The Design and Implementation of a Multimedia Storage Server to Support Video-On-Demand Applications

Anastasio Molano[1]

Universidad Autónoma de Madrid
Madrid (Spain)

Alberto García-Martínez and Angel Viña

Universidad de La Coruña
La Coruña (Spain)

Abstract

In this paper we present the design and implementation of a client/server based multimedia architecture for supporting video-on-demand applications. We describe in detail the software architecture of the implementation along with the adopted buffering mechanism. The proposed multithreaded architecture obtains, on one hand, a high degree of parallelism at the server side, allowing both the disk controller and the network card controller work in parallel. On the other hand, at the client side, it achieves the synchronized playback of the video stream at its precise rate, decoupling this process from the reception of data through the network. Additionally, we have derived, under an engineering perspective, some services that a real-time operating system should offer to satisfy the requirements found in video-on-demand applications.

1. Introduction

Recent developments in workstation technology, compression technology, high bandwidth storage devices and high speed networks, will make feasible the support of distributed video-on-demand applications.

A multimedia storage server for video-on-demand applications should provide simultaneous service to multiple clients that request the playback of video clips with some demanded QoS. System resources like disk bandwidth, processor capacity and network bandwidth should be checked before accepting a new session [14].

Intensive research has been undertaken in order to develop successful video-on-demand services. This research focus mainly on three different directions:

- The design of the storage server itself, that implies making some decisions concerning the playback policy, the buffering mechanism and the stream admission test [4, 6, 8, 11, 13].
- The system support for multimedia applications: the operating system services needed to satisfy multimedia applications requirements [3, 5, 12].
- The communication media, that should provide a high bandwidth and appropriate reservation mechanisms to guarantee the requested QoS [1, 2].

Our research focus in particular on the design of multimedia storage servers, and the support of the operating system for their implementation.

In this paper, we present the design and implementation of a client/server based multimedia architecture for supporting video-on-demand applications. An application has been developed where the clients, connected through an ATM based link to the multimedia storage server, request the playback of video clips with some QoS parameters. The server runs a stream admission test to check whether there are enough resources to satisfy the requested QoS.

In case of lack of resources, a renegotiation with lower quality parameters is carried out, until the client whether accepts or rejects the connection. Our prototype implementation is based on the MPEG-I decoder, handling constant bit rate (CBR) streams.

The experience gained through the design and implementation of the multimedia architecture has allowed us to identify, from an engineering perspective, the specific real-time operating system services needed for video-on-demand applications. We have reviewed the following aspects: process management, task scheduling, memory management and support for data intensive I/O applications.

1. This research has been supported by the Regional Research Plan of the Autonomous Comunity of Madrid under an F.P.I. research grant.

1089-6503/96 $5.00 © 1996 IEEE
Proceedings of EUROMICRO-22

The remainder of this paper is organized as follows. Section 2 describes in detail the adopted design approach: it explains the computational model of video streams that will be followed thereafter, the buffering mechanism applied and the stream admission test. An in-deep description of the software architecture of the implementation, both at the server and at the client side, is presented in section 3. The QoS negotiation protocol carried out between the server and the clients is also described in this section. Section 4 highlights the derived real-time operating system services. In section 5 we describe the multimedia experimental infrastructure installed in our laboratory. Section 6 presents conclusions and, finally, section 7 highlights future work.

2. The design of a client/server based multimedia architecture

Designing a multimedia storage server implies making some decisions concerning several parameters, for instance, the buffering mechanism, the playback process, the disk scheduling algorithm and the subsequent stream admission test. In this section we explain the specific parameters adopted in our implementation.

2.1. The computational model of video streams

MPEG video streams are characterized by the parameters indicated in table 1 [7], that are included in the MPEG sequence header at the beginning of the file containing the videoclip. The MPEG sequence header is preceded by the sequence start code that is parsed previously to check the correctness and validity of the MPEG video stream.

