Editing Extensions to the Berkeley Continuous Media Toolkit

J. Eric Baldeschwieler and Lawrence A. Rowe
Computer Science Division - EECS
University of Berkeley
Berkeley, CA 94720-1776

ABSTRACT

Extensions to the Berkeley Continuous Media Toolkit (CMT) are described that support video editing. These extensions were used to develop a Continuous Media Editor (CMEdit). The editor operates on and produces CMMovies that organize digital video and audio material into linear presentations. This source material can be located on remote servers or on a local machine. CMMovies can be played directly using CMT applications such as the CMPlayer or CMEdit. Alternatively, they can be used to assemble new source material. This approach supports the construction of highly-interactive media editors that take full advantage of the Internet.

1 Introduction

Tools have been developed recently that capture, transport and display video and audio material on the Internet. We refer to such time-based digital media as continuous media (CM). We have also seen the development of non-linear video editing systems such as AVID's Media Suite[Avid] and Adobe Premier[Adobe], which bring full-featured CM editing to general computer environments. However, these systems assume that all needed CM material is present in file systems mounted on the user's machine. They do not support the use of source material located on remote Internet sites. We have developed abstractions and an Application Programming Interface (API) that supports the development of CM editing applications that will allow direct, efficient, use of the large quantity of CM data available on the Internet, avoiding the down-load time and storage costs imposed by current editors.

The Berkeley Continuous Media Toolkit (CMT) was originally developed to support the transport and display of digital video and audio material across computer networks [Rowe94]. It has also been used to implement a video conferencing system. This paper describes additions to CMT to support interactive editing. These changes include: 1) The definition and implementation of a source file location scheme based on WWW URL's, 2) a textual representation for movies or presentations, called CM Movies, and 3) a data structure that supports efficient manipulation of the CMT objects needed to display presentations. An implementation of this API has been used to build a new version of the CMPlayer and a simple CM editor.

The remainder of this paper contains the following sections. Section 2 describes CMT and other foundations on which the API is built. Section 3 describes the CMT extensions that support editing. Section 4 discusses the CMPlayer, a video playback application using the API. Section 5 discusses CMEdit, a simple editor. Section 6 concludes with current status and future research directions.

2 Foundations

CMT is built on a number of existing tools, including Tcl/Tk [Ousterhout] and Tcl-DP [Smith93]. Tcl is a simple scripting language designed to easily incorporate new command extensions. Tk is an extension to Tcl that supports the development of Graphical User Interfaces (GUI). Tcl-DP supports the development of distributed programs. It includes a name server[Liu95], remote procedure calls (RPC), and unicast and multicast interprocess communications.

CMT is a Tcl extension that supports the development of distributed CM applications. It includes device abstractions (e.g., audio and video capture and playback devices), storage representations, file input/output, network transport, and time synchronization abstractions. CMT applications are typically written as Tcl/Tk scripts which are executed by a Tcl/Tk interpreter that includes the Tcl-DP and CMT libraries. The CMT abstractions are implemented as objects which a CMT application instantiates and interconnects to display or capture CM data. We refer to the collection of objects used to display a CM file as a *play chain*. A play chain that displays a single local file is shown in Figure 1. It is composed of two objects: a segment object and a play object. The segment object reads data from a file and passes it to other objects for processing. This object must be configured with the name of a file to read, the section of that file that should be passed, and the logical time at which to send it. A play object can use either hardware or software to decode the data stream and display it to the user. CMT provides play and segment object classes for many data formats (e.g. Motion JPEG, MPEG, H.261, etc.). In the example below, an MPEG Segment object is used to read the MPEG file "talk.mpeg" from disk and an MPEG play object displays this data in a window.

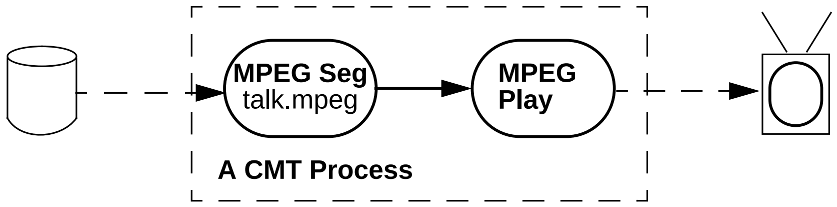


Figure 1: Simple CMT Play Chain

For a play chain to function, it must have an associated *logical time system* object (LTS). An LTS is the clock that synchronizes a CMT presentation. All objects in a play chain produce and/or receive data which is time-stamped with a desired display time, expressed in LTS time. The beginning of a presentation is LTS time zero. When a play chain is displaying data normally, the LTS time advances one second of logical time per second of real time. The display of a play chain can be controlled by setting the time and play rate of the LTS. Changes to the LTS state can stop or start a presentation, control the display rate and direction (e.g. forwards or backwards) and also jump to a random position in the presentation. Figure 1 shows a play chain with an associated LTS object.

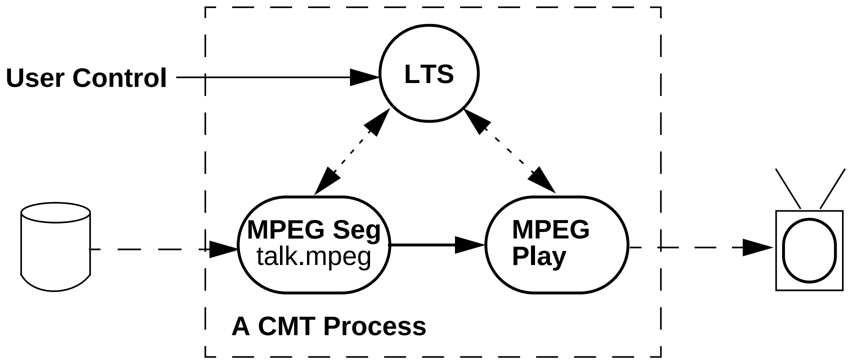


Figure 2: Play Chain with LTS

A CM source file may be located on a different machine from the one running the CMT application. In this case the application must transport CM data from the source machine to the user's local machine. This application requires the existence of a CMT server process on the remote machine that allows client applications to remotely create CMT objects. Using CM Servers, play chains can span networks. Play chains that cross machine boundaries use CMT network transport objects to transport the data between machines. The standard objects used to transport data over the internet are named *Packet Source* and *Packet Destination*. The packet source object transforms the data streams exchanged between objects within a CMT process into a stream of UDP packets. They are sent to a specified packet destination object in another CMT process. The packet destination object receives a data stream of UDP packets, translates these back into CMT internal buffers, and passes them on to another object for further processing. Figure 3 shows a play chain that will display the file 'talk.mpeg" located on a remote machine running a CMT server. A packet source object takes data from the MPEG segment object and sends it to a packet destination object on a different machine which passes it to the MPEG play object.

