Design and Performance of the Berkeley Continuous Media Toolkit

Ketan Mayer-Patel

Master's of Science in Engineering - Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720-1776
Prof. Lawrence A. Rowe, Advisor

ABSTRACT

The design and performance of the Berkeley Continuous Media Toolkit (CMT) is described. CMT provides a programming environment for rapid development of continuous media applications. CMT overhead is measured in the context of a simple video playback application and is found to be only a few milliseconds per frame played. As a demonstration of CMT as a research infrastructure, an experiment comparing adaptive frame rate control policies is described.

Keywords: continuous media, toolkit, prototyping.

1.0 INTRODUCTION

This paper describes the design of the Berkeley Continuous Media Toolkit (CMT) and the results of experiments conducted to test the performance of the system. CMT provides a rapid development environment for multimedia applications that use continuous media (e.g., video, audio, etc.). The system provides objects to perform media specific operations (e.g., capture, store, play, etc.) and a simple programming model for building applications. The goal of this paper is to show the viability of a general-purpose, portable continuous media toolkit by quantifying the overhead costs of CMT in the context of a simple video playback application and by demonstrating the use of CMT as a research tool. Measurements of CMT performance indicate that the toolkit overhead is only a few milliseconds per media unit (i.e., frame of video) for basic playback functionality. The flexibility of CMT is demonstrated in a simple experiment designed to measure the performance of a variety of adaptive rate control feedback algorithms for delivering and playing MPEG and MJPEG data.

Current continuous media research operates on a variety of different levels. At the lowest level, new media compression standards (e.g., MPEG[6], h.261[8], source channel encoding schemes, etc.) and communication protocols (e.g., RTP[20], RSVP[26], etc.) are explored. Often these new schemes are specifically designed for particular applications (e.g., video conferencing, animation, etc.) and are concerned with issues such as scalability and error concealment. The development and performance of specific applications is another level of continuous media research. Vic[12] and vat are examples of these research efforts. At the highest level, continuous media research involves large systems such as video-on-demand (VOD) and interactive television. A viable continuous media research system would allow different researchers to conduct experiments at any of these levels and utilize system infrastructure and applications at other levels to test end-to-end performance.

CMT was developed to facilitate the construction of new applications and provide a framework for research at all levels. CMT is an extensible, open system, and it is flexible enough to support a variety of application models. Cross-platform portability is provided to operate in the heterogenous environment of the Internet. Using CMT, it is possible to conduct experiments without having to reinvent and implement an entire multimedia system. For example, using CMT researchers have investigated a number of issues involved with continuous media including: 1) a command stream media type[4], 2) a protocol for delivering continuous media data over an ATM network, 3) a real time communication protocol[23], 4) new compression algorithms (e.g., VQ[10] and 3D subband coding[24]), and 5) new applications (e.g., software production video switcher[21], MBone question board[11], etc.). The cost of CMT is the overhead incurred to provide modular, media specific objects. My goal is to show that this overhead cost is small relative to the work required to manipulate continuous media data. The experiments presented here demon-

strate that continuous media applications reach bottlenecks in other basic system operations, such as disk I/O and software decoding, before CMT overhead becomes a limiting factor.

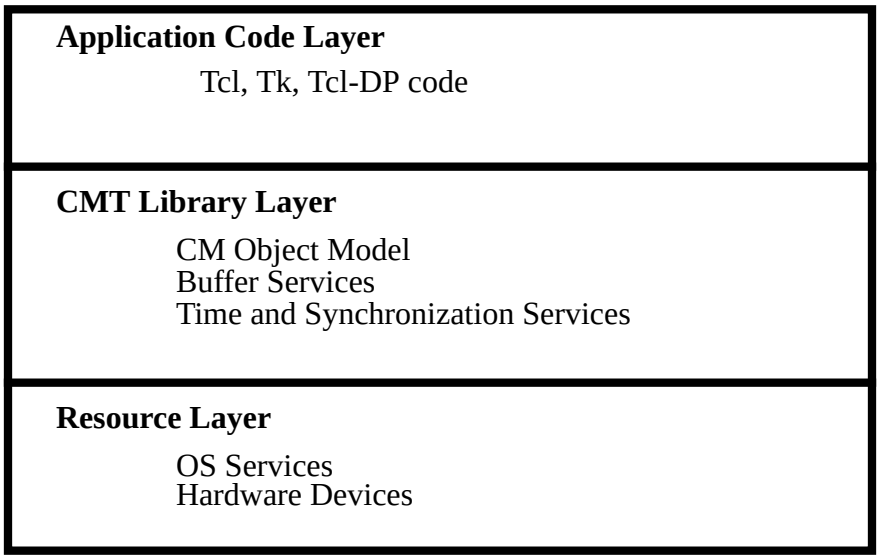
Related work in multimedia toolkits include a variety of research systems and commercial products. The ViewStation[25] project at MIT most closely matches the architecture of CMT. It was designed to operate on high-speed local area networks. The DAVE[15] system developed at Sandia Laboratories provides a high-level abstraction to continuous media devices. It, however, assumes that devices are sampled at regular time intervals and does not provide as flexible a time model as CMT. The Multimedia Component Kit[3] from Switzerland provides a C++ class library as an abstraction to specific devices but uses a separate analog transmission network for the delivery of continuous media data. Quicktime Conferencing[1] (QTC) from Apple is a commercial system that provides an API to Quicktime codecs and network modules for the development of video conferencing applications. InSoft's OpenDVE[5] toolkit is another commercial development system providing an API for application programmers to create collaborative and desktop video conferencing applications. Lastly, the ActiveMovie[13] architecture from Microsoft is greater in scope than either QTC or OpenDVE providing an API integrated with other development API's such as Direct3D[14] (a 3-D graphics API).

The remainder of this paper is organized into four sections. The design of the CMT system is presented in Section 2. Measurements of CMT system overhead are presented in Section 3. Section 4 demonstrates the use of CMT as a continuous media research infrastructure. Finally, Section 5 contains a discussion of CMT's use as a research tool and its future.

2.0 CMT SYSTEM DESIGN

This section describes the design of CMT and how it is used to build applications. The CMT system architecture is shown in Figure 1. At the top level, the application code layer represents the programming environment provided to application developers. CMT uses Tcl/Tk[16,17] as the programming language. A Tcl interpreter is extended with Tcl-DP[22] to provide distributed programming support on which CMT abstractions are built. The middle layer is the CMT library layer that extends the Tcl interpreter with continuous media objects and a variety of services. The bottom layer is the hardware and operating system resources layer which represents the CPU, network interface, memory, I/O devices, and other hardware. The following subsections describe the application code and CMT library layers in more detail.

