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 Engineering
lgao@ecs.umass.edu

University of Massachusetts
Amherst, MA 01003

UMass Computer Science Technical Report 2000-56

Abstract

Multimedia streaming applications consume a significant amount of server and network resources due to the high bandwidth and long duration of audio and video clips. *Patching* and *periodic broadcast* schemes use multicast transmission and client buffering in innovative ways to reduce server and network resource use. Current research in this area has focussed on the theoretical aspects of these approaches, rather than on the challenges involved in implementing and deploying such scalable video transmission services.

In this paper, we first describe the design and implementation of a flexible streaming video server and client testbed that can support emerging streaming services such as periodic broadcast and patching. We explore and present solutions to the system and network issues involved in actually implementing these services. Using this testbed, we conduct extensive experimental evaluations, measuring performance both at the server as well as end-end performance at the client, over the local network as well as over VBNS, and present key insights gained from our implementation and experimental evaluations.

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 arise from 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 a number of video applications. These characteristics place significant load on both network and server resources. The scenario is further complicated by the fact that the client population is likely to be both large and heterogeneous, with different clients asynchronously

[1]This research was supported in part by the National Science Foundation under Grants CDA-9502639,ANI-9977635,ANI-9805185, NCR-9508274, ANSIR-9977555, and ANIR-9875513, and a gift from the Intel Corporation
Any opinions, findings, and conclusions or recomendations expressed in this material are those of the authors and so not necessarily reflect the views of the National Science Foundation.

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 [5–9], described in more detail in section 2, are two such approaches that have received considerable recent attention. They 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.

To date, existing research on periodic broadcast and patching has been algorithmic in nature, with performance studied either analytically or through simulation. In either case, simplifying assumptions are necessarily made (e.g., abstracting out control and signaling overhead, operating system issues such as the interaction between disk and CPU scheduling, multicast group join and leave times, and more) in order to evaluate performance.

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 Linux-based, application-level video server and a collection of both Linux- and Windows-based clients; we conduct both LAN and WAN evaluations. While there are a number of existing production (e.g., Apple Darwin server, RealServer, Windows Media Server, Oracle Video Server) and experimental [3, 4] video server efforts, these use either unicast streaming to clients, or a single IP multicast stream; no empirical evaluations of either periodic broadcasting or patching algorithms have been made. The goal of this paper is thus to investigate the underlying system issues that arise when putting such idealized algorithms intro practice.

Our results show that network bandwidth, rather than server resources, is likely to be the bottleneck - 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 can come close to fully loading a 100Mb network connection with patched-in clients, again 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 seconds is sufficient to handle startup signaling and absorb data jitter induced at either the client or the server. Finally, our results show that dramatic performance improvements can be gained via application-level data caching and that further gains can be realized under an optimal data caching 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, identifies the system issues we face in implementing periodic broadcast and patching, and gives a high level overview of the platform architecture. Section III describes the salient features of the server architecture, and Section IV describes the signaling protocol. Our experimental measurements, analysis and evaluation are presented in Section V. Section VII reflects on the important lessons learned and describes ongoing work. Finally, Section VIII concludes the paper.

II. Design Overview

In this section we present background material, identify key design principles of our streaming media testbed, and then present an overview of the server and client architectures.

A. *Streaming multimedia, multicast, periodic broadcast, and patching*

Many Internet multimedia applications have asynchronous clients that may request the same video stream at different times. Making high-volume video services economically viable requires effective techniques that 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 transmission [1, 2, 5–8]. For example, the server could *batch* requests that arrive close together in time [1], 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 [5–9], 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 transmission bandwidth requirements, while still guaranteeing a bounded playback startup latency.

Periodic broadcast schemes [1, 5–8] 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 or stream tapping [6, 9–11], 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 transmit afresh 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 only on-demand, when new clients arrive. For a detailed description of periodic broadcast and patching, the reader is referred to the references cited above.

B. *Design Principles*

The design and implementation of our video server testbed is based on the following principles.

- *Modular, extensible architecture:* Since we envisage using our testbed for evaluating various streaming media techniques, the architecture of our server and end-client is designed in a modular and extensible manner. This facilitates easy addition of new streaming techniques and modifications of existing techniques in the server or client. The addition of a new technique such as patching, for instance, involves writing a new module that "plugs" into the existing server architecture and requires minimal changes to other server code.
- *Separation of control and data functionality:* Both server and client implementations separate control and data functionalities. Since the control and data path impose significantly different demands on the underlying system, such 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.
- *Standards-based architecture:* Our server and client implementations are based on existing streaming media standards such as RTP [14, 15], RTSP [16] and SDP [13]. 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 where available (unicast can used in environments where IP multicast is not yet deployed). 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, join/leave latencies for users and techniques to hide such latencies).
- *Scalability and resilience to network impairments:* A key goal of our server design has been to keep system overhead low. This allows our server to scale to a large number of concurrent users and handle large volumes of streaming media data with timeliness requirements. Our end-clients, on the other hand, are designed to be resilient to packet losses and delays in the network.
- *Use of off-the-shelf components:* Our server and client are designed to run on vanilla operating systems such as Linux and Windows. The limitation of using a generic best-effort operating system platform, however, is that we do not benefit from the numerous special-purpose resource management techniques (e.g., rate-based scheduling) that have been proposed recently.

C. Client and Server Overview

Figure 1 provides a high-level view of our video server testbed, showing the server and one of several clients. We close this section with a high-level overview of the

server and client architecture; subsequent sections then describe the server and signaling components in more detail.

As shown in Fig. 1, our server consists of two main modules, a *Server Control Engine (SCE)* and a *Server Data Engine (SDE)*, embodying our principle of of separating control and data functionalities.

