An Implementation of Dynamic Service Aggregation for Interactive Video Delivery [1]

P. Basu, A. Narayanan, R. Krishnan, and T.D.C. Little

Multimedia Communications Laboratory
Department of Electrical and Computer Engineering
Boston University
8 Saint Mary's St., Boston, Massachusetts 02215, USA
(617) 353-9877, (617) 353-6440 fax
{*pbasu,ashok,krash,tdcl*}*@bu.edu*

MCL Technical Report 11-01-97

Abstract–Supporting independent VCR-like interactivity in true-video-on-demand requires the allocation of channel to each interactive user. In the worst case, the system must be provisioned with a unique channel for each user. A variety of techniques have been proposed to ease this requirement by aggregating users into groups. Bridging of the temporal skew between users viewing the same content is one such technique that achieves this result.

In this paper we describe a prototype video-on-demand system that delivers continuous media content to multiple clients using a bridging technique called dynamic service aggregation. Rate adaptation and content insertion are used to achieve resource reclamation on-the-fly, thus improving performance and provisioning requirements in interactive scenarios. In this novel demonstration of aggregation via rate-adaptive merging, MPEG-1 system streams are used as the content format and IP multicast is used for video delivery. We present our experiences with building the system and address issues of server-directed channel switching at the client and stream merging through content acceleration. In addition to the proof-of-concept prototype, we show simulation results of aggregation applied to a large user population as anticipated for video-on-demand. Results indicate that our clustering and merging algorithms can achieve an average piggybacking of more than 2.5 users per physical channel.

Keywords: Video-on-demand, multicast, continuous media, aggregation, resource utilization, video servers.

---

[1]In *Proc. SPIE – Multimedia Computing and Networking*, San Jose, CA, January 1998. This work is supported in part by the National Science Foundation under Grant No. NCR-9523958.

1 Introduction

With the development of file servers specifically optimized for continuous media, much research has gone into improving the throughput and resource utilization of these servers. A typical system involves a video server (or a cluster of servers) reading digitized video from a disk farm and broadcasting it over a common network to a set of clients.

Cable television is an example of a *broadcast* system in which video is distributed over a number of *channels*. Load-reduction techniques have been suggested in such models, including the use of multicast technologies to reduce redundant transmissions. In a *video-on-demand* (VoD) system, the user selects video content to view from a library of available selections. Various classifications of VoD systems have been discussed in the literature [1].

True-VoD systems are expensive to implement because of the high resource requirements. Each user must be allocated enough resources to deliver one channel of video. Current server and network technologies make such an allocation prohibitively expensive. For example, OC-12 ATM at 622 Mb/s can carry just over 100 channels of 6 Mb/s MPEG-2 video. Near-VoD systems typically reduce the number of video streams being delivered by batching requests for the same movie into groups [2]. Thus, a user requesting a movie may have to wait for several minutes prior to playout. True-VoD is therefore, more attractive to the users than Near-VoD.

The high cost of video delivery coupled with the demand for such services over the Internet has prompted research into resource sharing, scalability of services, aggregation and resource reclamation techniques. Aggregation seeks to combine users into groups served by a single channel, thereby reducing channel requirements. IP Multicast is a resource sharing scheme suitable for aggregating users, allowing interactivity at the same time [3]. Schemes have been described for dynamic service aggregation in which the server continuously attempts to aggregate users [4]. Such systems are better suited to interactive scenarios. *Rate adaptation* and *content insertion* are schemes that attempt to reduce the temporal skew between users by modifying the content progression rates of streams.

In this paper, we describe a VoD server which implements dynamic service aggregation using rate adaptation and IP multicast groups. The primitive of "client switching channels under server control" is used for both client-initiated channel switching and aggregation attempts by the server. We also address issues involved in achieving seamless channel switching. We also show simulations of the EMCL clustering algorithm[5] and report substantial

gains through reclamation of channels. A demonstration of the aggregation techniques is achieved with a proof-of-concept prototype implementation.

The remainder of the paper is organized as follows: Section 2 furnishes background and pointers to related work. Section 3 covers major design issues involved in the development of the system. Section 4 describes implementation details. Section 5 describes the simulations and results. Section 6 concludes the paper.

2 Background and Related Work

Consider a scenario in which two users view the same video clip but at different positions in the stream (i.e., they began viewing at different times). If are able to bridge the temporal skew between them, then they can be served out of the same stream, thus releasing a channel. Clearly, this reduction in resource usage results in the ability to serve an increased number of clients with the same set of resources.[3] This is the basic objective of service aggregation and different techniques have been proposed to achieve this end. Essentially, each user has a virtual channel [4] and multiple virtual channels are mapped onto a single physical channel. However, this mapping need not be static. If two users are viewing the same content, each will have a distinct virtual channel (VCh). If both VCh's receiving the same content (i.e., if both users are at the same position in the content stream), then the two VCh's can be mapped onto the same physical channel (PCh). This releases the resources associated with the other PCh. Thus the mapping from VCh's to PCh's can be one-to-one or many-to-one depending upon the state of the system.

Service aggregation can be static or dynamic. In a static service aggregation scheme (SSA), aggregation is achieved at a fixed point, typically prior to initiating stream delivery. Here it is assumed that the aggregates formed at this time remain throughout the life of the stream. However, user interactions, such as pausing or other VCR-like functions, violate this assumption. In this case the system must either exclude interactivity or accept reduced resource utilization. Batching at multiple fixed points[2] and interval caching[3, 2, 1, 6] are examples of extensions to SSA. Dynamic service aggregation schemes (DSA) are more attractive because they continually seek to reclaim resources over the life of the stream. This improves resource utilization, thus permitting the system to support a larger number of users. Rate adaptation[7] and content insertion[5] are examples of dynamic service aggregation schemes.

