Multimedia Storage Servers: A Tutorial

D. James Gemmell

Department of Computer Science,
Simon Fraser University
E-mail: gemmell@cs.sfu.ca

Harrick M. Vin

Department of Computer Sciences,
The University of Texas at Austin
E-mail: vin@cs.utexas.edu

Dilip D. Kandlur

IBM T.J. Watson Research Center
E-mail: kandlur@watson.ibm.com

P. Venkat Rangan

Department of Computer Science and Engineering,
University of California at San Diego
E-mail: venkat@cs.ucsd.edu

Lawrence A. Rowe

Department of Electrical Engineering and Computer Science,
University of California at Berkeley
E-mail: larry@cs.berkeley.edu

Abstract

Multimedia storage servers provide access to multimedia objects including text, images, audio, and video. The design of such servers fundamentally differs from conventional servers due to: (1) the real-time storage and retrieval requirements, as well as (2) the large storage space and data transfer rate requirements of digital multimedia. In this paper, we present an overview of the architectures and algorithms required for designing digital multimedia storage servers.

Figure 1: Data flow for a multimedia network server

1 Introduction

Recent advances in computing and communication technologies have made it feasible and economically viable to provide on-line access to a variety of information sources such as books, periodicals, images, video clips, and scientific data. The architecture of such services consists of *multimedia storage servers* that are connected to client sites via high-speed networks (see figure 1). Clients of such a service are permitted to retrieve multimedia objects from the server for real-time playback at their respective sites. Furthermore, the retrieval may be interactive, in the sense that clients may stop, pause, resume, and even record and edit the media information if they have permission to do so.

The design of such multimedia services differs significantly from traditional text/numeric storage and retrieval services due to two fundamental characteristics of digital audio and video:

- *Real-time storage and retrieval*: Media such as audio and video are often referred to as "continuous" media (CM), because they consist of a sequence of media quanta (such as video frames or audio samples) which convey meaning only when presented continuously in time. This is in contrast to a textual object, for which spatial continuity is sufficient. Furthermore, a multimedia object, in general, may consist of several media components whose playback is required to be temporally coordinated.

- *Large data transfer rate and storage space requirement*: Playback of Digital video and audio consumes data at a very high rate (see Table 1). Thus, a multimedia service must provide efficient mechanisms for storing, retrieving and manipulating data in large quantities at high speeds.

Consequently, the critical components in the design of such multimedia services are: (1) multimedia storage servers that support continuous retrieval of media information from the storage subsystem, and (2) network subsystems that ensure synchronous and timely delivery of media information to the display sites. In this paper, we will focus only on the design of multimedia storage servers and we assume a network subsystem (or transmission channel) capable of delivering the CM information according to its real time specifications[1].

[1]Note that the "network" may be phone lines which clients can dial in to for access to the server, in which case simply providing lines of sufficient bandwidth satisfies the networking requirement

Media Type (specifications)	Data Rate
Voice quality audio (1 channel, 8 bit samples at 8kHz)	64 Kbits/sec
MPEG encoded audio (equivalent to CD quality)	384 Kbits/sec
CD quality audio (2 channels, 16 bit samples at 44.1 kHz)	1.4 Mbits/sec
MPEG-2 encoded video	0.42 MBytes/sec
NTSC quality video (640 X 480 pixels, 24 bits/pixel)	27 MBytes/sec
HDTV quality video (1280 X 720 pixels/frame, 24 bits/pixel)	81 MBytes/sec

Table 1: Storage space requirements for uncompressed digital multimedia data

The main goal of this paper is to provide an overview of the various issues involved in designing a digital multimedia storage server, and present algorithms for addressing the specific storage and retrieval requirements of digital multimedia. Specifically, to address the real-time recording and playback requirements, we discuss a set of disk scheduling algorithms that are optimized for retrieving multimedia information, as well as *admission control* algorithms which a multimedia server may employ to determine whether a new client can be admitted without violating the real-time requirements of the clients already being serviced (Section 2). To manage the large storage space requirements of multimedia data, we examine techniques for efficient placement of media information on individual disks, large disk arrays, as well as hierarchies of storage devices (Section 3). Additionally, we describe some of the data structures that a multimedia file system may need to maintain so as to facilitate random access as well as support efficient editing operations (Section 4).

2 Ensuring Continuous Recording and Retrieval of Digital Multimedia

Digitization of video yields a sequence of frames and that of audio yields a sequence of samples. We refer to a sequence of continuously recorded video frames or audio samples as a *stream*. Since media quanta, such as video frames or audio samples, convey meaning only when presented continuously in time, a multimedia server must ensure that the recording and playback of each media stream proceeds at its real-time rate. Specifically, during recording, a multimedia server must continuously store the data produced by an input device (e.g., microphone, camera, etc.) so as to prevent buffer overruns at the device. During playback, on the other hand, the server must retrieve data from the disk at a rate which ensures that an output device (e.g., speaker, video display) consuming the data does not starve. Although semantically different, both of these operations have been shown to be mathematically equivalent with respect to their real-time performance requirements [1]. Consequently, for the sake of clarity, we will

Figure 2: Ensuring continuous retrieval of media stream from disk

only discuss techniques for retrieving media information from disk for real-time playback. Analysis for real-time recording can be carried out similarly.

Continuous playback of a media stream consists of a sequence of periodic tasks with deadlines, where tasks correspond to retrievals of media blocks from disk and deadlines correspond to the scheduled playback times. Although it is possible to conceive of systems that would fetch media quanta from the storage system just in time to be played, in practice the retrieval is likely to be bursty. Consequently, information retrieved from the disk may have to be buffered prior to playback.

