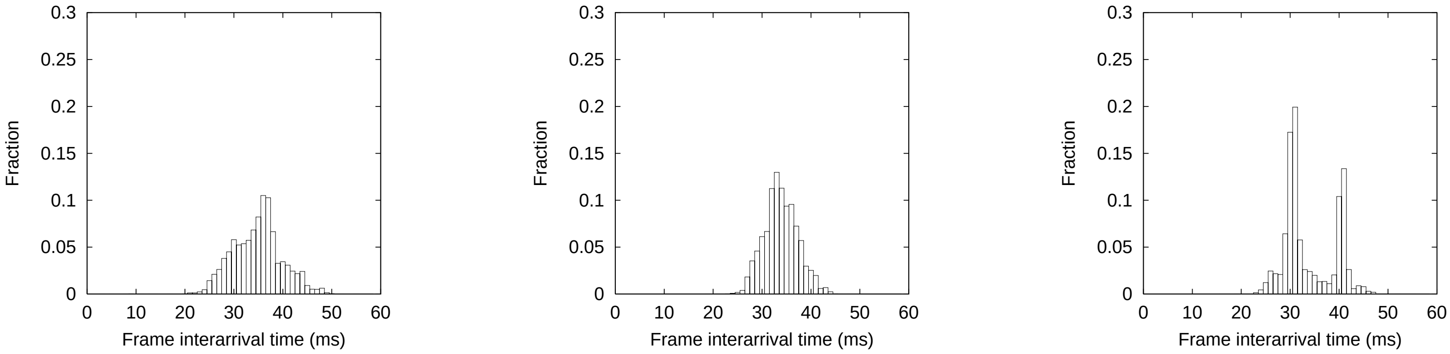
We next evaluate the server and client performance under periodic broadcast and patching schemes. We use three copies of *Blade2*. Each copy is treated as a distinct video by all components of the videos server and the underlying operating system. The total size of the videos (1011 MB) being served at the server is significantly larger than the size of the server's main memory (400 MB) making it impossible to cache the videos completely in main memory, and assuring that the disk subsystem will be exercised in the experiment. The workload generator creates requests for each of the three videos with equal probability. After the server load reaches a steady state, our measurement client requests one video every 40 minutes. Our measurements focus on the server and client performance of the video requested by the measurement client. Throughout this set of experiments, each video is allocated a 16MB application-level cache. Parameters specific to periodic broadcast and patching are described separately in the following corresponding subsections.

C.1 Periodic Broadcast

Our aim in this experiment is to evaluate the end-end performance of periodic broadcast. We use three GDB schemes at the server: 3-GDB, 10-GDB and 30-GDB. As described in Table II, the number of segments corresponding to three copies of *Blade2* for the three GDB schemes are 27, 21 and 15 respectively. In any GDB scheme, the amount of data sent out from the server into the network (SNT) does not depend on the client arrival rate. However a high arrival rate incurs more signaling and processing overheads at the server. The workload generator sends out requests at a low rate, 1 per minute, and a relatively high rate, 600 per minute.

We first investigate the server performance. Since the server only reports 15% of its processor being utilized (almost all of it in system time), deadline conformance percentage (DCP) proves to be a good indicator of stress observed at the server. We find that the DCP at the server under the various GDB schemes and arrival rate is always over 99% for the different scenarios. More streams and higher arrival rate do not necessarily lead noticeably poorer DCP.

We next examine the performance observed at the client. We find that the client does not any have problem receiving multi-



(a) 3-GDB (b) 10-GDB (c) 30-GDB

Fig. 9. Client Frame Interarrival Time (CFIT) histogram under 3-GDB, 10-GDB, and 30-GDB at 600 requests per minute.

ple streams simultaneously. Across these experiments, reception schedule latency (RSL) ranges from 15 to 40 milliseconds. We use the client frame interarrival time (CFIT) to measure the quality of transmission observed by the client. Figure 9 plots the histogram of CFIT under 3-GDB, 10-GDB and 30-GDB respectively when the request rate is 600 requests per minute. Each plot shows the result of one run. Other runs under the same configuration display similar behavior. For each GDB scheme examined, the CFITs under the arrival rates of 1 request per minute are very similar to those plotted in Figure 9, confirming that even at the high arrival of 600 requests per minute does not cause performance degradation at the server.