- **Server Control Engine (SCE):** The control engine is primarily responsible for handling (control) interac-

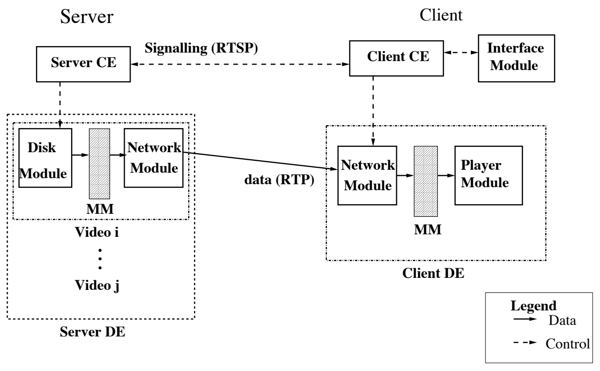


Fig. 1. Architecture of Streaming server and client testbed

tions between the server and end-clients. These interactions are based on the Real Time Streaming Protocol (RTSP) and the Session Description Protocol (SDP). The control engine listens for RTSP client requests on a well-known port. For each such request, it computes: (i) a *transmission* schedule that specifies how each video segment should be retrieved from disk and transmitted to the client over the network, and (ii) a *reception* schedule that specifies the order in which the end-client should receive this data. The transmission schedule is then handed over to the server data engine, which then retrieves and transmits data based on this schedule. The reception schedule is formatted as an SDP message and sent to the client in an RTSP response; the client then uses this schedule to receive data on the specified multicast channels.

- **Server Data Engine (SDE)**: The server data engine uses directives from the control engine to coordinate the retrieval of video data from disk, and transmit data to end-clients using the Real-time Transport Protocol (RTP). Conceptually, the data engine is an efficient data pump that is designed to handle a high volume of data at low overhead (for instance, by storing pre-packetized RTP streams on disk, which lowers packetization overheads at transmission time). Efficiency is an especially important consideration in our design, since the data engine will typically need to simultaneously retrieve and transmit multiple portions (segments) of video.

An end-client interacts with the server via signalling and receives data transmitted by the server; it supports a number of streaming techniques such as sequential unicast, periodic broadcast and patching. Conceptually, our client acts as middleware between the server and the actual video player—it receives data from multiple video segments, possibly in out-of-playback order, and presents the illusion of a logically sequential stream to the player. This clean separation of functionality between the end-client (which is responsible for signalling and receipt of data) and the video player (which is responsible for decoding and display) allows a great deal of flexibility. For instance, it allows our end-client software to interoperate with several widely used players, including mpeg2dec (in public domain [18]), the Real player and the Windows media player.[1]

Like the server, our end-client consists of two key components that separate the data and control functionalities:

[1]We have successfully integrated our end-client software with the mpeg2dec and the Real player; a port of our end-client software with the Windows Media player is currently in progress.

the client control engine ("Client CE" in Figure 1) and the data engine ("Client DE"). The client control engine is responsible for signalling with the server. It obtains user requests using a GUI interface and communicates them to the server using RTSP messages. Our client currently supports user interactions such as play, stop, pause, resume and indexed jump; other interactive operations such as fast forward and rewind are currently under implementation. The client data engine is responsible for receiving RTP packetized data transmitted by the server; the reception schedule sent by the server indicates the timing and amounts of data that will be received from each multicast channel. The received data is then presented to the player software in playback order for decoding and display. We next describe the server architecture and client-server interactions in further detail.

III. Server

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

A. *Server Control Engine (SCE)*

The SCE is responsible for (i) receiving client requests, and exchanging control information with the requesting clients, (ii) determining the server transmission schedule for each video and communicating this to the Server Data Engine, and (c) computing the video reception schedule for each requesting client and communicating this to a client. The SCE and SDE communicate with each other over a TCP connection.

The SCE is implemented as a multithreaded single process system. A single *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. One thread from the pool will wake up and retrieve an incoming request. This scheduler thread is then responsible for (i) subsequent communications with the client, (iii) generating an abstract *transmission schedule* for the server data engine, and (iii) generating and transmitting a *reception schedule* to the client.

The scheduler thread determines whether there is an already-scheduled transmission for that video that can be used to satisfy this client, or whether an additional transmission of the video (or part of it) is needed. In the latter case, the scheduler will be responsible for updating the data engine's *transmission schedule*, which is used by the SDE to determine what parts of the stream are to be sent out, and when. Based on the server transmission scheme, the scheduler thread computes a reception schedule for this client and transmit this schedule back to the client.

The data structure for the transmission schedule 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). Figure 2 illustrates this structure. For each media stream being currently transmitted, there is a data structure named *Media*, containing stream-specific information such as the file location, length, and type of the stream.

The *Media* data structure 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 a video can be transmitted on multiple

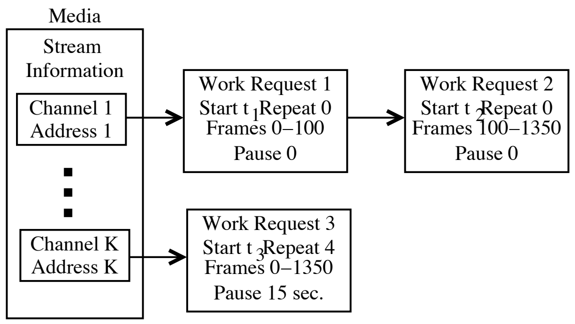


Fig. 2. Data Structure Organization

channels simultaneously, 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.

Each work request element corresponds to a sequence of consecutive frames in the media stream, and contains schedule information used by the SDE to determine when this sequence of frames is to be transmitted. Each work request contains the following items:

- *Beginning* and *end* frames numbers of the sequence.
- *Time* to initiate transmission of the sequence.
- *Repeat Count* : Number of times the current work request should be repeatedly serviced, before moving to the next work request.
- *Pause Time*: Amount of time to pause after a work request finishes before repeating it.

We note that an important advantage of specifying the transmission schedule at the frame level is that this allows 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 connection at some later time t_2. The work request list associated with channel 1 in Fig. 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.

Many scheduling schemes such as periodic broadcast require the repeated transmission of a sequence of frames. Suppose we also wish to transmit the above video on a second connection, once every minute starting at time t_3, for a total of five transmissions. We allocate a new channel (Channel K in Fig. 2) and fill out the appropriate address. We associate a single work request with this channel, with Start Frame=0, End Frame=1350. There is a 15 sec gap after each complete transmission of the video, and so the Pause field is set to 15 sec. The Repeat field

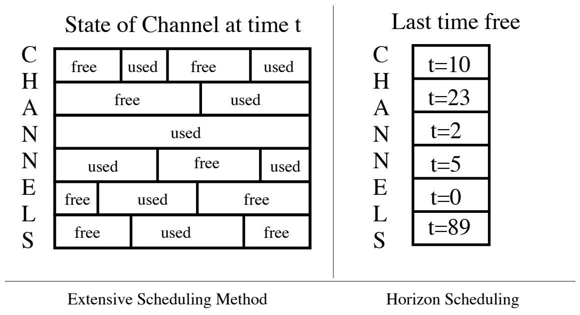


Fig. 3. Possible Schemes to Schedule Multicast requests

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 channel pool. Since multicast addresses are a finite resource, the server will need to perform some level of scheduling to allow for address/channel reuse; this is done via the channel pool.