Therefore, the challenge for the server is to keep enough data in stream buffers at all times so as to ensure that the playback processes do not starve [6]. In the simplest case, continuous playback can be ensured by buffering the entire stream prior to initiating the playback. Such a scheme, however, will require very large buffer space and may also yield a very large latency for initiating playback. Consequently, the problem of efficiently servicing a single stream becomes one of preventing starvation while at the same time minimizing buffer space requirement and initiation latency (See Figure 2). It has been shown that these two minimization problems are, in fact, one and the same - minimizing one will minimize the other [6]. Furthermore, since the data transfer rates of disks are significantly higher than the real-time data rate of a single stream (e.g., the maximum throughput of modern disks is of the order of 3-4 MBytes/s, while that of an MPEG-2 encoded video stream is 0.42 MBytes/s, and uncompressed CD-quality stereo audio is about 0.2 MBytes/s), employing modest amount of buffering will enable conventional file and operating systems to support continuous storage and retrieval of isolated media streams.

In practice, however, a multimedia server has to process requests from several clients simultaneously. In the best scenario, all the clients will request the retrieval of the same media stream, in which case, the multimedia server needs only to retrieve the stream once from the disk and then multicast it to all the clients. However, more often than not, different clients will request the retrieval of different streams; and even when the same stream is being requested by multiple clients (such as a popular movie), requests may arrive at arbitrary intervals while the stream is already being serviced. Thus, each client may be viewing a different part of the movie at the same time.

A simple mechanism to guarantee that the real-time requirements of all the clients are met is to dedicate a disk head to each stream, and then treat each disk head as a single stream system. This, however, limits the total number of streams to the number of disk heads. In general, since the data transfer rate of disks are significantly higher than the real-time data rate of a single stream, the number of streams that can be serviced simultaneously can be significantly increased by multiplexing a disk head among several streams. However, in doing so, the server must ensure that the continuous playback requirements of all the streams are met. To achieve this, the server must carefully schedule disk requests so that no stream starves. Furthermore, it must ensure that it does not admit so many clients as to make such disk scheduling impossible.

2.1 Disk Scheduling Algorithms

Traditionally, a variety of disk scheduling algorithms (e.g., first come first served (FCFS), shortest seek time first (SSTF), SCAN, etc.) have been employed by servers to reduce the seek time and rotational latency, to achieve a high throughput, or to provide fair access to each client. The addition of real-time constraints, however, make the direct application of traditional disk scheduling algorithms to multimedia servers inadequate.

The best known algorithm for real-time scheduling of tasks with deadlines is the *Earliest Deadline First* (EDF) algorithm. In this algorithm, after accessing a media block from disk, the media block with the earliest deadline is scheduled for retrieval. Scheduling of the disk head based solely on the EDF policy, however, is likely to yield excessive seek time and rotational latency. Hence, poor utilization of the server resources is to be expected with EDF.

One variant of this basic algorithm combines SCAN with EDF, and is referred to as the SCAN-EDF scheduling algorithm [10]. SCAN operates by "scanning" the disk head back and forth across the surface of the disk, retrieving a requested block as the head passes over it. By limiting the amount of "back-tracking" that the disk head does, SCAN can achieve a significant reduction in seek latencies. SCAN-EDF services the requests with earliest deadlines first, just like EDF. However, if several requests have the same deadline, then their respective blocks are accessed using the SCAN disk scheduling algorithm. Clearly, the effectiveness of the SCAN-EDF technique is dependent on how many requests have the same deadline. If deadlines for media block retrieval are batched together (e.g. by only initiating playbacks of media strands at certain intervals) then SCAN-EDF effectively reduces to SCAN.

SCAN-EDF is distinctive among CM disk scheduling approaches because it does not intrinsically batch requests together. All other approaches typically process requests in *rounds*. During each round, the multimedia server can retrieve a sequence of media blocks for each stream. [2] Processing requests in rounds is more than just a convenience; utlizing rounds exploits the periodic nature of CM playback.

Within each round, a disk scheduling algorithm still must be selected. The simplest of all such techniques is the *round-robin* scheduling algorithm, which services the clients in turn, using a fixed

[2]Note that the sequence may be of arbitrary length - even zero. Processing in rounds is not necessarily as rigid as it may seem at first glance

order that does not vary from one round to the next. However, the major drawback of round robin scheduling is that it, like EDF, does not exploit the relative positions of the media blocks being retrieved during a round. For this reason, data placement algorithms that inherently reduce latencies are sometime used in conjunction with round-robin scheduling.

To reduce the latencies in each round, it is possible to simply apply the SCAN algorithm. For CM servers, minor alterations to SCAN can lead to minimization of the seek latencies, and hence the round length [5]. However, minimization of the round length is not the only issue for the CM server. Notice that in the case of the round-robin algorithm, the order in which clients are serviced is fixed across rounds. Therefore, the maximum separation between the retrieval times of successive requests of a client is bounded by the duration of a round. However, using SCAN, the relative order for servicing clients is based solely on the placement of blocks being retrieved, so it is possible for a client to receive service at the beginning of a round and then at the end of the next round.

This difference has some implications in terms of playback initiation delay and buffer requirements. For round-robin, it is possible to initiate playback immediately after all blocks from the first request have been retrieved. With SCAN, however, playback must wait until the end of the round. To prevent starvation, round-robin needs enough buffer space to satisfy consumption for the duration of one round, while SCAN needs enough buffer space to satisfy consumption during approximately two rounds. However, SCAN's rounds will be shorter, so there is a trade-off between round length and the number of rounds between successive service.