In Figure 9, the histogram of CFITs under 3-GDB and 10-GDB are both unimodal in the range of 20 to 50 milliseconds with the peak at 38 and 33 milliseconds respectively. The variance and coefficient of variation for 3-GDB are 28.63 and 0.16, while the variance and coefficient of variation for 10-GDB are 12.01 and 0.10. The CFIT from 3-GDB is higher variation than CFIT from 10-GDB. The histogram of CFITs under 30-GDB is bimodal, with two peaks at 31 and 41 milliseconds respectively. These can be explained as follows. For a scheduled transmission of one frame per 33 millisecond, the frame interarrival times at the client are expected to be 40 milliseconds or 30 millisecond (if no jitter is generated over the server and the network) due to the 10 millisecond granularity of scheduling in Linux. If the server can keep up with the schedule, the percentage of 40-millisecond and 30-millisecond interarrival times are expected to be 67% and 33% respectively. The CFIT in 30-GDB is close to this expectation while the CFIT in 10-GDB and 30-GDB deviate from this expectation. This is due to heavier traffic over the network in the later two schemes. The total network loads in 3-GDB (73.59Mbps) and 10-GDB (57.81 Mbps) are 1.66 and 1.30 times of the network load in 30-GDB (44.43Mbps) respectively. Similar behavior is also observed in the end-end patching experiment described next. Finally, our experiments show that if the client starts playback around 80 milliseconds after receiving the first frame, it is able to receive all the frames before the playback time. This suffices that a very *small* amount of waiting time after receiving the first frame can guarantee continuous playback in periodic broadcast in our testbed.

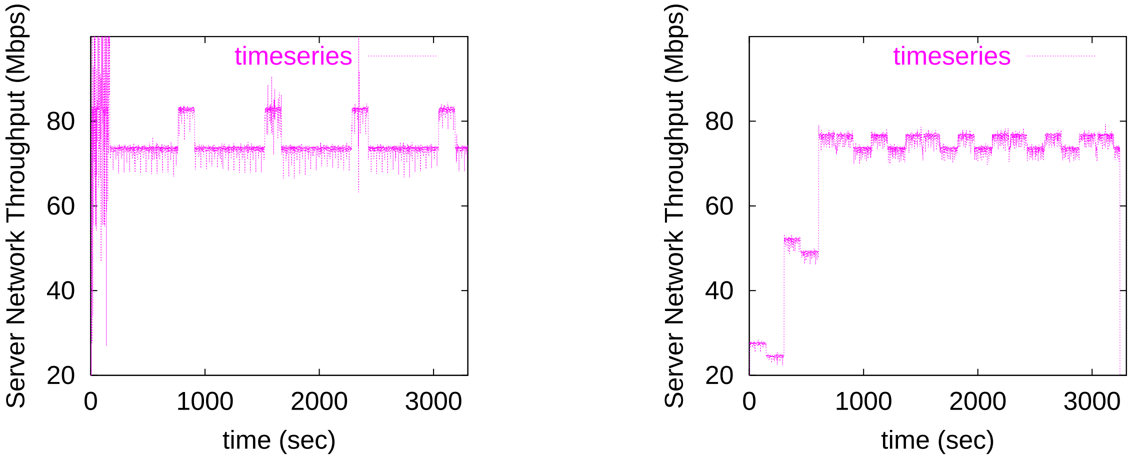
C.2 Patching

Our aim in this experiment is to explore the end-end performance for threshold-based patching. The aggregate request rates of the workload generator are chosen to be 1 and 5 requests per minute (for higher arrival rates the 100Mbit network link becomes a bottleneck).

The average network load for arrival rate of 5 per minute is 55.27Mbps, which is 2.65 times of the network load for arrival rate of 1 per minute (20.85Mbps). As with periodic broadcast, we find that the DCP remains steady at 99.9% for the two arrival rates, which demonstrates that the server does not have any difficulty in handling this range of request rates. On the client side, reception schedule latency is around 15 to 20 milliseconds. The CFIT histograms (not shown here due to space constraints) are bimodal, similar to the 30-GDB result in Figure 9 Another important observation is: if the client starts playback around 1.5 seconds after sending the request, it is able to receive all the frames before the playback time. This 1.5 seconds includes the latency for the first frame of the video to come in and some delay after that to accommodate the packets that arrive later than the supposed playback time. We conclude that the network becomes the bottleneck in 100Mbit switched Ethernet LAN settings for patching since the server is able to send near to the bandwidth of the link without experiencing poor quality video at the client.