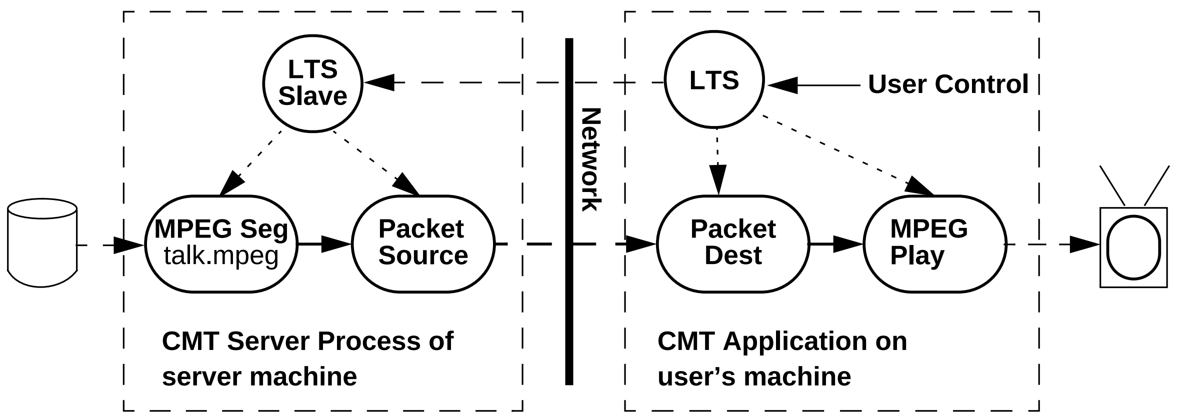


Figure 3: CMT play chain spanning two machines.

As shown above, an LTS can also span networks through the use of slave LTS objects. Slave LTS objects maintain the same time system as a master LTS on a remote machine. When the state of the master LTS object is changed, the slaves are automatically updated. Any difference in time between the system clocks of the master and slave machines must be known before the slave LTS can be synchronized. This clock skew is determined by requesting the clock of the target machine via RPC and compensating for network delay [Cristian 89].

All CM presentations are composed of groups of play chains. The CMT applications described in this paper display CMMovies. A CMMovie typically consists of two synchronized *streams* of CM data: one audio and one video. Each stream is composed of a collection of file segments, also called clips, presented in sequence. To play a CMMovie, an application must build a play chain for each segment, slave all of the play chain objects to a single LTS, and then instruct the LTS to "play" the movie. Appendix C shows a textual, graphical and play chain version of a CMMovie. The syntax of a CMMovie text file is described in Section 3.2. The graphical display shown in Appendix C is used by CMEdit and is described in Section 5.2.

3 Details of the Editor API

This section describes three object abstractions added to the CMT API to support editing: 1) URL, 2) PlayList, and 3) Movie.

3.1 The cmtp URL

The first step of any media editing process is to locate source material. In a traditional non-networked editing environment, source material is referenced by file name. A user manages his source material in his local file system. CMT applications may use source material located on any Internet machine running a CM server. This complicates locating and naming of source material. Locating source material is addressed by a video browser [Berger95]. Once a CM file is identified, a stable and portable name is required. A Universal Resource Locator (URL) is an Internet file naming syntax [Connolly] used by WWW browsers and other networked applications. A URL includes a file name, the Internet address of a server that can provide the file and the protocol used by the server. We have introduced a CMT URL protocol (CMTP) to support the transport and interrogation of CM files across the Internet. A CMTP URL contains the names of a CMServer and a file. The following are typical CMTP URLs:

```
cmtp://videoServer.edu:34/talk11.mpeg
cmtp://videoServer.edu/talk11.mpeg
```

The first URL indicates that the file "talk11.mpeg" can be requested from the CMServer found on Internet host "videoServer.edu" at port 34. The second URL is similar, except that the default CMTP port number is used, currently 5432.

Explicitly naming the target machine, port, and file name can be restrictive. A machine independent resource naming convention is needed. The eventual definition of a standard Universal Resource Name (URN) [Connolly] standard may solve this problem. In the interim we have partially addressed this problem by introducing a layer of indirection into the CMTP protocol. When using a URL, the first stage of the CMTP protocol is URL translation. URL translation allows the server named in a URL to be either a simple CM-Server or a translation server that maps the original URL to a new URL. This indirection provides flexibility to site administrators. Files can be moved or renamed without invalidating existing CMMovies if the URL's in these movies reference a directory server. More elaborate services such as load balancing and video on demand (VODS) can also be provided. We have implemented two translation servers, the URL directory server and the VODS server [Rowe95].

The directory server supports stable URL names and manages CM Servers on unstable development machines. Our development environment includes several video servers, and CM files may be moved from one host to another. Since a URL explicitly names the server providing the file, moving a file from one machine to another invalidates all URLs directly referencing that file. For example: a URL for the file "speech.au" located on the server "bugs-bunny.cs.berkeley.edu" would be "cmtp://bugs-bunny.cs.berkeley.edu/speech.au". If the file was moved to the machine "linus.cs.berekeley.edu", the file would be referenced by the new URL "cmtp://linus.cs.berkeley.edu/speech.au" and any use of the old URL would produce an error. Using the directory server, the file speech.au can be registered as "cmpt://plateau.cs.berkeley.edu/speech.au". This URL might originally be mapped to the bugs-bunny address, but after the file is moved the directory server can be updated to reflect this change, preserving the validity of any scripts referencing "speech.au", despite its change of location.

The second function of the directory server is to manage CM servers. A URL references a specific port number on a specific machine. If a server process is not running at that address, the URL is not currently valid. Many of our video servers are experimental or development machines which frequently crash and do not provide reliable service. The Name Server process manages other processes in such environments [Liu95]. It can restart failed servers and provide load balancing between lists of available servers. The directory server improves the reliability of URL's by contacting a name server during URL translation. Although conceptually separate servers, our current implementation of the directory server has been directly incorporated into the name server.

The Berkeley Distributed VODS system manages a large tape archive of CM material [Rowe95]. At any given time, a small fraction of this material is cached on hard disk, available for playback. CMMovies may reference material in this archive by using a URL that references the VODS server, which also functions as a CMTP translation server. When the CMPlayer attempts to play a CMMovie with a VODS URL, a translation request is sent to the VODS server for that URL. If the material is currently available on a high speed cache disk, the VODS server behaves in the same manner as the directory server, returning the address of a CMT server that can provide the CM material. If the requested file is not cached, it is not available without a delay and an error message is generated. The user can then use the VODS browser to request that the material be prepared for playback.

URL translation is a simple iterative process. The application contacts the server specified in the URL and issues a translate RPC, giving the server the full text of the URL. The contacted server can respond in one of two ways. It can return a tuple containing a filename, host, and port, or it can return a new URL. If a tuple is returned, translation is complete. The tuple is the address of a CMServer, which can handle CMT requests to build a play chain for the requested file. If a URL is returned, the application repeats the translation process by contacting the server named in the new URL and again requesting translation. In the simplest case, the initial URL directly names a CMServer. This server will return the same host, port, and file name found in the URL in tuple format. In more complicated cases, the initially contacted server is a translation server. Such a server will return a new URL, the result of translation the original URL. And the application requesting translation will begin a new translation request cycle

The most common URL translation case at our site, is that a URL names our directory server, not a specific CMServer. The name server is a translation server, not a full CMServer, and can not create play chains. It responds to translate requests by returning a new URL. This URL names a CMServer which supports the play chain objects needed to present CM data. The sequence of events in such a translation is shown in Figure 4. First, the application contacts the name server. The name server responds with a URL for a CM-Server. The application then contacts this server, which responds to translation requests by simply echoing the URL host, port, and filename in tuple format. The use of tuple format signals the completion of translation.

