Proc. 3rd Int. Workshop on Network and OS Support for Digital Audio and Video, San Diego CA (November 1992)

A Continuous Media Player[†]

Lawrence A. Rowe and Brian C. Smith
Computer Science Division-EECS, University of California
Berkeley, CA 94720, USA

Abstract. The design and implementation of a continuous media player for Unix workstations is described. The player can play synchronized digital video and audio read from a file server. The system architecture and results of preliminary performance experiments are presented.

1. Introduction

Our goal is to develop a portable user interface and continuous media support library that can be used to implement a variety of multimedia applications (e.g., hypermedia systems, video conferencing, multimedia presentation systems, etc.). A key component of these applications is a *continuous media* (CM) player that can play *scripts* composed of one or more synchronized data *streams*. Example data streams are: digitized video or audio, animation sequences, image sequences, and text.

The initial application we are implementing to test our abstractions is a video browser that allows a user to play high quality videos stored in a large database on a shared file server. Figure 1 shows a screen dump of the browser interface. The window on the left lists videos in the database, and the window on the right plays the video. The VCR controls below the video window allow the user to play the video forwards or backwards at several speeds or to access a particular position using the thumb in the slider.

The player runs on a Sun Sparcstation with a Parallax XVIDEO board which has a JPEG CODEC chip. The video stream is stored as a sequence of JPEG frames, and the audio stream is stored in a standard Sparc audio file.

The system has a flexible architecture that will allow other data representations and decompression technologies to be added. For example, we have implemented an MPEG video decoder in software that will be added to the system, and we are anxiously awaiting compression hardware for other Unix workstations.

A special-purpose datagram protocol was implemented to send CM packets from the file server to the client workstation. The current implementation runs on UDP, but it was designed to use the real-time IP protocol being developed by another research group at Berkeley [18]. We have run the player on a conventional ethernet and FDDI network.

The remainder of the paper describes the design and implementation of the player, the results of some initial performance experiments, and related work.

[†] This research was supported by the National Science Foundation (Grant MIP-9014940) and the Semiconductor Research Corporation with a matching grant from the State of California's MICRO program. Additional support was provided by Fujitsu America and Hewlett-Packard.

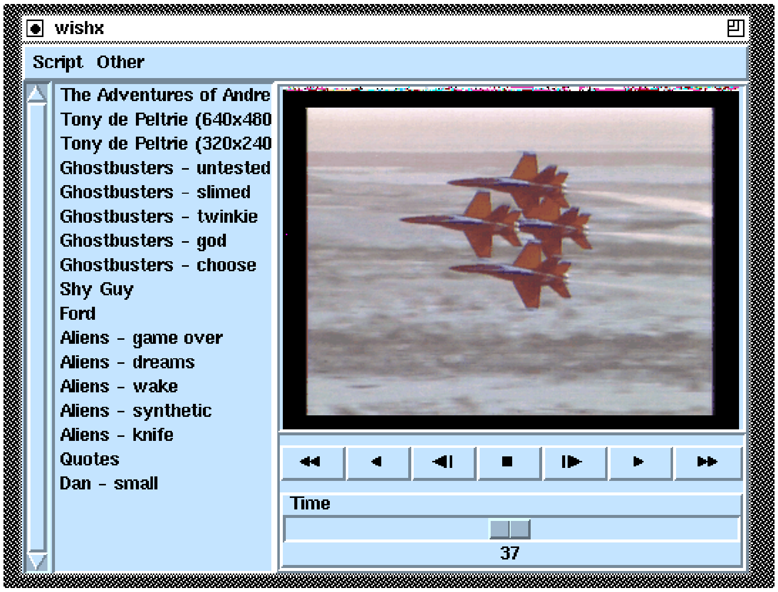

Figure 1. CM player user interface.

2. System Architecture

Figure 2 shows the architecture of the player. The playback application controls the user interface and the CM Server process. The application is responsible for creating windows, responding to input events, and sending commands to the CM Server.