FIGURE 1. CMT System Architecture



2.1 THE APPLICATION CODE LAYER

The use of Tcl/Tk in the application code layer provides the application developer with a programming interface that is already widely used and highly portable. Because Tcl/Tk is interpreted, application development does not require lengthy compilation, and the programmer can directly interact with the interpreter to change program behavior. In addition, Tk provides a powerful user interface toolkit that can be used to create continuous media applications that require complex user interfaces.

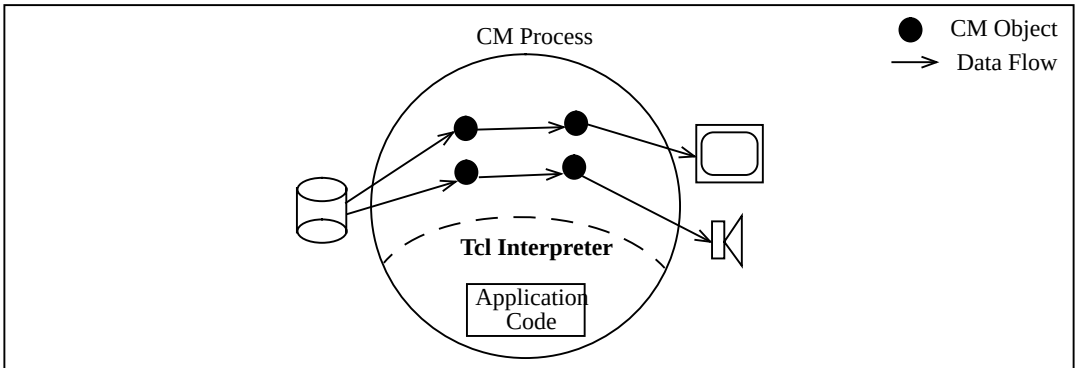
The interpreter used by CMT is extended with Tcl-DP, a distributed programming package built on top of Tcl. By using Tcl-DP, client/server and other distributed applications can be built. Three possible playback application models are shown in Figures 2, 3, and 4 with example Tcl code that implements each model. The first model is a local playback model. All CMT objects that implement this model exist in the same process. The second model is a more complicated distributed model. The application code executed in one CMT process creates a connection to another CMT process. CMT objects used to implement the playback application are created in both processes. In the third model, the application code executes in one process and connects to two different CMT processes. It creates and controls CMT objects in both. In this third model, continuous media data only flows between CMT objects in the two remote processes even though the application code itself is in a third process. The example Tcl code used to create the CMT objects for each application is remarkably similar despite the difference in complexity.

2.2 THE CMT LIBRARY LAYER

The CMT library layer is composed of extensions to the Tcl interpreter to support continuous media objects (e.g., video capture, audio play, etc.), buffer management services, time and synchronization services, continuous media event mechanisms, and continuous media stream and storage abstractions.

The continuous media object model is where the bulk of CMT functionality lies. CMT objects are created and controlled in much the same way as Tk user interface widgets. Each instance of a CMT object provides a mechanism for setting attributes and invoking object methods. CMT objects operate with the basic premise of receiving media-specific data from either a

FIGURE 2. Local Playback Application Model

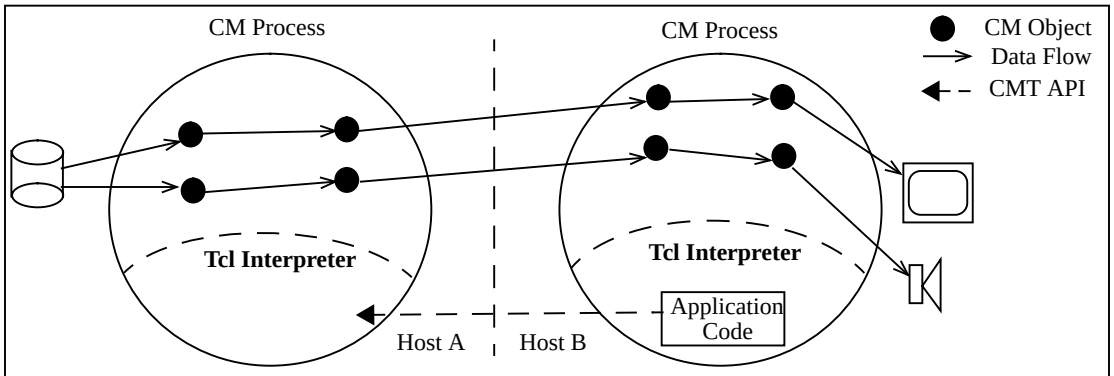



```
# This code creates the objects necessary to implement the
# application model above.

# Create a CMT API object for the local process.
set cmt_api [cmt “”];
$cmt_api open -local;

# Create objects for video stream.
set mpeg_file_obj [$cmt_api create mpegFile];
set mpeg_play_obj [$cmt_api create mpegPlay];

# Create objects for audio stream.
set au_file_obj [$cmt_api create auFile];
set au_speaker_obj [$cmt_api create auPlay];
```

device (e.g., disk, camera, microphone, etc.) or another object, operating on the data in some way, and then sending the data to either a device (e.g., disk, video display, speaker, etc.) or another object. Creating objects and linking them together creates a "pipeline" through which continuous media data flows. CMT objects are classified into four categories: sources, sinks, filters, and transport pairs. Source objects generate continuous media data, often reading or receiving data from devices. Source objects send data to sinks, filters, or transport objects. Examples of source objects are file, camera, and microphone objects. Sink objects receive data from source, filter, or transport objects without passing the data along to any other object. Record, display, and speaker objects are examples of sinks. Filter objects receive data from source, filter, and transport objects, perhaps transforming the data in some way, and then send the data on to filter, transport, or sink objects. Examples of filter objects include priority objects that reorder continuous media data according to some prioritizing algorithm and transcode objects that transform one media format to another format (e.g., MPEG video to H.261 video). Transport objects are responsible for sending and receiving data across a network. Typically, transport objects come in send/receive pairs where a send object communicates with and sends data to a receive object of the same transport type. Examples of transport objects are UDP and RTP send and receive objects.