Rate adaptive merging is a dynamic aggregation technique that seeks to merge skewed but identical video streams by varying the content progression rates of the skewed streams. Content progression rates are distinct from the data delivery rate or the frame rate. An accelerated content progression rate implies that the total duration of the video will be reduced and any given scene would occur earlier in time.

Content insertion is a coarse-grain rate adaptation technique. Here the content progression of a stream is stopped for a period of time while alternative content is delivered (e.g., "commercials"), thereby altering the overall progression rate of the original stream. Under this scenario the alternative content can be simultaneously delivered to many VCh's, freeing multiple PCh's, depending on the current mapping of VCh's to PCh's. Further details about content insertion scheme can be found in Krishnan et al.[5]

3 Building the System

In this section, we describe our prototype VoD system implementation comprised of a server and multiple clients. The main function of the server is to deliver streaming video to the clients while trying to continually reclaim bandwidth using our implementation of dynamic service aggregation, thereby servicing more users than the number of available channels.

Currently, our implementation supports interaction in the form of the VCR *pause* of a video stream. This type interaction best illustrates the concept of static user-VCh and dynamic VCh-PCh mappings. It also shows how the system aggressively performs aggregation. Additonal types of interactions (e.g., fast-forward, reverse) are left a future prototype.

We begin with a discussion of techniques used by the server to facilitate acceleration and to improve performance. We then describe the techniques for stream clustering and aggregation followed by issues of client-server communication and overall system function.

3.1 Conditioning of streams and metadata insertion

One of the primary difficulties in achieving our implementation (called dynamic service aggregation protocol or DSAP) was reconciling the video content format with the network transport format. We used the MPEG-1 system stream[8] as our content format with a realization that it is not particularly suitable for packetization delivery on an unreliable or multicast network. Because the MPEG-1 system layer and compression layer are syntac-

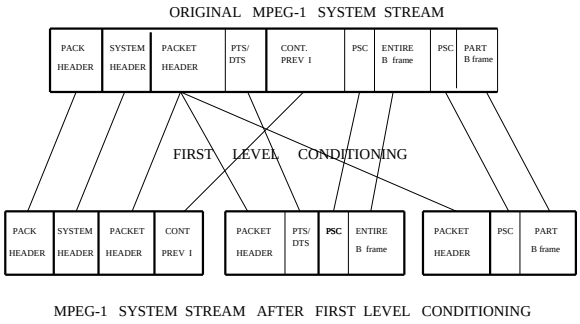

Figure 1: Conditioning of MPEG Streams

tically uncorrelated, the video frame boundaries do not necessarily coincide with network packet data boundaries. Hence, an MPEG-1 systems packet can contain parts of multiple MPEG frames making error recovery at the client difficult. For example, if a fragment of a packet containing a part of an I frame, a complete B frame and a part of a P frame, is lost, then errors in three frames result from the loss of a single packet. To deal with this incompatibility we opted to precondition the MPEG data to achieve a more suitable mapping of frames to packets while preserving compatibility to the standard.

Our *conditioned* format for system streams ensures that each access unit (a frame or a frame part) begins at an MPEG packet data boundary thus removing the non-correlation between the system and compression layers. The format of conditioned packets is shown in Fig. 1. After conditioning, each MPEG packet cannot contain parts of multiple MPEG frames. Hence, in the case of packet loss, identifying the lost frame and dealing with it becomes much simpler. Since the presentation timestamps (**PTS**) and the decoding timestamps (**DTS**) in a packet correspond to its first access unit, when a packet is split at the access unit boundary, the timestamps are removed from the first packet and copied into the newly generated packet. Packet lengths are modified appropriately.

Note that the conditioned stream remains compliant to the MPEG-1 system stream specification. The amount of data inserted via extra packet headers is minimal and does not violate the tolerance limit of the System Clock Reference (*SCR*) field in the *pack layer*. This implies that it is feasible to build an MPEG-1 encoder for capture that generates digitized video in this new conditioned format. Also, the new format allows frames to be dropped anywhere in the source-to-destination path (e.g., at the server) thereby permitting various forms of smooth scaling of service quality.

To deal with the variable-length characteristic resulting from Huffman coding in MPEG

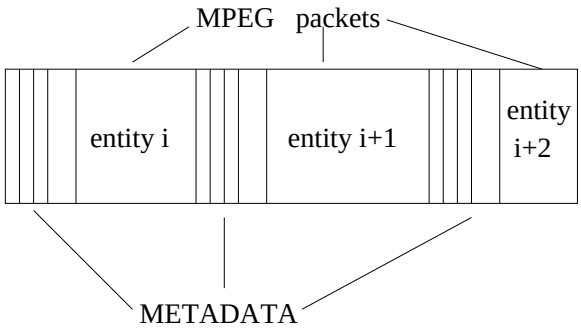

Figure 2: MPEG Stream with Metadata

compression and to avoid parsing the stream at each end, additional metadata are included in the conditioned streams in storage. We define an **entity** as a conditioned MPEG packet with associated metadata. Each entity has the following metadata fields associated with it: *Frame Type*, *Frame Number* within a GOP(*group of pictures*), *Fragment Number*, i.e., the sequence number of the particular fragment of a frame, and the *Size* of the entity in bytes. Offsets of the previous I frame (for rewind operations) and the next I frame (for fast-forward operations) can also be included as part of metadata to help in supporting VCR-like operations. Only a few bytes of metadata precede each entity in the file. The structure of the file after storing the metadata is shown in Fig. 1. In the final step, a table is generated which lists the absolute offset of each frame in the file. Using these metadata, the server can quickly retrieve the video in logical units rather than parsing the bitstream, thus significantly improving performance.