The CM Server receives CM data from the CM Source and dispatches it to the appropriate output device (e.g., the DSP chip to play audio or the video window to play video). The CM Server has a time-ordered play queue to synchronize the playing of audio and video packets. It communicates with CM Source processes on the file server through interprocess communication channels, and it communicates with the X server through shared memory. The system clocks on the different systems are synchronized by the Network Time Protocol (NTP) [11] so that actions in the CM Server and Sources can be synchronized.

The CM Server will eventually be merged with the X server as in the ACME Server [1], but for now it is convenient to separate the functionality for several reasons. First, it makes the CM Server easy to change. Second, it reduces maintenance when a new X server is delivered since we do not have to retrofit our changes. Lastly, source code for the X server is not required which is important because we want to use commercial video boards. Commercial video boards usually include a modified X server for which source code is often difficult to obtain.

The CM Source processes read CM data and send it to the CM Server. CM data is sent in 8k packets on a UDP connection. We have implemented retransmission and adaptive flow control to improve reliability, throughput, and playback quality. Eventu-

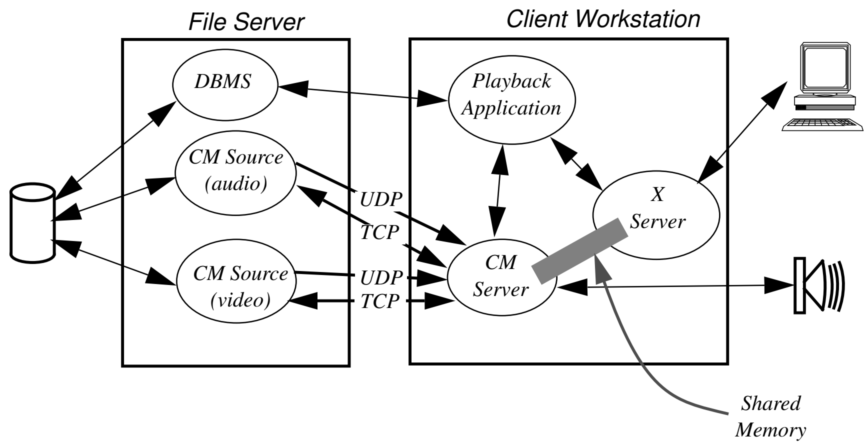

Figure 2. CP Player Architecture.

ally, we will implement a single CM Source server process rather than using a separate process for each stream.

Meta data about scripts is stored in a database. The raw CM data is stored in binary large object (*blob*) files. The meta data is separated from the raw CM data so that different scripts can include overlapping clips without having to make a copy of the CM data.

The remainder of this section describes the CM data model, the CM server abstractions, the CM network protocol, and their implementation.

2.1 CM Data Model

Figure 3 shows a logical picture of a script. Each stream is composed of a sequence of *clips* that represent a sequence of *frames*. A frame is a playable unit such as an image, a frame of video, or a block of audio samples. A clip is a contiguous sequence of frames stored in a blob file. The script has a *logical time system* (LTS) to which frames are synchronized.

CM data is stored in files and the meta data that represents the script is stored in a separate database. The database design for the meta data is shown in figure 4. The

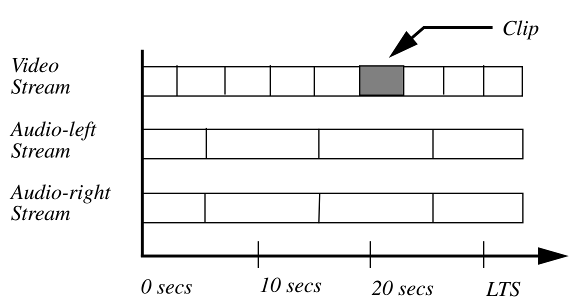

Figure 3. Script representation.

type cmtype: (AUDIO, VIDEO, ANIMATION, IMAGE, UNKNOWN)

type cmformat: (JPEGPLX, SPARCAUDIO, GIF, MPEGPARIS, ...)