To exploit this tradeoff, a generalization, known as the *Grouped Sweeping Scheme* (GSS) partitions each round into a number of groups. Each client is assigned to a certain group, and the groups are serviced in a fixed order in each round. For each group, the SCAN disk scheduling algorithm is employed. If all the clients are assigned to the same group, GSS reduces to SCAN. On the other hand, if each client is assigned to its own unique group, GSS degenerates to round-robin. By optimally deriving the number of groups, the server can balance the reduction of round length against the number of rounds between successive service (see Figure 3).

2.2 Reading and Buffering Requirements

As mentioned above, nearly all approaches to multi-stream CM retrieval process requests in rounds. Another nearly universal practice is to ensure that production keeps up with consumption in each round. During each round, the amount of data retrieved for a stream is at least as much as will be consumed by the playback of the stream. This means that on a round-by-round basis, the production of data never falls behind the consumption, and there is never a net decrease in the amount of buffered data. Algorithms having this property have been referred to as *workahead-augmenting* [1] or *buffer-conserving* [5].

It is also conceivable that an algorithm may be developed which proceeds in rounds, but is not buffer-conserving. Such an algorithm would allow production to fall behind consumption in one round, and then make up for it in a later round. However, this would necessarily be more complex. Furthermore, while buffer-conservation is not a necessary condition for preventing starvation, it can be used as a sufficient condition. For instance, before initiating playback, if enough data is pre-fetched so as to meet the consumption requirements of the longest possible round, and if each round thereafter is buffer-conserving, then it is clear that starvation is impossible.

To ensure continuous playback of media streams, the number of blocks retrieved for each client during a round must contain sufficient information so as not to starve the display for the entire duration of a round. The determination of the amount of read-ahead that ensure continuous playback requires

Figure 3: Trade-off between round length and time between service for SCAN, Round-robin, and GSS

the knowledge of the maximum duration of a round. Since the duration of a round is governed by the total time spent in retrieving media blocks from disk (also referred to as *service time*), it is critically dependent on the number of media blocks being accessed during each round as well as the disk scheduling algorithm.

In the simplest case, the server may retrieve exactly the same number of media blocks for each stream. In such a scenario, the duration of a round will be governed by the request with the maximum consumption rate. This is likely to be sub-optimal, because whereas the stream with the maximum consumption rate will have retrieved exactly the number of media blocks it needs for the duration of a service round, other streams whose consumption rates are smaller will have retrieved more media blocks than they need in each service round. Consequently, by reducing the number of media blocks retrieved per service round for such streams, it should be possible to accommodate more streams. In fact, it can be shown that in order to enable a multimedia server to service the maximum number of streams simultaneously, the number of blocks retrieved for each stream during each round should be proportional to its consumption rate requirements [1, 6, 7, 13].

Naturally, these reading requirements dictate that the server manage its buffers in such a way that there is always sufficient free space for the next reads to be performed. On a per-stream basis, the most suitable model for the buffer is a first in first out (FIFO) queue. Using a FIFO, contiguous files, and round-robin scheduling, the buffer can be approximately the same size as the maximum required read. In this case, each stream FIFO can simply be “topped up” in each round (i.e. enough is read to fill the FIFO). In contrast, a SCAN strategy would require at least a double-buffered scheme, with each buffer being the size of a maximum read. This is because SCAN may schedule the reads for a stream such that the stream is serviced last in one round and first in the next, i.e. back-to-back. If a buffer is not completely empty when it is time for reading, the "topping-up" strategy can still be used.

However, when files are not stored contiguously, a "topping-up" strategy can add extra intra-file seeks (see section 3.1), as the amount read will vary with the free space in the buffer. One way around this would be to use three logical disk-block sized buffers. With three buffers, the only time that a whole block could not be read would be when at least two were full, in which case there is already enough buffered so that reading is not necessary until the next round. Solutions are possible with less than three buffers [5].

2.3 Admission Control Algorithms

Given the real-time performance requirements of each client, a multimedia server must employ admission control algorithms to determine whether a new client can be admitted without violating the performance requirements of the clients already being serviced. So far we have assumed that the performance requirements of a client includes meeting all real time deadlines. However, some applications may be able to tolerate some missed deadlines. For example, a few lost video frames, or the occasional pop in the audio may be tolerable in some cases - especially if such tolerance is rewarded with a reduced cost of service. Furthermore, in order to guarantee that all real time deadlines are met, worst case assumptions must be made regarding seek and rotational latencies. In reality, the seek time and rotational latency incurred may be much less than the worst case. Hence, a multimedia server may be able to accommodate additional clients by employing an admission control algorithm that exploits the statistical variation in the access times of media blocks from disk (or statistical variations in compression ratios, where applicable).

In admitting clients, CM servers can offer three broad categories of *quality of service* (QoS):

1. *Deterministic:* all deadlines are guaranteed to be met. For this level of service the admission control algorithm considers worst-case scenarios in admitting new clients.

2. *Statistical:* deadlines are guaranteed to be met with a certain probability. For example, a client may subscribe to a service that guarantees that 90% of deadlines will be met over an interval. To provide such guarantees, admission control algorithms must consider statistical behavior of the system while admitting new clients.

3. *Background:* no guarantees are given for meeting deadlines. The server schedules such accesses only when there is time left over after servicing all guaranteed and statistical clients.

To implement deterministic service, resources are reserved in worst case fashion for each stream admitted. For processing of requests in rounds, this is usually a matter of checking whether increasing the round length by servicing another client will cause the amount buffered for existing clients to be inadequate to prevent starvation [1, 13]. Some schemes dynamically change the client buffer spaces based on the current round length. Alternately, all client buffer spaces may be allocated assuming a maximum round length, and for admission the new round length only needs to be compared to the maximum [5].