Since work is scheduled out in time, there are two naive ways to represent the schedules of a multicast channel. For a continuous-valued time, a set of linked lists could be used to represent free intervals; leading to a search across the multicast channels for an interval of time in which nothing else is being sent. This problem can be reduced into a discrete case by selecting only intervals of time, leading to a bitmap operation to search for intervals of free time (Fig. 3). However, this operation is also computationally expensive.

Instead we use a technique called Horizon Scheduling (HS) [19]. HS allows us to store the last free time for each multicast channel. When a channel is needed for an interval of time, the server will do a linear search across the multicast channels(Fig. 3). It will look for the multicast channel with the latest free time that is before the beginning of the new interval. Once found, the server will update the multicast channels free time to be the end of the interval.

B. Server Data Engine (SDE)

The SDE is responsible for retrieving video streams from disk and then transmitting them on one or more network connections in accordance with the abstract transmission schedule received from the Server Control Engine. The SDE must therefore handle large volumes of video data and with real-time deadlines. Since the SDE runs on top of Linux, which offers no real-time streaming support, the SDE is exposed to the occasional unpredictable timing behavior characteristic of non-realtime OSs.

The SDE is a multithreaded, single-process entity. For each video that is currently being transmitted, the SDE creates two new threads. A *disk thread (DT)* handles retrieval of the video 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 the equal-sized free memory blocks that form the buffer cache for this video. Both DT and NT operate in rounds. The disk round-length is δ

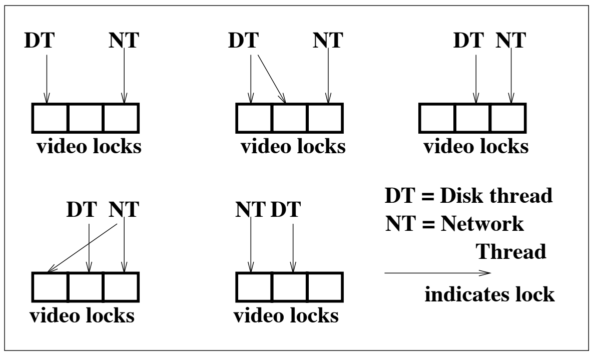


Fig. 4. How BufferLocks Work

and the network round-length is τ. In each δ-round, the disk thread wakes up, uses the abstract server transmission schedule to determine which parts of the video need 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 (from the abstract server transmission schedule) the data that needs to be transmitted on each channel in that round, transmits that data from the main memory buffer cache, and goes to sleep.

The separation of disk retrieval and network transmission activities is motivated by the very different nature of the disk and network subsystems. It is well-known that the disk subsystem can introduce significant unpredictability in the timing and has high overheads. To prevent starvation due to high and variable disk access times, data is prefetched from disk and temporarily staged in main memory. To reduce the impact of disk overheads, we (i) employ asynchronous read requests, and (ii) issue read requests for large chunks of data at a time. Therefore the disk round, δ, should be relatively large, and is currently set to 1 sec. For the network, it is desirable to avoid injecting bursts of traffic. Therefore the network round length, τ, is typically much smaller than δ, so the NT has to transmit only a small amount on any channel at a time.

Since the DT acts as the producer and the NT as the consumer of data in the main memory buffer cache, a mechanism is necessary to ensure that the NT does not attempt to go ahead of the DT or access the part of the buffer cache that the DT is simultaneously modifying. We outline below a method of synchronizing the NT and DT using only three locks(Fig. 4).

Let us define $m = \frac{\delta}{\tau}$, the ratio of δ-to-τ. Initially the disk and network thread each take a lock. For each completed loop of the disk thread, the disk thread attempts to grab the next lock. The three locks should be considered as a loop where the next lock moves along one direction. Once the disk thread gets the next lock it unlocks its previous lock and increments the δ counter to show how many times the disk thread has ran the loop. For the network thread, the current value of the τ is multiplied by m. If the product is greater than or equal to the current δ counter then the next lock is acquired.

Note that the disk thread must guarantee that the data will be in position before it moves to the next lock, and

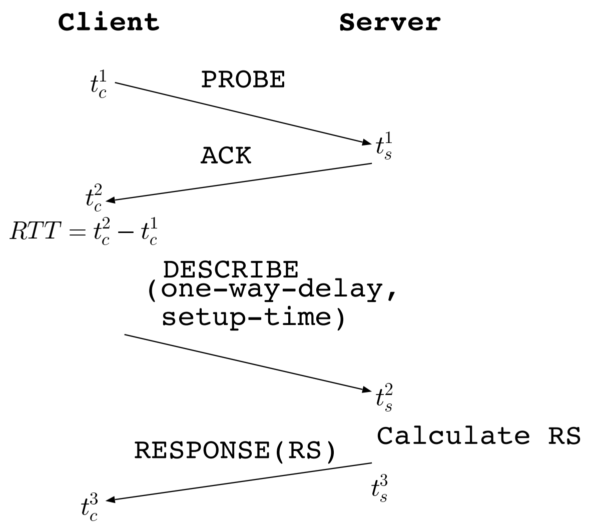


Fig. 5. The signalling between the server and the client.

the disk thread will not erase a block of data that has been called up less than 2δ rounds ago. As a consequence this places a lower bound of δ for start up latencies in the data engine. Note also that a key advantage of this novel 3-lock arrangement is that in each δ-round, the DT or NT needs to access only a single lock. This is a much more lightweight mechanism than conventional approaches which would have associated locks with each buffer cache block, and would require the DT or NT to access a potentially large number of locks in each round.

At the end of a DT or a NT's round, the thread must give up the CPU and wait for the next round. The SDE is running in a Linux-based non-real-time environment, with a very coarse (10ms) scheduling granularity. This means that sleeping threads are rescheduled as much as 10ms apart. Therefore is no guarantee that the thread will wake up at its designated wakeup time. Even if it wakes up on time, it is not guaranteed to get the CPU immediately. A mechanism is thus required to prevent these delays from accumulating and adversely impacting system performance. In our design, when a thread completes its activities for a round, it checks to see if it has fallen behind, i.e., if the start time the next round has passed. If so, the thread starts servicing the next round instead of going to sleep.