Since we generate metadata at the frame level, the server can easily maintain a 30 f/s delivery rate, independent of GOP boundaries. Support for interactivity can be enhanced by maintaining data about adjacent I- and P-frames. Also, these metadata aid the client in constructing a valid, playable MPEG stream in the event of packet reordering or loss.

As a result of conditioning of the MPEG system streams and metadata generation we increase video file size by about $1.5 - 2\%$. This negligible increase is achieved by an offline, one-pass conditioning and metadata generation process.

3.2 Aggregation

To perform rate adaptation, the server must have the capability to serve content at normal and accelerated rates. We achieve acceleration by dropping a B frame and some audio frames

in every group of pictures (GOP) containing 15 frames. Note that the actual frame rate is the same in both the cases (30 f/s) although the accelerated stream has a higher *effective frame rate* (32 f/s in our case). A technique for efficient storage of video in a single format to permit such dropping of frames from single or multiple disks is presented elsewhere.[9] It was experimentally observed that dropping a single B frame and 6.67% audio in every GOP has a negligible impact on the audio-visual quality of the movie. However, dropping additional frames does affect picture and sound quality perceptibly. Although no proper perceptual studies have been performed regarding this, we believe that deceleration can also be used to merge streams, especially if there is a cost associated per second of a movie. We are not familiar with studies related to frame dropping for non-MPEG codecs, but arguably the effects of dropping content in codecs with no inter-frame dependencies will be more perceptible since those codecs won't have smaller and less important frames (like B-frames in MPEG) which can be dropped.

Although rate adaptation is a good technique for bridging temporal skews, it is not suitable for merging streams that are wide apart. Content insertion helps to bridge wider skews that rate adaptation cannot. Consider two streams that are spaced N frames apart in a given program. We define the *catch-up window* to be the number of frames within which the trailer will catch up with the leader. Let it be K frames assuming no interactions in between. C_1. It is easy to see that $F_{adv} = \frac{15}{16}N_2$. Therefore alternative content of $\frac{15}{16}N_2$ frames needs to be inserted into C_1. Then C_3 would catch up in $15N_1 - 15.06N_2$) frames rather than $15N_1$ frames (without content insertion).

In practice, content insertion can be achieved by maintaining a single multicast alternative content channel (e.g., with advertisements) that the server switches users to during content insertion periods. To avoid problems with switching cut-outs during arbitrary points in the content progression (e.g., n the midst of a "nail-biting" scene), metadata describing viable cut points can be associated with the content. In this case the server can avoid content insertion at these times. The advantage of rate adaptation is that it has a very fine granularity. If the inserted content is in the form of 30 second clips, then a skew of 40 seconds cannot be smoothly bridged by content insertion alone. In this case, we can display one 30-second clip of secondary content, and use rate adaptation to bridge the final 10-second skew.

3.3 Channel switching

The difficulties of channel switching are generally obscured in mathematical models. However, our experience shows that this is a significant requirement which poses problems to the system designer. These issues must be addressed by any system designer and are considered below.

There are two kinds of switching scenarios. The *client-pull* switch, or on-demand switch, occurs as a direct consequence of a client action. For example, when the client requests a movie, switching must be performed to the channel broadcasting the movie. The *server-push* switch occurs when the server preemptively switches the channel that a client is receiving. An example of this occurs when a client who is receiving an accelerated stream, is deemed by the server to have caught up with another stream. In this case, the client must be switched onto a non-accelerated stream.

We simplify the system by treating client-pull switches as a special case of server-push switches. When a client requests a movie, the request is sent to the server. The server creates a new multicast group, begins playing the movie on this group, and sends a server-pushed channel switch directive to the client to begin listening on the new group.

Dealing with latencies in channel switching is also significant. Due to the frame interdependencies in the MPEG compression layer, decompression and display cannot begin from arbitrary frames in the stream. The client must *latch* onto the MPEG stream by discarding all data it receives until it encounters the start of an I-frame. With the standard 15-frame MPEG profile, it takes on average 7.5 frames for the client to latch on to the MPEG stream. Once the stream is latched, the MPEG player will take three frames (48K [8]) to begin playout. This raises the average channel switching latency to a minimum of 10.5 frames, or 0.35 seconds, which is a clearly perceptible delay. In practice, switching latencies can be even greater than this due to MPEG player implementation-specific buffering.

3.4 Server design

The server is the central component of our system. Broadly, it is responsible for scheduling reads from the disk, writing data to the network, handling communication with the client, and performing aggregation. It consists of two modules each performing a different set of functions: a DSAP agent and a scheduling agent. The DSAP agent is responsible for receiving client requests, maintaining the VCh/PCh mapping, and for controlling aggregation.

The long-term scheduling of the system is done by the DSAP agent. The term *long-term scheduling* refers to the task of picking and assigning appropriate movies to a client in a particular session. The scheduling agent, on the other hand, is responsible for short-term scheduling of data within a *scheduling interval*. In other words the scheduling agent reads conditioned MPEG data from the disk and writes data to the network in every scheduling interval. We utilize a scheduling interval of 15 frames, or 0.5 seconds of video. The scheduling agent does not need to parse the MPEG bitstream because the metadata in the stream helps it to read data quickly in logical units (frames). It also monitors the temporal skew between several pairs of streams and informs the DSAP agent when the skew is bridged.

The DSAP agent, however, is more idle than the scheduling agent and it performs only *higher level* tasks. Consider a system with two users A and B. When A requests a movie M, the DSAP agent checks to see if anybody else is viewing the same movie. If not, a new PCh is created, this user's VCh is mapped to it and playout begins. If B requests the same movie M after a delay d, the agent applies the clustering algorithm (discussed below) and may determine that A is watching the same movie and can be clustered with B. This would depend on whether d lies within the mergable distance for this movie. The server then proceeds to specify normal or accelerated channels for each group depending on the results of the clustering algorithm. The DSAP agent is also responsible for generating IP multicast addresses from a pool of addresses and assigning them to appropriate clients.

