Design of a Variable Bit Rate Continuous Media File Server for an ATM Network

Gerald Neufeld, Dwight Makaroff and Norman Hutchinson
{neufeld,makaroff,hutchinson}@cs.ubc.ca

Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z4
Canada

ABSTRACT

This paper describes the design and implementation of a file server for variable bit rate continuous media. Most continuous media file servers have been designed for constant bit rate streams. We address the problem of building a server where each stream may have a different bit rate and, more importantly, where the bit rate within a single stream may vary considerably. Such servers will be come increasingly more important because of Variable Bit Rate (VBR) compression standards such as is possible with MPEG-2.

1 INTRODUCTION

The motivation for the design of a specialized file server for continuous media such as video and audio is well established .[5,12,15] Data Access patterns, as well as the services provided to the client by a continuous media file server (CMFS) differ considerably from a conventional distributed file service such as NFS. A continuous media client typically transfers large volumes of sequential data. As well, the resource requirements of the network and server itself differ considerably. In order to guarantee continuity, the allocation of network resources such as bandwidth must be guaranteed. Similarly, the availability of resources at the server, such as processor cycles, RAM, and disk bandwidth, must be guaranteed to properly service the client.

In order to provide such guarantees at the server, some assumptions must be made about the resources available, such as the bandwidth of the disk and the resource requirements of the client. Most of the prior research in this area has made simplifying assumptions about these two aspects.[3,15] In reality, however, a continuous media file server must be able to handle many different transfer rates concurrently. Some recent work has been done that attempts to analyze such servers.[4] Such analysis still assumes, however, that each individual stream has a constant bit rate (CBR). Compression methods such as motion JPEG or MPEG-2 produce streams whose bit rates vary considerably. It is, of course, possible to use these schemes to produce a constant bit rate stream but then either too many resources are required or the quality of the stream is impaired. Computers and data networks are well-suited to handle bursty traffic. In particular, much effort has gone into making ATM networks capable of handling the bursty traffic characteristic of data networks. It seems reasonable to design a file service which can explicitly accommodate such variation in resource requirements and thereby increase the number of simultaneous streams supportable. This is similar to the approach labeled *planned bandwidth allocation* by Dey-Sircar.[6]

The CMFS described here addresses two major design issues:

Synchronization support A client typically requires multiple concurrent media streams. For example, a session may include video, audio and captioned text. Standards exist that allow all three of these to be combined into a single stream, but this severely limits the flexibility of the resulting system. For instance, a user may wish to display the same video together with a one of a variety of different audio streams depending on the preferred language (English, French, etc.). As well, the streams may come from different servers. In order to support such flexibility the server should not be restricted to a single media syntax such as MPEG-2, or even a limited selection of syntaxes, but should support any syntax a client wishes to store. Since such a server cannot assume that it understands the syntax of any media stream, it must provide a suitable abstraction for time such as media units per second. Section 3 presents the abstract playback interface that the server exports to its client.

VBR Admission Control The I/O scheduling is based on variable bit rate streams rather than constant bit rates. This permits the scheduling of streams that have been compressed using VBR schemes such as motion JPEG and MPEG-2. In addition to the variation inherent in the encoding of I, B and P frames, variation also arises on granularities of a second or more across video scene boundaries. Section 4 describes both the admissions control algorithm and the stream scheduler for variable bit rate traffic.

The next three sections describe the architecture of the system, how clients synchronize multiple streams and the admissions control/scheduling algorithm respectively. This is followed by a section on related work. We end the paper with some conclusions, a report on the status of our server, and a description of future work.

2 ARCHITECTURE

The design of the file server is based on a controller node and a set of server nodes, each with a processor and disk storage on multiple local SCSI-2 Fast/Wide buses. Each node is connected to an ATM network for delivering continuous media data to the client systems (see Figure 1).

A sufficient number of disk drives are attached to the SCSI buses to provide the required bandwidth. The disks can be striped along a single SCSI bus (usually to a maximum of four disks) or striped across multiple SCSI buses.