class SCRIPT(scriptOID: *oid*, name: *string*, owner, *string*, source:*string*, duration: *ltstimeperiod*, micon: *image*, comment: *string*)

class STREAM(streamOID: *oid*, name: *string*, type: *cmtype*, script: *oid*)

class STREAMREP(format: *cmformat*, maxBufferSize: *integer*, maxFrameRate:*integer*, minFrameRate: *integer*, imageWidth: *integer*, imageHeight: *integer*, imageDepth: *integer*, audioSampleRate: *integer*, clipSeq: *oid*)

class CLIPSEQ(clipSeqOID: *oid*, seqNum: *integer*, blob: *oid*, startFrame: *integer,* endFrame: *integer*, duration: *ltstimeperiod*)

class BLOB(blobOID: *oid*, name: *string*, numFrames: *integer*, type: *cmtype*, format: *cmformat*)

class VIDEOBLOB(frameRate: *integer*, width: *integer*, height: *integer*, depth:*integer*, maxFrameSize: *integer*) **inherits from** (BLOB)

class JPEGPLXBLOB(qFactor: *integer*, timecode[]: *timecode*, startTime[]: *ltstime*, endTime[]: *ltstime*, frame[]: *byteoffset*) **inherits from** (VIDEOBLOB)

class MPEGPARISBLOB(startTime[]: *ltstime*, forwardEndTime[]: *ltstime*, reverseEndTime[]: *ltstime*) **inherits from** (VIDEOBLOB)

class AUDIOBLOB(sampleRate: *integer*) **inherits from** (BLOB)

class SPARCAUDIOBLOB() **inherits from** (AUDIOBLOB)

class GIFBLOB(width: *integer*, height: *integer*, depth: *integer*, colorTable[]:*color*) **inherits from** (BLOB)

Figure 4. Database design for script meta data.

schema is specified using an object-oriented data model with inheritance, object identifiers, and user defined attribute types including arrays. Several points should be noted about the design. First, a stream may have several representations (c.f. *STREAMREP*) so a script can be played on workstations with different compression hardware and output devices.

Second, time is represented either by a timecode (i.e., hours:mins:secs:frame) or an LTS time. Script and clip durations are stored so that applications can support a slider and operations to seek to a particular time. Video clips include a start and end time for each frame to support playing forwards and backwards using the mapping from logical time (i.e., LTS time) to system time described below. MPEG clips need both a forward and reverse end time to recover synchronization on dropped frames.

Third, an instance of a BLOB class such as *JPEGPLXBLOB*, *SPARCAUDIOBLOB*, and *GIFBLOB* represents a blob file that contains data in that format. We expect this meta data to be replicated in the blob file. The design accommodates the addition of new file types such as Apple's Quicktime files [14].

Lastly, we expect blob files to move between different levels of a storage hierarchy that will include local disks at a workstation, large video file servers, and near-line tertiary stores such as an optical disk or a robot tape jukebox. The mapping from the name of a blob file to a location on which it resides will be handled by a dynamic name server.

2.2 CM Server Abstractions

The CM Server is an event driven process that uses a time-ordered priority queue. Events come from many sources including system clock events, network events (e.g., receive packet or remote procedure calls), X events, and idle events (e.g., software decompression processing). The Server receives packets from the CM source, performs required local processing (e.g., assembles data from several packets into playable units, requests retransmission of missing packets, etc.), calculates the system time at which frames should be played, and queues the play request. Every queued play request has a time period during which it must be executed that is represented by an *earliest start time* and a *latest start time*.

At some later time, provided the Server was able to process the queued play request within the designated time period, the request is executed. Examples of play requests are "put image in video window" or "send packet to audio device." If the Server gets behind, the late request is dropped.

An important feature of the system is that audio frames will be played at the right time regardless of whether the synchronized video frames are played because audio play requests are given high priority. Consequently, audio plays smoothly even when video frames are being dropped.