IP multicast groups is the primary means of achieving aggregation in our system. The paradigm of server pushed channel switching, though contrary in spirit to traditional client-server communication, is an ideal one in these situations. This is because rate adaptive merging is naturally achieved with the help of multicast groups and our paradigm. When a DSAP agent decides to merge nearby streams, it sends a *switch directive* to the trailing clients (who have caught up with the leader now) asking them to leave their old groups and to listen to the leader's multicast group. Resources allocated to the trailers' old PCh's are then released. Implementation issues are discussed in detail in Section 4.

We use the EMCL (*Earliest Maximal CLuster*) algorithm to achieve clustering of streams.[5] A *mergable cluster* is defined as a group of streams which can be merged within a given merging window. This algorithm takes the merging window as a parameter. A brief description of the algorithm follows:

1. Sort the streams carrying the same program in descending order of program position.
2. Generate the *earliest maximal mergable* cluster from this group. The earliest maximal

cluster is that cluster which combines the maximum number of streams, and whose leader is earlier in the program than that of any other cluster combining the same number of streams.

3. Continue generating non-intersecting clusters until all the streams have been considered.

It has been shown [5] that the EMCL algorithm releases the maximum possible number of channels for a given merge window. Performance of the EMCL algorithm as measured in channel reclamation gains are presented in Section 5.

3.5 Client design

The client consists of three components: the transport entity, the MPEG player, and the UI component. The transport entity is responsible for receiving the video packets from the server, stripping the headers, and preparing the video for playout. Since ordered delivery is not guaranteed in UDP, the transport agent must reorder the received packets if necessary. This is accomplished by using a nondecreasing packet count, applied to each packet by the server for this purpose. The set of buffered packets is scanned to determine if any packets were lost in the network. If the agent finds that a packet was lost, it looks to see which frame of which GOP the packet is from and this frame is now unplayable. It then calculates what other frames, if any, were dependent on this frame, and these frames are also unplayable. In some cases the entire GOP must be discarded (e.g., when an I-frame packet is dropped). By discarding these data, the player generates valid MPEG-1 data which is subsequently passed to the MPEG-1 player.

The MPEG-1 player component is an MPEG-1 system stream player. Any one of a number of commercial hardware/software players can be used; however, the player must be capable of receiving streaming MPEG data. The UI component is responsible for handling the interaction of the transport agent with the user and the server. It displays a GUI interface to the user and allows the selection of video clips and the interaction with their playout (e.g., the pause/resume operations). When a user requests a clip or interacts, the UI agent sends a signal packet to the DSAP agent on the server. We use UDP for this communication as well, since it was deemed too expensive to open/close a TCP connection for such a small data transfer. The problem of UDP packet losses can be solved by using one of many standard techniques such as stop-wait.

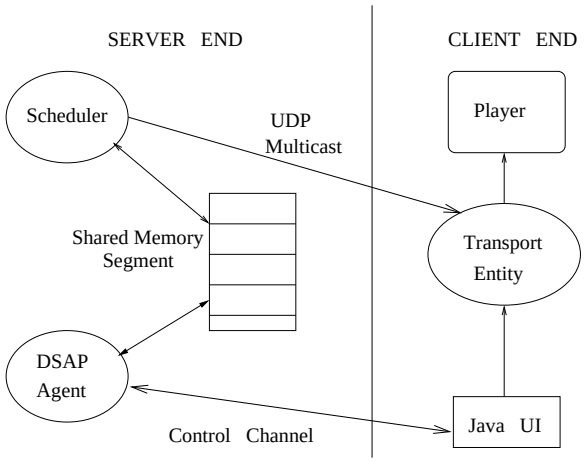

Figure 3: Architecture of the System

Consider the following typical scenario of operation. When a user selects a clip and presses the *Play* button, the UI agent sends a request packet to the DSAP agent. The DSAP agent schedules the playout of the movie and generates a IP multicast address from a pool of available addresses, which it returns to the client. The UI agent then receives the multicast address and sends it to the transport agent, which will join the multicast group after leaving any other group it is currently a member of. The same mechanism is also used for server-pushed channel switches. For example, if the DSAP agent decides to switch a client from a normal to an accelerated stream, it generates an accelerated stream on a new channel and sends a switch directive to the UI agent of the client in question, which then forwards the request to the transport agent.

4 Implementation Details

Fig. 3 depicts a high-level schematic of the system architecture. A client requests a movie from the Java-based user interface. The UI talks to the DSAP agent through a control channel and sends the movie ID to the agent in a datagram packet. The DSAP agent writes the requests into the *shared memory* segment which is read by the scheduling agent once every scheduling period. The scheduling period in the current implementation is 15*(1/30) s = 0.5 s. Then the scheduling agent serves the request. The details of the server operation are shown next.