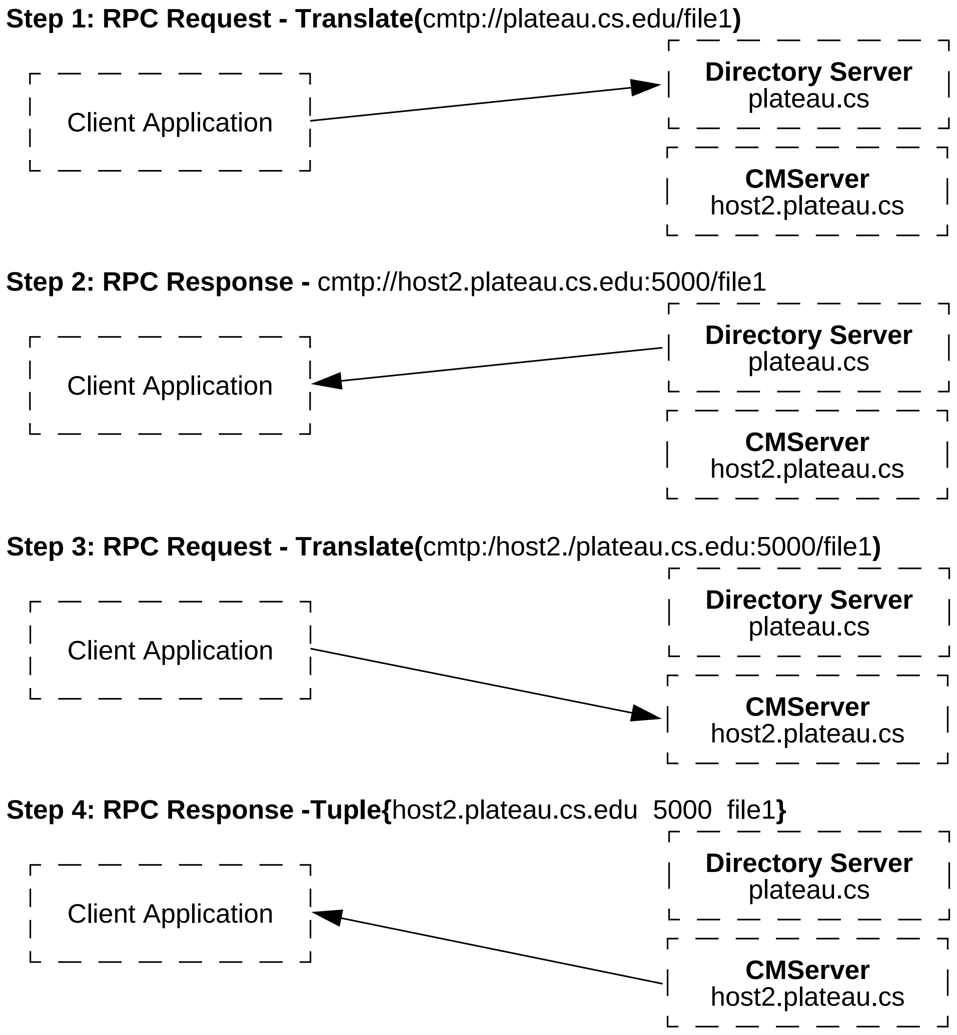

Step 1: RPC Request - Translate(cmtp://plateau.cs.edu/file1)
Client Application
Directory Server plateau.cs
CMServer host2.plateau.cs
Step 2: RPC Response - cmtp://host2.plateau.cs.edu:5000/file1
Client Application
Directory Server plateau.cs
CMServer host2.plateau.cs
Step 3: RPC Request - Translate(cmtp:/host2./plateau.cs.edu:5000/file1)
Client Application
Directory Server plateau.cs
CMServer host2.plateau.cs
Step 4: RPC Response -Tuple{host2.plateau.cs.edu 5000 file1}
Client Application
Directory Server plateau.cs
CMServer host2.plateau.cs

Figure 4: URL Translation with a Directory Server

The CMT application library function `CMApp_TranslateURL` performs the entire translation process described above. URL translation is potential very inefficient. Our experience indicates that the cost of the RPCs for translation is insignificant compared to the overhead of play chain creation. However, the cost of the TCP connection setup and tear-down required to make these RPC calls can be significant. It is very inefficient to establish a TCP connection to another process simply to issue a single translation request. A typical application translates many URLs. Further, after translating a URL the application immediately reconnects to the last host contacted in each translation exchange in order to create play chain objects. The CMT application library exploits this regularity by caching connections made to server processes during URL translation. Translation connections are silently kept open so that any following translation or object creation commands issued to the same process execute without the expense of connection setup. With this caching scheme in place, translation has not proven to be a performance bottleneck.

3.2 CMMovies

A CMMovie is an abstract object that can be stored in a file, a Tcl variable (as a *PlayList*), or a more complicated *Movie* data structure. Playback applications only deal with the first two representations (i.e., the script file and the *PlayList*) because they do not edit the CMMovie. The *Movie* data structure, designed to support interactive editing, is discussed in the next section.

CMMovies, sometimes called *scripts*, are composed of a collection of typed streams of CM data. A typical video script contains one video stream and one audio stream, but may have any number of streams, each of any supported type. Currently, audio and video streams are supported. An animation stream type is under development [Konstan]. Each stream consists of a sequence of clips. A clip is a tuple specifying the following: 1) a CM file URL, 2) logical start and end times, and 3) source start and end times. Logical times specify when the clip appears in a presentation, while source times specify the sub-range of the referenced CM file that will be played during that presentation period. For example, a clip with a logical start time of 0 and a source start time of 5 will be seen at presentation time zero, which is the beginning of the movie. Because of the source start time, material from the fifth second of the file onward will be presented. Notice that including the logical and source start and end times of a clip allows us to specify not only when a clip is played, but also the direction and speed at which it is played. An example of such a clip is shown below.

We make a distinction between stream type and file format. The type of a stream specifies the destination device, a screen or display window in the case of a video stream. Format encompasses type but also includes encoding details. Format designators use MIME-like syntax and semantics. For example, the format "video/mjpeg" specifies a motion JPEG file usable in a video stream. CMT allows a single stream to contain clips of many formats, but all clips must share the same type as the stream. In other words, a video stream can contain mjpeg and mpeg clips.

The language for CMMovie scripts was designed to be easy to parse both by machines and humans. It is composed of comments and command lines. White space and empty lines are ignored. Comment lines begin with the '#' character. The first line of a script is a `movie` command that specifies the number and type of streams in the movie. Other lines specify a clip in a stream. Clip lines must include the stream number of the clip and a CM file URL. Following these fields are optional fields that specify the logical and source times for the clip. Reasonable default values are supplied for these times if they are not present.

```
# CMT-Movie-1.0

movie video audio
clip 0 cmtp://host.edu/talk1.mpeg
clip 0 cmtp://host.edu/talk2.mjpeg
clip 1 cmtp://host.edu/dance.au
```

Figure 5: A simple CMMovie.

Figure 5 specifies a movie with a video stream (numbered 0) and an audio stream (numbered 1). Stream numbers are assigned based simply on the order of types in the `movie` command. The video stream contains two clips "talk1.mpeg" and "talk2.mjpeg" which will be played in sequence. The audio stream contains one clip stored in the file "dance.au". Notice that the two video files have different formats, one is MPEG and one is motion JPEG. No start times are specified, so the presentation will default to beginning at logical time zero. Because no source times are specified, each source file will be played in its entirety. Figure 6 defines the various optional arguments to the clip command that specify clip times.

```
-ls    - logical start time
-le    - logical end time
-ss    - source start time
-se    - source end time
```

Figure 6: Definitions of the `clip` command switches.