Continuous media data is passed between the objects using either a push model or a pull model. To clarify explanation of the two models of data passing, consider the following example. Suppose that data is to be passed between two objects. The object that has the data will be called the producer. A producer may be a source, filter or transport receive object. The object to which

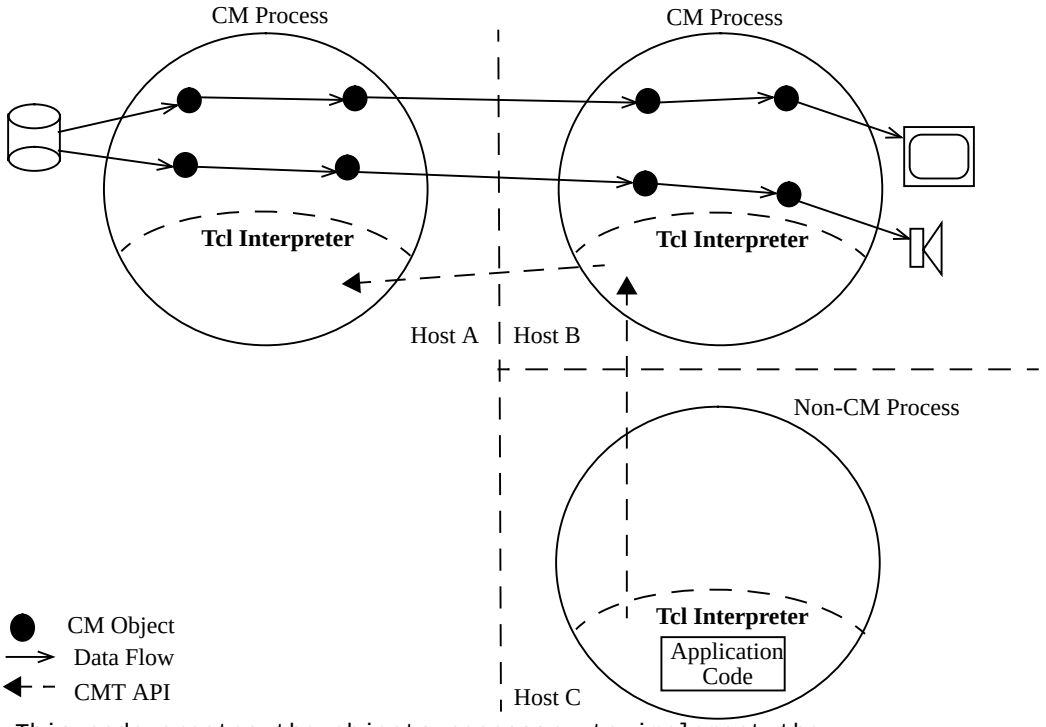
FIGURE 3. Simple Remote Playback Application Model




```
# This code creates the objects necessary to implement the
# application model above.

# Create a CMT API object for the local process.
set local_cmt [cmt ""];
$local_cmt open -local;

# Create a CMT API object for the remote process.
set remote_cmt [cmt ""];
$remote_cmt open -host HostA -port 6000

# Create objects for video stream.
set mpeg_file_obj [$remote_cmt create mpegFile];
set mpeg_send_obj [$remote_cmt create pktSrc];
set mpeg_recv_obj [$local_cmt create pktDest];
set mpeg_play_obj [$local_cmt create mpegPlay];

# Create objects for audio stream.
set au_file_obj [$remote_cmt create auFile];
set au_send_obj [$remote_cmt create pktSrc];
set au_recv_obj [$local_cmt create pktDest];
set au_speaker_obj [$local_cmt create auPlay];
```

the data is being passed will be called the consumer. The consumer can be a sink, filter or transport send object. In the push model, the producer initiates the transfer of data by invoking a method of the consumer and providing a representation of the data as a parameter to the method. In the pull model, the consumer initiates the data transfer by invoking a method of the producer which will result in a representation of the data to be transferred. The specific method to be called by either the producer in the push model or the consumer in the pull model is an attribute set by the application programmer. This attribute can be changed at any time, and it can be any arbitrary Tcl command, allowing the programmer to create Tcl procedures that act as data producers and consumers. For example, a programmer can define a Tcl procedure that accepts a Tcl representation of continuous media data which in turn calls the accept method of two or more consumer objects. In this case, the programmer was able to construct a simple multiplexer between one producer object and many consumer objects.

FIGURE 4. Complex Remote Playback Application Model




```
# This code creates the objects necessary to implement the
# application model above.

# Create a CMT API object for the remote process on Host B.
set hostb_cmt [cmt “”];
$hostb_cmt open -host HostB -port 6000;

# Create a CMT API object for the remote process on Host A.
set hosta_cmt [$hostb_cmt create cmt];
$hosta_cmt open -host HostA -port 6000

# Create objects for video stream.
set mpeg_file_obj [$hosta_cmt create mpegFile];
set mpeg_send_obj [$hosta_cmt create pktSrc];
set mpeg_recv_obj [$hostb_cmt create pktDest];
set mpeg_play_obj [$hostb_cmt create mpegPlay];

# Create objects for audio stream.
set au_file_obj [$hosta_cmt create auFile];
set au_send_obj [$hosta_cmt create pktSrc];
set au_recv_obj [$hostb_cmt create pktDest];
set au_speaker_obj [$hostb_cmt create auPlay];
```

The representation of continuous media data in CMT is handled by the CMT buffer manager. The buffer manager is a service provided by the CMT library to both the application programmer and CMT objects. The buffer manager supports shared memory segments that may be attached to a windowing system (e.g., X, Windows, etc.), tagging buffers with labels, reference counting to allow the same buffer of data to be used by more than one object at a time, and buffer reuse. Representing media data in memory buffers is done with a structure called a scatter buffer list. This structure allows different memory buffers to be treated as one contiguous piece of memory, thereby avoiding the need to copy large areas of memory to do simple tasks such as appending a network protocol header structure. Tcl and C representations of scatter buffer lists are defined to be used by CMT objects when passing data.

The CMT Logical Time System (LTS) is provided in the CMT library layer. It allows applications to maintain a concept of where in time the application is and how fast time is progressing. More than one LTS can be constructed and different LTS's can be synchronized to each other. Two LTS's in different CMT processes on different machines on a network can also be synchronized. An LTS is essentially a mapping between system time (i.e., the machine's internal clock) and an infinite logical timeline centered on the value 0. An LTS has three components: speed, value, and offset. The value of an LTS determines a point along the logical timeline. The speed of an LTS determines how fast the value of the LTS is changing. The LTS offset establishes a relationship between logical time and system time. Conversion between logical time and system time is done with the following formula:

$$value = speed * system\ clock + offset$$

Each continuous media data unit (e.g., frame of video, block of audio samples, etc.) is mapped to a segment of logical time. Objects use LTS's to schedule action taken with continuous media data by converting logical time to system time. Each object using an LTS is notified whenever an application explicitly changes the value or speed of the LTS so it can take appropriate action and possibly schedule future action. In this way, random access into a continuous media data stream can be implemented by simply changing the value of the LTS. Time can be sped up or slowed down by setting speed to values greater than 1 or less than 1 respectively. Time can be made to flow backwards by setting speed to be negative. Time can be stopped by setting speed to 0.