D. Control and Data Engine Interaction

In this section we examine some of the implications of having the control and data engine run as separate entities. From Section V-C we see that for a deadline conformance percentage $> 99\%$, the clients receive good service. Furthermore, we observe in Section V-B.2 that all DCP values remained $> 99\%$ for



(a) 3-GDB synchronized

(b) 3-GDB non- synchronized

Fig. 10. Synchronization between transmission schedules could lead to bursty behavior

all levels client request rates. To understand why this occurs we need to look at what is happening for each active thread in the server. The disk thread needs very little computation time each second and has a negligible effect on the processor. Using the measurements observed for configuration II in Section V-B.2 (see Table III), we can calculate the amount of time that the network thread needs each second. We find that the NT needed $5.08ms$ per round and there are 30 rounds in a second. Therefore, the NT needed $(30/sec) * (5.08ms) = 152.4ms/sec$ in order to perform its task. In general the amount of time that the NT will need each second is 3 ms per Mbit of the data transmitted. The only other active entities are the 5 scheduler threads. This means there are total of six active threads in the server, excluding the DT. Assuming a simple processor sharing model we see that each thread will receive $166ms$ of processor time each second. This should allow the NT thread to essentially run in isolation from the scheduler threads. To test this hypothesis, we re-ran experiment II with 20 scheduler threads and used a high request rate. While the server was able to service more clients, the resulting DCP was *only* 4.26%. This indicates that without some form of guarantee for processor time, the number of active threads needs to be chosen carefully.

E. Scheduling Among Videos

When supporting 3 videos (copies *Blade2*) using periodic broadcast, we observe that the server generated bursty network traffic (see Figure 10(a)). In particular, bursts of traffic were found to occurred every 768 seconds. This can be explained as follows. The l-GDB algorithm periodically repeats each segment of the video at a certain rate. Due to the fact that the last segment is smaller than the repetition rate, the server will transmit the last segment for a short amount of time; the channel remains idle until the next repetition. If several videos are transmitted with this last channel synchronized, the server will generate a burst in network traffic. In order to avoid the bursty behavior, we examined what happened when the schedules were started separately at an interval of three minutes apart. Figure 10(b), shows three 3-GDB broadcasts, staggered to prevent them from synchronizing the retrieval of the last segment, and find that the sustained bursts disappear. Scheduling to avoid synchronization removes the necessity of provisioning high peak server network throughput. This example illustrates the benefit of using techniques for smoothing out the offered load, especially for high loads.

VI. Conclusions

The high transmission bandwidth requirements of streaming video, coupled with the best-effort service provided by today's IP networks makes it a challenging problem to provision network resources for delivering such media to remote clients. In this paper, we presented the design and implementation of an experimental streaming media testbed for investigating scalable streaming solutions like periodic broadcast and patching. The testbed consists of a distributed video server and client software running on top of off-the-shelf PCs executing commercial Linux and Windows operating systems.

Experimental evaluations indicate that the server is able to support the real-time, bandwidth intensive data delivery requirements imposed by schemes like periodic broadcast and patching, vindicating many of the key design principles incorporated in the architecture. Under periodic broadcast, our server can easily process a client request rate of 600 requests per minute (returning periodic broadcast schedule information to each client), while at the same time streaming video segments over multiple multicast groups and missing few data transmission deadlines. Under patching, our server again comes close to fully loading a 100 Mbps network connection with patched-in clients, while missing few data transmission deadlines. Our measurements also show that in a loaded LAN environment, an initial client startup delay of less than 1.5 sec. is sufficient to handle startup signaling and absorb data jitter induced by the non real time operating systems at either the client or the server, as well as any network jitter.

Our evaluations show that dramatic reductions in the bandwidth demand on the underlying server operating system can be gained via application-level data caching even when using a simple Least Recently Used (LRU) cache replacement policy, and that substantial additional performance gains can be realized under a Least Frequently Used (LFU) Replacement Policy. Furthermore, we have shown that LFU is an optimal cache replacement policy for for periodic broadcast and patching schemes.