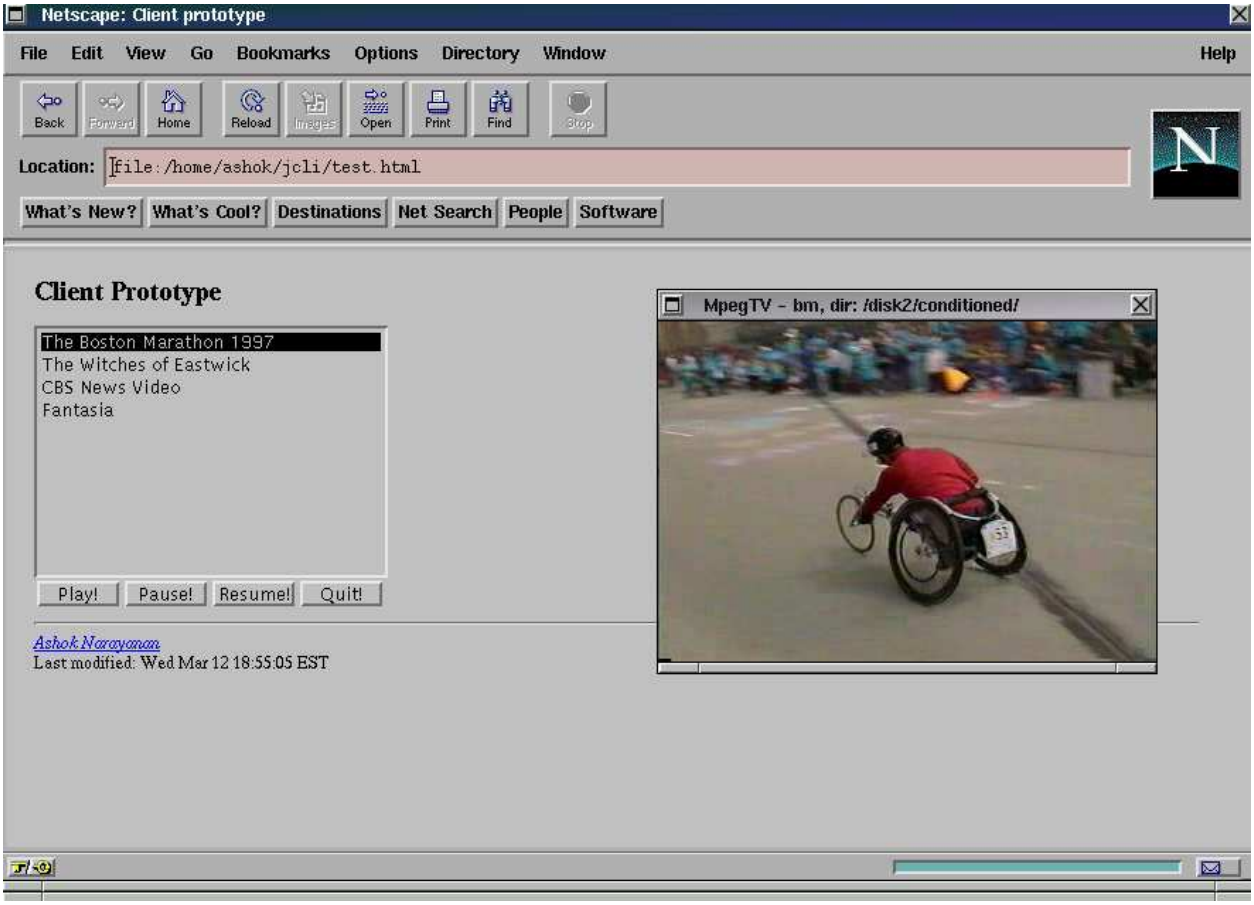

Figure 4: Screen Shot of the Client

4.1 The server

The server is implemented on a Unix platform and has been ported to Linux. The server consists of two processes: the *DSAP agent* and the *scheduling agent*. These two processes communicate using a *shared memory* segment.

4.1.1 The DSAP agent

The DSAP agent handles the control channel communication with the clients. In the current implementation, we handle the following requests: start a movie, pause, resume, and quit. The DSAP agent maintains the state of active streams (the multicast groups on which they are playing etc.) and that of active clients.

When a *start* request occurs, the DSAP agent creates a new PCh by generating a free multicast group address from the pool of addresses. A VCh is created for this client, and is associated with the newly created PCh. The agent signals the client with a packet specifying the multicast group address of the assigned PCh. It then creates an entry in the shared memory segment with the relevant details of this PCh, namely the movie pathname, the multicast group, and the port number. It marks the status of the entry as *active* and *new*. Next, the DSAP agent runs the clustering algorithm followed by the merging algorithm.

On receiving a *pause* request, the DSAP agent checks whether the client requesting the pause is alone in its multicast group. If so, then the PCh is marked as *paused*. Otherwise, a new PCh is created, and the VCh for this client is promoted into the new PCh, which is then marked as *paused*. With a low probability (which depends on the system parameters) such a request will not be honored and hence a new PCh will not be created if there are insufficient resources at the server end.

When the user resumes playout, the DSAP agent can map this VCh into a normal or an accelerated PCh depending upon the distance from its neighbors. In case of the latter, the agent needs to determine the corresponding frame offset in the accelerated stream. Since the normal MPEG stream has 15 frames in a GOP and the accelerated stream has 14 frames in a GOP, frame number F in the normal stream corresponds to frame no. $14 * (F/15)$ in the accelerated stream. Since the scheduling interval is 15 frames, this expression evaluates to an integer.

If a quit request comes, the DSAP agent marks the corresponding shared memory entry

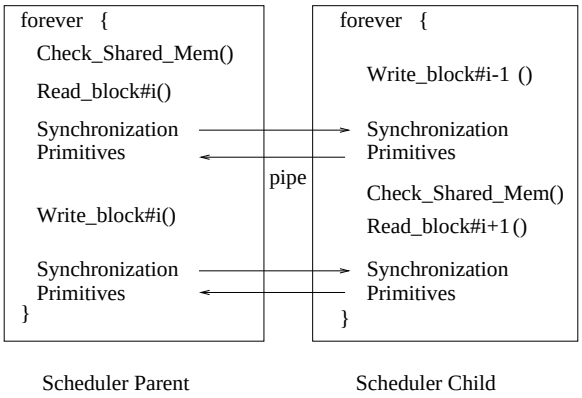

Figure 5: The Scheduling Agent

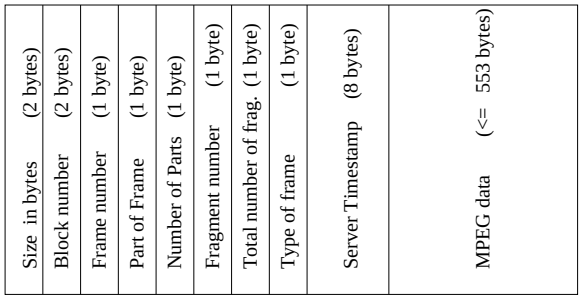

Figure 6: The Application Packet

as
STOP_REQUEST and within two scheduling periods, the entry is removed from shared memory. It also deallocates the VCh and PCh for this client.