The mapping from logical time to system time is

$$\mathrm{LTS} = \mathit{Speed} \times (\mathrm{SystemClock} - \mathit{Start})$$

where *Speed* is the rate at which the script is being played and *Start* is the *SystemClock* time for *LTS* equal zero. The advantage of this abstraction is that conventional VCR controls can be implemented by setting the *Speed* and *Start* variables as follows:

Function	**Implementation**
stop/pause	Speed := 0
play forward	Speed := 1
play backward	Speed := -1
goto *lts*	Start := *now* - *lts*
step forward	Start := Start + 1 / *fps*
step backward	Start := Start - 1 / *fps*
fast forward	Speed := 2.5
fast reverse	Speed := -2.5

Speed represents the relationship between LTS time and *SystemClock* time. *Speed* equal one means LTS time advances at a real-time rate, *Speed* equals 2 implies that LTS time should advance at twice the rate of *SystemClock* time, and so forth. This definition is better than using frame rate as a metric for *speed* because frames per second (fps) can vary during a stream.

Notice that the fast forward and backward speed can be varied. This capability allows an application to implement a jog-shuttle control similar to the mechanical controls found on some video tape recorders.

Another feature implemented in the CM Server to produce high quality user interfaces is resampling audio data in real-time so that synchronized sound can be played when playing a script backwards or forwards at speeds other than normal. Taken together, prioritizing audio packets higher and audio resampling produce a perceptibly better user interface.

2.3 CM Network Protocol

The CM network protocol was implemented when we discovered that a normal TCP connection incurred too much overhead and was too slow.

CM Source processes send packets one second before they are needed. This delay is insignificant when the user begins to play the script, but it gives the Server a buffer against delayed packets due to network or file server load. Our experience has been that audio packets always are delivered on time, but that video packets are often delayed because the data volume and rate is beyond the capabilities of the system. This point is discussed in more detail in the next section. The CM Server periodically requests retransmission of lost packets.

The Server uses an adaptive feedback algorithm to match packet flow to the available resources.[1] The flow rate is based on the *fps* being played. Every 300 msec the CM Server calculates a *penalty* of 10 points if a frame is queued, but not played (i.e., missed) and 10 points if two consecutive frames are missed. For example, if two consecutive frames are missed, the *penalty* is 30 points. In addition, a 10 penalty is assessed if a frame was lost in the network. The maximum allowable penalty in a time period is 100 points. Thus, a *penalty* of 0 means every frame was played and a *penalty* of 100 means many frames were missed.

The *penalty* is sent to the CM Source. Each stream has a minimum and maximum frame rate specified either in the database or when play was initiated. The CM Source also maintains a current frame rate at which the stream is being played. The Source uses the *penalty* to adjust the current frame rate as follows

$$currentRate = currentRate \times \left(1 - \frac{penalty}{100}\right) + minimumRate \times \left(\frac{penalty}{100}\right)$$

Thus if the penalty is 0, no adjustment is made. If the *penalty* is between 0 and 100, the current rate is reduced. If the *penalty* is 100, the current rate is set to the minimum rate.

---

[1] The real-time IP protocol will guarantee a bandwidth when the connection is established. However, this adaptive mechanism will still be required because the delivery rate that can be guaranteed may be below the rate required by the script.

At the same time the CM Source periodically increments the current rate until the maximum rate is achieved. The effect of this algorithm is to slow the rate quickly when the system is overloaded and to increase it incrementally to an achievable throughput. The system may be overloaded because of contention with other processes or because the video being played requires too much bandwidth. This point is discussed in more detail below.

We believe that reduced variability in the frame rate produces higher quality video playback than minimizing the number of dropped frames. Users are more sensitive to random frame drops than to regular drops. Consequently, the adaptive algorithm attempts to reduce the variation of *fps* played.