IV. Signaling Protocol

Client-server communication in our testbed is based on the Real Time Streaming Protocol (RTSP) [16] and occurs over a separate TCP connection. The steps from when a client issues a request to when it receives a Response message (see Figure 5) are as follows:

1. At time t_c^1, the client sends a PROBE message to the server. The server receives the message at time t_s^1 and immediately returns an ACK message.
2. The ACK message reaches the client at time t_c^2. The client calculates the RTT as $t_c^2 - t_c^1$. Based on the RTT, the client estimates the one-way-delay from the server to the client. Exact estimation of this one-way-delay is not trivial, because the route from server to client and from client to server are not necessarily the same. Also routes followed by multicast and unicast sessions may be different. As an initial estimate, we choose half of the RTT as the one-way-delay.
3. The client composes an RTSP DESCRIBE message and piggybacks with it additional information (using

Video	Format	Length(min)	Frame rate	Bandwidth(Mbps)	File size(MB)	# of RTP pkts
Leno	RealMedia	8.4	13	0.196	12.74	25180
Hacker	MPEG-1	18.6	11	0.998	153.69	131939
Blade1	MPEG-1	12	30	1.99	180.1	155146
Blade2	MPEG-1	15	30	3	337	296706

TABLE I
SAMPLE VIDEOS FOR THE EXPERIMENTS.

the SDP description language) such as its estimates of the one-way-delay and initial setup time. The setup time includes time for creating sockets, setting initial parameters, etc. This is obtained based on off-line profiling of the client code.

4. The server receives the RTSP DESCRIBE message at time t_s^2. It calculates a reception schedule for this client and piggybacks the Reception Schedule (in SDP format) on the RTSP RESPONSE message to the client. The client receives the RS at time t_c^3.

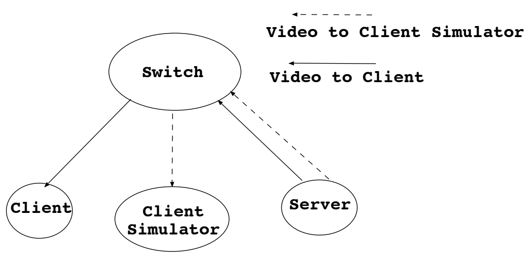
V. EXPERIMENTAL SETTINGS

In this section, we describe the evaluation of the platform. The measurements are divided into two sets. One set is carried out locally. It is used to show the system's ability to run broadcast schemes in an ideal local area network. In particular, we explore the end-end performance and the caching effect. The other set is carried out over *vBNS*. This initial study of a wide area network focuses on the server's performance at delivering streams across the network in preparation for our future work.

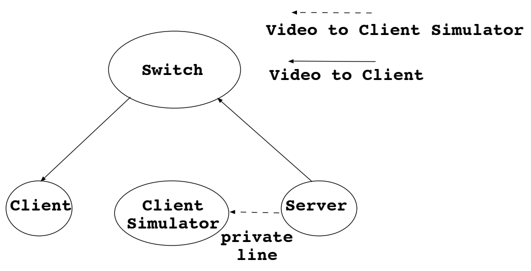
The videos used for the experiments are listed in Table I. The bandwidth of the videos varies from 196.8Kbps to 3Mbps. The length of the video is in the order of several minutes. The number of RTP packets refers to the total number of packets after RTP packetization based on RFC2250 [15]. The maximum size of each packet is 1500 Bytes. In all experiments, the server transmits each stream at the playback rate on each channel. 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 demonstrate here the tuning experiments on δ and τ. The experiments show that improper setting of δ and τ lowers the server performance and the settings of δ to 1 sec and τ to 33 msec leads to good server performance. Throughout the experiment, we use this setting for δ and τ.

A. Local Area Configurations

The configurations for end-end measurement in the local environment is shown in Fig. 6. Both configurations contain a server, a client, and a client simulator. All three machines are 400 MHz Pentium II with 400MB RAM running a Linux OS. The server sends out three separate copies of the *Blade2* video listed in Table I using a selected broadcast scheme. The client simulator generates a background load of client requests to the server in a Poisson manner choosing one of the three videos with equal probability. The server sends the video to the client



(a) Configuration I



(b) Configuration II

Fig. 6. Experimental configurations for local end-end measurements.

simulator according to the request sent. The client simulator does not actually collect video data from the server. Once the background load reaches a steady state, we run a real client to request the full stream and monitor the statistics on the stream arrival data.

The differences between the two configurations is where the background load is directed to. In Configuration I, the background load is sent out to the switch where it will interfere with the data being sent to the real client. In this configuration it is noticed that the 100Mbit link will become a bottleneck in the system. In Configuration II the background traffic is sent on a private link to the client simulator so that results collected at the client can be determined to be caused by bottlenecks of the server other than the network link.

B. vBNS Configuration

Our vBNS tests include four sites. EC1 and EC2 are located on east coast and connected to vBNS. Similarly, MW1 and WC1 are located in the midwest and west coast respectively. All sites use Linux based servers and clients to make measurements on the stream quality sent from other sites.

C. Broadcast Algorithms

We chose a representative algorithm from the periodic broadcast family and the patching family of broadcast 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 [6] segmentation scheme. In GDB we divide the stream into segments. The length of the first stream is denoted by l. 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. Throughout the experimental section we use *l-GDB* to indicate a GDB scheme where the initial length is of size l. The 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

Scheme	Number 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 2 DIFFERENT GDB SEGMENTATIONS FOR THE 3 MBPS, 15 MIN MPEG-1 BLADE USING FIRST SEGMENT LENGTHS OF 3 AND 10 SEC RESPECTIVELY.

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 are sent out every 480 seconds.

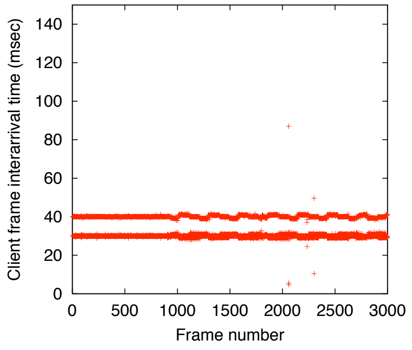
- *Patching:* For patching, we consider the threshold based Controlled Multicast scheme proposed in [17]. In this scheme, a threshold T is used to control the frequency at which a complete stream of video is transmitted. The first request for the video is served using a complete transmission. Subsequent requests that arrive within T time units of beginning transmission of the previous complete stream will share the complete stream and obtain only a prefix of the video from the server. A request arriving after T time units have elapsed is served by initiating a new complete transmission for the video . Under this scheme, when the client arrival rate for a video is λ, the length of the video is L seconds, the threshold is chosen to be $(\sqrt{2L\lambda + 1} - 1)/\lambda$ seconds to the minimize the average transmission required to serve a client [17].