Implementing statistical service would proceed as with deterministic service, but instead of using worst case values in computing the change to round length, some statistical distributions would be used. For instance, instead of using a worst case rotational delay value, an average value may be used, which can be expected some percent of the time based on a random distribution of rotational delays. In servicing clients during a round, deterministic clients must be ensured service before any statistical clients, and all statistical clients must be ensured service before any background clients. Some algorithm must be implemented to ensure that when deadlines must be missed the same clients are not dropped each time, i.e. that missed deadlines are distributed in a fair way.

Providing statistical service guarantees is essential not only due to the variation in the seek time and rotational latency, but also due to the variation in the data transfer requirements of compressed media streams. To provide statistical service guarantees, a server could employ precise traffic characterizations, rather than the worst-case or the average-case values. It is also possible that when variable rate data is stored, a complete and accurate description of the rate changes could be computed, so that the server could use the information during playback to reserve only the required amount of server resources.

For background and statistical traffic, there are different strategies for dealing with missed deadlines. For example, it may be desirable not to skip any blocks of data so as to ensure that the information received is intelligible. However, such a policy would increase the effective playback duration of media streams. On the other hand, if playback of multiple media streams are being temporally coordinated, it may be preferable to drop media blocks so as to maintain the playback time-aligned.[3] A significant departure from these simplistic schemes are techniques which dynamically vary the resolution levels so as to adjust to the overloaded system state. For instance, the quality of audio being delivered can be degraded simply by transmitting only the higher order bits. In a similar way, some compression schemes can be made *scalable*, that is, they encode the data so that subsets of the media stream can be extracted and decoded to achieve lower resolution output. In general, techniques used to vary resolution to deal with missed deadlines will be very similar to those used for implementing fast forward (see Section 4.3).

3 Managing the Storage Space Requirement of Digital Multimedia

A storage server must divide video and audio streams into *blocks* while storing them on disks. Each such data block may occupy several physical disk blocks. In this section, we will first describe models for storing digital continuous media on individual disks, and then discuss the effects of utilizing disk arrays as well as storage hierarchies.

3.1 Placement of Data Blocks

In the broadest terms, the blocks belonging to a file may be stored contiguously or scattered about the storage device. Contiguous files are simple to implement, but they are subject to fragmentation problems and can necessitate enormous copying overheads during insertions and deletions in order to maintain contiguity. In contrast, scattered placements avoid fragmentation problems and copying overheads. Thus, contiguous layouts may be useful in read-only systems (e.g. video on demand) but are not viable for flexible, read-write servers.

With regard to continuous media, the choice between contiguous and scattered files relates primarily to intra-file seeks. When reading from a contiguous file, only one initial seek is required to position

[3]It may be that the choice between these methods is best left up to the user.

the disk head at the start of the data. No additional seeks are required, as the data is contiguous. However, when reading several blocks in a scattered file there may be a seek incurred for each block read. Furthermore, even when reading just a small amount of data, it may be possible that half of the data is stored in one block and the other half in the next block. So even with small reads it is possible to incur intra-file seeks by crossing block boundaries.

Intra-file seeks can be avoided in scattered layouts if the amount read for a stream always evenly divides a block. One logical approach to achieve this result is to select a block size that is sufficiently large, and to read one block in each round. There are several advantages to this technique, especially for large video servers. It improves the disk throughput substantially, and consequently increases the number of streams that can be served by the disk. Furthermore, since a file system has to maintain indices for each media block, choosing a large block size also yields a reduction in the overhead for maintaining indices. In fact, this may permit a server to store the indices in memory during the playback of a media stream.

If more than one block would be required to prevent starvation prior to the next read, then intra-file seeks are a necessity. Instead of avoiding intra-file seeks, another approach is to attempt to reduce them to a reasonable bound. This is referred to as the *constrained placement* approach [9]. Constrained placement systems ensure that the separation between successive blocks of a file are bounded. However, it does not do this for each pair of successive blocks, but only on average over a finite sequence of blocks (see figure 4). Thus, the latency due to intra-file seeks is constrained. [4]

Constrained placement is particularly attractive when the block size must be small (e.g., when utilizing a conventional file system with block sizes tailored for text). However, implementation of such a system may require elaborate algorithms to ensure that the separation between blocks conforms to the required constraints. Furthermore, for constrained latency to yield its full benefits, the scheduling algorithm must retrieve all the blocks for a given stream at once before switching to any other stream. If an algorithm like SCAN is used, which orders blocks regardless of the stream they belong to, then the impact of constrained placement is marginal [5].

An approach to the problem of seek reduction comes from the adaptation of "log-structured" file systems [7]. When modifying blocks of data, log-structured systems do not store modified blocks in their original positions. Instead, all writes for all clients are performed sequentially in a large contiguous free space (see figure 4). Therefore, instead of requiring a seek for each client writing, and possibly intra-file seeks, only one seek is required prior to a batch of writes. This leads to a dramatic performance improvement during recording. However, such an approach does not guarantee any improvement in the playback performance, and is more complex to implement (because modified blocks may change position). Consequently, log-structured file systems are most suitable for multimedia servers which support extensive editing operations, and are inappropriate for systems which are primarily read-only (for example video-on-demand servers, which could likely implement writes in non-real time).

There are special placement considerations when the media is encoded using variable bit rate compression. In this case, conventional fixed-sized clusters would correspond to varying amounts of time, depending on the compression achieved. Alternately, the system may store data in clusters which correspond to a fixed amount of time, with a variable cluster size. Furthermore, compressed media quanta may not correspond to an even number of disk sectors, which raises questions about "packing"

[4]A constrained scheme is also briefly discussed in [1].

Figure 4: Contiguous, Constrained and Log-Structured Placement Schemes

Figure 5: Striped data accessed in parallel

data [6]. With scalable compression, placement of data must be carefully managed to ensure that the extraction of low-resolution subsets is efficient.