TABLE 1. MPEG Video Sequence Parameters

MPEG Sequence Header
Picture Width (horizontal size of image space)
Picture Height (vertical size of image space)
Aspect Ratio Code
Frame Rate (frames per second)
Bit Rate (bytes per second)
Buffer size (minimum buffer size)

Among these parameters, the frame rate and the bit rate can be used to infer the timing properties of the playback process. For instance, the playback period, T_i, can be obtained as the inverse of the frame rate, and the logical unit size (following the terminology of [8]), L_i, i.e. the amount of information to be played back every T_i time units, can be computed as the ratio between the bit rate and the frame rate:

$$T_i(\text{sec}) = 1/\text{frame rate} \quad \text{(EQ 1)}$$

$$L_i(\text{bytes/frame}) = \frac{\text{bit rate}}{\text{frame rate}} \quad \text{(EQ 2)}$$

These parameters represent the requested QoS, QoS = (frame rate, bit rate). We will assume a minimum QoS, QoS_{MIN} = (frame $rate_{MIN}$, bit $rate_{MIN}$), for each video stream, that it will be used in the negotiation of QoS parameters as it will be explained below.

In order to be able to satisfy the requested QoS, the multimedia storage server should retrieve from disk L_i bytes of information for the stream i every T_i time units. However, the logical unit, L_i, is usually very small and retrieving from disk the information in small chunks of L_i bytes each, would produce a high overhead due to the high rate of I/O system calls performed. In common practice, it is better, in order to improve throughput, to fetch the information from disk in larger chunks of $m_i \cdot L_i$ bytes each, that will be read consequently every $m_i \cdot T_i$ time units, storing the data previously in some buffer before being played back. The multiplier m_i can be computed by taking into account the amount of available memory, considering a target buffer size, such as 64 KBytes, that represents a tradeoff between large disk accesses that improve throughput and the available memory in the computer [8].

In our design the information is retrieved from disk in this way (see figure 1), so $m_i \cdot L_i$ bytes (DL_i for short) are fetched from disk every $m_i \cdot T_i$ time units (DT_i for short).

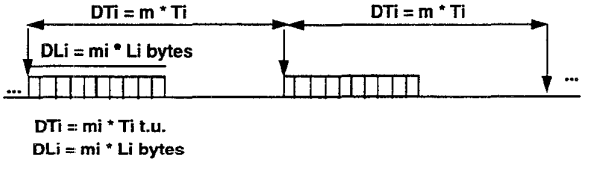

Figure 1. Fetching of video streams from disk.

2.2. Buffering mechanism

The stringent real-time requirements of the isochronous media playback process can be relaxed by buffering a set of frames before outputting them to the destination device.

With a buffering mechanism, the operating system does not need to retrieve from disk the information at exactly the same rate as is needed in the playback process.

Generally, a double buffering scheme is adopted (see figure 2), consisting of two equally sized buffers that store, one of them, the frames currently being fetched from disk, while the other, the frames currently being delivered to the destination device (whether to the presentation device or to the network).

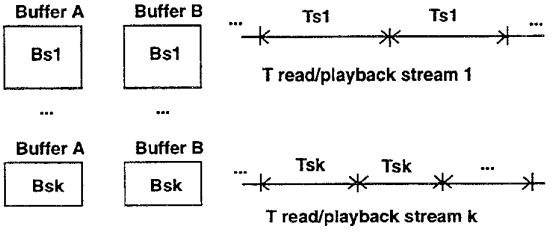

Figure 2. Video server buffering mechanism.

In the double buffering scheme a global reading/ playback period is defined, under which the storage server fetches data from disk and stores it in the first buffer (the producer buffer), while consuming the data stored in the second buffer (the consumer buffer). Once the first buffer is full and the second is empty, a buffer switch is performed so the role of each buffer is interchanged.