D. Metrics

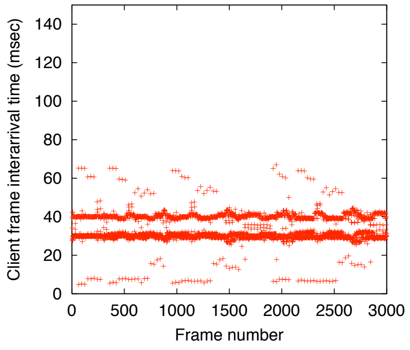
We focus on the the following key performance metrics in our experimental evaluations.

D.1 Server Metrics

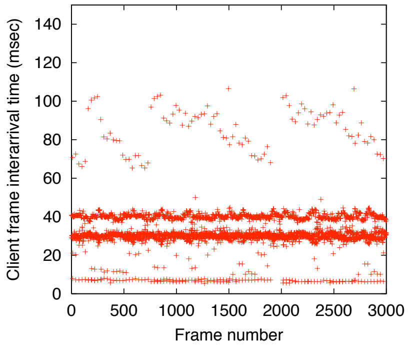
- *System Read Load (SRL)* : This is the volume of video data requested per unit time by the application from the underlying operating system. A read request is initiated only if a required data block is not present in the application-level cache. SRL therefore presents a measure of the workload associated with the data path that is imposed on the underlying system by the application. The system may satisfy the request from the kernel buffer cache if possible, and otherwise fetch the block from disk. Therefore, this metric also acts as an upper bound on the read workload experienced by the disk subsystem.
- *Server Network Throughput (SNT)* : This 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 buffering.
- *Deadline Conformance Percentage (DCP)* : Given a transmission schedule, this is the percentage of frames that the server was able to transmit to the network by scheduled deadlines created by the server control engine. This measures how well the server was able to keep up with the schedule.



(a) 30-GDB, 1 request per min.



(b) 10-GDB, 1 request per min.



(c) 10-GDB, 600 requests per min.

Fig. 7. Client Frame Interarrival Time under different segmentation schemes and request rate for periodic broadcast.

Arrival Rate(per min)	Scheme Used	Server Network Throughput(Mbps)	Deadline conformance Percentage
1	30-GDB	14.81	99.9%
1	10-GDB	19.27	98.3%
600	10-GDB	19.27	97.9%

TABLE III
SERVER STATISTICS FOR PERIODIC BROADCAST.

D.2 Client Metrics

- *Network Jitter:* Suppose S_i is the time that the ith packet is sent at the server, and R_i is the time that the ith packet is received at the receiver. In [14], the difference in packet spacing at the receiver compared to the sender for a pair of packets is defined as: $D(i, j) = (R_j - R_i) - (S_j - S_i)$. We define the average of $\{|D(i, i + 1)|\}$s over a certain period of time as the network jitter in this period.
- *Client Frame Interarrival Time (CFIT):* Suppose r_i is the time that the last packet of frame i reaches the client. The difference of r_{i+1} and r_i is the client frame interarrival time. For a smooth transmission, the frame interarrival time should be near to a constant. The variability of CFITs reflects the jitter caused by both the server and the network.
- *Reception Schedule Latency:* The Reception Schedule Latency is the time from when the client requests the video at t_c^2 to when it receives the RS at t_c^3 (see Fig. 5).
- *Stream Latency:* The stream latency is the time from when the client requests the video at t_c^2 to when the first frame of the video reaches the client (see Fig. 5).

VI. Performance Evaluation

A. End to end performance

A.1 Periodic Broadcast

In l-GDB the data are sent out regardless of the number of clients that request the video. This allows us to test two dimensions of load on the server. We can inspect loads where the server is sending varying numbers of streams onto the network along one axis and vary the number of requests for that video along another axis. For these experiments we used local configuration II (see Fig. 6) to minimize the impact of the network bottleneck. We used 10-GDB and 30-GDB with arrival rates of one request per minute generated by the client simulator to test the effects of server network throughput on the system. The results are in Table III. From these results we can view the deadline conformance of each of these runs. We use the deadline conformance percentage to show the percentage of the frames that are sent out by their deadline. Since the server only reports 15% of its processor being utilized (all of it in system time), DCP proves to be a good indicator of stress seen at the server. As the SNT at the server increases for these schemes, the deadline conformance decreases. However, most deadlines missed were less than 100ms.

The client sees very few problems for both 30-GDB and 10-GDB schemes. Figure 7 plots the client frame interarrival time for the transmission experienced by the client. For this video, the frames should arrive every 33ms from each other. By examining Fig. 7(a) we notice a strange effect. The plot of times is divided into two layers, 30ms and 40ms. This is due to the 10ms granularity of scheduling on Linux. As would be expected two thirds of the mass lies at 30ms and one third lies at 40ms. This shows that the server is able to deliver frames with little jitter for 30-GDB. In Fig. 7(b) we can see differences when load is increased at the server running 10-GDB. We notice that several frames fall well away from 33ms interarrival times. As a result the difference of time till the next frame is very low and explains the line of points that form of the bottom of the graph. In most cases, the client can expect to receive the next frame in less than 100ms. In this LAN setting, we can see that the server is easily able to handle these loads and provide the client with high quality video.

Next, we examine the effects of the number of clients requesting videos. For this we examine 10-GDB with request rates of 1 and 600 per minute generated by the client simulator. This measurement is very important for these schemes. In theory, segmented delivery should scale up to an infinite number of clients since for each new client the server will not have to transmit any more data. However, as we can see in Table III increasing the request rate causes the DCP to decrease. By examining Fig. 7(b) and Fig. 7(c) we can see that the client experiences higher variation in the CFIT plot between frames. However, this rate is still well behaved.

The end result of this experiment shows us two things. First, the server sees little difficulty in working with the large number of streams sent out and the client is similarly able to listen to 5-7 streams with little difficulty. Secondly, high client requests do affect the ability of the server to deliver the stream. This could eventually lead to scaling problems or at least considerations in the implementation of future systems.

