Berkeley CMT Media Playback API

MacDonald Hall Jackson III
U.C. Berkeley

October 4, 1996

Abstract

The Berkeley Continuous Media Toolkit provides low-level, modular tools for developing distributed continuous media (CM) applications. The programming interface to the toolkit requires application developers to create and manage objects required to play back audio and video. These objects are distributed to different processes possibly on different hosts. This paper presents an application programming interface (API) that frees the application writer from the details of managing the underlying CM objects. It also provides an easily configurable framework in which CM object developers can place their objects. A simple video editing application is shown to demonstrate the benefits of using the API and framework.

1 Introduction

This paper describes the design and implementation of an application programming interface (API) for continuous media (CM) application developers. Applications written using the CMT Media Playback API are presented a simple Tcl/Tk interface that allows them to treat CM streams as a collection of media clips and timing instructions. All connection and resource management details are hidden from the application developer. The API is built on the Continuous Media Toolkit (CMT) which provides an extensible collection of polymorphic objects that can be linked together to play audio and video over arbitrary networks [MPR96]. Audio and video material can be captured live or played from storage. Developers who use the toolkit can modify the objects to meet their needs, and after integrating new objects into the framework, applications will seamlessly take advantage of new toolkit capabilities.

CMT is an open, portable toolkit that can be used for research experiments. CMT currently runs on PC's (Windows and UNIX) and UNIX workstations, and it will be ported to the Macintosh. Source code is available to encourage experimentation. Related work in multimedia toolkits include a variety of research systems and commercial products. The ViewStation project at MIT [Tea95] most closely matches the architecture of CMT. It was designed to operate on high-speed local area networks. The DAVE system developed at Sandia Laboratories [MFY94] provides a high-level abstraction to continuous media devices. However, assumes devices are sampled at regular intervals and does not provide as flexible a time model as CMT. The Multimedia Component Kit [dMG93] 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. Apple Quicktime Conferencing [App] (QTC) is a commercial system that provides an API to Quicktime codecs and network modules for the development of video conferencing applications. InSoft's OpenDVE [Ins] toolkit is another commercial development system providing an API for application programmers to create collaborative and desktop video conferencing applications. Lastly, the ActiveMovie [MSa] architecture

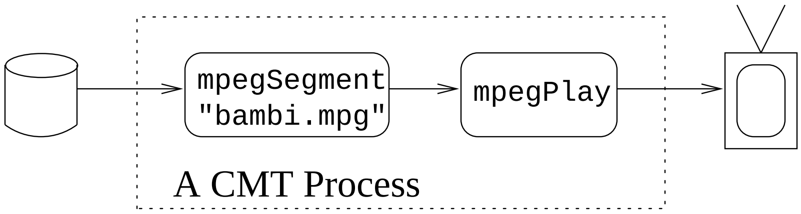


Figure 1: An idealized playchain.

from Microsoft is greater in scope than either Apple QTC or InSoft OpenDVE providing an API integrated with other development API's such as Direct3D [MSb] (a 3-D graphics API).

The remainder of this paper is organized as follows. Section 2 presents background on the issues encountered when developing CM applications and describes problems this API is designed to solve. Section 3 presents a high level introduction to the API software architecture. Section 4 presents a detailed explanation of the API. Two auxiliary functions and an example application are given in section 5 to demonstrate the power of the API. And section 6 discusses issues to be addressed and work to be done.

2 Background

CMT is built on several existing tools, including Tcl/Tk [Ous94] and Tcl-DP [SRY93]. Tcl is an interpreted scripting language that provides a rapid prototyping environment. Tk is a collection of Tcl commands that implement a graphical user interface development toolkit. Tcl-DP is a scripting language for distributed application programming. It extends Tcl with remote procedure calls, unicast and multicast communication support, and a nameserver [LSR95].

CMT supports development of continuous media applications. The system includes abstractions for various audio and video streams (e.g., μ-law audio, MPEG, and MJPEG), file I/O, a logical time system (LTS), and objects that send and receive data over a network. These objects are accessed using Tcl commands, thus CM applications are written as Tcl/Tk scripts. This section describes fundamental concepts of CMT. Section 2.1 explains how to use the abstractions to play CM. Section 2.2 discusses how to resolve universal resource locators (URL). And section 2.3 shows how to create CM objects on remote hosts.

2.1 CMT Introduction

Let us look at an idealized example of how to play a movie. A user who wants to play the movie `bambi.mpeg` must read the data from a storage device (e.g., disk), decode it, and display it on a monitor. CMT provides two objects that can be used together to implement this playback application. The `mpegSegment` object reads data from the disk and passes it to an `mpegPlay` object. The `mpegPlay` object decodes the data it receives and displays the reconstructed frames on the monitor. These objects are created in a CMT process which is a Tcl/Tk process with commands added to create and control CMT objects. Figure 1 shows a graphical representation of the objects in this example. The set of objects necessary to play a movie, or any type of media, are called a *playchain*.