3.2 Multiple Disk Configurations

So far, we have considered storage on a single disk. If an entire multimedia file is stored on a single disk, the number of concurrent accesses to that file are limited by the throughput of that disk. One approach to overcome this limitation is to maintain multiple copies of the file on different disks. However, this approach is expensive because it requires additional storage space. A more effective approach to this problem is to scatter the multimedia file across multiple disks. This scattering can be achieved using two techniques: "data striping" and "data interleaving".

Data Striping RAID (redundant array of inexpensive disks) technology has popularized the use of parallel access to an array of disks. Under the RAID scheme, data is "striped" across each disk (see figure 5). Physical sector 1 of each disk is accessed in parallel as a large logical sector 1. Physical sector 2 of each disk is accessed as logical sector 2, and so on. In this configuration, the disks in the set are *spindle synchronized* and they operate in lock-step parallel mode. Because accesses are performed in parallel, the access time is the same for a logical block and a physical block. Therefore, the effective transfer rate is increased by the number of drives involved.

With their increased effective transfer rate, disk arrays are a good solution to the problem of the high bandwidth requirements of continuous media. Furthermore, with disk arrays, a physical block is accessed from each drive in parallel, yielding an effective block size that increases with the number of drives in the array. Rather than being a problem, as they may be to conventional systems, large block sizes are usually desirable for continuous media file systems.

Observe, however, that while striping can improve the effective transfer rate, it cannot improve the seek time and rotational latency incurred during retrieval. Hence, the throughput of each disk in the array is still determined by the ratio of the useful read time to the total (read + seek) time. As with the single disk configuration, the throughput of the disk may be increased by increasing the size of the physical block. However, this would result in an increase in the logical block size, and consequently increase the startup delays and the buffer space requirements for the stream.

Round	Disk 1	Disk 2	Disk 3
1	File A, block 1	File B, block 1	File C, block 1
2	File C, block 2	File A, block 2	File B, block 2
3	File B, block 3	File C, block 3	File A, block 3
4	File A, block 4	File B, block 4	File C, block 4

Table 2: Reading interleaved data (method 2).

Data Interleaving In this scheme, the blocks of the media file are stored interleaved across the set of disks. Successive blocks of the file are stored on different disks. A simple interleave pattern is obtained by storing the blocks in a cyclic manner across a set of N disks. In this scheme, the disks in the set are not spindle synchronized and they operate independently.

With this organization, there are two possible methods of data retrieval. One method of retrieval follows the data striping model in which for each stream, in every round, one block is retrieved from each disk in the set. This retrieval method ensures a balanced load for the disks, but requires more buffer space per stream. In the other retrieval method, for a given stream, in each round, data is extracted from one of the disks in the set (see table 2). Hence, the data retrieval for the stream cycles through the set of disks in N successive rounds. In order to maximize the throughput of the N disks, it is necessary to ensure that in each round the retrieval load is balanced across the disks. Given that each stream cycles through the set, this load balancing can be achieved by “staggering” the streams. With staggering, all the streams still have the same round length, but each stream considers the round to begin at a different time, so that their requests are staggered rather than simultaneous.

A combination of data striping and data interleaving can be used to scatter the media file across a large number of disks attached to a networked cluster of server machines. This technique makes it possible to construct a scalable video server that can serve a large number of streams from a single copy of the media file. Moreover, redundancy techniques can be applied to the media file to increase availability and throughput.

3.3 Utilizing Storage Hierarchies

The preceding discussion has focused on fixed disks as the storage medium for the multimedia server. Although sufficient for providing efficient access to a small number of video streams (e.g., 25-50 most popular titles maintained by a video rental store), the high cost per gigabyte of storage makes such magnetic disk-based server architectures ineffective for large-scale servers. In fact, the desire for sharing and providing on-line access to a wide variety of video sequences indicates that large-scale multimedia servers must utilize very large tertiary storage devices (e.g., tape and optical jukeboxes). These devices are highly cost-effective and provide very large storage capacities by utilizing robotic arms to serve a large number of removable tapes or disks to a small number of reading devices. Because of these long seek and swap times, however, they are poor at performing random access within a video stream. Moreover, they can support only a single playback at a time on each reader. Consequently, they are inappropriate for direct video playback. Thus, a large-scale, cost-effective multimedia server will be required to utilize tertiary storage devices (such as tape jukeboxes) to maintain a large number of video streams, and then achieve high performance and scalability through magnetic disk-based servers.

	Magnetic Disk	Optical Disk	Low-End Tape	High-End Tape
Capacity	9 GB	200GB	500 GB	10TB
Mount Time	0 sec	20 sec	60 sec	90 sec
Transfer Rate	2MB/s	300 KB/s	100 KB/s	1,000 KB/s
Cost	$5000	$50,000	$50,000	$500,000 - 1,000,000
Cost/GB	$555/GB	$125/GB	$100/GB	$50/GB

Table 3: Tertiary storage devices.

In the simplest case, such a hierarchical storage manager may utilize fast magnetic disks to cache frequently accessed data. In such a scenario, there are several alternatives for managing the disk system. It may be used as a staging area (cache) for the tertiary storage devices, with entire media streams being moved from the tertiary storage to the disks when they need to be played back. On the other hand, it is also possible to use the disks to only provide storage for the beginning segments of the multimedia streams. These segments may be used to reduce the startup latency and to ensure smooth transitions in the playback [8].