Arrival Rate(per min)	Threshold (minute)	SNT (Mbps)	DCP
1	7.39	20.25	99.9%
5	4.05	57.7	99.9%

TABLE IV
SERVER STATISTICS FOR PATCHING.

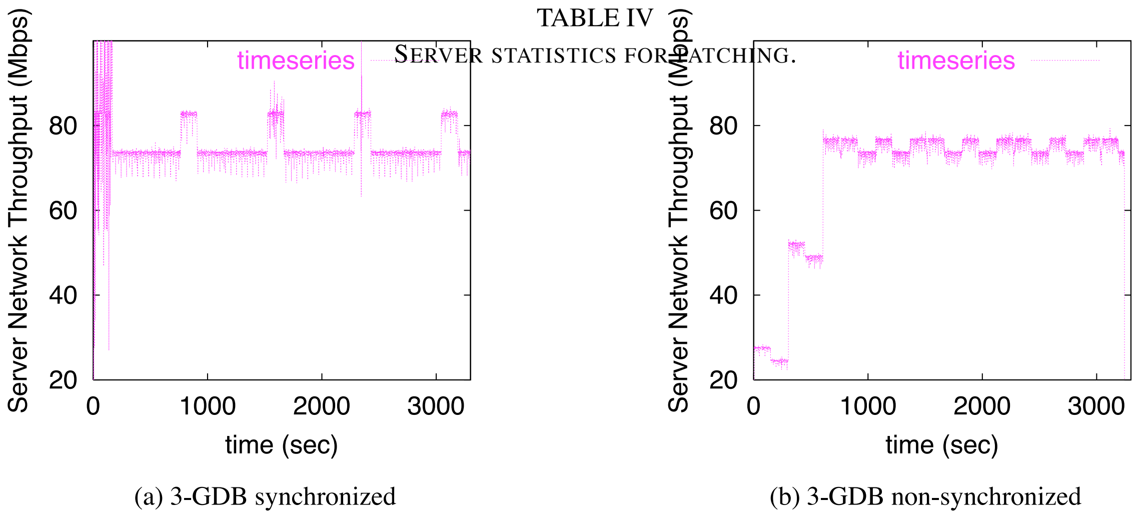


Fig. 8. Synchronization between transmission schedules could lead to bursty behaviors

A.2 Patching

For patching algorithms the number of streams sent out is a function of the client arrivals. This means that the streams a client must receive (i.e. from multicast groups) is a function of when clients arrived before it. We are forced to use local configuration II (see Fig. 6) so that the real client can interact with the requests of the simulated clients. Since the arrival rate and the SNT are related, we increase the client arrival rate from 1 to 5 requests per minute and record the performance. (For higher arrival rates the network link becomes a bottleneck.) Table IV shows the parameters of the experiment. The server's DCP is remains steady at 99.9%.

On the client side, we see similar CFIT distribution as in periodic broadcast. We observed that 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 stream latency and some delay after that to accommodate the packets that come later than the supposed playback time. We can conclude that the network becomes the bottleneck in LAN settings since the server is able to send at the bandwidth of the link without experiencing poor quality video at the client.

B. Scheduling Among Videos

When running 3 copies of the video *Blade2*, we find an unexpected effect. The amount of data placed on the network seen in Fig. 8(a), showed spiking effects. To understand this we looked back to the way that l-GDB works. The l-GDB algorithm periodically repeats each segment of the video at a certain rate. The last segment might be smaller than the repetition rate. Therefore, the server will send the segment and then send nothing until the next repetion. After realizing that the server was seeing three extra channels of information, we examined

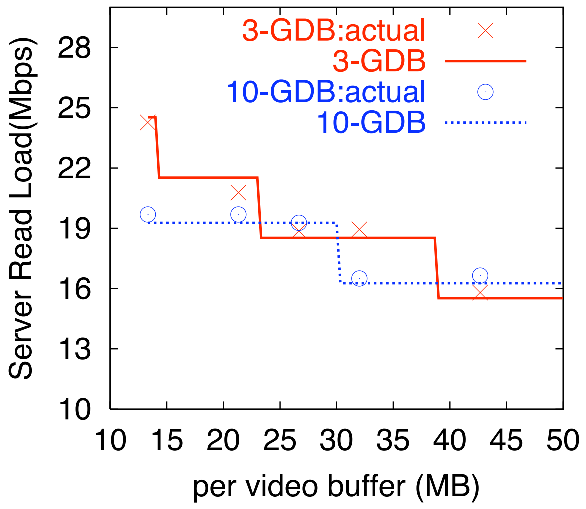


Fig. 9. Caching effects on Periodic Broadcast: plots the experimental and analytic values of the read overhead under LRU.

what happened when the schedules were started separately at an interval of three minutes apart. Figure 8(b), shows three 3-GDB staggered broadcasts to prevent them from synchronizing the retrieval of the last segment, and find that the sustained bursts disappear. This is clearly useful as it removes the necessity to provision high peak SNT. This example illustrates the benefit of using techniques for smoothing out the offered load, especially for high loads.

C. *Caching Implications for Periodic Broadcast and Patching*

Earlier research on periodic broadcast and patching has focussed on the server network throughput requirements of these schemes. We now focus on the server read load imposed by such schemes. Server end-systems today possess significant high speed memory. Caching effects (at the application and system level) would impact the full demand of the server. In this study, we investigate the use of *application-level* caching and *application-specific* caching policies for reducing the bandwidth demand on the underlying server operating system. We consider Least Recently Used (LRU) cache replacement as a baseline. The choice was motivated by the fact that this is widely used in literature and many conventional operating systems implement this policy in their underlying kernel buffer caches.

C.1 Periodic Broadcast

We first consider two GDB segmentation schemes 3-GDB and 10-GDB. Our application locks a tunable amount of main memory for application-level caching for a video. The video uses a local caching policy to reduce the load on the system. A read is generated only if a block is not present in the application-level cache. The metric of interest is the system read load.