The LTS mechanism is free of any assumptions about the relationship between continuous media data units. Each data unit is mapped to logical time independently, allowing for non-uniform media rates (i.e., frame rates). Media data units may even be overlapped in time. The interpretation of irregular time mapping constructions is media and object specific.

The CMT library layer provides event handling through a collection of different services and mechanisms including the *at queue, priority groups*, and the *cmBind* mechanism. An *at queue* allows CM objects to schedule a C callback to be made at a particular point in system time. The object can specify a range in time when the callback is to be made. The object can also arrange for a different callback to be made if the time constraints of the event cannot be met. *Priority groups* is a mechanism for prioritizing the actions of a group of objects over those of another group of objects. A Tcl interface is available to the application programmer to create and order priority groups. The priority group of an object is generally an attribute that can be set by the application programmer. This allows, for example, the programmer to prioritize the actions of objects involved in an audio stream over actions involved in a video stream. The *cmBind* mechanism is analogous to the Tk bind mechanism that allows an action to be associated with user interface events. CMT objects can register continuous media events with the cmBind mechanism, and the application can associate an action to be taken whenever these events occur. Examples of continuous media events include receiving a media unit of data (i.e., frame of video, block of audio), playing a media unit of data, dropping a media unit of data, and so forth. These events can report internal object data (e.g., decode rates, frame type, etc.) to external monitors[2].

3.0 CMT PERFORMANCE OVERHEAD

This section presents the results of a series of experiments designed to measure overhead in the CMT system. The overhead was found by measuring the performance of a simple playback application called the CMPlayer. The CMPlayer can be used to play video and audio stored in local files as well as streams that are served from a simple server process. The CMPlayer implements the first and the second application models for local and remote playback in Figures 2 and 3 respectively.

Unfortunately, traditional profiling is an inadequate measure of system performance since the interactions of CMT objects, especially in terms of CPU scheduling and synchronization, are highly affected by the slow performance of profiled code. To measure the performance overhead incurred by CMT, I constructed four versions of CMPlayer in which I incrementally eliminated processing done on the video. In the first version, the video stream is read from the disk by a file reading object, packaged into frames, sent to a decoding and display object where it is decoded, dithered, and finally displayed on the screen. The second version establishes the same pipeline of objects, but the final decode and display object does not actually dither or display the video. In the third version, the final decode and display object does not decode the data. In the fourth version, the file reading object does not read the data from disk, and the decode and display object does not decode, dither, or display the data. In each version, the objects operate as if they are processing the video normally. They create buffers and headers, pass the buffers to the next object, and schedule processing work. In the fourth version, all overhead involved in processing the video is present without actually processing the video in any way. The performance of each version of the application was measured by attempting to play video at a various frame rates and measuring the actual frame rate achieved. Performance was measured several times with the results averaged and a standard deviation was calculated. A graph of these results has the general shape of a line that initially rises at a 45 degree angle, eventually falls off the 45 degree angle but continues to rise, and finally begins to fall back toward zero. The initial 45 degree rise represents that range of frame rates that the application can easily achieve (i.e., achieved frame rate equals desired frame rate). As desired frame rate exceeds achievable frame rate, the line falls off the 45 degree rise while still increasing until the application's peak sustainable frame rate is achieved. Achieved frame rate falls back toward zero as desired frame rate rises past the sustainable peak because the processing of dropped frames increasingly consumes CPU resources. The difference in peak sustained performance between the different versions of CmPlayer represents the time required for the eliminated operations to be executed per frame played. The peak sustained performance of the final version represents an upperbound on the overhead.

The measurements were performed with an MPEG-I video sequence on an HP 712. Decoding of the MPEG stream was done in software by the decoding and display object. Figure 5 shows the results with the four versions of the application labelled "Full," "No Dither/Display," "No Decode," and "No Read." Table 1 shows the measured upper bounds for each of the processing components in milliseconds per frame played. It is evident from the large jump in peak performance between the "No Dither/Display" version and the "No Decode" version that the bulk of the time required is spent decoding the MPEG frames. The peak sustained performance of the "No Read" version initially establishes CMT overhead per frame played at 5.5 ms.

FIGURE 5. 320x240 MPEG performance, local playback.

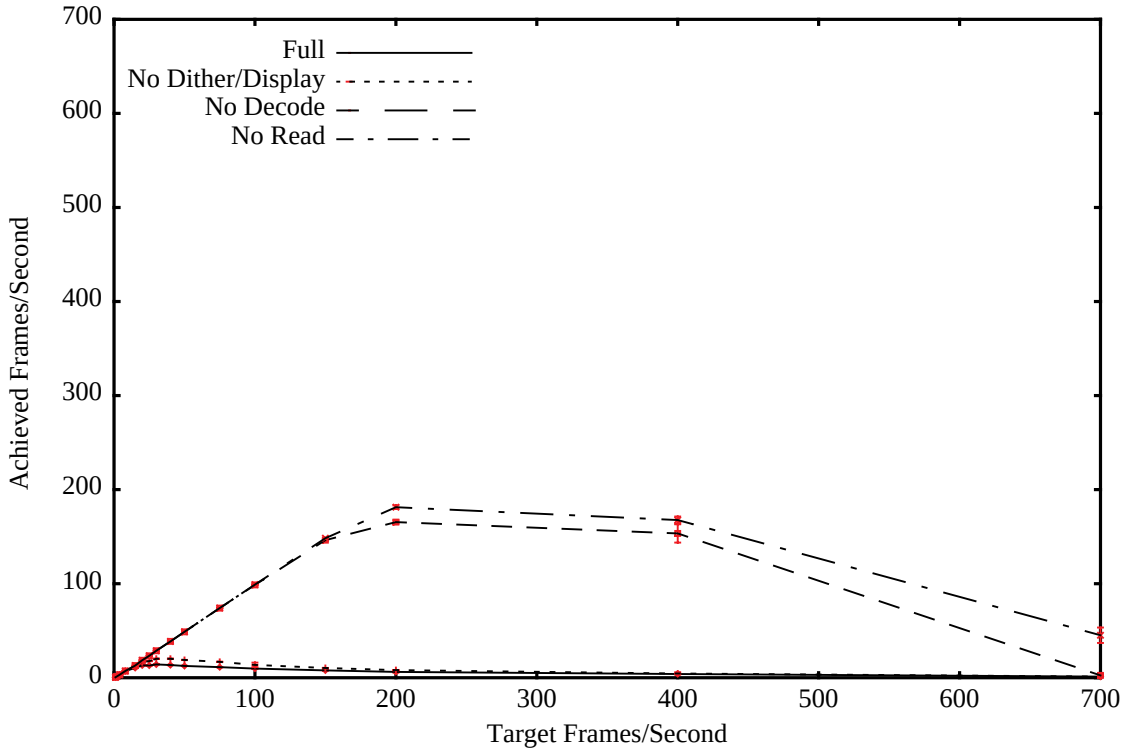


TABLE 1. Measured Upper Bound for Processing Components Per Frame Played

Component	Time (msec)
Dither/Display	21.1
Decode	43.5
Read	0.5
Overhead - Initial Measurement	5.5
Overhead - Optimized Measurement	3.5