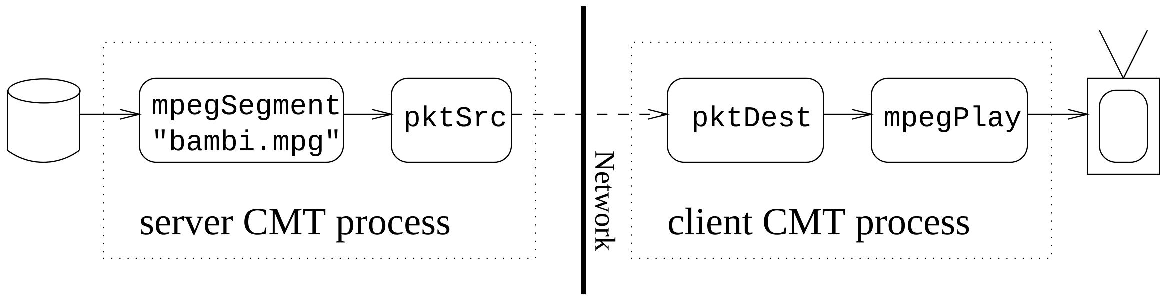


Figure 2: A playchain spanning a network.

A more complicated *playchain*, shown in Figure 2, illustrates how the addition of two more CMT objects, named `pktSrc` and `pktDest`, allows the movie to be played when the media file is stored on a remote server. The `pktSrc` and `pktDest` network transport objects fit transparently between the `mpegSegment` and `mpegPlay` objects. The `mpegSegment` is configured and behaves the same as before, except that it sends the data to the `pktSrc` instead of to the `mpegPlay` object. The `pktSrc` and `pktDest` objects work together to transfer data from one host to another over a network, independent of data format. On the client machine, the `pktDest` passes the data to the `mpegPlay` object, which proceeds as above.

CM data has a time characteristic not found in traditional media such as text and images. Every movie and sound file has an associated length in time. Every frame i in a movie, or audio block i in a sound file, should be played at a specified time t_i. CMT provides a time abstraction, called the *logical time system* (LTS), that is essentially a user configurable clock. This clock encapsulates a mapping from real time to playback time. The LTS abstraction has two important properties: *speed* and *value*. An LTS object behaves like a time line, the current time is *value*, which changes by *speed* every second of real time.

The properties of an LTS object can be changed by the configure command. For example, the following commands create an LTS and configure it:

```
set local_lts [lts ""]
$local_lts configure -speed 2 -value 0
```

The first line creates an LTS and stores its name in the variable `local_lts`. The second line configures the LTS to start at `value` 0 and increment it by 2 every second, essentially "fast forwarding" at twice real time.

Every movie or audio file (a.k.a. clip-file) has beginning and ending times. The beginning time is 0 and the ending time is n seconds later. CMT provides a way of specifying which portion of a clip-file to play back. Source start (`ss`) denotes how many seconds of the clip-file to skip before playing. Source end (`se`) specifies how far into the clip-file (from the beginning) to stop playing. For example, the expression

```
bambi.mpg -ss 10 -se 30
```

specifies a clip of the movie `bambi` that is 20 seconds long, beginning 10 seconds into the movie.

Most objects, such as `mpegPlay`, `pktDest`, etc., require an LTS object to operate. CMT objects use the LTS to schedule events like data decoding and packet transmission. Figure 3 shows how an LTS fits into the *playchain* in Figure 2. For example, if the LTS that controls the `mpegSegment` object in Figure 3 has speed zero, the `mpegSegment` will not read data from disk. `pktDest` and `pktSrc` also use the LTS to determine which packets of data to forward and which to drop.

There must be an LTS in every CMT process involved in the *playchain*. LTSs on different hosts can be "slaved" to one another to synchronize them. The user or application manipulates the LTS, and the rest of the objects in the *playchain* do their best to play the correct frame of video at the correct time. For example, if the user control in Figure 3 increases the value of the LTS by thirty, the LTS slave's value will also increase by thirty. Consequently, other CMT objects will change their behavior: 1) the `mpegSegment` object will begin sending data from the new time, 2) the `pktSrc` and `pktDest` objects will ignore all data they are supposed to have transmitted at the old time, and 3) the `mpegPlay` object will only play frames at this advanced time.