The DSAP agent can also get a *merged* signal from the scheduling agent. This occurs when the scheduling agent determines that the temporal skew between two PChs P_1 and P_2 (previously scheduled for merging) has been bridged. All VCh's previously mapped to P_1 are now remapped to P_2. The DSAP agent sends a switch directive to all the clients of these VCh's directing them to begin listening to the multicast group of P_2. P_1 is then terminated and the scheduling agent is signalled to this effect.

When the movie clip associated with PCh P terminates, the scheduling agent signals the DSAP agent. All clients associated with VCh's mapped to P are informed to stop listening. Then this PCh and all VCh's associated with it are terminated and their resources are released.

4.1.2 The scheduling agent

The scheduling agent has been patterned after a dual-buffering reader-writer process model as shown in Fig. 5. Each process alternates as a reader and writer. In every scheduling interval, the reader process checks the shared memory for new requests or changes in the status of existing entries. The following steps are taken to create a new entry.

1. The DSAP agent creates an entry in the shared memory table and sets its status to NEW_REQUEST
2. The first reader process to see this request creates a new session in its own internal process variables. It then sets the stream status to STARTING.
3. The second reader process to see this request creates a new session for itself. Since the status is STARTING, it knows that the other reader process has registered this stream and it is ready to begin playout. Therefore, it sets the stream status to ACTIVE.

When the stream status is ACTIVE, the reader process will read in data for this stream and buffer them for playout in the next cycle. A status of PAUSED represents a paused stream. A similar interlocking process is used for stopping a stream.

1. The DSAP agent sets the process status to STOP_REQUEST.
2. The first reader process removes this stream from its internal variables and sets its status to STOPPING.
3. The second reader process then removes it from its internal variables and sets the status to STOPPED.
4. The DSAP agent notes the STOPPED status and deallocates the multicast group for this PCh.

After every scheduling interval the two processes switch roles. The reader process assumes the role of a writer process for the next scheduling interval. Similarly, the writer process becomes the reader. Both these processes have local data buffers from/to which they read/write the video data for one scheduling interval. The essential advantage of this

model is that reading and writing proceed concurrently. Also, the need for large shared-memory segments (and the associated performance overhead) is removed. When there are multiple sessions, the reads for the sessions are done sequentially and so are the writes.

Synchronization between the two processes is achieved by means of pipes. Process P_1, after finishing its read/write job, writes a byte to *pipe1* and blocks until it receives data from *pipe2*. Process P_2 finishes its write/read job, blocks until it receives data from *pipe1*, and then writes a byte to *pipe2*. Race conditions are thus avoided.

To prevent transient network buffer overloads, the writer process performs *bandwidth smoothing* by writing one frame at a time to the network in a round-robin fashion. Every $1/30th$ of a second each PCh receives a frame. Thus, the idle time is spread evenly across the scheduling interval. While writing to the network, the writing process fragments each entity into UDP packets (553 bytes each), generates some metadata for those fragments (1-byte fragment number, 1-byte max. fragment count, 8-byte server-assigned nondecreasing packet count), and then sends the packets to the specified multicast group. Fig. 6 illustrates the format of the payload of a UDP packet containing video data.

Rate adaptation in the scheduler

In the `check_shared_mem()` routine the `rate_adapt()` module is called. The module has access to the shared memory segment and thus has an access to the values of the current frame numbers of existing sessions. If there are two sessions, one serving the normal stream and the other serving the accelerated version of the same stream, the `rate_adapt()` module monitors the temporal skew between them. If it finds that the current frame number of the normal session is $\frac{16}{15}$ times the current frame number of the accelerated session, it declares that the temporal skew has been bridged and that the streams can be merged. It then sends a *merged* signal to the DSAP agent which carries out the merging procedure (described in Section 4.1.1).

4.2 The client

We experimented with a variety of technologies in the development of the client. These include Java, hardware MPEG-1 playout boards, Microsoft ActiveMovie [10], the emerging Java Media Framework standard from Sun, SGI and Intel [11], and the Intel Media for Java JMF implementation [12]. In many cases our experiences with the products fell somewhat short of expectations.

For the MPEG-1 player component, we chose to use MpegTV, a commercially-available MPEG-1 software player [13]. MpegTV can play MPEG video received through a UNIX pipe or a TCP socket. In this manner, we were able to integrate it into our system without requiring access to the player's source code. We had previously considered using a system stream MPEG player to which we had the source code [14]. For the latter player, based on the Berkeley MPEG-1 player, we were unable to obtain more than 21 f/s on our testbed machine (a Pentium Pro 200 MHz running the Linux). MpegTV delivered almost 30 f/s under no-load conditions and 26 f/s under loaded conditions.

The transport entity is written in C. It receives the video packets from the network, resequences them, strips the headers and generates valid MPEG-1 video, which is then handed-off to the player via a pipe. Since the MPEG-1 player is a black-box component, it is the responsibility of the transport agent to ensure that video data are always available for the player when they are needed. To this end, a startup buffer is filled before playout begins.

One of the significant problems with using a black-box player is the inability to gain access to internal buffers that introduce latencies. Because the player was written to accept data from a network or disk, it has a sizeable internal buffer which we empirically determined to be worth about 4-5 GOPs. Therefore, it is clear that we cannot begin playout before the transport agent has buffered at least this much data. This caused our channel switching latencies to increase by approximately 2 s, and often more. Channel switching delays are a significant problem for system developers who need to utilize downstream black-box MPEG players.