The playback process must be continuous over time and the multimedia storage server should avoid undesired interruptions that could happen if the producer buffer overflows or the consumer buffer gets empty. In order to guarantee that the buffers never overflow or get empty, the storage server should be able to fill up the producer buffer before the information stored in the consumer buffer had been played back. If this condition holds, the playback process will not be interrupted and it will be continuous over time. That should stand for all the streams being simultaneously played back.

We have made use of a simple double buffering mechanism, both at the server and at the client side. In the former case, the information is fetched from disk and sent to the network, in the later, the information is received from the network and displayed on the screen.

In our design, the playback of each stream is carried out in an independent fashion, so each of them is read/played back at its own global reading/playback period. For the stream i the global reading/playback period is DT_i time units long. We read DL_i bytes of information for each stream every DT_i time units. Consequently we need to allocate for each stream a double buffer of $2 \cdot DL_i$ bytes.

Our scheme differs from other approaches that define a common global reading/playback period of equal length for all the streams being serviced [4, 6, 8]. For each stream the storage server retrieves from disk the exact amount of data needed for its playback during the common global period, that it will depend on its bit rate, the larger the rate the larger the amount of data to be read. Consequently the buffer allocated will be different for each stream, and equal to its data consumed during the common global period. A round robin policy is generally used for choosing the next stream to be read within the cycle. This approach suffers mainly from read postponement due to data accumulation [8], and also from a lack of flexibility when new sessions need to be accommodated, because of the need to recompute the common global period each time a client requests a new session.

2.3. The stream admission test

The multimedia storage server must be able to determine if a new stream can be accepted. In order to do so, it computes a stream admission test to check the existence of enough resources to satisfy the requested QoS.

Depending on the particular disk scheduling policy applied, the stream admission test will differ. In our design we have made use of the EDF scheduling policy [9], assigning a dynamic priority to each disk request depending on the current deadline. The assigned priority will be the highest if the deadline of its current request is the nearest.

The EDF scheduling algorithm can achieve utilization factors of 100 percent, and, in this sense, it is an optimal algorithm. Other scheduling algorithms like the fixed-priority based Rate Monotonic algorithm cannot guarantee a 100 percent utilization factor (Rate Monotonic is limited up to 69% in the worst-case. when the periods of the requests are mutually prime).

Assuming EDF priority assignment, the stream admission test will be [9]:

$$\sum_{i=1}^{s} \frac{t_{read}(m_i \cdot L_i) + 2 \cdot Seek_{MAX}}{DT_i} \leq 1 \qquad \text{(EQ 3)}$$

Disk seek times, coming from disk head movements and rotational delays, have been included in the test, taking into account that worst-case seek costs are incurred in each activation.

3. Implementation of the multimedia architecture

The software architecture has been developed in a SPARC 20 machine under SunOs 5.5, using Solaris Threads Library.

3.1. The server software architecture

In order to carry out the playback of multiple video streams, several activities should be performed concurrently. For instance, for each stream being serviced, new data blocks are read from disk at the same time that previously stored data is consumed by the destination device (the destination device is just the network at the server side).

To allow such a high number of concurrent activities, we have made use of a multithreaded architecture, where each thread performs a different task (see figure 3). For each stream, the multimedia server process creates two threads, a reader thread and a consumer thread. The reader thread is activated for the stream i every DT_i time units, and request DL_i bytes from disk in each activation, storing the information in one of the buffers. The consumer thread is activated at the same rate, and sends DL_i bytes from the second buffer to the network on each activation. Once they have finished their activities they sleep until the next activation period, DT_i time units later.

Hence, the multithreaded architecture is comprised of a total of $2 \cdot k$ threads (k reader threads and k consumer threads) being k the number of video streams serviced. There is a bank of buffers organized following the double buffering scheme, and each stream has a double-buffer associated with it, $2 \cdot DL_i$ bytes sized, that is accessed in a private way (see figure 3).