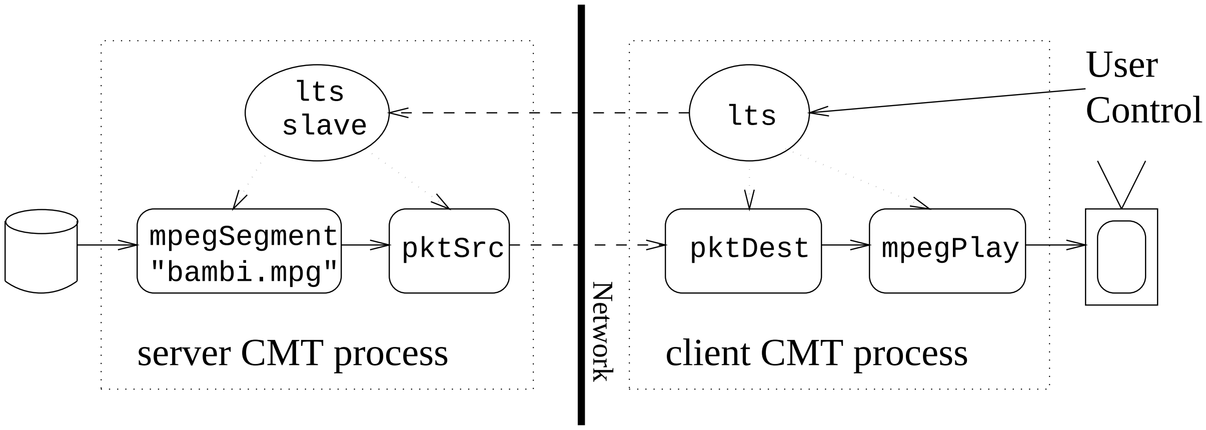


Figure 3: A complete playchain, including the LTS.

2.2 URL Resolution

CMT provides a name server that can map a URL into a specific CM clip-file. The URLs introduce a layer of indirection between the user and CM server processes. This indirection provides the flexibility necessary to use a centralized server which maintains a database of media stored on different sources [RBB95]. CM sources are specified using URLs similar to:

```
cmtp://name-server.edu/bambi.mpg
cmtp://name-server.edu:512/video/cinderella.mjpg
```

The protocol, identified by `cmtp`, specifies the name server and file. The name server returns a 3-tuple of the form:

{*host port filename*}

where *host* is the name of the machine that stores the clip-file *filename*, and *port* is the port to which the CMT process on *host* is listening. CMT provides a function `CMApp_TranslateURL` which contacts the name server and returns the 3-tuple. The function behaves as follows:

```
CMApp_TranslateURL cmtp://name-server.edu/bambi.mpg
```

returns the 3-tuple:

```
video-server.edu 1490 /video/mpg/bambi.mpg
```

By decoding the URL, an application can reference the video clip and the system can determine where the clip is located [RBB95].

2.3 The *CMT* object

CMT provides a mechanism to create objects on remote hosts. The CM process on a remote host is represented by an object, named `cmt`, on the client process. By using the 3-tuple returned by `CMApp_TranslateURL`, a user can create the objects required on the video server. Here is a set of commands which create the objects necessary for playback on the server side:

```
set server_process [cmt ""]
$server_process open -host video-server.edu 1490
$server_process create mpegSegment
$server_process create pktDest
$server_process create lts
```

The first two lines create the `cmt` object and connect it to the server process on `video-server.edu`. The next three lines create the `mpegSegment`, `pktDest`, and `lts` objects, respectively in the server process. Each `create` command returns the name of the newly created object. These objects can be found in the box labeled "server CMT process" in Figure 3. Note that initialization of these objects has been left out for simplicity.

The `cmt` object connected to the server process acts as a communication link between the client and server CMT processes. As shown above, the `cmt` object allows the user to create objects on remote processes. Each remote object has a proxy object with the same name on the client process. Commands can be executed on remote objects by using the proxy objects instead. Communication between the proxy objects and the actual objects is maintained through the `cmt` object. Figure 4 shows a complete playchain including the proxy objects, whose outlines are dotted.

The user must keep track of the objects in order to modify the *playchain*, to remove portions of the *playchain*, and to delete the *playchain* when finished. While maintaining this state is not a complex task, it obviously increases the complexity and size of application code. The next section introduces an API that manages the necessary state information automatically.

3 API Design

This section introduces the two abstractions that constitue the CMT Media Playback API. It also presents an example of how to use the abstractions, and a comparison with the code required if the API was not used.

The API specifies two new abstractions: *Stream* and *mediaPlayer*. A *Stream* is a media specific sequence of non-overlapping CM segments. Segments can be specified by URLs, and the source material clip-files can be located on arbitrary hosts. A *mediaPlayer* plays a given *Stream* on a local host. *MediaPlayers* are type specific. Currently there are two *mediaPlayers*, one for playing audio (`audioPlayer`) and one for playing video (`videoPlayer`).

Using these two abstractions, a user can write an application to play the video specified by the URL