2.4 Implementation

All processes in the player are implemented with the Tool Command Language (Tcl) and the Tcl Toolkit (Tk) [12, 13]. Altogether, the player is approximately 20K lines of code of which 10% is written in Tcl. The application process uses both Tcl and Tk, includes 1.7K lines of code, and requires 1.5 MBytes at runtime. The CM Server and Source only use Tcl and a library of Tcl and C code developed for distributed applications (e.g., an RPC mechanism, client/server abstractions, CM abstractions, etc.). The library is 9K lines of C code. The CM Server and Source each have 500 lines of Tcl code and 4K lines of C code. The Server requires 1.8 MBytes at runtime and each Source is about 0.6 MBytes at runtime.

The majority of the communication between processes is accomplished by sending Tcl commands to a remote process to be executed. These commands are sent as strings, evaluated by the embedded Tcl interpreter in the remote process, and a string-valued result is returned. This mechanism is a simple remote procedure call (RPC). The application was very easy to develop because remote commands could just be defined and sent rather than requiring the definition of a shared header file that was compiled by a stub compiler as in other RPC mechanisms. Another advantage of Tcl/Tk is that it is very easy to prototype abstractions in Tcl, and when a time critical abstraction is discovered, it can be recoded in C. The entire application was written in under 10 person-months.

3. Performance

This section reports the results of some preliminary performance experiments. Two video scripts were used: “The Adventures of Andre and Wally B” and “Tony De Peltrie” [4]. Both are 24 *fps* computer graphics generated videos. Wally was digitized at 320 by 240 pixels and Tony was digitized at 640 by 480 pixels. Both streams were JPEG compressed using the XVIDEO board. The following table shows the static size of the data.

Video	Total Number Frames	Minimum KB/Frame	Maximum KB/Frame	Average KB/Frame	StdDev KB/ Frame
Wally	1806	7.7	12.7	11.3	1.3
Tony	2530	12.8	24.9	20.9	1.6

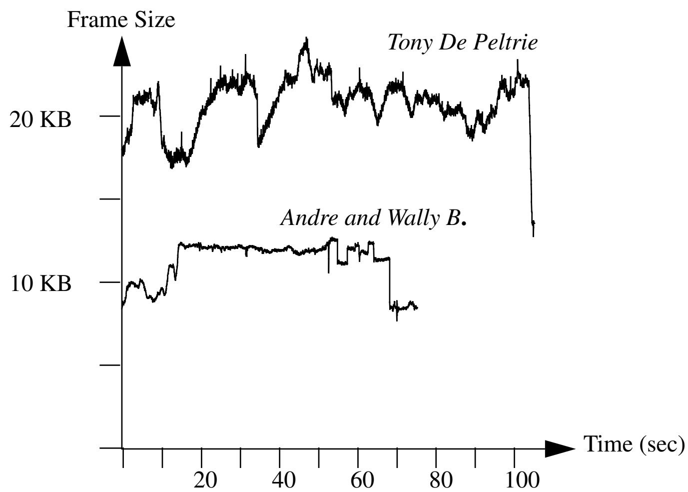

Figure 5. Frame size versus time plot.

Figure 5 plots frame size versus time. Notice that Wally remains somewhat constant at 12KB/frame whereas Tony varies between 13KB/frame and 25KB/frame.

When we play these videos, all audio packets are received and played at the right time, but depending on which video is being played and the system load at the time, video frames are dropped. Wally plays correctly most of the time, but Tony always drops many frames. The problem is the maximum throughput required to play the videos. For example, the maximum throughput required by Wally is 305KB/sec (2.4 Mbit/sec) whereas Tony requires 598KB/sec (4.8 Mbit/sec). The problem is either in the network or the file server, because essentially all frames received by the CM Server are played (e.g., one video frame is received but not played about every 2 seconds). We ran experiments on both ethernet and FDDI networks and the same problems were observed, so it is not the throughput on the network. Since essentially all packets sent by the server are successfully received, we conclude that the bottleneck is in the file server.