We must carefully consider the correct synchronization between the threads execution. On one hand, for the stream i, both reader and consumer threads should be activated every DT_i time units, and there should be no jitter between their activations, or the buffer switch could not be correctly performed. On the other hand, the activities carried out by the reader and consumer threads have different durations, and must be synchronized to accomplish the buffer switch in consistent way. The earlier thread in completing its activity must wait for the other before updating the status of the buffers and the pointers to them.

Activating both the reader and the consumer threads in an independent fashion every DT_i time units, would lead to an unavoidable jitter because of the impossibility to synchronize both timers. In order to avoid this jitter, only the reader thread is activated periodically every DT_i time units, and it synchronizes its execution with the consumer thread by means of a counting semaphore initialized to zero. The consumer thread blocks waiting on the semaphore, and the reader thread increases its value at the beginning of its activation, once the timer expires, allowing the execution of the consumer thread. Doing this way, both reader and consumer threads initiate their executions at the same time with zero jitter (the only jitter is the time to await the consumer thread).

Performing the buffer switch requires more complex synchronization primitives. For instance, the buffer switch can only be done once the buffer of the reader thread has been filled up, and the buffer of the consumer thread emptied. This condition has been implemented using a condition variable guarded by a mutex. When the reader/ consumer thread finishes its work, it acquires the mutex and checks whether the other thread has completed its work, reading the state of the other buffer (FULL or EMPTY). If the other thread has not finished yet, it sleeps waiting on the condition variable. Otherwise, in case that the other thread

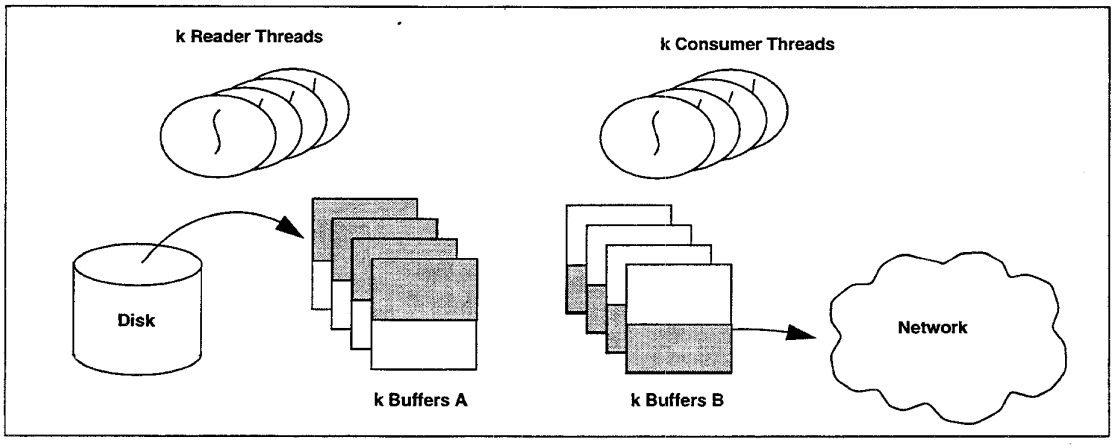

Figure 3. Video server software architecture (k streams).

has finished, the currently executing thread performs the buffer switch, updating the status of the buffers and the pointers to them, and finally it awakes the other thread by signalling the condition variable. The reader thread will sleep on the timer again, waiting for the next activation period. The consumer thread will sleep on the semaphore until the reader thread awakes and releases it.

The use of the multithreaded architecture has allowed us to increase the degree of parallelism at the server side without affecting the performance. Threads can be created dynamically with low overhead, because the operating system maintains a minimum amount of private context for each thread, that can be allocated on creation time at low cost. The high number of concurrent activities that must be carried out in a multimedia storage server, makes the chosen software architecture the most appropriate for our application. Additionally, when both the reader and consumer threads send their requests to the disk drive and to the network drive respectively, they are put to sleep, and their respective requests are serviced simultaneously, since the disk controller and the network card controller can work in parallel (through DMA requests), improving performance.