A distributed hierarchical storage management extends this idea by allowing multiple magnetic disk-based caches to be distributed across a network. In such a scenario, if a high percentage of clients access data stored in a local (or nearby) cache, the perceived performance will be sufficient to meet the demands of continuous media. On the other hand, if the user accesses are unpredictable or have poor reference locality, then most accesses will require retrieval of information from tertiary storage devices, thereby significantly degrading the performance. Fortunately, for most video on-demand applications, user accesses are likely to exhibit high locality of reference. A small set of video streams are likely to be popular at any given time, while older or more obscure streams will be seldom accessed. Moreover, for a large class of applications, the access pattern of users may be predictable well in advance. For instance, an instructor may predict that recent class lectures as well as other footage related to topics being discussed in class are likely to be viewed more frequently than the lectures on the previous topic. Finally, since most video streams are likely to be read-only, distributed cache consistency problems do not occur in video on-demand systems.

The architecture of such a distributed hierarchical storage management system will consist of several video storage servers (denoted by VFS) that will act as on-line cache for the information stored permanently on archive servers (denoted by AS) (see Figure 6). In addition to maintaining one or more tertiary storage devices that contain the video streams as well as the corresponding metadata, each AS will also provide an interface that will permit users to query the database to locate pertinent video streams, and then schedule their retrieval [4, 11].

4 Implementing a Multimedia File System

4.1 Interfacing With The Client

Based on their access models, servers can be roughly classified as *file-system-oriented* or *stream-oriented*. A client of a file-system-oriented server sees the multimedia object as a large file and uses file-system operations such as *open*, *close*, *read*, to access the file. It issues read requests to the server

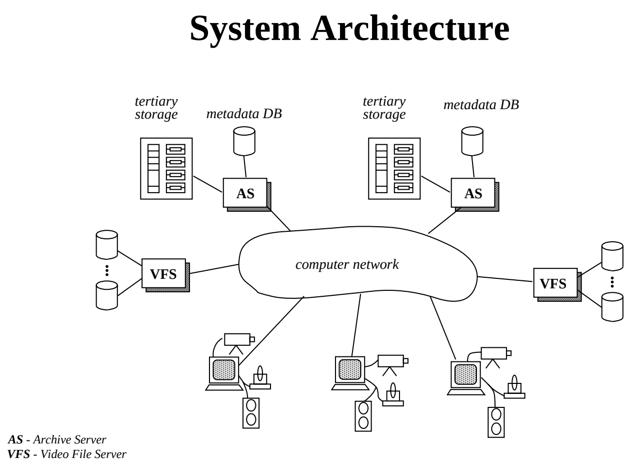

Figure 6: Architecture of a distributed hierarchical video on-demand server

periodically to read data from the file. The server may use the open operation to enforce admission control and initiate pre-fetching of the multimedia file. The server can also do periodic pre-fetching from the disk system into memory buffers to service read requests with a minimum delay. In this model, the client can implement operations such as *pause* and *resume* by simply ceasing to issue read requests. On the other hand, a client of a stream-oriented server issues commands such as *play*, *pause*, and *resume* to the server. The server uses the stream concept to deliver data continuously to the client. After the user initiates playback of the stream, the server periodically sends data to the user at the selected rate without further *read* requests from the user.

Another important issue in the server-client interface (and elsewhere) is the movement of data. Typically, data being transferred from one process (e.g. the server kernel) to another process (e.g. the client) is copied. For CM streams copying is unecessary, takes extra time, and produces extra traffic on the system bus. Because of the high throughput requirements of CM, it is desirable to share memory, or re-map the memory into another address space to avoid copying of data.

4.2 File Structures

A fundamental issue in implementing a file system is to keep track of which disk blocks belong to each file; keeping a map, as it were, of how to travel from block to block in a file. For contiguous files, of course, this is not an issue. For scattered files a number of approaches are possible. In this section, we consider conventional approaches to this problem, and their relative merits for multimedia file systems.

A simple solution for mapping blocks is to use a *linked list*, with each block containing a pointer to the next block in the file. In such a scenario, the file descriptor may need to contain only a pointer to the first block of the stream. A serious limitation of this approach, however, is that random access is highly inefficient as accessing a random block requires accessing all of the previous blocks.

To improve the performance of random access, some conventional file systems (e.g. DOS) have utilized a file allocation table (FAT), with an entry in the table for each block on the disk. Each entry in the table maintains a pointer to the next block of a file. Assuming that the entire FAT is kept in main memory, random access can be very fast. However, it may not be feasible to keep a FAT in main memory for the large file systems expected in multimedia servers.

A FAT contains information about the entire file system, but only a portion of this information relating to files which are currently open is needed. To exploit this, it is possible to store an *index* for each file separately (e.g. I-nodes in UNIX). These indices can be simple lists or hierarchical structures such as binary trees (so as to make the process of searching more efficient). Thus, rapid random access is still possible, but the need to keep the entire FAT in main memory is alleviated.

One possible drawback to this approach occurs when all open file indexes cannot be kept in main memory in their entirety (as is possible with very large CM files [5]). In this case, retrieving a continuous media file will involve retrieving blocks of the index in real time, in addition to the blocks of the file itself. It is true that the index retrieval is much less demanding in terms of bandwidth, but it nonetheless will consume resources. In fact, managing such small bandwidth “streams” may require special algorithms to keep them from using a disproportionate amount of system resources. An obvious way around this is to implement a linked list as well, so that real-time playback can follow the pointers contained in the blocks of data, while random seeks can be achieved quickly through the index without reserving real

[5] For example, a server with a small selection of long videos may have all the videos open for playback at once. Therefore the open indexes would map all of the allocated file space.

time resources. This would add system overhead in keeping both the index and the link pointers up to date, but for applications which perform little editing, such as video on demand, the overhead may be worthwhile. [6]

Finally, since each multimedia object may contain media information in various forms: audio, video, textual, etc., in addition to maintaining file maps for each of the media streams, a multimedia server will be required to maintain characteristics of each multimedia object, such as its creator, length, access rights, and most importantly, inter-media synchronization relationships.