Multiple server nodes can be configured together to increase the capacity of the service. Since each server node is independent of the others, any number can be added subject only to the capacity of the network switch configuration. Configurations can consist of either completely independent computers, or processor cards interconnected via an I/O bus such as VME. In the latter case, the nodes can communicate over the I/O bus rather than the ATM network. In either case, the initial client *open* request goes from the client to the controller node. This node determines which of the server nodes has the requested stream and forwards the request to the appropriate server node along with a detailed description of the bandwidth for each display unit required by the stream, known as its *playout vector* (See Section 4) . Communication from then on takes place directly between a particular server node and the client.

In our configuration, the ATM network consists of NewBridge switches, connecting the the clients and server nodes with 100 Mb/s multimode fibre. The nodes are IBM RS/6000s (350) running AIX. The local bus is a micro channel operating at a peak of 80 MBytes per second. We currently have four disks attached to a single SCSI-2 Fast/Wide bus. Software striping is done across these four disks. Currently we have two server nodes.

Client applications have been written for both Sun workstations and IBM RS/6000s. On both platforms, the client can perform hardware JPEG decoding (using a Parallax card on the Sun and an IBM JPEG card on the RS/6000s). Using hardware decoding, frame rates of 30 frames per second can be supported at a resolution of 640x480. A client application that performs software

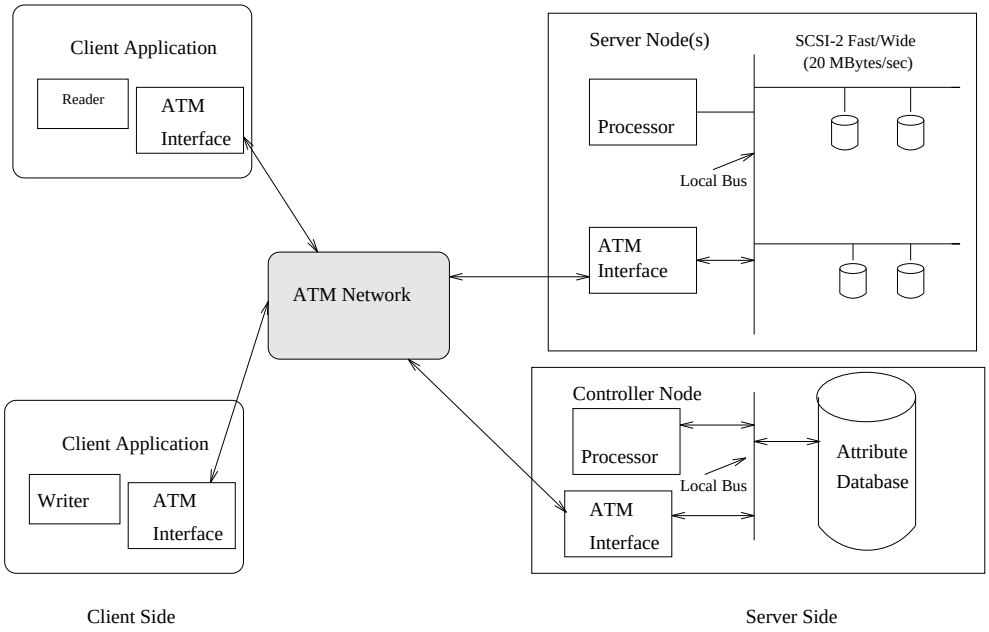

Figure 1: Organization of System

MPEG decoding is supported although at a much reduced frame rate and resolution. These clients can playback audio, video and/or text in a synchronized fashion on an Xterminal (or NTSC video display for IBM's decoding card). Further, uncompressed video display clients are supported, but again at reduced frame rate due to the large amount of data that must be transmitted. Both platforms also have clients that write continuous media objects to the CMFS.

3 SYNCHRONIZATION

Server support for synchronization is required since a client may request multiple media objects, potentially from different servers, which need to be synchronized at the client. For example, if a video object is to be combined with an audio object at the client, it is typically expected that this will be done in such a way that lip-synchronization is achieved. In our design, the video and audio objects may be completely independent – they may even exist on different server nodes. Another example of where synchronization is required at the client is in support of scalable video. In scalable video, a video stream is composed of a base stream and one or more additional streams which can be combined with the base stream to produce higher quality video.[7] Depending on the quality demanded by and bandwidth available to the client, it may require zero or more of the additional streams, each of which is stored as a separate object at the server. Since the server does not have knowledge of the video syntax, it is not possible for the server to combine these streams; therefore, it must provide a mechanism which permits multiple streams to arrive at the client sufficiently close to the same time to allow the streams to be combined at the client.