3.2. The client software architecture

The clients request the playback of at most one video clip each time, therefore, the concurrency is more limited than in the previous case. Only two threads, a producer and a consumer thread are needed in the multimedia client process (see figure 4). This does not preclude the execution of several clients in the same machine, under the user control.

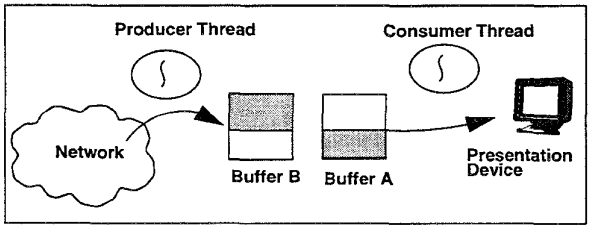

Figure 4. Client software architecture.

A simple double buffering scheme is employed, being the size of the buffers the same as the peer buffers for the stream i at the server size, for instance $2 \cdot DL_i$ bytes both.

The producer thread fetches data from the network and stores it in the buffer assigned for it. Ideally, it receives DL_i bytes from the network every DT_i time units. The consumer thread reads data from the consumer buffer, initiates the decoding process, and displays the frames on the screen after decoding a complete logical unit. The playback process is performed at a fixed rate of $1/T_i$ frames per second, the natural playback rate of the stream. This rate should be preserved in order to achieve a successful presentation to the user.

Despite the limited number of threads within the multimedia client process, the software architecture is similar to the server case, because there are the same synchronization problems. Both, the producer and the reader threads must be activated every DT_i time units, with zero jitter, and the buffer switch must be performed in a consistent fashion, once both of them have finished their work. The synchronization mechanisms used to solve these problems are similar to those employed for the server case.

At the client side there is an additional problem derived from the display of frames on the screen: it should be carried out at a rate of one frame per T_i time units, while the activation period of the consumer thread is $m_i \cdot T_i$ time units. In order to tackle this problem, the consumer thread is activated within the global period m_i times, one activation each T_i. During the inner activations the consumer thread decodes a logical unit, and after completing the decoding process, it displays the resulting frame on the screen.

At the client side, the proposed multithreaded architecture allows us to isolate the playback process from the reception of data through the network. The playback process is carried out at its own natural playback period, T_i, despite of receiving the data in chunks of DL_i bytes every $m_i \cdot T_i$ time units.

The communication protocol used in our implementation (TCP/IP) do not guarantee a strict timely delivery in the reception of data, and in general, there will be time jitters in its reception. The double buffering mechanism avoids the interruption of the playback process, as far as the average receiving rate be sustained, no matter whether there are reception time jitters in between.

3.3. Negotiation of QoS parameters

The multimedia storage server is connected with its clients via TCP connections. At the initiation of the connection, the client sends a `clientRequest` packet (see figure 5), where the desired QoS parameters are specified (QoS = (frame rate, bit rate)). It also includes the minimum value of such parameters, QoSmin (QoS_{MIN} = (frame $rate_{MIN}$, bit $rate_{MIN}$)), still valid to accept the connection.

The server computes the multiplier m_i to be used in the

buffers allotment and in the actual retrieval of data from disk. Then it runs the stream admission test, using the disk accessing period DT_i, and the amount of data retrieved each period DL_i, derived from the basic QoS parameters (see EQ1 and EQ2).

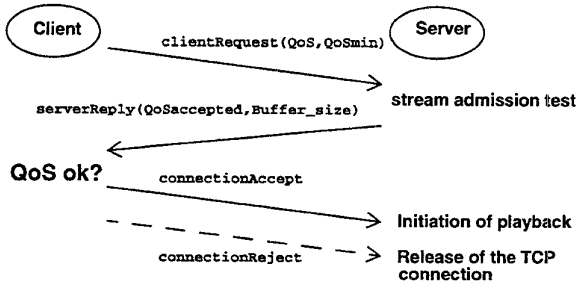

Figure 5. Negotiation of QoS parameters.