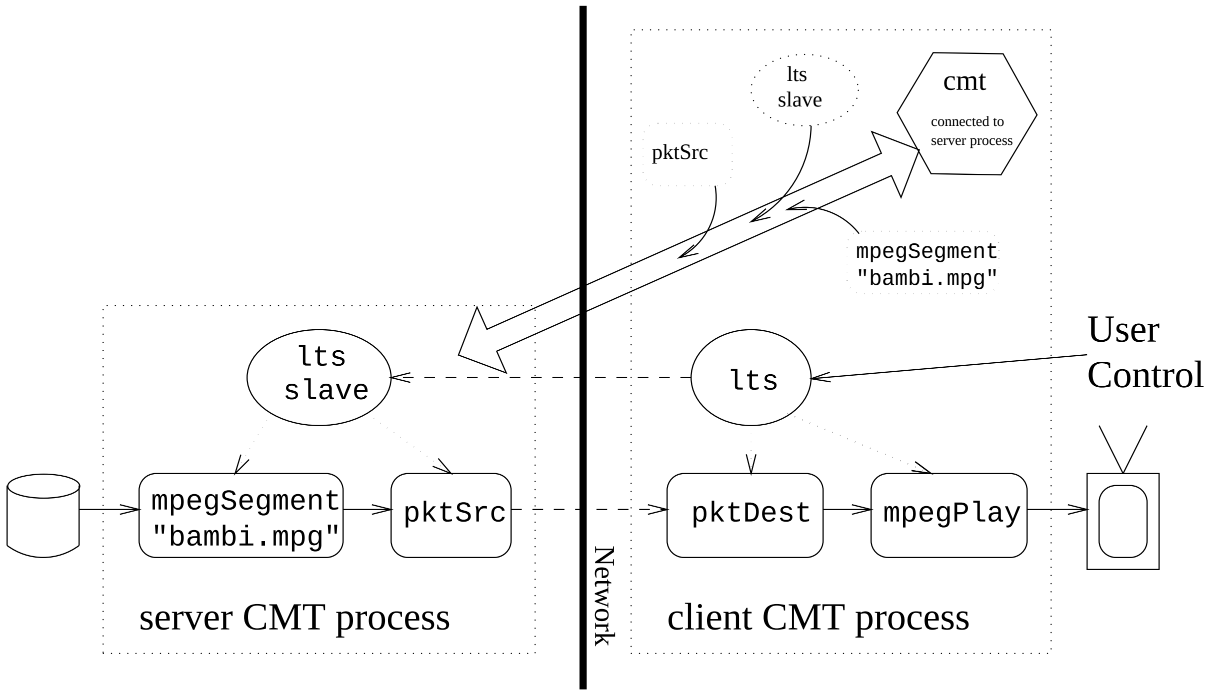


Figure 4: A playchain showing the proxy objects.

```
cmtp://name-server.edu:512/bambi.mpg
```

with the following six lines of code:

```
Stream create vid_stream video
vid_stream addclip "cmtp://name-server.edu:512/bambi.mpg"
videoPlayer create vid_player -stream vid_stream
vid_player ready
pack .vid_player
vid_player speed 1
```

The first line creates a *Stream* object named `vid_stream` with the type `video`. The second line appends a video clip to the *Stream*. The third line creates the `videoPlayer` object `vid_player`. The `-stream vid_stream` option specifies that `vid_player` will play the contents of `vid_stream`. The last three lines ready the player, display a window in which the video will be played, and start the video at speed 1 (normal speed).

For comparison, to achieve the same result without use of the API, the developer must write 27 lines of code to create and configure the objects in the *playchain*. The actual code is given and explained in Appendix A. The *playchain* created by the above code is similar to the *playchain* shown in Figure 3.

The user can play the audio portion of the bambi movie, specified by

```
cmtp://name-server.edu:512/bambi.au
```

with these additional 5 lines of code:

```
Stream create au_stream audio
au_stream addclip cmtp://name-server.edu:512/bambi.au
audioPlayer create au_player -stream au_stream
au_player configure -lts [vid_player cget -lts]
au_player ready
```

The only differences between the code that plays video and the code that plays audio are: 1) a window must be displayed for the video, and 2) the `audioPlayer` is configured to use the LTS that controls the `videoPlayer`. The two *mediaPlayers* are synchronized because the same LTS controls both players. It is possible to control playback using either the `speed` and `value` commands or by directly controlling the LTS. Both methods have the same effect, the first allows the application writer to avoid using the LTS.

Using the API, an application can dynamically add, modify, or delete video segments from a *Stream*, and the *mediaPlayer* will adjust the *playchain* to match. To play two video segments after the bambi movie, the application issues three more commands. One for each movie, and one to signal the `vid_player` object to ready itself to play the additional clips:

```
vid_stream addclip cmtp://name-server.edu/dumbo.mjpg
vid_stream addclip cmtp://name-server.edu/beauty.mpg
vid_player ready
```

These commands result in the *playchain* shown in Figure 5. Without the API, an application must execute an additional 50 lines of Tcl code.

In order to modify a clip, an application must first locate the proper *segment* object, then configure the object. An application must also destroy all CM objects when finished with a specific *playchain*. Some applications need to construct multiple *playchains* (e.g., a video editor), which adds more complexity to the application code.

The objects provided by the API keep track of the CM objects in the *playchain*. Changes to clips in the stream are made with only two lines of Tcl code, one to make the change and one to `ready` the *mediaPlayer*. A *playchain* can be destroyed using only one line of Tcl code. The *mediaPlayer* objects even conserve resources by reusing CM objects when possible. For example, Figure 5 shows a `pktDest` object shared by two `pktSrc` objects on different server processes. Every CM process in a *playchain* has only one LTS, one `pktSrc` per media format, one `pktDest` per media format, etc. In essence, the *Stream* and *mediaPlayer* objects represent *playchains*. Hence, commands on these objects are transformed into commands on the objects that implement the *playchain*.