Media objects are composed of sequences of data objects called "segmented objects" or simply "segments." The duration of a segment is determined at the time the media object is created and may depend on the syntax of the media. For example, if the media object is an MPEG stream, a segment could be an I-B-P sequence of frames. For motion JPEG a segment may be an individual frame.

Reading a media object from the server is done via three RPC requests: *open*, *prepare* and *read*. The *open* request is sent as an RPC to the Server Controller. The media object to be read is specified with a universally unique object identifier. The controller determines if the object exists and on which node (or nodes) it resides (objects may migrate from node to node, or be replicated). The request, along with the playout vector is forwarded to the selected node. The node then determines the client's buffering requirements and passes this information along with the transport-level protocol connection open request to the client. We assume that the client is capable of double buffering for the worst case. That is, given a variable bit rate stream, the maximum buffering requirement is the maximum data required in any two successive time intervals. This is the minimum amount of buffering required for the *read* operations to have no external (network and server) latency. The server completes the *open* request and returns the worst-case delay that a *prepare* request will take. No stream data is transmitted over the connection at this time.

When the client is ready to begin receiving the data, it sends a *prepare* request to the server indicating the *positions* within the media object at which the server should start and end the playout (specified as segment identifiers), the *speed* of the playout, and whether each segment should be sent or whether segments should be *skipped*. The latter two parameters provide the capability to play back the stream at speeds other than that at which it was recorded. The analysis in[5] indicates that a reasonable way by which fast motion display can be done without immense increase in bandwidth is to skip sequences that are a reasonably long (1 second or more) length of time. The user will be able to observe fast forward (or fast rewind). Rewind is indicated by specifying a ending position that is earlier in time than the starting position. The server runs the admissions control algorithm and if successful, schedules the disk manager to begin reading the media blocks for the client. When the first interval of data has been read from the disk and sent to the client, the *prepare* request returns to the client. Since the stream is now scheduled, however, the server continues to read and transmit blocks subject to buffering constraints at the server and client. The client is now ready to read and process the media stream. This is done via *read* requests. *Reads* are local client operations which simply pass the data from the network buffers to the application. Once *prepare* has returned, the client must begin reading within a designated interval of time determined by the buffering allocated at the client and server.

This interface has been designed to support the synchronization of multiple streams, as explained in the following example. Assume that the client must synchronize three streams; one for video, one for audio and one for text. The client issues three *open* requests. Each open returns the maximum time a *prepare* will take for that stream. Based on this information, the client schedules the three *prepares* so that the *read* operations can begin immediately thereafter. This is easily implemented using a real-time threads[8] environment where each thread controls a stream. Each *prepare* may take a different amount of time to return. Since the client cannot start reading the data until all of the *prepare* operations complete, there may be a significant lag time between the completion of the final *prepare* and the earlier ones. During this lag time, the server will continue to schedule and read blocks for each stream. As a result, sufficient buffer space must exist at either the client or the server to accommodate this lag. In order to determine whether sufficient resources exist, the client specifies the maximum lag time in each *prepare* request. If the combination of the buffering available at the client and the server is not sufficient, the *prepare* request will fail.

4 ADMISSIONS CONTROL ALGORITHM AND I/O SCHEDULER

When a client first stores a media object, a presentation unit vector is created which contains the number of bytes which must be read for each presentation unit for the stream. For example, a presentation unit can be a video frame or a second of audio. Because the stream may be VBR, the values for the vector may vary considerably.

In order to schedule variable bit rate (VBR) streams, the server divides time into fixed length

intervals called "slots" or "rounds".* When a media object is *prepare*d, the data required is divided into these slots as explained below.

The admissions control algorithm must know the minimum number of blocks that the server can read in a single slot, called *minRead*. This value is determined by running a calibration program. The minimum number is calculated by uniformly spacing the blocks across the disk thus maximizing the seek times (assuming a SCAN algorithm). This value most accurately reflects the actual capacity of the server since it includes all transfer delays (through SCSI bus and I/O bus to memory) as well as server software overhead.