If the stream admission test do not hold using the basic QoS parameters, the server computes the test again, but this time using a frame rate one unit smaller than the previous one. This procedure is successively repeated in an iterative fashion until whether the test holds, or the frame rate_{MIN} is reached. If that value is reached and the test does not hold, the connection will be rejected. The server sends to the client a **`serverReply`** packet, where it indicates if the connection has been accepted or not, and the granted QoS.

The user can accept or reject the negotiated quality parameters. In case of acceptance, the client sends a **`connectionAccept`** packet to the server and the playback process begins thereafter. Otherwise it sends a **`connectionReject`** packet to the server and it releases the TCP connection.

4. Real-time operating system services for video-on-demand applications

The experience gained through the design and implementation of the multimedia architecture has allowed us to identify, from an engineering perspective, the specific real-time operating system (RTOS) services needed for video-on-demand applications. In this section we review the main features that an operating system should have to suit multimedia applications requirements.

4.1. Process/Threads management

A multimedia storage server must perform a high number of concurrent activities. As it has been shown in a real implementation, the number of threads within a multimedia server process can be as large as 2 k threads, being k the number of video streams.

New clients can request the playback of video clips at any time, and in order to cope with that, new threads must be dynamically created to service them. Therefore, taking into account the high number of threads and the dynamic management of their associated resources, the overhead should be maintained as low as possible to achieve high performance.

In order to guarantee a low overhead, the private context of the threads should be very small to require a subsequent small thread creation time. Additionally, the thread context switch time should be as short as possible to reduce the performance penalty incurred because of context switches between threads.

4.2. Processor scheduling

Video-on-demand applications have to satisfy stringent real-time constraints derived from the real-time nature of the video playback process. We have shown that multiple threads must execute concurrently to serve the client requests. These threads of execution must be correctly scheduled to guarantee that media streams are properly processed and meet their deadlines.

Current commercial RTOS offer simple support for task scheduling. The Posix 1003.1b Standard specifies two different scheduling models, the *system thread scheduling model* and the *process scheduling model*. In the former, each thread competes for execution resources against any other thread in the system, that can belong to the same or to a different process. In the later, threads are scheduled only against all other threads within the same process.

Only the system thread scheduling model guarantees that no other lower priority thread belonging to a different process can interfere the execution of the current higher priority thread, therefore it is the right model to apply in multimedia applications.

Several policies are considered within the Posix 1003.1b standard (SCHED_FIFO, SCHED_RR, SCHED_OTHER). Each thread has its own policy and priority, and the priorities are assigned in a fixed basis. Fixed priority based scheduling policies as the Rate Monotonic policy [9] can be directly supported using the standard interface.

However, more suitable dynamic scheduling policies such as the EDF policy that achieves 100 percent of the utilization factor [9], are not directly supported, so the programmer must resort to "ad hoc" techniques to implement this scheduling policy.

The programmer can execute all the threads at the highest priority and then, once they have been chosen for execution, adjust their actual priority at the beginning of their execution according to their deadlines. A second method is establishing the actual thread priority in the context of "hook functions" that are linked to the thread,

and executed before its code (some RTOS like VxWorks incorporate "hook functions").

RTOS should offer better support for EDF dynamic priority assignment, and the kernel-level scheduler should assign priorities to the threads dynamically according to their deadlines.

4.3. Memory management

In a multimedia storage server, memory is dynamically allocated to threads in order to serve new requests (e.g. allocation of thread management resources and buffers for the playback process). Such dynamic allocation of memory can lead to unavoidable unbounded delays. It only happens at the beginning of the stream playback process, and it appears as a delay in the playback starting time.