Fig. 9 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 performance for the same cache size. The small deviations between the analytic and experimental values are due to the large application level memory blocks (100 KB) used for these experiments. This graph demonstrates the role of caching in reducing the read load. As expected the SRL is a non-increasing function of increasing buffer

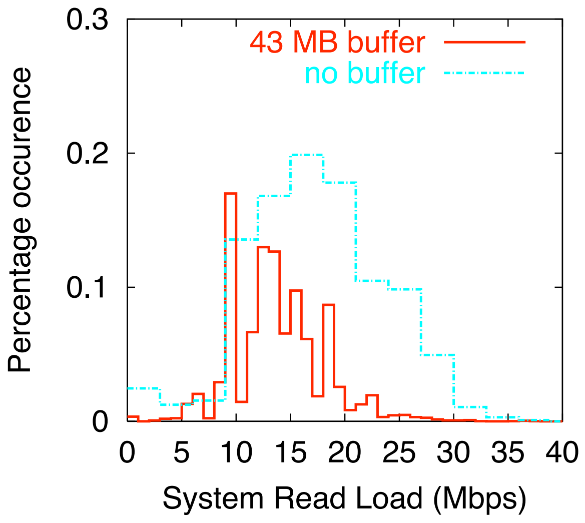


Fig. 10. Shows the distribution of server read loads for patching with and without buffer.

size. In addition, larger and larger increases in buffer size yield lower and lower returns.

In the absence of any caching, the read loads for the 3-GDB and 10-GDB segmentation schemes would be : 24.53 Mbps and 19.27 Mbps respectively. Even with a relatively small 32 MB buffer which is 9% of the size of the video, the load under LRU reduces by 23% and 14%.

Periodic broadcast exhibits some interesting qualities under LRU caching. In order for caching gains to be realized the buffer must be large enough to allows for an entire segment to be cached. Otherwise the LRU will choose to replace the blocks that are actually needed next. This explains the step-like behavior. A step corresponds to the region where LRU has buffer to cache another segment. Hence the step sizes are 3Mbps, which corresponds to the bandwidth for one segment. This is followed by a horizontal portion where the additional buffer is not sufficient to fully cache the next segment.

Network-centric studies of periodic broadcast trade off larger stream latency for reducing the SNT for a particular periodic broadcast scheme. Under this approach, 10-GDB should require less SNT than 3-GDB. However, this does not necessarily hold in the context of the SRL, as caching effects may cause 3-GDB to impose a lower SRL than a 10-GDB (See Fig. 9). For example, with 27 MB of buffer, the 10-GDB induces a read load of 19.27 Mbps, while the 3-GDB results in a load of 18.88 Mbps. The graphs suggest that instead of increasing the length of the first segment, a better approach may be to use caching to reduce the read load on the server.

C.2 Patching

We next consider the impact of application-level caching on patching. For a given arrival rate of 1.33 client per minute, we examine the changes in the SRL. In the absence of caching, the expected SRL is 15.90 Mbps. As can be expected, the resultant SRL is lower to 13.69 Mbps with 43MB of buffer. Fig. 10 shows the distribution of read loads across a five hour run, for no buffer and 43 MB of buffer. The graph illustrates that for a given buffer size, the instantaneous read load can be much higher than the mean, and that LRU caching with even a modest amount of buffer can significantly reduce the instantaneous read load.

Trace	Server-client	Time	RS lat.(ms)	Stream lat.(s)	Pkt. Loss	Fraction of frames without loss
L1	EC1-EC2	9/09 21:00	242	2.16	0.2%	99.5%
L2	EC2-EC1	9/09 21:30	87	2.40	2.00%	92.7%
H1	EC1-EC2	9/09 16:00	226	1.63	0.1%	98.9%
H2	EC2-EC1	9/09 16:30	162	1.70	17.4%	14.6%
H3	EC1-MW1	7/26 15:00	198	1.70	1.12%	91.9%
B1	EC1-EC2	10/16 16:00	85	0.81	0.13%	99.2%
B2	EC2-EC1	10/16 16:30	24	1.37	31.9%	6.3%
B3	EC1-WC1	10/03 20:00	183	0.90	5.3%	93.2%
B4	WC1-EC1	10/06 20:00	178	1.32	24.9%	12.7%

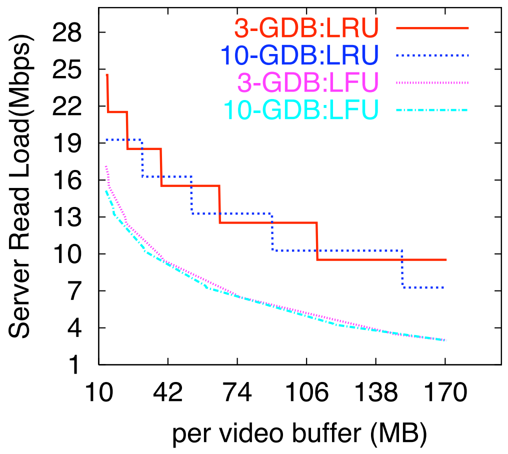
TABLE V
EXPERIMENTAL DATA OVER *vBNS*.

D. Network Evaluation

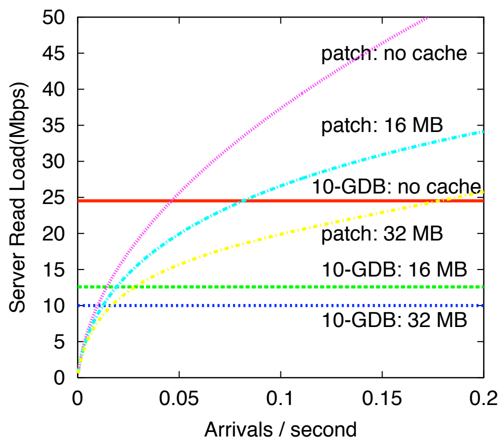
Ideally, we could evaluate the platform using multicast over *vBNS*. Unfortunately, the sites for our WAN experiment do not have multicast interconnectivity. Given this, our preliminary experiment is testing the simplest case, that is, transmitting the video in one channel using unicast. Clark, etc. carried out application-level measurement of performance on the vBNS based on simulated video data [12]. In our experiment, the server transmits videos with bandwidth ranging from 196.8Kbps to 2Mbps over *vBNS* and the goal is to measure the resultant loss, jitter and latencies. We disable the decoding and playback at the client to remove their load on the processor.

Table V summarizes some of the experiment results. Traces L1 and L2 are collected using the video *Leno*. Traces H1, H2 and H3 are collected using the video *Hacker*. Traces B1, B2, B3 and B4 are collected using the video *Blade1*. All the data in the table are measured at the client site. The packet loss ratio is the number of packets lost over the total number of packets transmitted. We assume all the losses occur in the network because our local experiments shows that there is no loss caused by the deficiency of the server or the client. To confirm this, we use tcpdump to collect traces at the server and the client network interface. The comparison of the tcpdump traces with the log data recorded by the server and client confirmed our assumption.