This single value is the only piece of information required from the disk system. The admissions control algorithm is therefore independent of the mechanism used to layout blocks on the disk or any other disk management technique such as striping. Clearly, the more optimized the disk management is, the higher the value for the minimum number of disk blocks that can be read, and therefore, more clients can be accepted and scheduled. The admissions scheme itself, however, is not affected by these optimizations. The value for *minRead* must be reasonably close to the actual number of reads per slot. Otherwise, the admissions algorithm will be very conservative. It is possible to relax the value for *minRead* if only statistical guarantees are required.

The scheduling of the disk reads for a stream is done whenever a media object is prepared (Section 3). At this time the presentation unit vector, is converted to a "block schedule" which records the number of blocks to be read per slot for that stream. This number is influenced by the flexible parameters that the user can select in *prepare*. These are the values of the *start* and *stop* positions and the values of *speed* and *skip* specified in the *prepare* request. More data may be read than the required number of bytes in any given slot, due to block boundaries and offsets into blocks, but this is compensated for in subsequent slots, unless the settings of prepare parameters cause discontiguities in the data locations on the disk.

A simple admissions algorithm is to sum the block schedules for all the active streams. The resulting sum is called the "server schedule." The server schedule indicates for each slot the number of blocks that must be read for all active clients. If any slot in the resulting vector is greater than the minimum number of blocks the server can read per slot (*minRead*), the admission fails. This is a very conservative estimate, however, because any single slot may have considerably less than *minRead* I/O operations. In this case, the server is idle at the end of a slot. In order to admit more streams, we permit the server to read ahead as fast as it can subject to buffering constraints. By permitting the server to read ahead we can admit a stream where the total number of blocks required to be read within a slot is greater than the minimum number of blocks the server can read for a slot.

The following example illustrates this method. Assume that the server is capable of reading a minimum of 10 (fixed size) disk blocks per slot. Figure 2 shows the current schedule for the server and a new stream to be admitted. In these vectors, the numbers represent the number of disk blocks that must be read in each slot (block schedule). For instance, in the current server schedule, at slot i the server must read 3 disk blocks, at slot $i+1$ the server must read 5 disk blocks, etc. These blocks represent the total number of blocks for all active streams. The vector for the new stream to be admitted represents the blocks which must be read just for that new stream. For instance, in the first slot for the new stream the server must read 1 block, then 1 again for the next slot, etc.

In the conservative admissions control algorithm, we would simply add the new block schedule to the current schedule. In this case, the $i + 2^{nd}$ slot would have a value of 12 which is higher than the minimum number of blocks the server can read in that slot (10). If we permit the server to read more than a single slot's worth, however, the first two slot's worth of disk blocks would be read in one slot time. This read-ahead permits the given schedule to be accepted. By the time the server reads slot $i+2$ it will still be in the 2^{nd} slot of time assuming the server reads at the minimum number of blocks per slot.

The server in practice reads more than the minimum number of blocks per slot. While we cannot

*A reasonable length for such a slot is 500 milliseconds. If the slot size is too large, the granularity of control is compromised; if too small, the overhead incurred is too great.

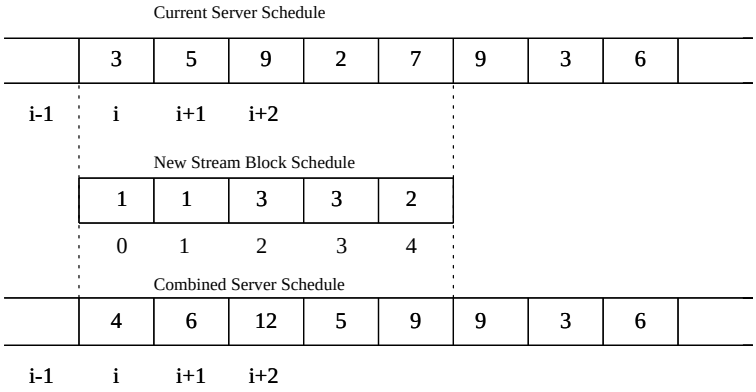

Figure 2: Server Schedule During Admission

count on this for the future, we can take advantage of read ahead that has already been accomplished in the past. The algorithm starts with this number of actual read-ahead blocks and then continues assuming the server will read a minimum number of blocks in each future slot.