However, once the session has been initiated, new delays may occur because of page faults of parts of the program that have been swapped to disk. These delays appear as an interruption in the playback process and the multimedia storage server should prevent from the occurrence of these undesired interruptions. To avoid the potential latencies caused by page faults, current RTOS offer memory locking, which is a facility to bind application programs into memory. Memory locking guarantees the residence of portions of the address space in memory.

In our multimedia server, pages are locked into memory at the beginning of the execution of the threads once the memory has been allocated. The stream admission test must check the memory availability when accepting a new session.

4.4. Operating system support for I/O data intensive applications

Multimedia storage servers represent a good example of I/O data intensive applications. The information must be retrieved from disk and sent to the network with a strict timely pattern. In order to obtain a high throughput the operating system should minimize the amount of data copies performed between address spaces, incorporating techniques like memory mapped streams [5], and in-kernel data paths [3].

Another related issue is the problem of scheduling disk I/O requests. In order to effectively apply disk scheduling techniques like EDF or Rate Monotonic, disk I/O requests should be prioritized and preemptable.

Assigning a priority, either static or dynamic, to the reader threads does not overcome the disk scheduling problem, because, the threads sleep while the disk I/O operations are being carried out, and their priorities are not taken into account meanwhile.

Disk I/O operations should be prioritized, in order to perform a higher disk I/O operation before a lower priority one. To implement Rate-Monotonic assignment we should guarantee that the thread priority is preserved along the disk I/O operation, and thus each disk I/O operation is performed at the same priority level as the one of the thread that invoked the I/O system call.

In case of dynamic priority EDF assignment, each individual disk I/O operation should be performed at a different priority in each activation, accordingly to the current deadline. Rate Monotonic and EDF scheduling, assume that tasks are preemptable, so by extension, disk I/O operations should be preemptable as well.

Current real-time operating systems do not offer support for preemptive prioritized disk I/O operations. At most, they allow the prioritized queueing of disk I/O requests within the disk driver, but in no way they take care of the preemption of such disk I/O requests.

Intensive research has been undertaken in our Laboratory in that direction. We have developed a prototype implementation that allows the invocation of preemptive priority-based disk I/O operations and we have applied it to our case study. Either static or dynamic priority assignment can be carried out. A more detailed description of that work is beyond of the scope of this paper and can be found in [10].

5. The CESATlab Multimedia Testbed

In this section we present the infrastructure installed in our laboratory (see Figure 6), consisting of conventional workstations connected via an ATM based link. The multimedia storage server runs on a SPARC 20 machine and the multimedia clients on a SPARC 4, connected both workstations through a LattisCell 10114 ATM switch from Synoptics.

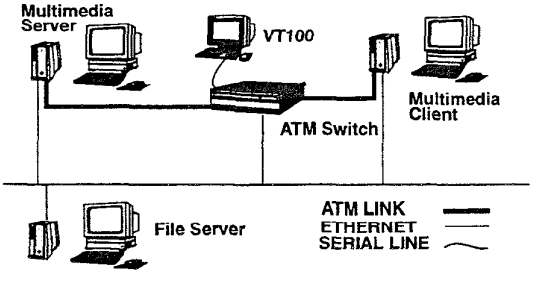

Figure 6. Multimedia experimental infrastructure.

We have not integrated multimedia specific hardware, such as video compression/decompression cards, so the decoding process is carried out entirely by software, using a MPEG-I decoder.

6. Conclusions

We have presented, in a comprehensive fashion, the design and implementation of a client/server based multimedia architecture for supporting video-on-demand applications.

We have described in detail the software architecture of the implementation along with the adopted buffering mechanism.

The use of a multithreaded architecture has allowed us to increase the degree of parallelism without affecting the performance. Threads can be created dynamically with low overhead, because the operating system maintains a minimum amount of private context for each thread, that can be allocated on creation time at low cost. The high number of concurrent activities that must be carried out in a multimedia storage server, makes the chosen software architecture the most appropriate for our application.