The table shows that there is tremendous variability in terms of the packet loss ratio along different paths and the bandwidth of the video. The packet loss ratio of the low bandwidth videos (e.g. *Leno*) is usually significantly lower than that of the high bandwidth videos (e.g. *Blade1*). Most of the losses occur as single packets, this is consistent with the result in [12]. However when transmitting the highbandwidth video *Blade1*, we see bursty losses (over 10 packets loss in the row). The last column of the table shows the fraction of frames without any loss. It is a rough measurement of the video playback quality assuming players can not tolerate any loss in a frame. For more resillent players, we need more accurate measurements which take account of the importance of the different frames and the dependence of the frames. Network jitter per second is measued as defined in Section V-D.2. We compare the packet loss and jitter for every second and do not see correlation between them.



(a) Plots the comparison of LRU and LFU caching against server read load.



(b) System Read Load for CM patching and 10-GDB for video *Blade2* with LFU caching

Fig. 11. LFU characteristics for periodic broadcast and patching

The high loss rate on some paths implies that loss recovery scheme is needed to provide quality stream delivery.

VII. Lessons Learned

We have gained several key insights from the development of this streaming media platform and subsequent experimental evaluations. We discuss these insights in this section.

A. Caching Schemes

During the process of evaluating the performance of our periodic broadcast and patching schemes, we realized that LRU is a poor caching scheme. Subsequently, we concluded that a new replacement policy, Least Frequently Used (LFU) can provide better performance.

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

LFU can be shown to be optimal and provides a lower bound for theoretical server read loads. When we compare the LFU with the LRU scheme used previously, we can see significant improvement (See Fig. 11(a)). With the same buffer size, LFU reduces SRL much more effectively than LRU. For example, in 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 LFU decreases SRL more smoothly than LRU. We are in the process of implementing this simple yet remarkably promising scheme in our server.

It has been shown in [20] that there exists a request arrival rate above which periodic broadcast is preferred and below which controlled multicast is preferred. We will now examine the effect that LFU caching has on this crossover point. In Fig. 11(b) we see LFU used for patching and periodic broadcast across an axis of request arrival rates. (Note that previous graphs of SRL include overheads imposed by the system architecture. Here we view the upper bounds for comparison.) 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 24.53 Mbps. As the buffer cache increases, we find that the SRL decreases as expected for both schemes, but 10-GDB sees more advantages. With increasing buffer size, the crossover point for these 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. Inter-Scheduling Interactions

Most broadcast schemes have been examined in the context of a single video. It's important to note that, possibly unwelcome, interactions can occur between schedules for different videos. In Section VI-B, we presented a case where the lack of attention to this resulted the synchronizing the schedules of three video sessions. This produced situations where the server generated request rates well above the average rate for short period of times. These synchronizations can be reduced or eliminated through careful scheduling.

C. Server-Client Synchronization

One of the lessons we have learned which is not reported on in this paper is the difficulty of synchronizing the server and the client. For previous RTSP interfaces, the client selects a port on which it will receive the stream. With the broadcast schemes considered in this paper, the client must know not only where to listen to but also when to listen. If the client listens too early to a multicast stream, that client will receive data that is not relevant. Conversely, if the client listens too late, it will miss data. We presented an approach which uses relative timing along with estimations of round trip time. However, the variability of multicast join latencies from site to site can cause problems with scheduling.

D. The use of a non-real time operating system

In the design and implementation of our server, one of the main concerns was the use of a non-real time system. Our experimental results show that, although the server runs on top of Linux, without any underlying real-time support, with access to clocks with coarse time granularities of at least 10ms, the server is nonetheless able to meet real time deadlines. For loads that nearly saturate a 100Mb network connection, the server misses surprisingly few data transmission deadlines. Furthermore, we are able to show that with an initial startup delay of less than 1.5 sec the client can absorb jitter and hide the startup signaling latency in a LAN environment. This clearly justifies our design decisions.

VIII. 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. The platform uses the Real Time Protocol for data transmission and Real Time Streaming Protocol for signaling. The platform has been designed to be flexible and extensible to support a wide range of streaming delivery schemes.

Through measurement and analysis we find that the server is able to operate on a LAN setting with network bandwidth as the main bottleneck. Jitter imposed by the non real time operating systems and the network can be absorbed by a small amount of latency at the client. The use of caching in the server helps to reduce the load on the systems without sacrificing the quality of services. Furthermore, we have shown that LFU allows us to optimally cache the streams for periodic broadcast and patching schemes. Initial evaluations over the vBNS show that error-recovery schemes are needed to handle the losses in the network.

We are implementing LFU caching policy in our system to further improve the performance of the server. Also we are in the process of developing a network proxy server testbed that in conjunction with the current video server will allow us to investigate how proxies can be used to deliver high quality streaming video to clients over best-effort IP networks.

IX. 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] 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.

[2] 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.

[3] 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.

[4] 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 *Proceedings of ACM Multimedia'95*, November, 1995. San Francisco, California.

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

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

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

[8] 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.

[9] 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.

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

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

[12] M. Clark, and K. Jeffay, "Application-Level Measurement of Performance on the vBNS", in *IEEE Intl. Conference on Multimedia Computing and Systems*, June 1999.

[13] M. Handley and V. Jacobson, "SDP: Session Description Protocol", RFC2327, April, 1998

[14] H. Schulzrinne, S. Casner, R. Frederick and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", RFC1889, January, 1996

[15] D. Hoffman, G. Fernando, V. Goyal and M. Civanlar, “RTP Payload Format for MPEG1/MPEG2 Video”, RFC2250, January 1998

[16] H. Schulzrinne, A. Rao and R. Lanphier, “Real Time Streaming Protocol (RTSP)”, RFC2326, April 1998

[17] 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

[18] ftp://mm-ftp.cs.berkeley.edu/pub/multimedia/mpeg2/conformance-bitstreams/video/verifier/

[19] J. Turner, “Terabit Burst Switching,” Journal of High Speed Networks, 1999

[20] L. Gao, Z. Zhang, and D. Towsley, “Catching and Selective Catching: Efficient Latency Reduction Techniques for Delivering Continuous Multimedia Streams,” in *Proc. of ACM Multimedia’99*, 1999