Figure 7 contains a CMMovie script with fully specified times. Notice that the time difference between the logical start and end is different from the source time difference. This movie will play the first 10 seconds of the file "lightShow.mpeg" backwards at double speed (in 5 seconds).

```
#
# CMT-Movie-1.0

movie video
clip 0 cmtp://host.edu/lightShow.mpeg -ls 0 -le 5 -ss 10 -se 0
```

Figure 7: A CMMovie with a clip played at double speed

A PlayList is a run-time representation of a CMMovie script. The CMT application library contains a function, named `CMApp_ReadURL`, that takes a CMMovie file name or URL and returns a PlayList. `CMApp_WriteMovie` performs the inverse operation. A PlayList data structure contains the same information as the script file, but during the construction of the PlayList, CMT interrogates the clip files to determine the format and length of each clip. Default times and formats can be found for clips that are not fully specified in the script. Given a PlayList, it is easy to construct the play chains for the movie. The PlayList data structure is a Tcl/Tk list that can be manipulated and stored using built in Tcl mechanisms. Figure 8 shows the PlayLists returned by `CMApp_ReadURL` for the CMMovies shown in Figure 5.

```
{video
  {cmtp://host.edu/talk1.mpeg 0 10 0 10 video/mpeg}
  {cmtp://host.edu/talk2.jpeg 10 20 0 10 video/mjpeg}}
{audio
  {cmtp://host.edu/dance.au 0 20 0 20 audio/au}}


{video
  {cmtp://host.edu/lightShow.mpeg 0 5 10 5 video/mpeg}}
```

Figure 8: PlayLists for CMMovies in Figure 5 and Figure 7.

3.3 Movie Abstraction

The PlayList abstraction described above provides all information needed to build play chains for a playback application such as the CMPlayer. However interactive editing requires the ability to make changes to CMMovies and quickly update the play chains to reflect these changes. In other words, play chains must be manipulated and managed at run-time. While implementing CMEdit, we introduced the Movie abstraction. The Movie function takes a PlayList and returns an "object" that encodes the information in a PlayList. This object also has methods for manipulating the stored PlayList and associated play chains.

Movie objects do not support file I/O commands. However, commands exist to convert between Playlists and Movies and vice versa. PlayList file I/O is supported by the CMT application library. All modifications of movies can be performed using a few simple operations, such as `delete`, `shift`, and `addClip.`Other edit commands exist to perform such operations as copying an entire movie or merging two movies together, but these commands are implemented using the methods listed above.

Figure 9 shows the visual representation of a CMMovie used by the CMEditor. This visual representation will be used to demonstrate the effects of movie object editing methods.

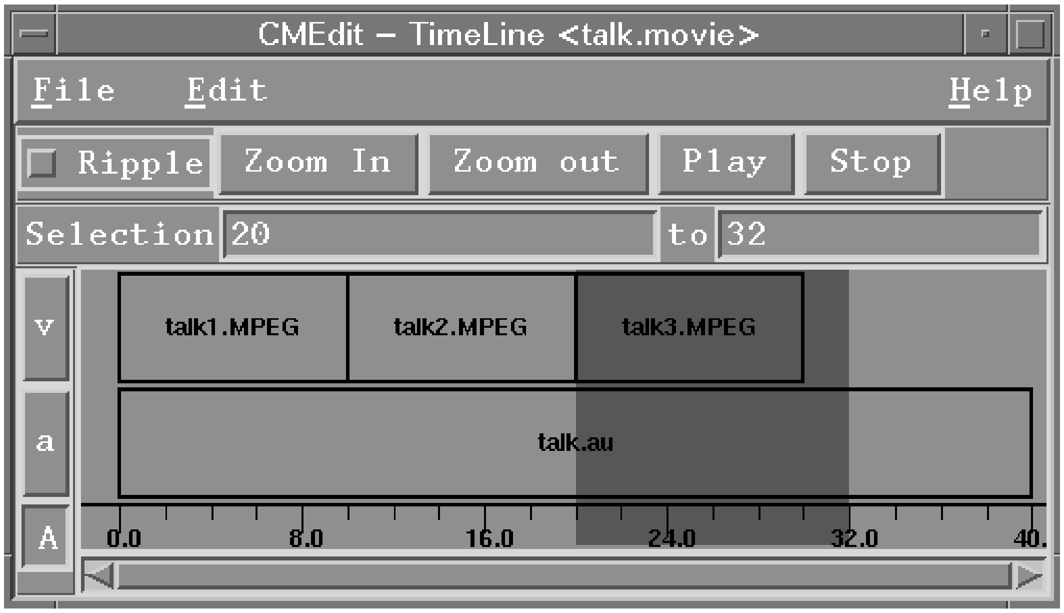


Figure 9: A Sample movie before editing.

The delete method removes all clips from a specified time-range in a movie. It leaves empty space. Figure 10 shows the effect of the following edit command on the CMMovie shown in Figure 9.

```
    movieObject delete 10 20
```

Note that if a clip starts before the deleted range and ends after it, the delete method will split it, creating new clips. The "talk.au" clip is split in this manor in the example below.

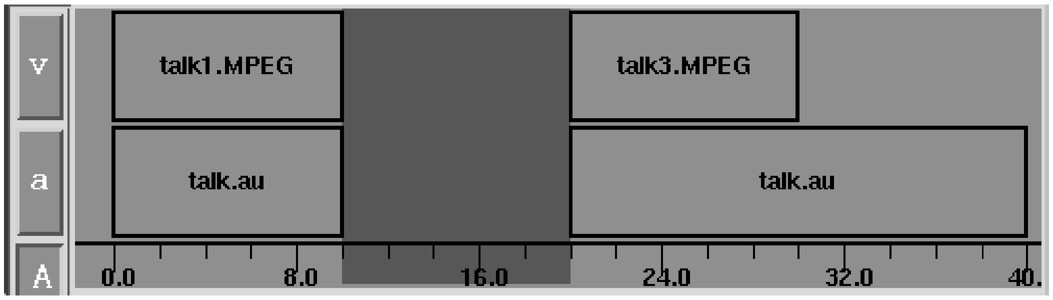


Figure 10: The effect of the command "`delete 10 20`".

The shift method adds or subtracts a constant from the logical start and end times of all clips in a movie after a given time. This operation shifts one section of the movie in relationship to another, by either inserting or deleting space.

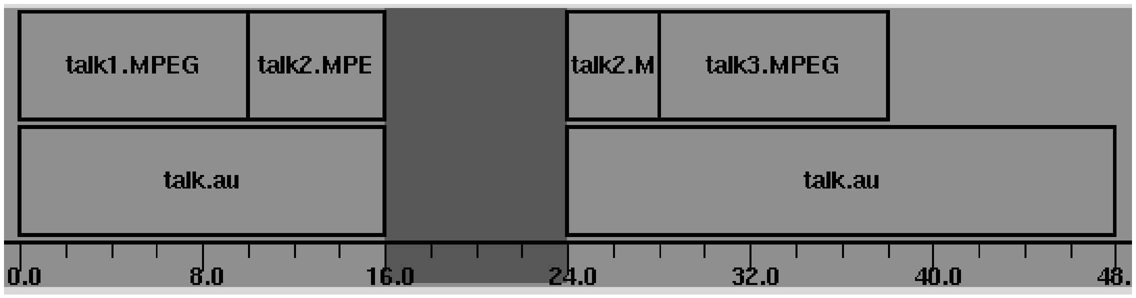


Figure 11: The effect of the command "`shift 16 8`".