4 API Details

This section describes the *Stream* and *mediaPlayer* abstractions, and their implementation. The first two subsections focus on the *Stream* and *mediaPlayer* abstractions. The object-oriented framework used to create the abstractions is discussed in the last subsection.

4.1 Stream

A *Stream* is a media-type specific segment. It contains an ordered sequence of media clips. A stream has a logical time line, and clips are ordered by the logical time at which they are to be played back. Each clip in a stream has a logical start (`ls`) and logical end (`le`) time. Clips in one stream are not allowed to overlap in logical time. Typically, the first clip in the stream begins at

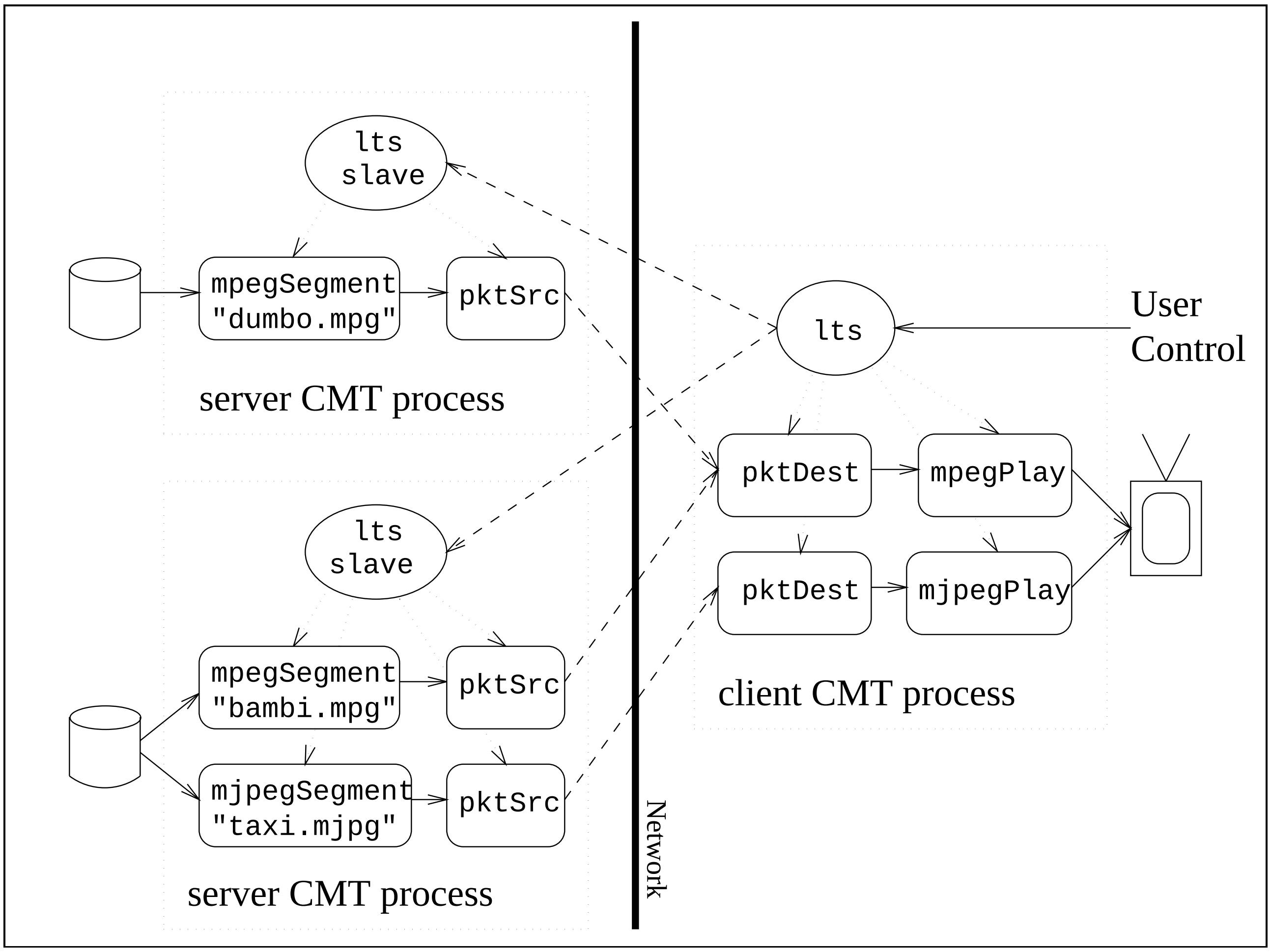


Figure 5: A complex playchain that plays 3 segments of video.

`ls` time 0. The clip data is actually stored in a clip-file. It has a source start (`ss`) and source end (`se`) time that specifies which portion of the clip-file should be played. Clips are added to streams using the `addclip` command:

stream_obj `addclip` *url* `[time specifiers]`

where `[time specifiers]` is any combination of the options shown in Figure 6.