Unfortunately, this measurement can not separate the overhead of each frame played from time spent on frames that were dropped (i.e., not read or sent by the file object or discarded by the decode and display object). In the initial experiment, I configured the file object to “read-ahead” and send one second’s worth of data (e.g., 24 frames) at a time. Since many frames are discarded by the decode and display object, the overhead measurement for frames played is inflated. The file object is configured to send multiple frames at a time to trade off the penalty for scheduling future work with the amount of work actually done (e.g., send 5 frames every 1/6 sec. instead of 1 frame every 1/30 sec). The cost of sending more than one frame at a time is blocking all other objects from utilizing the CPU while the frames are being read. To examine the utility of this trade-off as well as to reduce the number of frames sent that will eventually be dropped, the read-ahead of the file object was reduced and performance was remeasured. The performance of the “No Read” version of the application was remeasured five times with the file object configured to read-ahead 1 second, 3/4 second, 1/2 second, 1/4 second, and to send each frame individually (i.e., no read-ahead). These results are presented in Figure 6. Performance improves consistently as the number of frames read at a time decreases. This result implies that the cost for scheduling future work is small and is not worth blocking the CPU for more time than is necessary. Using the peak sustained performance of the system when sending only 1 frame at a time establishes an optimized upper bound for CMT overhead per frame played at only 3.5 msecs.

Given these measurements, it is still unclear how much overhead is incurred in the file reading object and how much is in the decode/display object. By examining the frame receive rate for the decode/display object, we can get a sense of how well the

FIGURE 6. 320x240 MPEG, No Read version with varying read-ahead.

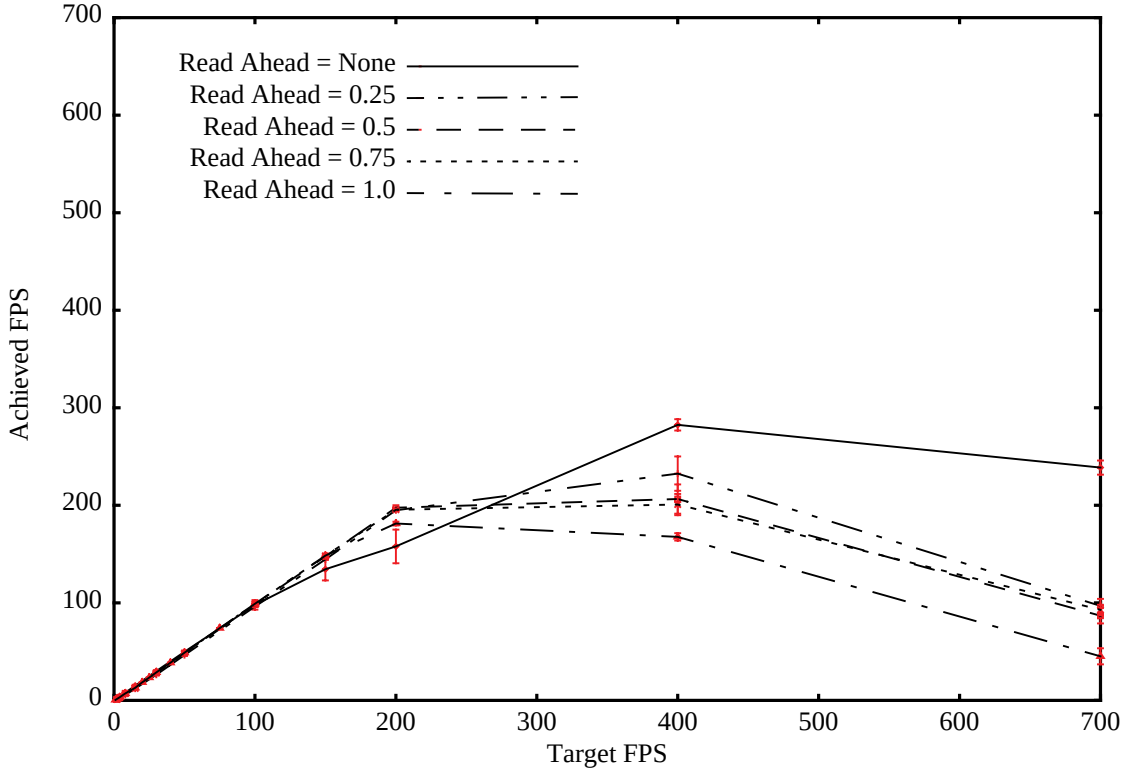


FIGURE 7. Frames Received and Frames Played by Decode/Display object for No Read version.