The addClip command inserts a clip into a given track at a given time. It takes the logical and source times needed to place the clip, and the URL of the source file. AddClip takes a target stream number. The other operations can take a stream number or operate on all streams.

Two details of the CMT implementation encouraged the addition of the Movie abstraction. First, some CMT objects are heavy weight, requiring significant time to create. This creation time is small compared to the time needed to start a player application and therefore is not an issue in most CMT applications. However, interactive editing requires the ability to replay the CMMovie immediately after changing it. A several second delay after each edit is not acceptable. Consequently, existing play chains must be modified, instead of building a new set of play chains after each edit. The second factor encouraging the introduction of the movie abstraction was that play chain objects require absolute (logical) start and end times. Any modification of a movie using the `shift` method changes the logical start and end times of all clips following the shift start time. This design implies that any sequence of edits that changes the length or position of any clip may require the reconfiguration of many play chain objects.

For example, consider the video stream of the CMMovie shown in Figure 9 and the corresponding PlayList and play chains shown in Appendix C. If the duration of the second clip, "talk2.mpeg", was changed from ten to eight seconds, the logical times of the third clip also must be changed to avoid creating a two second gap in the movie. Changing the logical times of the clips within the Movie is inexpensive, but even simple edits can have large effects, requiring the modification of many clip records, display images, and play chain objects. The movie object was designed to provide simple mechanisms to manage this complexity.

3.4 Movie Object Facilities for Managing Play Chains

Movie objects include a property list for each clip and a change marking facility to address the efficiency and management problems discussed in the previous section. These property lists allow arbitrary key and data pairs to be stored with each clip in a movie. CMEdit uses the clip property lists of each movie object to store a "_SEGMENT" property for each clip. The values of these "_SEGMENT" properties store pointers to the CMT objects that have been created to display the clips. This data allows the application to locate the play chain objects that must be adjusted when a movie is edited.

The change marking scheme is implemented as a series of flag properties in each clip. Whenever a clip is modified or added to a movie it is flagged as changed. After performing one or more edits, the application can iterate over all flagged clips, executing application supplied code. This approach allows display and play chain updates to be delayed, much like screen updates are managed by windowing systems. A complication to this solution is that the two cases mentioned, display and play chain update, can occur at different frequencies. This fact requires separate change tracking for the two events. Which is solved by allowing an application to define up to 32 change flags per movie object. The movie object method `addFlag` associates a symbolic name with an unallocated flag. Later, the application may test and clear the named change flags of any clip without effecting other flags of that clip. CMEdit allocates two change flags, one for screen updates and one for play chain updates.

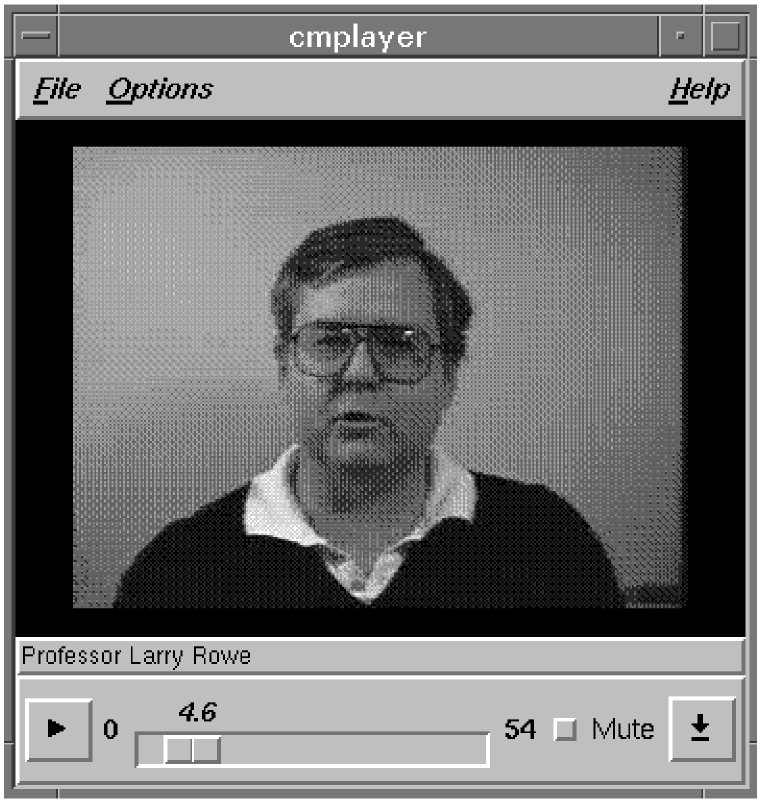


Figure 12: CMPlayer

4 The CMPlayer Application

The CMPlayer, shown in Figure 12, is a CMMovie playback application. This application allows a user to select a CMMovie file or CM source file and play it. The CMPlayer supports full-function VCR controls, including play, stop, fast forward, fast rewind and random positioning. The player demonstrates the flexibility of CMT. It allows a movie to be played at any speed and in either direction. Because the CMPlayer is a playback-only application, CMMovies are static. The user has the ability to choose and play a CMMovie, but he may not modify it. Consequently, the play chains used by the CMPlayer are also static. When the user selects a movie, the application builds a play chain for each stream in the movie. Because, VCR commands only effect the LTS of the play chains, not the structure of the play chains themselves, the CMPlayer does not need to modify them. For this reason, the CMPlayer does not use Movie objects. It simply generates the PlayList for the selected movie, using `CMApp_ReadURL`, and then traverses the resulting list, generating play chains.

The original CMPlayer [Rowe93] predated the editor extensions discussed in this paper. Previous versions supported a more primitive input script format, closer in structure to a PlayList then a CMMovie. The new CMMovie script format is human editable, which allows simple CMMovies to be edited using a text editor. The extensions also support playing remote scripts via URL's. `CMApp_ReadURL` takes a file name or the URL of a CMMovie and returns a PlayList. Previously, the movie script itself had to be located on the user's machine, but could reference remote CM source files. Lastly, the editing extension added URL indirection which allows the use of name and archive servers without specifically coding these features into the CMPlayer.

5 The CMEdit Application

CMEdit is a simple non-linear editing application. It was developed in parallel with the CMT editing abstractions which it utilizes. The application supports the interactive construction and editing of CMMovies via a graphical user interface. Users can make small changes to the movie and evaluate them by immediately viewing the result. This requirement encouraged the development of the movie object abstraction. CMEdit is a simple test bed for the CMT editing extensions and as such does not attempt to provide many of the features found in commercial non-linear editing packages. It has demonstrated the viability of a networked editing approach, allowing the construction of CMMovies that use material without requiring that it be located on the file system of a single site. The CMEdit user interface is built upon three distinct windows: Collections, TimeLines, and Players. Figure 13 shows a typical CMEdit session in which one of each of these window is in use.