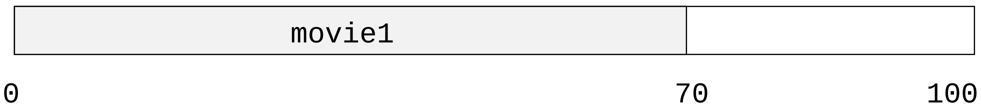
The `ss` and `se` options refer to the portion of the source clip to play. The source start defaults to 0 (the beginning), and the source end defaults to the length of the clip-file. The logical start (`ls`) and logical end (`le`) times specify the time to play the clip relative to the other clips in the stream. Logical start defaults to the logical end of the last clip in the stream. Logical end defaults to logical start plus the length of the clip to be added. Logical and source start times must be less than logical and source end times, respectively. Figure 6 shows a summary of the options and their default values.

Figure 7 shows several commands and a graphical representation of the corresponding time line for the stream. Assume `vidstream` initially contains no clips. The first command adds the 70 second movie, named `movie1.mpeg`, at the end of the (empty) clip-list. This clip is scheduled to

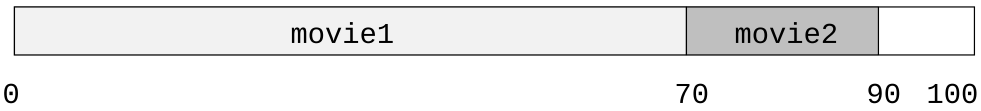
option	mnemonic	default value
[`-ss` *time*]	source start	0
[`-se` *time*]	source end	length of segment
[`-ls` *time*]	logical start	logical end of last segment in the stream
[`-le` *time*]	logical end	ls + actual length of segment in the stream

Figure 6: Timing options for streams.

```
vidstream addclip movie1.mpeg
```




```
vidstream addclip movie2.mpg -ss 10 -se 30
```




```
vidstream addclip movie3.mpeg -ls 20 -se 30
```

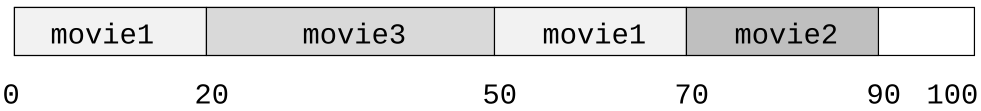


Figure 7: Depicts the effects of successive addclip commands.

play at logical time 0 because it is the first clip added to the empty stream. The second command adds a clip from the movie `movie2.mpg`, starting 10 seconds into the movie and ending 20 seconds later. Because no logical start time is specified, the clip is appended to the end of the stream. The final command specifies that the first 30 seconds of `movie3.mpeg` should be added to the stream beginning at logical time 20. The stream automatically splits the first clip into two smaller clips (one playing from 0 to 20, the other 50 to 70) and inserts the 30 seconds of `movie3.mpg` in the middle. At this point, the stream contains four clips.

After clips have been added to a *Stream*, they can be referenced by their position in the sequence. The following commands are supported:

stream_obj `getcliplist`	return the list of clips in the stream
stream_obj `getclip` *clipNum*	return clip number *clipNum*
stream_obj `deleteclip` *clipNum*	delete clip number *clipNum*
stream_obj `numclips`	return the number of clips in the stream

The clips returned by the stream object are in the form:

$$\left\{\ ss \quad se \quad ls \quad le \quad URL \quad format \quad \{\} \quad \{host \ \ port \ \ file\}\ \right\}$$

The first four elements of the clip are the timing information computed as described above. The URL is the specification added with `addclip`. The `format` specifies the media type and format (e.g., `video/mpeg`). And the last element of the clip is a triple specifying the location of the file specified by the URL.

A stream can also be manipulated using logical time as an index. The following commands manipulate logical time:

stream_obj `getstart`	return the *ls* of the first clip in the stream
stream_obj `getend`	return the *le* of the last clip in the stream
stream_obj `find` *time*	return a list of the clip index to played at logical time *time* and a non-zero value iff there is a clip scheduled to play at *time*
stream_obj `clear` *start end*	remove all clips scheduled to play between *start* and *end*
stream_obj `shift` *start delta*	shift all clips after time *start* by *delta* seconds

The commands `getstart` and `getend` return the logical start and end of the first and last clips, respectively, in the clip-list. `find` returns a list of two values: the index of the clip that is to be played at the specified time and a boolean value specifying that the clip is in the stream. `clear` removes all media to be played in between the specified start and end times. The `shift` command takes two arguments, *start* and *delta*, which specify the starting and amount of time by which to reschedule the stream.