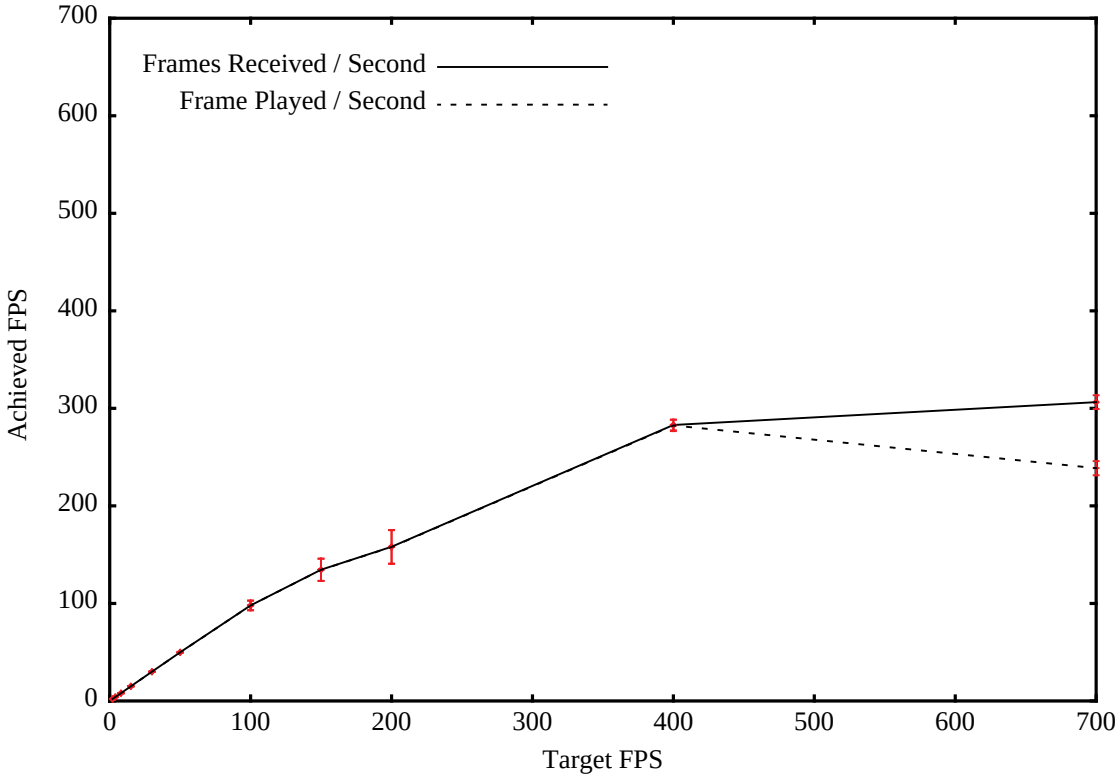
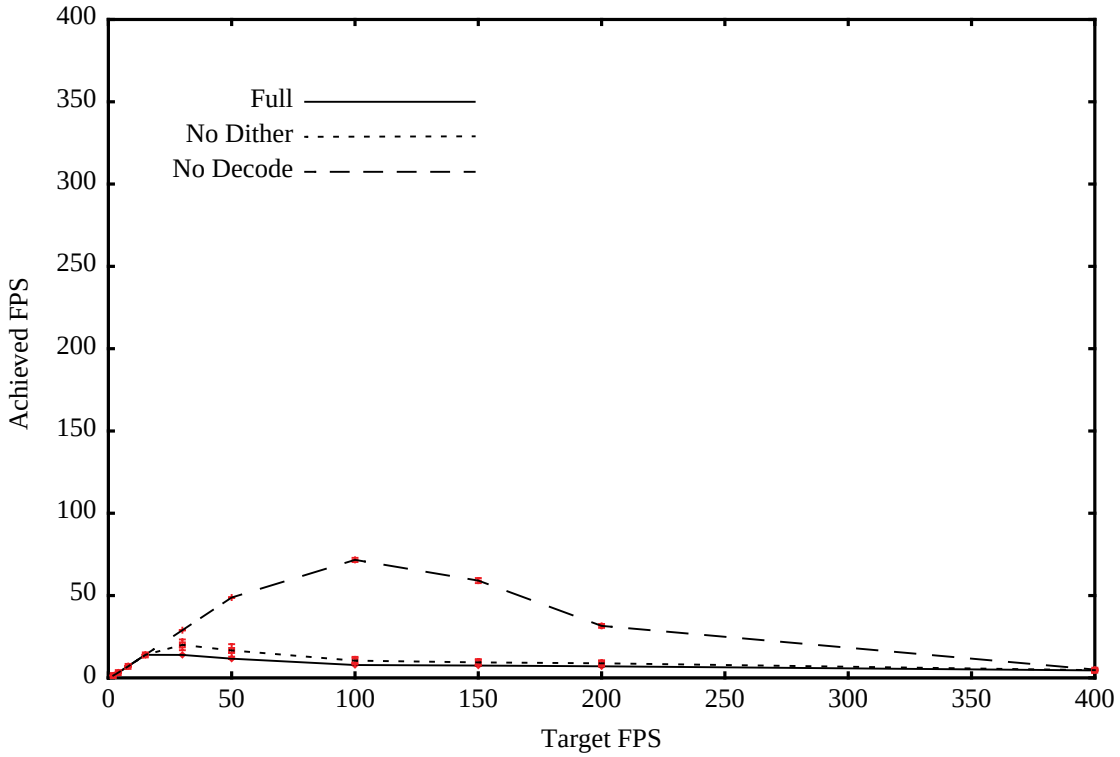


FIGURE 8. 320x240 MPEG, Networked Playback.



file reading object is performing. Figure 7 graphs the number of frames received by the decode/display object along with the number of frames played for the “No Read” case when the file object sends frames one at a time (i.e., no read ahead). This graph shows that system performance is most limited by the file reading object since only 0.4% of frame drops occur in the decode/display object (i.e., the decode/display object plays almost every frame it receives).

In the case of networked playback, I expected that peak sustained system performance would be limited by network bandwidth at some point before reaching the local playback sustained peak performance. Figure 8 shows the results of running CMPlayer with video streamed from a networked video server on a local area network. The “No Read” version of CMPlayer is not shown since the file reading object is no longer in the same process as CMPlayer. Peak sustained performance is greatly reduced and an examination of the frame receive rates shows that the decode/display object only accounted for 3.5% of the frame drops, implying that 97.5% of all frame drops occurred in the network. This result confirms the intuition that network bandwidth is the limiting factor.

To understand how much overhead is incurred by MPEG specific functions (e.g., resolving frame references and dependencies, etc.) as compared to generic functions involved with creating and passing buffers, performance was measured using MJPEG[7] data instead. The MJPEG objects are much simpler since there are no interframe dependencies in the MJPEG stream. Overhead with MJPEG data was measured to be 2.6 msecs per frame played as opposed t 3.5 msecs per frame played for MPEG data. This result indicates that 25% of the overhead measured in the MPEG case can be attributed to the algorithms implemented in the MPEG file reading and decode/display objects which are needed to handle the additional complexity of MPEG data.

4.0 ADAPTIVE RATE CONTROL EXPERIMENT

To demonstrate the use of CMT as a research infrastructure, I conducted a simple experiment to evaluate the use of adaptive rate control in remote video playback. Since cmBind is CMT’s mechanism for performance reporting and feedback, the key to CMT’s usability as a research infrastructure lies in the overhead incurred by cmBind. CmBind overhead is dominated by the time required to evaluate the Tcl scripts associated with various events. These scripts can be arbitrarily complex. The adaptive rate control experiment described below represents a reasonably complex use of cmBind which involves computing various running averages and issuing feedback commands to alter program behavior.