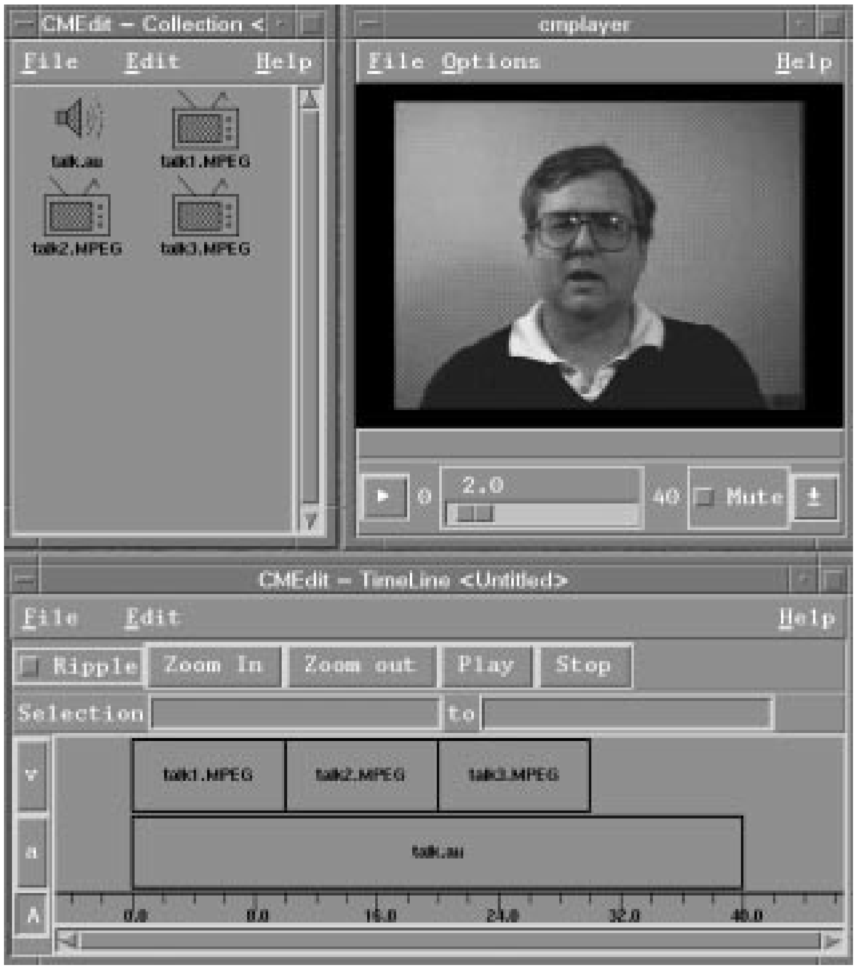


Figure 13: CMEdit Screen Shot.

5.1 Collection Window

Before a user can construct a CMMovie, he must have source material. Locating this material is a different problem than editing. It involves finding and organizing collections of video and audio source material. A collection window displays a list of clips with icons indicating their type. These clips can be viewed by doubling clicking on them, and they can be selected and pasted into other collections or timelines using the standard copy/paste paradigm. Collections may be saved to files for use in later editing sessions. The cur-

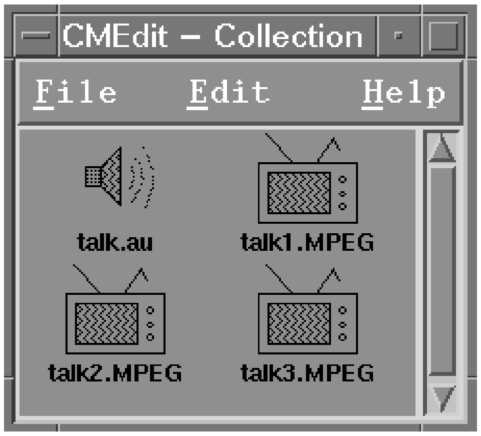


Figure 14: The Collection Window

rent implementation supports only a simple iconic view of the material. A collection is not a CMMovie, it is a list of references to source material that a CMEdit user may wish to use.

To add a clip to a collection, the user may either paste a selection from another window or enter a file name or URL via the "Add URL..." item in the **File** menu. If adding a clip via the selection mechanism, the user can either select a clip in another editor window or select a file name or URL in an editor or shell window. In either case, the URL is interrogated once it is added to the collection to determine the type and duration of the clip.

5.2 The Timeline Window

The timeline window, shown in Figure 15, is the main window of CMEdit. Each timeline window displays a graphical representation of a CMMovie. Normally, a CMEdit user will open only one timeline window, but CMEdit allows multiple timelines to be opened and supports copying material from one window to another using copy/paste mechanisms. Each clip in a movie is shown as a separate named rectangle. Each stream in the movie is displayed on a separate row. Time, in seconds, is shown on the horizontal axis. The window

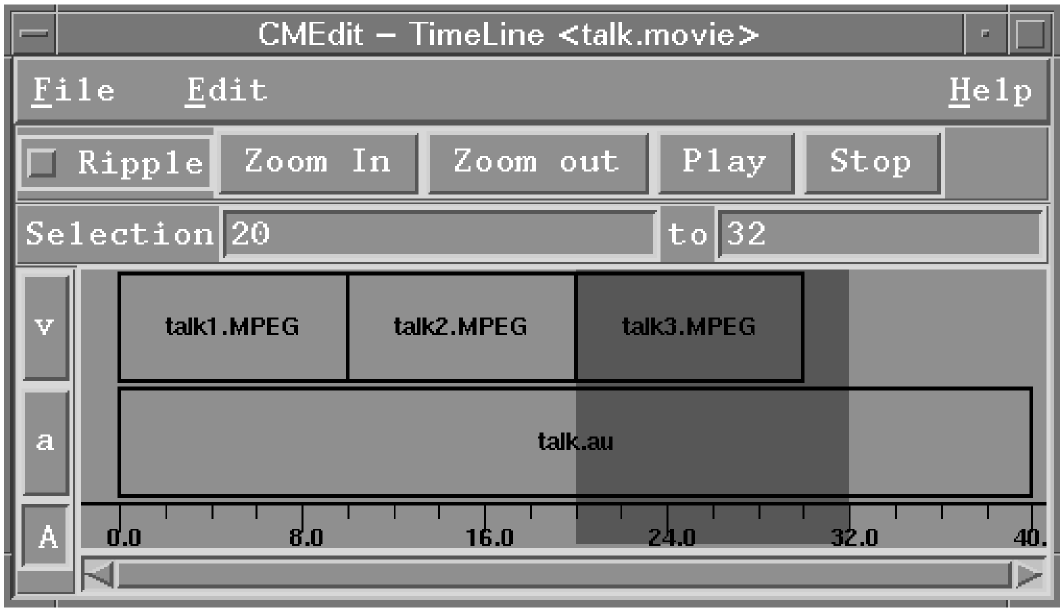


Figure 15: The TimeLine Window

supports zooming and time range selection. The play button causes the current contents of the TimeLine window to be presented in the play window.

Selection and operations in the window are done on time ranges. Double clicking within a clip selects the time range of that clip. Operations can be performed either on a single stream or all streams. Stream selection is controlled by the column of buttons on the left side of the window. These buttons are labelled with the type of the stream, "a" for audio and "v" for video. The button labelled "A" at the bottom causes operations to operate on all streams at once. To select a specific stream, the user clicks the button adjacent to that stream.

Operations are selected via menus or keyboard short-cuts. The most common operations are insert clip and delete. Insert adds a clip, filename or URL to the CMMovie at the time indicated by the selection bar in the window. The selection bar is shown as a white bar in Figure 15. Both the "Add URL..." menu item in the **File** menu and the paste mechanism use this command. The clip is added to the first stream which matches the type of the clip. Delete removes the selected time range from the CMMovie.