To demonstrate, let us look at `vidstream` in its state at the end of Figure 7. Below is a table of commands and the corresponding output that would be returned.

```
vidstream getstart   ⇒  0
vidstream getend     ⇒  90
vidstream numclips   ⇒  4
vidstream find 54    ⇒  2 1
vidstream getclip 1  ⇒  {0 30 20 50 movie3.mpeg video/mpeg \
                            {} {localhost 0 movie3.mpeg} }
```

Figure 8 shows the effects of the commands `deleteclip`, `shift`, and `clear` on the stream `vidstream`. The `deleteclip` command removed the first clip in the stream. The `shift` command

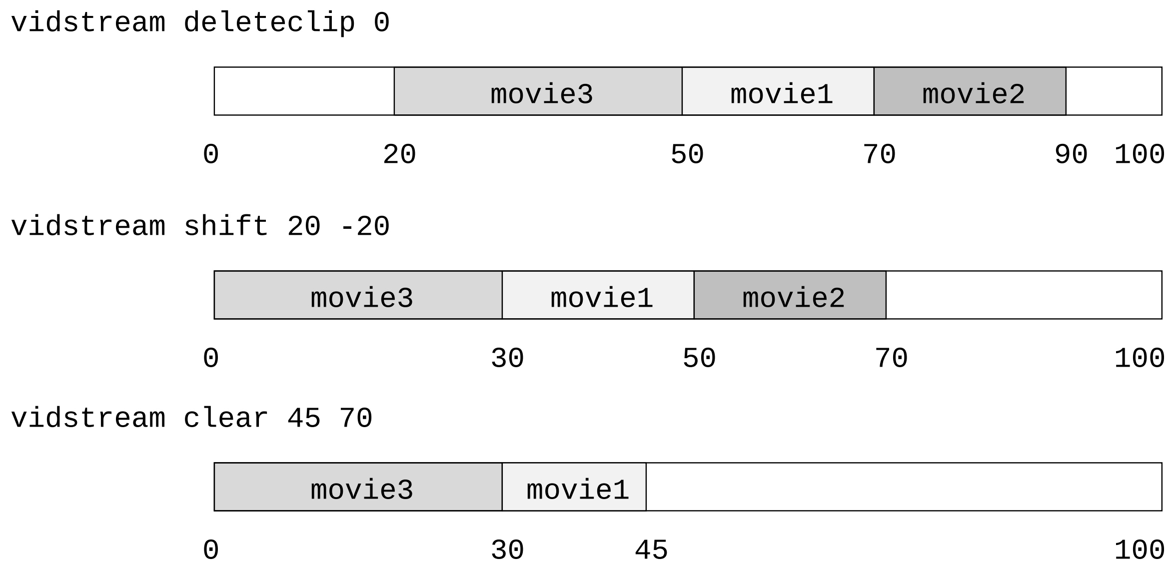


Figure 8: Depicts the effects of various mutative commands.

scheduled every clip to play 20 seconds earlier. The `clear` command deleted the third clip and truncated the second.

A complete listing of commands supported by the *Stream* abstraction are given in Appendix B.

4.2 MediaPlayer

A *mediaPlayer* plays back the contents of a CM *Stream*. It manages the *playchain* necessary to play the *Stream* and automatically adjusts the *playchain* when the stream is modified. Currently, there are two variants of the *mediaPlayer*: `audioPlayer` and `videoPlayer`. The two abstractions are classes that share a common abstract parent class, named `playerBase`. `playerBase` contains all media independent code such as setting up network connections (i.e., `cmt` objects), creating LTSs, `pktDest`, `pktSrc`, segment objects, and so forth. The segment objects are considered "media independent" because all segment objects are initialized the same way, and the clips contain the media format (e.g., mpeg, mjpeg, au) necessary to create the correct object. The `audioPlayer` similarly configures the `auPlay` object to send the audio to the proper device. Figure 9 shows the relationship of the three classes.

MediaPlayers also have user configurable slots accessed using `configure` and `cget`, just as slots in Tk widgets are accessed. All *mediaPlayers* require the `stream` slot to be initialized with an existing stream of the proper type. The `lts` slot specifies which LTS object controls playback. If an LTS is not specified, one is created automatically. The last slot common to all *mediaPlayers* is `autoready`. If the value in this slot is nonzero, the *mediaPlayer* will automatically ready itself whenever the stream is changed. The `autoready` slot defaults to 0. Setting `autoready` to 0 allows an application that wants to make several changes to a stream to defer changes to the *playchain* objects until all stream changes are specified.

The `audioPlayer` has two more configurable options: `device` and `gain`. `device` specifies which output device to use. It can be set to one of three options: "default," "sparc," and "af." `gain` controls the output gain at which sound should be played.

The `videoPlayer` has one additional slot, named `frame`. By default, a `videoPlayer` will automatically create a Tk frame of the same name as the `videoPlayer` (with a '.' prepended), and video will be displayed in that window. The application can provide a different Tk frame by setting the frame slot.

All *mediaPlayers* support a common set of five commands:

mediaPlayer	`ready`	readies the *playchain* for playback
mediaPlayer	`unready`	unreadies objects in the *playchain*
mediaPlayer	`speed` n	sets the playback speed to n
mediaPlayer	`value` n	sets the playback time to n
mediaPlayer	`destroy`	deletes the *playchain*

MediaPlayer objects do not react to changes to a *Stream* object until a `ready` command is given unless the `autoready` slot is non-zero. Upon receiving a `ready` command, the *mediaPlayer* checks each of the clips in the *Stream*. If the clip is new, the *mediaPlayer* creates and configures all CMT objects necessary to play that clip. If the clip's timing specifiers have changed, the *mediaPlayer* updates the necessary CMT objects. A `ready` command has no effect when the *playchain* is up to date.