At the server side, the reader and consumer threads send their requests to the disk drive and to the network drive respectively, they are put to sleep, and their respective requests are serviced simultaneously, since the disk controller and the network card controller can work in parallel (through DMA requests), improving performance.

At the client side, the proposed multithreaded architecture allows us to isolate the playback process from the reception of data through the network. The playback process is carried out at the stream natural playback period despite of receiving the data with a different period.

We have also derived, following an engineering perspective, the services that a real-time operating system should offer to satisfy video-on-demand application requirements.

Among these requirements we have shown that a RTOS should offer the following features: efficient process/thread management capability, dynamic deadline-oriented priority assignment, memory locking, and finally, appropriate support for preemptive prioritized disk I/O operations.

7. Future work

We are now working out on new disk scheduling techniques derived from basic EDF scheduling policy, that be able to reduce the performance penalty incurred because of disk seeks. We would like to push our implementation to their limits in high workload conditions. In order to do that we need to develop metric tools to evaluate its behaviour in such conditions. We also want to integrate in our implementation an MPEG-2 decoder, so VBR streams could be serviced as well.

8. Bibliography

1 Campbell A., Coulson G., García F., Hutchison D., Leopold H., "Integrated Quality of Service for Multimedia Communications". Proceeding of the IEEE Infocom' 93, Hotel Nikko, San Francisco, CA, March 1993.

2 Deloddere D., Verbiest W., Verhille H., "Interactive Video On Demand". IEEE Communications Magazine, pp. 82-88, May 1994.

3 Fall K., Pasquale J., "Improving Continuous-Media Playback Performance with In-Kernel Data Paths". Proceedings of the Int. Conf. on Multimedia Computing and Systems, pp. 100-109, Boston, May 1994.

4 Gemmell D. J., Han J., Christodoulakis S., "Delay-Sensitive Multimedia on Disks". IEEE Multimedia Magazine, pp.56-67, Fall 1994.

5 Govindan R., Anderson D.P., "Scheduling and IPC Mechanisms for Continuous Media". Proceeding of the 13th ACM Symposium on Operating Systems Principles, pp. 68-80, Pacific Grove, California, Oct 1991.

6 Kenchammana-Hosekote D.R., Srivastava J., "Scheduling Continuous Media in a Video-On-Demand Server". Proceedings of the Int. Conf. on Multimedia Computing and Systems, pp. 19-28, Boston, May 1994.

7 Le Gall G., "MPEG: A Video Compression Standard for Multimedia Applications". Communications of the ACM, vol. 34, No. 4, pp. 46-58, April 1991.

8 Lougher P., Shepherd D., "The Design of a Storage Server for Continuous Media". The Computer Journal (special issue on multimedia), pp. 32-42, 36(1), February 1993.

9 Lui C.L., Layland J.W., "Scheduling Algorithms for Multiprogramming in a Hard Real Time Emvironment". Journal of the ACM 20, 1, pp.47-61, 1973.

10 Molano A., "A Preemptive Priority-based disk I/O Subsystem for the management of Hard Real-Time Disk Traffic". CESATlab Technical Report CESAT-TR-96-09, 1996.

11 Reddy A.L.N., and Wyllie J., "Disk Scheduling in a Multimedia I/O System". Proc. ACM Multimedia Conference, ACM Press, New York, 1992, pp. 225-233.

12 Steinmetz R., "Analyzing the Multimedia Operating System". IEEE Multimedia Magazine, pp. 68-84, Spring 1995.

13 Tindell K., Burns A., "Scheduling Hard Real-Time Multi-Media Disk Traffic". Technical Report YCS 204, Department of Computer Science, University of York, 1993.

14 Viña A., López J., Molano A., and del Val D, "Real-Time Multimedia Systems". Proceedings of the 13th IEEE Symposium on Mass Storage Systems, Annecy (France), pp. 77-83, June 1994.