Using the cmBind mechanism, I modified CMPlayer to send feedback control messages to the file reading object in the remote server to attenuate the amount of data sent by the server to match the estimated decoding capabilities of the client process. The parameters of the experiment included: 1) the format of the video, 2) the decoding rate based on frames per second or bandwidth, 3) the frequency of measuring decode performance, and 4) the measurement time period. The experiment was run without frame rate control as a base case. In addition, to measure the overhead of the control mechanism, the experiment was conducted with all control mechanisms in place, but without sending the actual control messages. Table 2 summarizes the parameters of the experiment.

TABLE 2. Parameters of Rate Control Experiment

Parameter	**Possible Values**
Video format	MPEG, MJPEG
Frequency of feedback	Once every second
Basis of rate estimate	Frame Rate, Bandwidth, Empty (estimates made but no feedback done), None (no estimates made, no feedback done).
Period of estimate	Instantaneous, 0.25 seconds, 0.5 seconds, 1.0 second, Running Average

Figures 9 and 10 show the results of the experiment. Although MJPEG streams benefitted slightly from the frame-based rate control mechanism, both bandwidth and frame-based control mechanisms failed for MPEG streams. Although the adaptive frame rate strategies tested failed, the minute difference in performance between the base case of doing nothing and the case of having the control mechanisms without issuing the control messages is evidence that the overhead introduced by the cmBind mechanism is small. Clearly, more work might expose the reasons why rate control failed for MPEG data, but the point of this

FIGURE 9. 320x240 MJPEG Rate Control Results

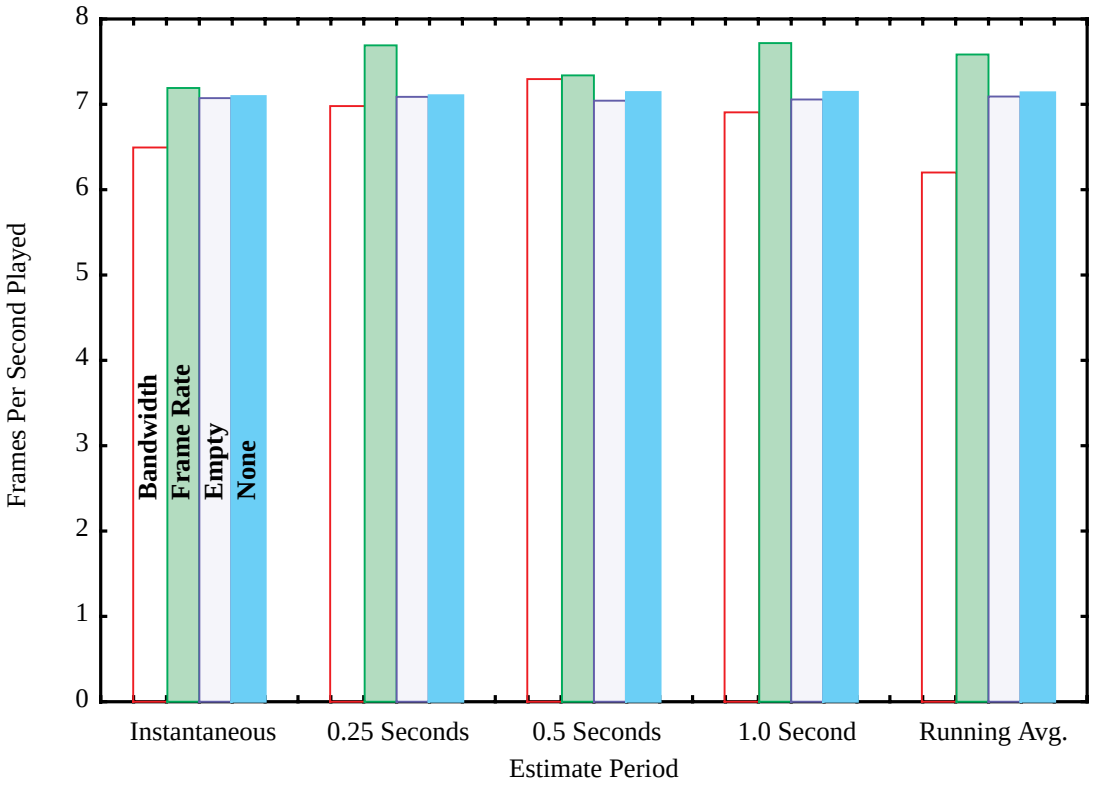
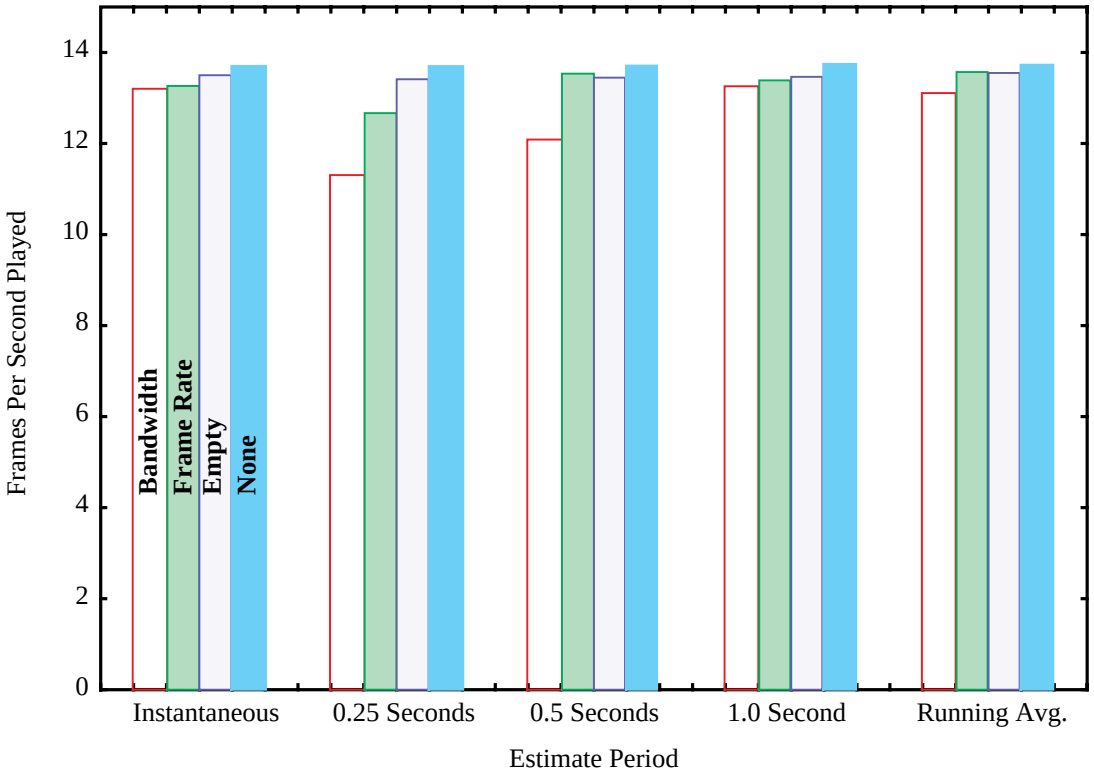


FIGURE 10. 320x240 MPEG Rate Control Results



exercise was to show that CMT can be used to prototype and run CM experiments rapidly. This experiment required only 350 lines of new Tcl code, and it was conducted in 2 days.