4.3 Operating on Multimedia Objects

The file system must provide facilities for creating, editing, and retrieving multimedia objects. In order to guarantee continuous retrieval, editing operations on multimedia objects, such as insert and delete, may require substantial copying of their component streams. Since the streams can be very large in size, copying can consume significant amount of time and space. In order to minimize the amount of copying involved in editing, the multimedia file system may regard streams as immutable objects, and perform all editing operations on multimedia objects by manipulating pointers to streams. Thus, an edited multimedia object may contain a list of pointers to intervals of streams. Furthermore, many different multimedia objects may share intervals of the same media stream. A media stream, no part of which is referred to by any multimedia object, can be deleted to reclaim its storage space. A garbage collection algorithm such as the one presented by Terry and Swinehart in the Etherphone system [12], which uses a reference count mechanism called *interests*, can be used for this purpose.

Another problem arises when material is cut from or pasted into multimedia files. The simplest approach to such a change in the material is to simply re-write the file from the edited point onwards. Unfortunately, for large multimedia files this may be a very time consuming process. To limit the time involved, it would be desirable to simply insert or delete blocks from the file map. However, if the amount of material being cut/pasted does not correspond to an integral number of blocks, then the file will be left with at least one block which is partially empty. This block may not be able to supply enough data to satisfy real time demands. However, by implementing a scheme where blocks are required to be filled to a certain level, and then spreading out data among adjacent blocks to ensure this fill level, it is possible to perform cuts and pastes in time proportional to the size of the cut/paste, rather than the whole file.

Additionally, a multimedia server must also support interactive control functions such as pause/resume, fast forward (FF) and fast backward (FB). The pause/resume operations pose a significant challenge for buffer management since it interferes with the sharing of a multimedia stream between different viewers [3]. The FF and FB operations can be implemented either by playing back media at a rate higher than normal, or by continuing playback at the normal rate while skipping some data. Since the former approach may yield significant increase in the data rate requirement, its direct implementation may be impractical. The latter approach, on the other hand, may also be complicated by the presence of inter-data dependencies (e.g. in compression schemes that store only differences from previous data).

There are several approaches possible to achieve FF using data skipping. One method is to create a separate, highly compressed (and lossy), file. For example, the MPEG-2 draft standard proposes the creation of special highly compressed 'D' video frames that do not have any inter-frame dependency to

[6]To support fast forward and rewind, it may be necessary to store extra pointers, as the blocks will not be visited in normal sequential order

support video browsing. During retrieval, when FF operation is required, the playback would switch from the normal file, (which may itself be compressed, but still maintains acceptable quality levels) to the highly compressed file. This option is interesting in the fact that it does not require any special storage methods or post-processing of the file. However, it requires additional storage space and, moreover, the resulting output is of poor resolution due to the high compression.

Another approach is to categorize each block as either relevent or irrelevent to fast forward. During normal operation both types of blocks are retrieved and the media stream is reconstructed by recombining the blocks either in the server or in the client station. On the other hand, during FF operation, only the FF blocks are retrieved and transmitted. Scalable compression schemes are readily adapted to this sort of use. A drawback of this approach is that it poses additional overheads for splitting and recombining blocks. Futhermore, with compressions schemes that store differences from previous data, the majority of data will be relevant to FF. For example, the I and P frames of MPEG are much larger than the average frame size. These means that the data rate required during FF operation would be higher than the normal rate.

Chen, Kandlur, and Yu [2] present a different solution for FF operations on MPEG video files. Their method performs block skipping using an intelligent arrangement of blocks (called segments) that takes into account the inter-frame dependencies of the compressed video. During FF operation entire segments of video are skipped, and the viewer sees normal resolution video with gaps. Their solution also addresses the placement and retrieval of blocks on a disk array using block interleaving on the disk array.

5 Concluding Remarks

Multimedia storage servers differ from conventional storage servers to the extent that significant changes in design must be effected. These changes are wide in scope, influencing everything from the selection of storage hardware to the choice of disk scheduling algorithms. This paper provides an introduction to the problems involved in multimedia storage server design and to the various approaches of solving these problems.

6 Acknowledgements

The authors would like to acknowledge the co-ordination and excellent detailed suggestions of Arturo Rodriguez in putting this paper together. This work was supported in part by MPR Teltech Ltd. and the British Columbia Science Council.

References

[1] D. Anderson, Y. Osawa, and R. Govindan. A File System for Continuous Media. *ACM Transactions on Computer Systems*, 10(4):311–337, November 1992.

[2] Ming-Syan Chen, Dilip D. Kandlur, and Philip S. Yu. Support For Fully Interactive Playout in a Disk-Array-Based Video Server. In *Proceedings of the ACM Multimedia'94, San Francisco*, October 1994.

[3] A. Dan, D. Sitaram, and P. Shahabuddin. Dynamic Batching Policies for an On-Demand Video Server. In *Proc. ACM Multimedia'94*, pages 15–24, San Francisco, October 1994. also, to appear in ACM Multimedia Systems.

[4] C. Federighi and L.A. Rowe. The Design and Implementation of the UCB Distributed Video On-Demand System. In *Proceedings of the IS&T/SPIE 1994 International SYmposium on Electronic Imaging: Science and Technology, San Jose*, pages 185–197, February 1994.

[5] D. James Gemmell and Jiawei Han. Multimedia Network File Servers:Multi-Channel Delay Sensitive Data Retrieval. *Multimedia Systems*, 1(6):240–252, 1994.

[6] J. Gemmell and S. Christodoulakis. Principles of Delay Sensitive Multimedia Data Storage and Retrieval. *ACM Transactions on Information Systems*, 10(1):51–90, 1992.