So far in this discussion, we have assumed that there are an arbitrary number of buffers. That is, the server can read ahead with out fear of running out of buffers. Clearly this is not the case. Therefore, we have to stop read-ahead in the admissions algorithm once we have run out of buffers. For purposes of buffer consumption we again assume the server reads a minimum number of blocks per slot. As buffers are transmitted on the ATM network, they are freed. Thus, we factor in the number of buffers that are being freed into the admissions algorithm. Data is transferred to the client, and the corresponding buffers freed, however, at a rate which depends on the amount of buffer space available at the client and the negotiated bit rate of the network connection. We therefore maintain another vector called the "buffer allocation vector." This vector is initially the same as the server schedule vector. As data is transferred to the client and buffers are freed, however, the values in the vector are decremented. Note that there is a 1 slot delay in recovering buffers. That is, buffers containing data to be sent to a client in slot i are not reclaimed until the start of slot $i + 1$.

For both the server schedule and the buffer allocation vector there is a "current slot" index. For the server schedule, this index identifies the next slot to read. For the buffer allocation vector it indicates the point of division between buffers that have been filled and buffers that will be required for future reads. As well, there is a "should be" index. This index is incremented by one after each slot time. The difference between "current slot" and "should be" is the number of slots read ahead. As the server reads a slot, it sets the value in the corresponding entry in the server schedule to zero, indicating the slot has been read. Figure 3 illustrates the manipulation of these values. Notice in this example that in the first three slots the network was able to transmit slightly earlier than required by the playout vector. As a result we were able to free five extra buffers and add them to the total pool.

Figure 4 describes the algorithm in detail. In this algorithm *newStream* is the block schedule for the stream to be added (which may have different numbers of blocks for each slot) and *slotCount* is the size of *newStream.* The total number of free buffers at the start of running the admissions test is given in the variable *totalFreeBuffs*. The *serverSchedule* and *bufferAllocate* vectors are the disk and buffer allocations per slot. The variables *shouldBe* and *currentSlot* refer to the "should be" and "current slot" indices respectively. For illustration reasons we assume that these vectors are arbitrarily long.

This algorithm takes into account the actual amount of read-ahead and the actual number of free buffers at the time the algorithm is run. As a result, it uses the actual performance in the past

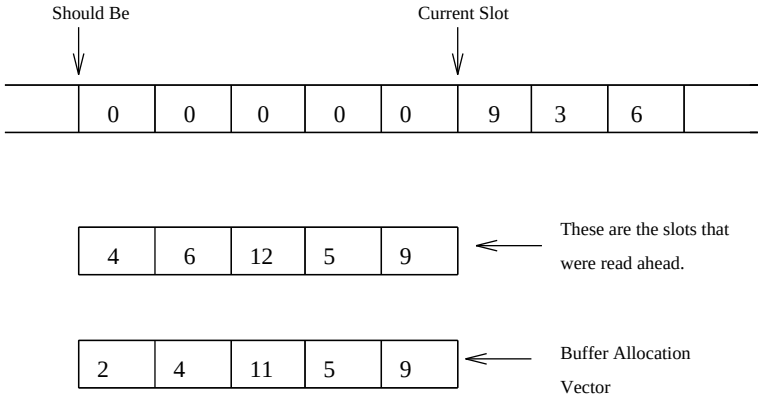

Figure 3: Example of Server Schedule and Buffer Allocation Vectors

and assumes the minimum performance in the future based on *minRead*. If *minRead* is considerably smaller than the actual number of blocks read per slot, then we could reject streams which could be adequately supported by the server. One simple approach to this problem is to increase the value of *minRead* to a value approaching the actual number of blocks read. This would prevent us from offering hard guarantees to clients, but may be a reasonable alternative in some environments.

Another aspect of the system is that if we allow the server to read ahead arbitrarily far it will do so and thereby, use up all of the buffers. The algorithm may easily reject clients due to insufficient buffers. A simple approach to this problem is to always keep in reserve some number of buffers for new clients. Knowing how many buffers to withhold for this purpose, however, would be difficult to determine. Another approach is to free some number of buffers that contain data with the latest "deadline" (i.e. data that will not be needed for the longest time). If we knew the correct number of buffers to reclaim, we could add them back to the server schedule, reset the value of *currentSlot*, free the buffers, and then run the admissions algorithm in Figure 4. It is easy to see that both freeing too few buffers and freeing too many buffers will cause the admissions algorithm to fail. Freeing too few buffers (in the limit, 0) will not provide sufficient read ahead on the new stream to smooth out the peaks of its data rate requirements. Freeing too many (in the limit, all of them) eliminates read ahead that streams which have already been accepted are relying on to smooth out their peak requirements.