Our testbed allows us to collect benchmarking measurements for individual components in the server. Using this data we can determine bottlenecks and search farther for lessons on how the individual components interact to effect the quality of service that the end client sees.

Building on our experience and the insights we have gained, we are developing a network proxy server testbed for exploring proxy-based techniques for deliver high quality streaming video over best-effort IP networks.

VII. Acknowledgements

We would like to recognize Abhinav Garg for his contribution to the initial design of the experimental testbed. We also aknowledge Ellen (Xiaolan) Zhang for her continuing work in developing client middleware support for the testbed.

References

[1] K. Hua and S. Sheu, "Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems," in *Proc. ACM SIGCOMM*, September 1997.

[2] L. Gao, J. Kurose, and D. Towsley, "Efficient schemes for broadcasting popular videos," in *Proc. Inter. Workshop on Network and Operating System Support for Digital Audio and Video*, July 1998.

[3] D. Eager and M. Vernon, "Dynamic skyscraper broadcasts for video-on-demand," in *Proc. 4^{th} Inter. Workshop on Multimedia Information Systems*, September 1998.

[4] D. Eager, M. Ferris, and M. Vernon, "Optimized regional caching for on-demand data delivery," in *Proc. Multimedia Computing and Networking (MMCN '99)*, January 1999.

[5] S. Sen, L. Gao, J. Rexford, and D. Towsley, "Optimal patching schemes for efficient multimedia streaming," Tech. Rep. 99-22, Department of Computer Science, University of Massachusetts Amherst, 1999.

[6] S. Carter and D. Long, "Improving video-on-demand server efficiency through stream tapping," in *Proc. International Conference on Computer Communications and Networks*, 1997.

[7] K. Hua, Y. Cai, and S. Sheu, "Patching: A multicast technique for true video-on-demand services," in *Proc. ACM Multimedia*, September 1998.

[8] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole, "Demonstrating the effect of software feedback on a distributed real-time mpeg video audio player," in *Proc. ACM Multimedia*, November 1995.

[9] K. Almeroth and M. Ammar, "An alternative paradigm for scalable on-demand applications: Evaluating and deploying the interactive multimedia jukebox," in *IEEE Transactions on Knowledge and Data Engineering Special Issue on Web Technologies*, July/August 1999.

[10] S.-F. Chang, A. Eleftheriadis, and D. Anastassiou, "Developement of columbia's video on demand testbed," *Image Communication Journal:Special Issue on Video on Demand and Inerative TV*, 1996.

[11] D. Anderson, Y. Osawa, and R. Govindan, "A file system for continuous media," *ACM Trans. Computer Systems*, pp. 311–337, November 1992.

[12] M. Buddhikot, G. Parulkar, and J. Cox, "Design of a large scale multimedia storage server," *Journal of Computer Networks and ISDN Systems*, pp. 504—524, Dec 1994.

[13] R. Haskin, "Tiger shark–a scalable file system for multimedia," *IBM Journal of Research and Development*, vol. 42, pp. 185–197, March 1998.

[14] W. J. Bolosky, R. P. Fitzgerald, and J. R. Douceur, "Distributed schedule management in the tiger video fileserver," in *Proceedings of the sixteenth ACM symposium on Operating Systems Principles (SOSP'97), Saint-Malo, France*, pp. 212–223, December 1997.

[15] C. Martin, P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz, "The Fellini multimedia storage server," *Multimedia Information Storage and Management , Editor S. M. Chung, Kluwer Academic Publishers*, 1996.

[16] F. A. Tobagi, J. Pang, R. Baird, and M. Gang, "Streaming raid – a disk array management system for video files," in *Proceedings of ACM Multimedia '93, Anaheim, CA*, pp. 393–400, 1993.

[17] M. Vernick, C. Venkatramini, and T. Chiueh, "Adventures in building the stony brook video server," in *Proceedings of ACM Multimedia'96*, 1996.

[18] M. Buddhikot, X. Chen, D. Wu, and G. Parulkar, "Enhancements to 4.4 bsd unix for efficient networked multimedia in project mars," in *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS'98), Austin, TX*, pp. 326–337, July 1998.