5.3 The Play Window

The play window duplicates the functionality of the CMPlayer. It allows the user to display the currently selected CMMovie. The current movie can be changed from the other CMEdit windows. Double clicking on a clip in the collection window causes it to appear in the play window. In addition, the timeline window has a play button that will play the current movie.

6 Conclusion

This work has suggested several further avenues of development. Many additional features can be added to CMEdit. One promising addition is to integrate the RIVL video processing language[Swartz95]. Currently, CMEdit only operates on a scripting level. It does not produce new CM source files, it only builds CMMovies that reference preexisting source files. We would like the ability to produce new material to support final mastering and special effects such as fades, wipes and mixes.

The addition of a RIVL back-end for mastering would be very simple. It would involve translating a CMMovie into a RIVL script and extending RIVL to handle networked fetches of input material. These extensions would allow the construction of a single new CM file, such as a multiplexed MPEG, from the CMMovie produced by CMEdit.

This project also suggests several additions to CMT. Play chain construction and management proved difficult using the tools provided by the CMT application library. Integrating the play chain management code implemented in CMPlayer and CMEdit back into the CMT application library will greatly simplify the task of CMT application development. Also a set of robust widgets that can display a PlayList stream of a given type, either audio or video, and support simple edit operations could be constructed from the CMEdit code.

Although hardly complete or professional in quality, the editing application resulting from this work demonstrates the viability and potential of a networked editor. A CMEdit user can work on a large editing project involving source material distributed throughout the Internet, without the cost of collecting the material on a local workstation. Once completed, the resulting CMMovie is a small text file that can be easily distributed through email or WWW applications and then immediately viewed on other sites, again without the cost of down-loading large source files. The entire CMEdit application, including the CMPlayer code and code to implement the movie object, consists of only 9000 lines of C and Tcl code. Given the complexity of the operations that it performs, this demonstrates the great power and flexibility of the CMT and the Tcl/Tk approach to developing continuous media applications.

APPENDIX A: References

[Adobe]
Adobe Premiere Web Site, http://www.adobe.com/Apps/Premiere.html

[Avid]
Avid Technology Inc. of Tewksbury, Mass., can be reached at (508) 640-6789 or (800) 949-2843; fax (508) 640-1366.

[Berger95]
D.A. Berger, Masters Degree Report, University of California at Berkeley, 1995

[Connolly]
Daniel W. Connolly and Tim Berners-Lee, "UR* and The Names and Addresses of WWW Objects", http://www.w3.org/pub/WWW/Addressing/.

[F. Cristian 89]
F. Cristian, "A Probabilistic Approach to Distributed Clock Synchronization," *Proceedings 9th International Conference on Distributed Computing Systems,* Newport Beach, CA, June 1989

[Konstan]
Joseph A. Konstan and J.Herlocker, "Tcl Commands as Media in a Distributed Multimedia Toolkit," Proceedings of the 1995 Tcl/Tk Workshop (Usenix Association).

[Liu95]
P. Liu, B.C. Smith, and L.A. Rowe, "Tcl-DP Name Server," *Procceedings Tcl/Tk Workshop 95*, Toronto, Canada, July 1995

[Ousterhout]
J.K. Ousterhout, "Tcl and the Tk Toolkit," Addison-Wesley Publishing Co., Reading, MA 1994.

[Rowe93]
B.C. Smith and L.A. Rowe, "A Continuous Media Player," in Network and Operating System Support for Digital Audio and Video, Springer-Verlag Lecture Notes in Computer Science #712, pp376-386, 1993

[Rowe94]
L.A. Rowe, " Continuous Media Applications," Presented at Multipoint Workshop held in conjunction with ACM Multimedia '94, San Francisco, CA, Nov. 1994.

[Rowe95]
L.A. Rowe, D.A. Berger, and J.E. Baldeschwieler, "The Berkeley Distributed Video-on Demand System," Multimedia Computing - Proceedings of the Sixth NEC Research Symposium, Ed. T. Ishiguro, SIAM, 1995.

[Smith93]
B.C. Smith, L.A. Rowe, and S. Yen, "Tcl Distributed Programming," Proceedings Tcl 1993 Workshop, Berkeley, CA 1993.

[Swartz95]
Jonathan Swartz and Brian C. Smith, "A Resolution Independent Video Language," Proc. of the Third ACM International Conference on Multimedia, San Francisco, CA, November 5-9, 1995.

Appendix B: Movie Command Man Page

Movie <PlayList>

Returns a new movie object constructed from the given playList. Note that an empty list is a valid play list, as is one containing only a list of stream types, IE "Movie {video audio}". The result of the Movie command is a movie object. The following switches may be used

Movie objects have the following methods:

<MovieObject> init ?playList?

Movie is init'ed to be equivalent to playList

<MovieObject> getnumstreams

Returns the number of streams in the movie.

<MovieObject> getstreamtype <idx>

Returns the type of the idx'th stream of the movie.

<MovieObject> getstream <idx>

Returns the stream object for the idx'th stream of the movie.

<MovieObject> getclipstart <trk> <idx>

Returns the logical start time of the idx'th clip in stream $trk.

<MovieObject> getclipend <trk> <idx>

Returns the logical end time of the idx'th clip in stream $trk.

<MovieObject> allocflag

Returns the flag number of an unused change flag. This flagNumber can then be used in calls to the clipflag method. These flags are automatically set whenever an editing method changes a clip.

<MovieObject> addstream <idx> <strmTxt>

Inserts a stream at position 'idx' in the movie stream list. $idx can equal "end" and will then cause an append. The stream text should follow the same format as a stream description in a playList. A minimum strmTxt would be just the stream type, i.e. video or audio.

<MovieObject> astext ?trks?

Returns a playList description of the movie. If a list of tracks is given, then only those tracks are included in the output.

<MovieObject> getclip <trk> <indx>

Returns a fully resolved clip description (a text list) for the indexed clip.

<MovieObject> find <trk> <time>

Returns the index of the clip that contains the given logical time in the given stream. If no clip contains the time, then the first following lcip is returned. The return is a tcl list of {clipIdx flag} where flag is true if the time falls within a clip.

<MovieObject> snap <trk> <time> <dist>

If there is a clip boundry within $dist of $time in stream number $stream, then the time of the closest clip boundry is returned. Otherwise $time is returned unchanged. This is usefull for implementing "snapping" functions in GUIs.

<MovieObject> getstart ?trks?

Returns the start time of the given track or the minimum starting time if multipule stream indexes are given ($trks can be a list of indexes or "all"). If no clips are defined in any of the given streams, then zero is returned.

<MovieObject> getend ?trks?

Returns the end time of the given track or the maximum starting time if multipule stream indexes are given ($trks can be a list of indexes or "all"). If no clips are defined in any of the given streams, then zero is returned.

<MovieObject> clear <trks> <start> <end>

Deletes all information in the given time range for the listed stream or streams ($trk can be a stream index, a list of indexes or "all"). If clips are partially in the time range, they are cropped.

<MovieObject> shift <trks> <start> <delta>

Moves shifts all clips on or after $start time in the given track(s) by $delta time units (e.g. adds delta to the logical start and end times). If delta is negative, clips in the region behind $start are deleted. Clips which span $start are split in two. $start may equal "start".

<MovieObject> foreach <trks> <start> <end> <streamVar> <clipVar> <script>

Executes $script for each clip in the given time range on the specified track(s). Within script $streamVar is set to the name of the stream object currently executing. $clipVar is set to the clip index of the current clip.