Incoming video packets (Fig. 6) are stored in a client buffer in the transport entity in the order of sending (as determined by the 64-bit monotonically increasing packet stamp applied by the server). The buffer is maintained in the form of a queue, with new packets being added at the tail (or inserted in the queue) and packets being pulled off at the head. Cooperative multitasking is implemented using the Unix `select(2)` system call. The agent has three threads: thread 1 listens on a multicast socket and receives video data, thread 2 writes video data to the player pipe, and thread 3 listens on a control socket for messages from the UI agent. Future versions of the agent will be written using kernel threads. When the playout thread pulls a packet off the queue to strip it and hand it to the player, it checks to see if this packet is the start of a new GOP. If it is, the GOP is scanned for missing packets. If any packets are determined to be lost, the agent drops all other packets belonging to the same frame, and in addition, drops all packets belonging to frames which are dependent on this frame. The algorithm we used to determine frame dependency is outlined as follows:

1. If the packet belongs to a 'B' frame, drop all packets belonging to this frame.
2. If the packet belongs to a 'P' frame, drop all packets belonging to this frame. Also, discard all packets belonging to frames which follow this frame in the GOP, and discard packets belonging to any B frames between this frame and the prior I or P frame.
3. If the packet belongs to an 'I' frame, drop all packets belonging to this GOP.
4. If the packet belongs to an 'A' frame, drop all packets belonging to this frame.

In the above algorithm, each case has a different packet drop policy due to the dependencies between the different types of frames in MPEG.

The UI agent is written in Java. It communicates with the DSAP agent on the server, and the transport entity, through UDP. Two threads have been written – one to interact with the user through the GUI, and the other to listen for commands from the DSAP agent. We also experimented with a complete Java solution at the client end. This was written for browsers under Windows 95 utilizing an ActiveMovie client. However, the Intel implementation of the JMF is still incomplete, so we were unable to test it. We did successfully modify the Java client to write to a TCP socket, and demonstrated it under Linux with the MpegTV player. The advantages of the Java client are: executable on multiple client platforms, standard interface to MPEG players on Windows 95 (through Intel Media for Java and JMF), and native thread support. A screen shot of the client is shown in Fig. 4.

5 Simulations and Analysis

In this section we report the results of simulations on our stream clustering and merging algorithms applied to large user populations. Although we have demonstrated successful behavior with a prototype system, we cannot test its performance for large user populations due to logistical difficulties. Hence simulations are used to understand potential gains in resource reclamation under dynamic service aggregation. The primary metrics of interest are channel reclamation rate and bandwidth reclamation rate for increasing numbers of streams. We also present some analysis of the algorithms.

We assume a library of 50 movies, each one hour long. The popularity distribution of the movies is *Zipfian*. The current simulations are based on a snapshot of the system. We assume that the program positions of various streams within each movie are uniformly distributed.

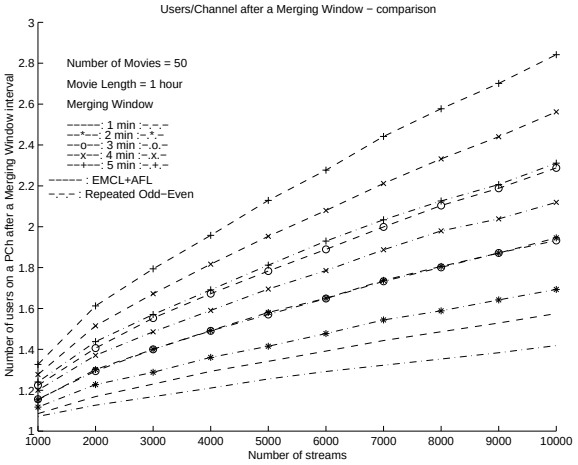

Figure 7: Users/Channels vs. Number of Streams

We vary the number of streams from $1,000$ to $10,000$ and the size of the *merging window* from 1 to 5 minutes where the merging window refers to the time (in units of frames) within which the streams merge. We have compared the channel reclamation rate of EMCL and odd-even merging [7] in a given time interval.

Mergable clusters are generated using the EMCL algorithm [5]. Having generated a mergable cluster, the next task is to determine a strategy for merging the channels in this cluster. The *all-follow-leader* (AFL) approach accelerates all streams in the cluster except the leader. The complete sequence is called *EMCL+AFL*.

The *AFL-content-insertion* (AFL-C) algorithm would accelerate all streams except the leader, who is slowed down by content insertion. But this would result in the leader always getting alternative content which is unacceptable. A constraint should be placed on the maximum "ad dosage" which makes the problem somewhat analogous to a scheduling problem. The implications of this constraint are not considered here. In Fig. 7, we plot the average number of users who are on a single physical channel as a function of the number of streams for different widths of the merging window. We plot the results of both the *EMCL+AFL* and the repeated *Odd-Even* merging heuristic on the same scale for comparison. The latter repeatedly attempts to merge streams in pairs if the distance between two contiguous (in time) streams allows the merge to occur within the merging window.