An approximation to this scheme is the following: run the admissions control algorithm, if it succeeds we report success; if it fails, we move the current slot back one, free the buffers indicated in the buffer allocation vector at that slot and try again. If it fails again, then we back up two slots and try again. We repeat this with some back off strategy (either linear or exponential may make sense in a variety of circumstances) until either the algorithm succeeds or we have moved *currentSlot* all the way back to *shouldBe* in which case we are unable to admit the new stream. Note that in all these cases, the original values for *serverSchedule* and *bufferAllocate* must be reset if the algorithm fails.

A factor in running the algorithm this many times is the duration the algorithm takes to complete. However, since the algorithm is linear in the length of the longest stream currently scheduled, this cost is minimal, making this approach feasible.

When the admissions control algorithm is examined in detail, we discover that we can accomplish the simulation of freeing buffers in the future dynamically as we schedule the new stream. This is done by modifying the processing when there are not enough free buffers. If this modified algorithm rejects a new stream, there does not exist a number of buffers to free that would allow us to accept this stream. The intuition behind this claim is that the new algorithm never wastes disk bandwidth

```
AdmissionsTest( newStream, slotCount )
begin

    for i = 0 to slotCount do
        serverSchedule[shouldBe + i] = serverSchedule[shouldBe + i] + newStream[i]
        bufferAllocate[shouldBe + i] = serverSchedule[shouldBe + i] + newStream[i]
    end

    for i = shouldBe to MAX VECTOR SIZE do

        { Note that readAhead may be negative.}
        readAhead = minRead − serverSchedule[i]

(1)     if (totalFreeBuffs < minRead) then
            totalReadAhead = totalReadAhead − serverSchedule[i]
        else
            totalFreeBuffs  = totalFreeBuffs − minRead
            totalReadAhead = totalReadAhead + readAhead

        if totalReadAhead < 0 then ABORT

        { Buffers released from previous slot. }
        totalFreeBuffs = totalFreeBuffs + bufferAllocate[i − 1]

    end

end
```

Figure 4: Admissions Control Algorithm

due to a lack of buffers, and at every step aggressively reclaims buffers containing data with the latest deadline and fills them with data with the earliest deadline. If such an approach cannot find enough bandwidth to service the combined set of streams, then no approach can do better. The details of this modification are given in Figure 5 which replaces the code identified by (1) in Figure 4.

The actual reclaiming of buffers is not performed at admissions time. When the disk schedule gets executed, each disk read attempts to locate a free buffer. If none exist, the buffer with the latest deadline is reclaimed. Not all of the buffers predicted for reclamation may end up being reclaimed, since transmission may occur more quickly than anticipated, a user may stop playback or close a connection on an unrelated stream, or other similar circumstances may occur. The admission algorithm need only determine that, even in the worst case, sufficient buffers will be available to be reclaimed to make the schedule feasible.

The algorithm mentioned in this section has execution time linear in the number of slots in the entire server schedule, because every time period must be checked to see if the bandwidth required can be supported. We have performed preliminary testing which indicates that, for a 2 hour schedule (14400 slots @ 500 msec), the amount of time necessary to perform the admissions test on a stream is less than 14 msec on a 66 SpecInt CPU.

```
while (totalFreeBuffs < minRead) and (currentSlot > i+1) do
    currentSlot = currentSlot − 1
    reclaim = min(minRead − totalFreeBuffs, bufferAllocate[currentSlot])
    totalFreeBuffs = totalFreeBuffs + reclaim
    serverSchedule[currentSlot] = serverSchedule [currentSlot] + reclaim
end

read = min(totalFreeBuffs, minRead)
totalFreeBuffs = totalFreeBuffs − read
totalReadAhead = totalReadAhead + read − serverSchedule[i]
```