Figure 6 shows the effect of varying the requested play rate of the Tony video without the adaptive frame rate control algorithm. The figure plots the number of frames played per second at requested playback rates of 12, 16, and 24 *fps* versus time. As you can see, at 12 and 16 *fps* most frames are played but that many frames are dropped at 24 *fps*. Further investigation suggests that the problem is the number of packets per second that must be sent by the CM Source. The limiting factor appears to be the overhead of sending a packet.

Figure 7 plots the requested frame rate, shown by the thick line, and the actual frame rate, shown by the thin line, with the adaptive frame rate control algorithm. This plot shows the requested frames and the played frames when playing Tony with bounds of 11 to 16 *fps*. You can see that the play rate closely follows the requested rate. Although it does not show in these graphs, the quality of the video is perceptibly better when rate control is used. However, playback could still be improved if predictive data on the

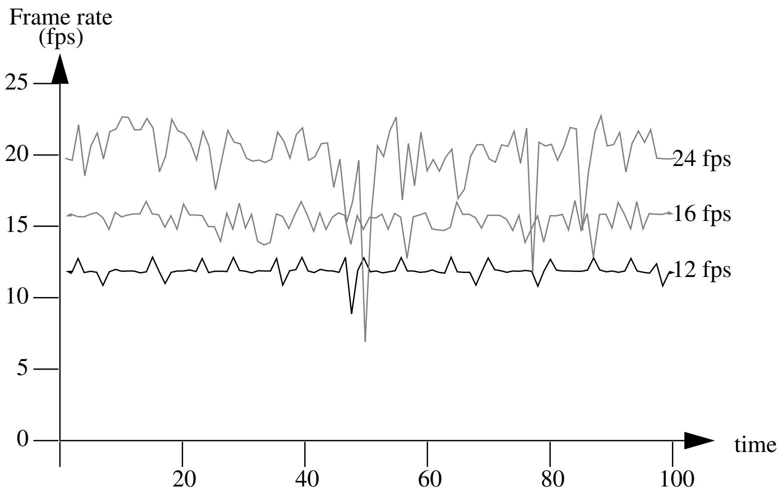

Figure 6. Frames played versus time without adaptive control.

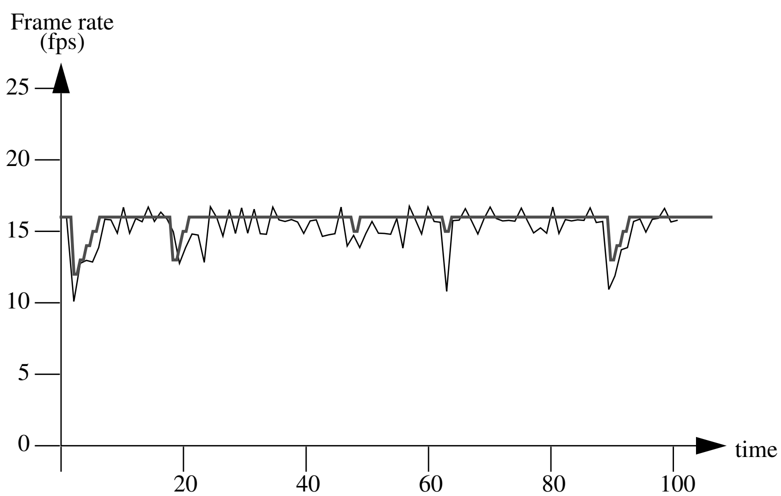

Figure 7. Frames played versus time with adaptive control

bandwidth that will be required during the next few seconds was available so the feedback algorithm could begin reducing the frame rate in anticipation of a change in the required bandwidth.

4. Related Work

Many groups are working on multimedia applications that include playing continuous media [1, 2, 3, 6, 7, 8, 14, 15, 19]. None of these systems report an applicationlevel adaptive control algorithm to vary frame rate dynamically. More research is needed to validate the claims made here and to explore algorithms that look ahead at future resource requirements as suggested by Little and Ghafoor [9]. In addition, we believe that this system is the first one to use NTP and globally synchronized clocks to synchronize transmission of CM data between processes on different machines. The LTS abstraction we used is similar to an analogous abstraction used in Apple's Quicktime.