[19] C. Aggarwal, J. Wolf, and P. Yu, "On optimal batching policies for video-on-demand storage servers," in *Proc. IEEE International Conference on Multimedia Computing and Systems*, June 1996.

[20] L. Golubchik, J. Lui, and R. Muntz, "Adaptive piggybacking: A novel technique for data sharing in video-on-demand storage servers," *ACM Multimedia Systems Journal*, vol. 4, no. 3, 1996.

[21] J.-F. Paris, S. Carter, and D. Long, "A low bandwidth broadcasting protocol for video on demand," in *Proc. 7^{th} Inter. Conference on Computer Communications and Networks*, October 1998.

[22] L. Gao and D. Towsley, "Supplying instantaneous video-on-demand services using controlled multicast," in *Proc. IEEE International Conference on Multimedia Computing and Systems*, 1999.

[23] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "Rtp: A transport protocol for real-time applications," January 1996.

[24] D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar, "Rtp payload format for mpeg1/mpeg2 video," January 1998.

[25] H. Schulzrinne, A. Rao, and R. Lanphier, "Real time streaming protocol (rtsp)," April 1998.

[26] M. Handley and V. Jacobson, "Sdp: Session description protocol," April 1998.

[27] C. Diot, B. Levine, B. Lyles, H. Kassan, and D. Balsiefien, "Deployment issues for the ip multicast service and architecture," *IEEE Network*, January 2000.

[28] J. Turner, "Terabit burst switching," *Journal of High Speed Networks*, 1999.

[29] Public domain MPEG2 encoder/decoder software.`ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/mpeg2/conformance%-bitstreams/video/verifier`.

[30] R. Rejaie, H. Yu, M. Handley, and D. Estrin, "Multimedia proxy caching mechanism for quality adaptive streaming applications in the internet," 1999.

[31] S. Acharya and B. Smith, "Middleman: A video caching proxy server," in *Proc. International Conference on NOSSDAV*, 2000.

[32] J. M. Almeida, D. L. Eager, and M. K. Vernon, "A hybrid caching strategy for streaming media files," in *MMCN*, 2001.

[33] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existance of a spectrum of policies that subsumes the least recently used LRU and least frequently used LFU policies," in *SIGMETRICS*, 1999.

[34] L. Gao, Z. Zhang, and D. Towsley, "Catching and selective catching: Efficient latency reduction techniques for delivering continuous multimedia streams," in *Proc. ACM Multimedia*, 1999.

[35] A. Silberschatz, J. Peterson, and P. Galvin, *Operating System Concepts, 3rd edition*. Addison-Wesley Publishing Company, Inc., 1991.

Appendix

I. Optimality Proofs of LFU

Theorem 1: LFU is optimal for threshold-based controlled multicast patching, under a Possion arrival process, and periodic broadcast, with any arrival process, in the sense that it minimizes the average bandwidth from the underlying operating system.

Proof: We prove this from a well known result in the OS community [35]:

> An optimal page-replacement algorithm that has the lowest page-fault is: Replace the page that will not be used for the longest period of time.

If the access frequency is known beforehand, LFU satisfies the property that the data that will not be used for the longest period of time are replaced first. Hence it is an optimal policy. We next show that the expected access frequency in periodic broadcast and threshold-based patching can be predetermined.

In periodic broadcast, each segment is broadcast periodically irrelevant to the request rate of the clients. Therefore, if the length of segment i is s_i, then the access frequency of the segment from the memory is $1/s_i$. The smaller the segment, the more frequent it is accessed.

In threshold-based patching under a Possion arrival process, let T be the threshold and λ be the arrival rate. We consider the interval between two complete streams. The video block after the threshold is required once. For video block $[x, x+\delta], x < T$, where δ is small enough, this block is required $1 + \lambda(T - x)$ times. The smaller the x, the more frequent it is being required.

Therefore, LFU minimized the miss rate hence the read overload on the system in both periodic broadcast and threshold-based patching.

In effect, the more frequently accessed segments (blocks) are not replaced but kept cached by using LFU. It should be noted that for segmentation schemes such as GDB where the length of the segments are in an increasing order, a prefix of the video ends up cached. In threshold-based patching, since the earlier video blocks have higher access frequency, it also turns out that a prefix of the video is cached.