Figure 5: Modified Admissions Control Algorithm

5 RELATED WORK

Much of the previous work in Continuous Media File Systems has been based on the constant bit-rate delivery of individual streams.[3,11,13,15] The structure of the servers described in Anderson and Homsy[2] and Lougher and Shepherd[11] is similar to ours, but does not provide the linear scalability of our approach. Hillyer and Robinson[10] and Tierney et al.[17] do seem capable of this type of scaling, although the focus in the former is on a more generic file system while the latter focusses on extremely high volume data transfer of image data. Some work[3,10] attempts to balance the performance of real-time and significant amounts of non real-time data in the same server. This needs a more integrated approach to the entire system design and adds somewhat to the complexity of achieving the performance goals.

The synchronous playback of multiple streams is addressed by several approaches, in particular Agarwal et al.[1] and Anderson and Homsy.[2] These methods provide comprehensive models for synchronization that involve both the client and the server. Our model provides the interface to synchronize many independent streams without inserting complexity into the data itself. Variable speed playback mechanisms are described in Chen et al.[5] and Dey-Sircar et al.[6]

Since the resource requirements of a stream must be known and reserved, there is a limit to the number of simultaneous users that can be supported by any continuous media system. Every previous system must provide a method of admission control that does not result in violation of continuity requirements. Many of these approaches[3,9,15,18] provide acceptance tests based on disk bandwidth and network bandwidth assuming the maximum possible data transmission rate at any point in the life of a stream connection. Their admission control is efficient because it can be done as a constant-time calculation, but does not take advantage of the variability of the stream to accept more connections. Vin et al.[19] describes an admissions test that incorporates the distribution of bit-rates into an algorithm that provides statistical guarantees of acceptable continuity. None of the other approaches address the the exact peaks and valleys in data requirements experienced by a given set of continuous media streams. Careful layout of disk blocks is used to increase the disk bandwidth (often of a set of streams expected to be played out in some kind of synchronous fashion) to increase the likelihood of a successful admission.[11,15,17]

6 CONCLUSIONS

In this paper, we have described a Continuous Media File Server that we have both designed and implemented. This server delivers scalable performance because the nodes of which it is comprised are completely independent. The server provides an API that permits client applications to schedule

multiple, concurrent streams in a simple, direct fashion, given that they are informed of a bound on the latency of a *prepare* operation. The API also provides a flexible scheme for alternative playout requests, rather than the "complete object, full speed, no stop" scenario.

Another contribution of this paper is a unique approach to admission control that examines the disk bandwidth requirements at a fine granularity. This admission control introduces very little overhead and allows a greater number of simultaneous requests to be serviced. The benefit is achieved because we permit the server to read ahead of its schedule at the maximum possible rate, subject to buffering constraints. We assume a minimum rate of disk reads, which is calibrated *a priori*. based on worst case assumptions of block layouts and use that amount to provide the ability to smooth out peaks in individual stream requirements. This does not additionally restrict future admissions, because filled buffers which are needed for the new stream can be aggressively chosen for reclamation (at no cost) without altering the feasibility of the schedule, keeping the disk busy as much as possible. Our design particularly complements both the variable nature of digitized audio and video and the ability of ATM networks to handle bursty traffic.

7 CURRENT STATUS AND FUTURE WORK

A multiple node version of the file server has been implemented. The server runs on IBM RS/6000s (350) or Sun Sparcstations over a 100 Mbps ATM link (or Ethernet, or Token Ring) to multiple clients running on either of those same two architectures. The system environment uses a real-time threads package developed for this project which operates within a single Unix process.[8] This package also provides an operating system shield to native systems which do provide some form of real-time threads such as AIX 4.1.

Future work is underway to investigate the details of performing real-time transmission of the continuous media across the network. The variable bit-rate schedule and information about client resources provides information which the server can use to schedule the network sends in a manner that effectively and fairly utilizes the available network bandwidth. This scheduling is complicated due to the fact that the clients may have varying amounts of buffer space or may have negotiated a network resource allocation that is not sufficient to utilize the available buffer space.

ACKNOWLEDGMENTS

The authors would like to acknowledge the efforts of David Finkelstein, Ann Lo, and Roland Mechler for their efforts in design and implementation of the Real-Time Threads package, the XTP network communication protocol and video/audio display clients to test many of our ideas.