5.0 DISCUSSION

The CMT system has evolved into its current state over the course of 4 years. In 1992, the first network playback application was developed[19]. A library of media specific functions was used to facilitate building other continuous media applications. This library constitutes the earliest version of CMT. Version 2 of the system was written in 1993 to support a desktop video conferencing application[18]. The experience gained from building these applications highlighted the need for a general purpose toolkit that could be used to support any application model and simplify the implementation of new compression algorithms and communication protocols. The current version of CMT (version 3.1) was designed to meet these needs. Current development projects include: 1) high level API's for format and location independent video and audio widgets[9], 2) objects for MPEG audio and system layer streams, and 3) RTP and multicast support. The most current version of CMT can be found at http://www.bmrc.berkeley.edu/cmt.

6.0 ACKNOWLEDGMENTS

I would like to acknowledge Brian Smith and Soam Acharya of Cornell University, Andrew Swan and David Simpson of the University of California, and Gordon Chaffee for their contributions to the development of CMT. I also acknowledge the numerous researchers around the world that have used and tested CMT.

7.0 REFERENCES

1. Apple Corp., "Apple Quicktime Conferencing," `http://qtc.quicktime.apple.com/qtc/qtc.faq.tech.html`.
2. Banks, D., "Analysis Tools for MPEG-I Video Streams," Master's Report Plan II, University of California, Berkeley. Dept. of Electrical Engineering and Computer Science, 1995.
3. de Mey, V. and S. Gibbs, "A Multimedia Component Kit," *ACM Multimedia 93 Proceedings*, June 1993, Anaheim, CA, p. 291-300.
4. Herlocker, J.L. and J.A. Konstan, "Tcl Commands as Media in a Distributed Multimedia Toolkit," *Proceedings of the Tcl/Tk Workshop, 6-8 July 1995*, Toronto, Ont., Canada, p. 205-12.
5. InSoft Corp., "OpenDVE Home Page," `http://www.insoft.com/ProductOverview/OpenDVE`.
6. International Standards Organization, "Coded Representation of Picture, Audio, and Multimedia/Hypermedia Information," *Committee Draft of Standard ISO/IEC 11172*, 6 December 1991.
7. International Standards Organization, "Digital Compression and Coding of Continuous Tone Still Images," *JTC1 Committee Draft of Standard ISO/IEC 10918*, February 1991.
8. International Telecommunication Union, "Video codec for audiovisual services at p*64kb/s," *ITU-T Recommendation H.261*, 1993.
9. Jackson, M., "An Application Programming Interface to the Berkeley Continuous Media Toolkit," Proposed Master's Report Plan II, University of California, Berkeley. Dept. of Electrical Engineering and Computer Science, 1996. `http://www.bmrc.berkeley.edu/~trey/masters.ps`
10. Long Jr., A.C., "Full-motion Video for Portable Multimedia Terminals," Proposed Master's Report Plan II, University of California, Berkeley. Dept. of Engineering and Computer Science, 1996.
11. Malpani, R., "Floor Control for Large-Scale MBone Seminars," Class Project for CS294-3, Fall 1995, University of California, Berkeley, `http://www-plateau/people/radhika/cs294-3/project/proj.html.`

12. McCanne, S. and V. Jacobson, “vic: A Flexible Framework for Packet Video,” *ACM Multimedia 95 Proceedings*, November 1995, San Francisco, CA.

13. Microsoft Corp., “ActiveMovie Streaming Format - Product Information,” `http://www.microsoft.com/advtech/activemovie/productinfo.htm`.

14. Microsoft Corp., “Interactive Media Technologies: Direct3D,” `http://www.microsoft.com/imedia/direct3d/direct3d.htm`.

15. Mines, R. and J. Friesen, C. Yang, “DAVE: A Plug and Play Model for Distributed Multimedia Application Development,” *ACM Multimedia 94 Proceedings*, October 1994, San Francisco, CA, p. 59-66.

16. Ousterhout, J., “Tcl an embedded command language,” *Proc. 1990 Winter Usenix Conference*, 1990.

17. Ousterhout, J., “An X11 toolkit based on the tcl language,” *Proc. 1991 Winter Usenix Conference*, 1991.

18. Rowe, L.A., “Continuous Media Applications,” Multipoint Workshop held in conjunction with ACM Multimedia 1994, San Francisco, CA, November, 1994.

19. Rowe, L.A. and B. Smith, “A Continuous Media Player,” *Proceedings of the 3rd International Workshop on Network and OS Support for Digital Audio and Video*, San Diego, CA, November 1992.

20. Schulzrinne, H. and S. Casner, R. Frederick, V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications,” *RFC 1889*, January 1996.

21. Simpson, D. and R. Fromm, “A Packet Digital Video Production Switcher,” Class Project for CS294-3, Fall 1995, University of California, Berkeley, `http://http.cs.berkeley.edu/~rfromm/Courses/FA95/cs294-3/Project/Index.html`.

22. Smith, B. and L.A. Rowe, S. Yen, “Tcl Distributed Programming,” *Proceedings of the 1993 Tcl/Tk Workshop*, Berkeley, CA, June 1993.

23. Staunton, P., “Tenet Suite 1 and the CM Toolkit,” Master’s Report Plan II, University of California, Berkeley. Dept. of Electrical Engineering and Computer Science, 1995.

24. Tan, W.-T. and R. Tang, “Scalable Video: A CMT Implementation,” Class Project for CS294-3, Fall 1995, University of California, Berkeley, `http://www-video.eecs.berkeley.edu/~dtan/proj.html`.

25. Tennenhouse, D. et al., “The ViewStation: a software-intensive approach to media processing and distribution,” *Multimedia Systems*, vol. 3, 1995, p. 104-15.

26. Zhang, L. et al., “RSVP: A New Resource ReSerVation Protocol,” *IEEE Network Magazine*, September 1993.