<MovieObject> addclip <trk> <clip> <args>

returns the {trackNum clipNum} of new clips, as a list.

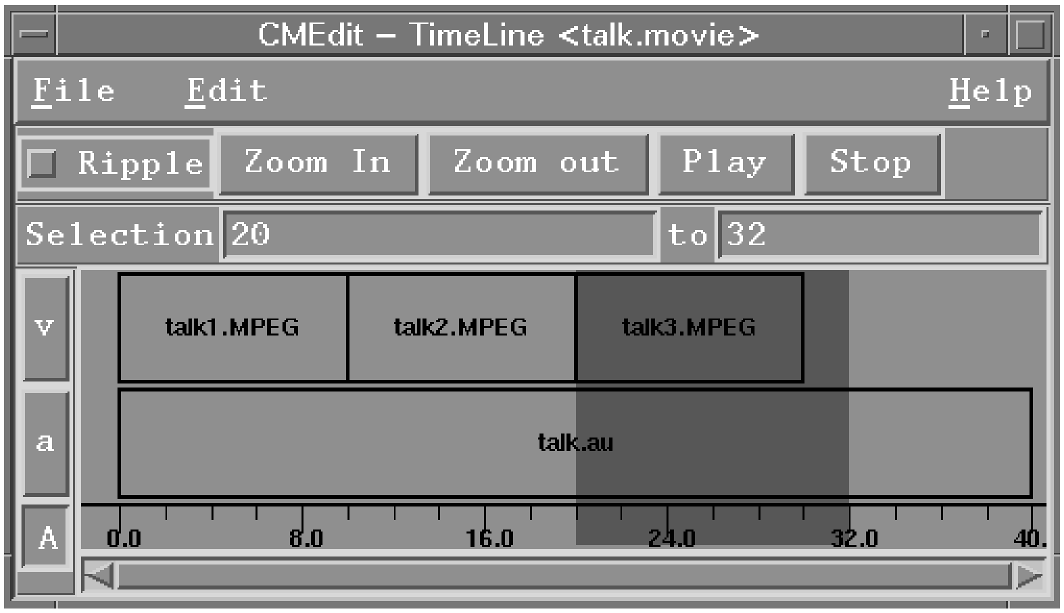
<MovieObject> mergestream <trks> <src>

Maps equivalent typed traclks for src to self add addclip's all the clips. Only $trks are effected. Unmatched tracks are discarded.

Appendix C: Several Representations of a CMMovie

Each of the following sub-sections contains an alternate view of the same CMMovie.

C.1 TimeLine View



C.2 CMMovie File

```
# CMT-Movie-1.0

movie video audio
clip 0 cmtp://host1.edu/talk1.mpeg
clip 0 cmtp://host1.edu/talk2.mpeg
clip 0 cmtp://host2.edu/talk3.mpeg -ss 10 -se 20
clip 1 cmtp://host2.edu/talk.au
```

C.3 PlayList

```
{video
  {cmtp://host1.edu/talk1.mpeg 0 10 0 10 video/mpeg}
  {cmtp://host1.edu/talk2.mpeg 10 20 0 10 video/mpeg}}
  {cmtp://host2.edu/talk3.mpeg 20 30 10 20 video/mpeg}}
{audio
  {cmtp://host2.edu/talk.au 0 40 0 40 audio/au}}
```

C.4 Play Chains

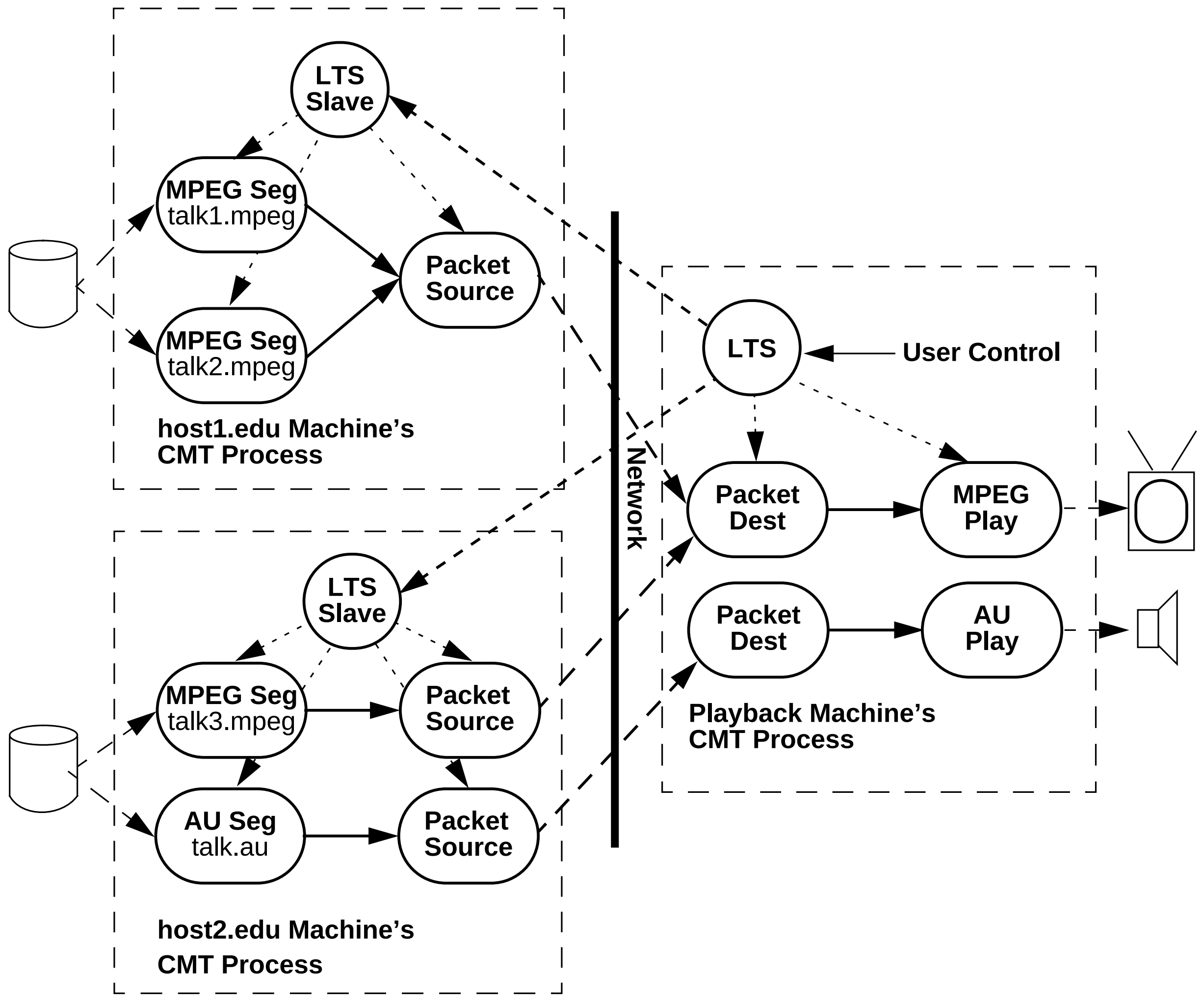


Figure 16: Play Chains for a more complex CMMovie.

Note that the play chains shown here have been folded into trees where they share common tail object types. This is an optimization used by all CMT applications. Each play tree could have been represented as a single chain of objects, the result would be an identical presentation.