8 REFERENCES

[1] N. Agarwal and S. Son. Synchronization of Distributed Multimedia Data in an Application-Specific Manner. In *ACM Multimedia*, pages 141–148, San Francisco, October 1994.

[2] David P. Anderson and George Homsy. A Continuous Media I/O Server and Its Synchronization Mechanism. *IEEE Computer*, 24(10):51–57, October 1991.

[3] David P. Anderson, Yoshitomo Osawa, and Ramesh Govindan. A File System for Continuous Media. *ACM Transactions on Computer Systems*, 10(4):311–337, November 1992.

[4] S. Bikash, M. Ito, and G. Neufeld. The Design and Performance of a Continuous Media Server for a High-Speed Network (to appear). In *IEEE Multimedia*, Washington, D.C., May 1995.

[5] Ming-Syan Chen, Dilip D. Kandlur, and Philip S. Yu. Support for Fully Interactive Playout in a Disk-Array-Based Video Server. In *ACM Multimedia*, pages 391–398, San Francisco, October

1994.

[6] K. Salehi Dey, J. Kurose, and D. Towsley. Providing VCR Capabilities in Large-Scale Video Servers. In *ACM Multimedia*, pages 25–32, San Francisco, October 1994.

[7] E. Dubois, N. Baaziz, and M. Matta. Impact of Scan Conversion Methods on the Performance of Scalable Video Coding. In *IS&T/SPIE Proceedings*, San Jose, February 1995.

[8] D. Finkelstein, R. Mechler, G. Neufeld, D. Makaroff, and N. Hutchinson. Real-Time Threads Interface. Technical Report 95-07, University of British Columbia, Vancouver, B. C., March 1995.

[9] D. James Gemmell. Multimedia Network File Servers: Multi-channel Delay Sensitive Data Retrieval. In *ACM Multimedia*, pages 243–250, June 1993.

[10] Bruce K. Hillyer and Bethany S. Robinson. Communications Issues in BBFS, a Broadband Distributed Filesystem. In *GlobeCom 91*, pages 1097–1101. IEEE, October 1991.

[11] Phillip Lougher and Doug Shepherd. The Design of a Storage Server for Continuous Media. *The Computer Journal (special issue on multimedia)*, 36(1):32–42, February 1993.

[12] Gene Miller, Greg Baber, and Mark Gilliland. News On-Demand for Multimedia Networks. In *ACM Multimedia*, pages 383–392, June 1993.

[13] Sape J. Mullender, Ian M. Leslie, and Derek McAuley. Operating-System Support for Distributed Multimedia. In *USENIX High-Speed Networking Symposium Procceedings*, pages 209–219, Oakland, California, August 1-3 1994. USENIX Association.

[14] G. Neufeld, D. Makaroff, and N. Hutchinson. Internal Design of the UBC Distributed Continuous Media File Server. Technical Report 95-06, University of British Columbia, Vancouver, B. C., April 1995.

[15] P.V. Rangan and H.M. Vin. Designing File Systems for Digital Video and Audio. In *Proceedings 13^{th} Symposium on Operating Systems Principles (SOSP '91), Operating Systems Review*, volume 25, pages 81–94, October 1991.

[16] W. T. Strayer, B. J. Dempsey, and A. C. Weaver. *XTP: The Xpress Transport Protocol*. Addison Wesley Publishing, October 1992.

[17] Brian Tierney, Willian Johnston, Hanan Herzog, Gary Hoo, Guojon Jin, Jason Lee, Ling Tony Chen, and Doron Rotem. Distributed Parallel Data Storage Systems: A Scalable Approach to High Speed Image Servers. In *ACM Multimedia*, San Francisco, October 1994.

[18] Fouad A. Tobagi, Joseph Pang, Randall Baird, and Mark Gang. Streaming RAID - A Disk Array Management System For Video Files. In *ACM Multimedia*, pages 393–400, June 1993.

[19] Harrick M. Vin, Pawan Goyal, Alok Goyal, and Anshuman Goyal. A Statistical Admission Control Algorithm for Multimedia Servers. In *ACM Multimedia*, pages 33–40, San Francisco, October 1994.

[20] XTP Forum. *Xpress Transport Protocol Specification Revision 4.0*, March 1995.