The synchronization model is similar to the time-based models described by others. We eventually plan to add hierarchical synchronization similar to the model suggested by Steinmetz [16].

Lastly, the data model we developed uses ideas from many sources including [5, 10, 14, 17]. One important point in our design is the replication of meta data in a database that will allow us to manage a tertiary store where CM data can be archived.

5. Conclusions

The design and implementation of a CM player was described. The player uses a globally synchronized clock to synchronize the process that plays the CM data on a client with the file server processes that delivery the data. This design along with synchronizing streams on a logical time system simplified the implementation and made it possible to play audio packets correctly even when video packets were being dropped. Lastly, preliminary performance experiments were reported that illustrate the need for adaptive control of the rate at which video is played and the effect of a simple feedback control algorithm.

Acknowledgments

Several other people contributed to the CM Player software: Steven Yen implemented the playback application, Dan Wallach implemented the audio resampling code, and Greg Paschall worked on an earlier prototype system.

References

[1] Anderson, D.P. and G. Homsy, A continuous media I/O server and its synchronization mechanism, *IEEE Computer* (October 1991).

[2] Blakowski, G., et.al., Tools for specifying and executing synchronized multimedia presentations, *Proc. 2nd Int. Wkshp on Network and OS Support for Digital Audio and Video*, Heidelberg (November 1991).

[3] Coulson, G., et.al., Protocol support for distributed multimedia applications, *Proc. 2nd Int. Wkshp on Network and OS Support for Digital Audio and Video*, Heidelberg (November 1991).

[4] Dream machine the visual computer - an anthology of computer graphics, Laserdisc, The Voyager Company (1986).

[5] Gibbs, S., Composite multimedia and active objects, Proc. OOPSLA '91, Phoenix (October 1991).

[6] Gusella, R. and M. Maresca, Design considerations for a multimedia network distribution center, *Proc. 2nd Int. Wkshp on Network and OS Support for Digital Audio and Video*, Heidelberg (November 1991).

[7] Hodges, M.E., et.al., Athena Muse: a construction set for multi-media applications, *IEEE Software* (January 1989).

[8] Lamparter, B, et.al., X-MOVIE: transmission and presentation of digital movies under X, *Proc. 2nd Int. Wkshp on Network and OS Support for Digital Audio and Video*, Heidelberg (November 1991).

[9] Little, T.D.C. and A. Ghafoor, Scheduling of bandwidth-constrained multimedia traffic, *Proc. 2nd Int. Wkshp on Network and OS Support for Digital Audio and Video*, Heidelberg (November 1991).

[10] Little, T.D.C. and A. Ghafoor, Synchronization and storage models for multimedia objects, *IEEE Journal on Selected Areas in Comm.*, 8 (3) (April 1990).

[11] Mills, D., Measured performance of the network time protocol in the internet system, Network Working Group, RFC 1128 (October 1988).

[12] Ousterhout, J., Tcl an embedded command language, *Proc. 1990 Winter Usenix Conference* (1990).

[13] Ousterhout, J., An X11 toolkit based on the tcl language, *Proc. 1991 Winter Usenix Conference* (1991).

[14] Quicktime, software product, Apple Computer Inc. (1991).

[15] Rosenberg, J., et.al., Presenting Multimedia documents over a digital network, *Proc. 2nd Int. Wkshp on Network and OS Support for Digital Audio and Video*, Heidelberg (November 1991).

[16] Steinmetz, R., Synchronization properties in multimedia systems, *IEEE Journal on Selected Areas in Comm.*, 8 (3) (April 1990).

[17] Stevens, S.M., Embedding knowledge in continuous time media, *Proc. 2nd Int. Wkshp on Network and OS Support for Digital Audio and Video*, Heidelberg (November 1991).

[18] Verma, D., and H. Zhang, Design documents for RTIP/RMTP, unpublished manuscript (1991).

[19] XMedia Toolkit, software product, Digital Equipment Corp. (1992).