From the graph we observe that our clustering algorithm outperforms repeated odd-even merging. For example in the case of $7,000$ streams after a merging window of 5 minutes,

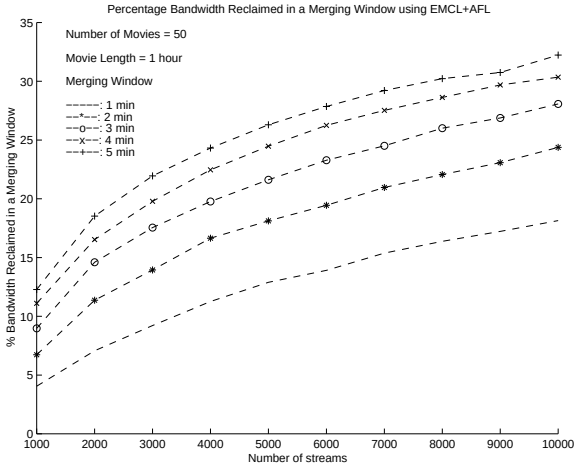

Figure 8: Percentage Bandwidth Reclaimed During Clustering and Merging vs. Number of Streams

odd-even merging is able to piggyback 2 users onto a single physical channel whereas EMCL is able to piggyback 2.4 users on average, which is about a 25% improvement. In the case $5,000$ streams and a window size of $5,000$, EMCL reclaims as much as 45% of the channels. We also observe that the percentage of reclaimed channels increases as the number of streams increases. This is because more streams appear in any mergable window, resulting in more mergings and hence the release of more channels.

The EMCL algorithm has been shown to possess $\mathcal{O}(n)$ time complexity. It has also been shown to be optimal in the number of channels reclaimed with a given merging window.[5]

In Fig. 8 we plot the percentage of bandwidth reclaimed during the clustering process (over a merging window) against the number of streams. (In the previous graph we discussed the number of channels freed at the end of merging.) The amount of bandwidth reclaimed is computed as follows. We first compute the bandwidth required without any clustering and then subtract from it the amount of bandwidth spent during execution of the AFL algorithm. This computation is done over all the mergable clusters which were output by EMCL. We can see from the graph that for all values of window size, the bandwidth reclaimed increases with the increase in the number of streams. Also, with the increase in the size of the merging window, the reclaimed bandwidth increases.

It is also evident from the graph that the gains decrease as the merging window increases. Also, because a real system will be likely to support interactive functions and users will arrive and depart, (in contrast to the static case considered here), the merge window cannot be

too large. Once the clusters have been generated, algorithms other than *AFL* can also be used to merge the streams within a cluster. An odd-even merging heuristic could prove to be better than AFL when the program positions within a stream are uniformly distributed.

In the above simulations, we assume that no interactions occur within the merging window. If interactivity occurs within a catch-up window, the interactive user would have to be removed from their cluster and the clusters would need to be recomputed by re-invoking the clustering algorithm. We are currently investigating the effects of interactivity on the performance of the EMCL algorithm.

6 Conclusion and Future Work

Dynamic service aggregation schemes are ideally suited for resource reclamation in continuous media servers running interactive applications such as video-on-demand because they continuously seek to reclaim resources without using a large buffer space or introducing latencies. In this paper we demonstrated the viability of dynamic service aggregation using rate-adaptive merging in an interactive VoD system. We addressed several issues relating to service aggregation, server design and channel switching in the client. We demonstrated how a server can aggregate clients by making them switch channels to appropriate multicast groups. We also proposed a conditioning scheme and a metadata insertion scheme which enhance the server performance and help the client in recovery from packet losses. Finally, we demonstrated a prototype of our system and how the system would perform under conditions of a large-scale deployment. These results indicate that our clustering algorithm yields approximately a 25% increase in performance over existing approaches.

We are currently extending this work to understand the effects of varying degrees of user interaction on our clustering schemes, and plan to investigate issues related to bandwidth utilization during the clustering process. We are also developing improved merging algorithms involving both rate adaptation and content insertion that account for constraints on dosages of secondary content. These are being considered in conjunction with the use of pricing policies to influence user behavior.

Acknowledgements

We thank L. Kuczynski for her work on the early versions of the server.

References

[1] T.D.C. Little and D. Venkatesh, "Prospects for interactive video-on-demand," *IEEE Multimedia* **1**(3), pp. 14–24, 1994.

[2] A. Dan, D.M. Dias, R. Mukherjee, D. Sitaram, and R. Tewari, "Buffering and caching in large-scale video servers," in *Proc. Compcon*, pp. 217–224, March 1995.

[3] K.C. Almeroth and M.H. Ammar, "On the use of multicast delivery to provide a scalable and interactive video-on-demand service," *IEEE Journal on Selected Areas in Communication* **14**, pp. 1110–1122, August 1996.

[4] D. Venkatesh and T.D.C. Little, "Dynamic service aggregation for efficient use of resources in interactive video delivery," in *Lecture Notes in Computer Science – Proc. 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, T.D.C. Little and R. Gusella, Eds., Vol. 1018, pp. 113–116, Springer-Verlag, November 1995.

[5] R. Krishnan, D. Venkatesh, and T.D.C. Little, "A failure and overload tolerance mechanism for continuous media servers," in *Proc. ACM Multimedia '97*, November 1997.

[6] W.D. Sincoskie, "System architecture for a large scale video on demand service," *Computer Networks and ISDN systems* **22**, pp. 155–162, 1991.

[7] L. Golubchik, J.C.S. Lui, and R.R. Muntz, "Reducing I/O demand in video-on-demand storage servers," in *Proc. SIGMETRICS'95/Performance '95*, pp. 25–36, May 1995.

[8] "ISO/IEC IS 11172, Information Technology - Coding of Moving Pictures and Associated Audio for Digital Storage Media up to about 1.5 Mbit/s," 1993.

[9] R. Krishnan and T.D.C. Little, "Service aggregation through a novel rate adaptation technique using a single storage format," in *Proc. 7th Intl. Workshop on Network and Operating System Support for Digital Audio and Video*, May 1997.

[10] "Microsoft Activemovie Standard." http://www.microsoft.com/developer/tech/amov1doc.

[11] "Java Media Framework." http://java.sun.com/products/java-media/jmf.

[12] "Intel Media for Java." http://developer.intel.com/ial/jmedia.

[13] "MpegTV." http://www.mpegtv.com.

[14] "MPEG-1 System Stream Software Player." http://hulk.bu.edu/pubs/software.html.