[7] Philip Lougher and Doug Shepherd. The Design of a Storage Server for Continuous Media. *The Computer Journal*, 36(1):32–42, 1993.

[8] T. Mori, K. Nishimura, H. Nakano, and Y. Ishibashi. Video-on-Demand System using Optical Mass Storage System. *Japanese Journal of Applied Physics*, 1(11B):5433–5438, November 1993.

[9] P. Venkat Rangan and Harrick M. Vin. Efficient Storage Techniques for Digital Continuous Multimedia. *IEEE Transactions on Knowledge and Data Engineering*, 5(4):564–573, August 1993.

[10] A.L. Narasimha Reddy and J. Wyllie. I/O Issues in a Multimedia System. *COMPUTER*, 27(3):69–74, 1994.

[11] L.A. Rowe, J. Boreczky, and C. Eads. Indexes for User Access to Large Video Databases. In *Proceedings of the IS&T/SPIE 1994 International SYmposium on Electronic Imaging: Science and Technology, San Jose*, pages 150–161, February 1994.

[12] D.B. Terry and D.C. Swinehart. Managing Stored Voice in the Etherphone System. *ACM Transactions on Computer Systems*, 6(1):3–27, February 1988.

[13] Harrick M. Vin and P. Venkat Rangan. Designing a Multi-User HDTV Storage Server. *IEEE Journal on Selected Areas in Communications*, 11(1):153–164, January 1993.

D. James Gemmell

Jim Gemmell is a Ph.D. candidate at Simon Fraser University. He received his B.Sc. in computer science from Simon Fraser University in 1988, and his M.Sc. in Computer Science from the University of Waterloo in 1990. His research is in the area of delay sensitive multimedia systems, focusing on server storage and retrieval.

He may be contacted at the School of Computer Science, Simon Fraser University, Burnaby, B.C. Canada, V5A 1S6, or by email at gemmell@cs.sfu.ca.

Harrick M. Vin

Dr. Harrick Vin received the B. Tech. in Computer Science and Engineering in 1987 from the Indian Institute of Technology, Bombay, India, the M.S. in Computer Science in 1988 from Colorado State University, and the Ph.D. in Computer Science in 1993 from University of California, San Diego. He is currently an Assistant Professor of Computer Science, and the Director of Distributed Multimedia Computing Laboratory at the University of Texas at Austin. His research interests are in the areas of multimedia systems, high-speed networking, mobile computing, and large-scale distributed systems. He has co-authored more than 35 papers in leading journals and conferences in the area of multimedia systems.

Dilip D. Kandlur

Dilip D. Kandlur received the B.Tech. degree in Computer Science and Engineering from the Indian Institute of Technology, Bombay, in 1985. He received the M.S.E. and Ph.D. degrees, also in Computer Science and Engineering, from the University of Michigan, Ann Arbor, in 1987 and 1991 respectively.

From 1987 to 1991 he was a member of the Real-Time Computing Laboratory at the University of Michigan, involved in the design and development of the HARTS experimental real-time system. He developed algorithms for broadcasting, clock synchronization, and time-constrained communication in real-time systems. Since 1991 Dr. Kandlur is a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY. He has worked on the design and implementation of the Multimedia Multiparty Teleconferencing (MMT) System, a high-quality desktop videoconferencing system which has been deployed as part of the CNRI Aurora Gigabit Testbed. His research has focused on several aspects of multimedia systems such as video/audio support for desktop collaboration, multimedia networking, and multimedia storage management.

Dr. Kandlur is a member of the IEEE Computer Society and he has served on the program committees of the IEEE Real-Time Systems Symposium and the SPIE Symposium on Electronic Imaging.

P. Venkat Rangan

Dr. Venkat Rangan founded the Multimedia Laboratory at the University of California, San Diego, where he serves as its director and Associate Professor of Computer Science. Dr. Rangan is well known for his contributions in the areas of multimedia on-demand servers, media synchronization, multimedia communication and Collaboration. Dr. Rangan has numerous publications to his credit, and holds two patents on optimal video on-demand delivery systems.

Dr. Rangan received his Ph.D. in computer science from the University of California at Berkeley in 1988, and the B.Tech in electrical engineering from the Indian Institute of Technology, Madras, India, in 1984, where he received the President of India gold medal. Recently, Dr. Rangan received the NSF National Young Investigator Award. He serves on several program committees and editorial boards, and served as the program chairman of ACM Multimedia 93: the first ACM International conference on Multimedia. Currently, Dr. Rangan serves as an editor-in-chief of the ACM/Springer-Verlag Multimedia Systems journal.

Lawrence A. Rowe

Professor Rowe received a BA in mathematics and a PhD in information and computer science from the University of California at Irvine in 1970 and 1976, respectively. Since 1976 he has been on the faculty at the University of California at Berkeley where he is now a Professor of Electrical Engineering and Computer Science.

He has designed and implemented several programming languages and database application development systems including the Rigel Programming Language, the Forms Application Development System, and the Picasso Graphical User-Interface Development Environment. Many ideas in these systems were integrated into commercial products.

His current research interests are multimedia applications and databases, video conferencing, hypermedia courseware, and audio and video compression. He is the head of the Berkeley Plateau Multimedia Group that developed the Berkeley MPEG video tools (i.e., software decoder, parallel encoder, and utilities), a network playback system, algorithms to compute special effects on compressed images, the Berkeley Distributed Video-on-Demand System, and a desktop video conferencing system.

Professor Rowe has published over fifty papers, organized and chaired several conferences, served on numerous program committees, and was a co-winner of the Best Experimental Paper Award at OOPSLA '91.

Professor Rowe was a co-founder and member of the Board of Directors of Ingres Corporation until the company was sold in 1990. His primarily responsiblity was the user interface and application development tools strategy for the company.