The `speed` and `value` commands have the effect of configuring the object in the *mediaPlayer's* LTS slot to have the same value. For example, the following two lines are equivalent:

```
vid_player value 2
[[vid_player] cget -lts] configure -value 2
```

The one exception is that the `value` command takes arguments of `begin` and `end`, which configure the LTS to 0 and the logical end of the *Stream* respectively.

The `unready` command causes a media player to issue `unready` commands to all CM objects in the *playchain*, thereby disabling playback. The `destroy` command causes a *mediaPlayer* object to destroy all objects in the *playchain* and remove itself from the current CMT process.

Appendix C provides complete specifications of the `playerBase`, `audioPlayer` and `videoPlayer` classes.

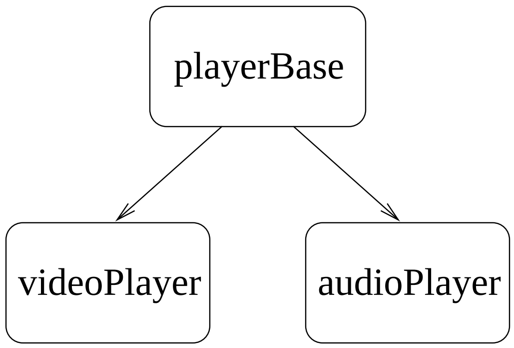


Figure 9: API Object Hierarchy

4.3 Object Tcl

The CMT Media Playback API was written using an object-oriented extension to Tcl, called Object Tcl (OTcl) [WL95]. OTcl provides an object-oriented framework for managing complex data types. OTcl objects can be defined using Tcl or C.

One reason CMT uses a Tcl interface is because it is well-suited for rapid code development. Nevertheless, many internal objects are written in C for efficiency. We decided that rapid development was more important than speed because the API does not perform time-critical tasks. Consequently, the difference between a Tcl and C implementation was not an important factor. Since the API is written in Tcl it is easy for others to modify and extend the abstraction to suit their needs.

The following four features were important criteria for using an object-oriented extension to implement the Media Playback API:

- Provides data encapsulation.
- Provides inheritance (code reuse).
- Object interface is Tcl-like.
- Easy to integrate into Tcl.

First and foremost the API is designed to make the task of writing CM applications easier. A primary part of the API's function is to maintain the state of *playchains*. Tcl does not provide a convenient mechanism for encapsulating data, but an object-oriented extension does.

An object-oriented system also provides inheritance. All the functionality common to *videoPlayers* and *audioPlayers* has been placed in the *playerBase* definition. Once the *videoPlayer* class had been written and debugged, the *audioPlayer* was written in 15 minutes. Adding other new types of media may be just as easy.

Two specific object-oriented extensions to Tcl were considered: [incr Tcl][McL93] and OTcl. While both systems provide the functionality necessary to implement the API, OTcl was chosen because its interface follows the same conventions as built-in Tcl and Tk objects. And it is easy to integrate into Tcl. [incr Tcl] requires changing the core of the Tcl interpreter, and that means CMT would have to ship and support a modified Tcl interpreter. This requirement increases the overhead for CMT developers, and inhibits application developers. [incr Tcl] also uses a C++ style interface, which is awkward and looks out of place in Tcl code. Code uniformity was an important factor in the design.

5 Helpful Tools

This section describes two auxiliary functions written to assist developers using the CMT Media Playback API and presents an example application that demonstrates the power of the API.

Two functions, `CMApp_ReadStream` and `CMApp_WriteStream`, save the description of streams. Each takes the name of a stream and a filename. `CMApp_ReadStream` creates a stream with the name *stream* and initializes it with the stream description from *filename*. `CMApp_WriteStream` does the opposite, it writes the description from *stream* to *filename*. Here are examples:

```
CMApp_WriteStream vid_stream  /streams/stream1
CMApp_ReadStream new_stream  /streams/stream1
```

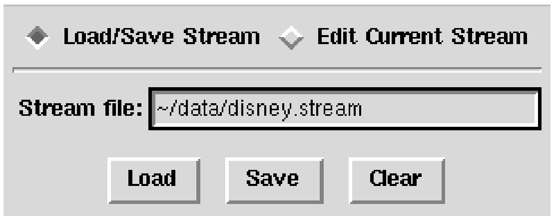


Figure 10: Stream File window

Figure 11: Display window

The first line creates a plaintext file named `stream1` whose contents describe the sequence of clips stored in `vid_stream`. The second line creates a *Stream* object, named `new_stream`, and initializes it with a clip-list identical to that in `vid_stream`.

To gain some insight into the benefit of using the API, we wrote a very simple video editor. The editor allows the user to read and write files containing descriptions of *Streams*, add to and delete from the clip-list, and play the desired stream. The entire program contains 157 lines of Tcl, 26 of which deal directly with CM objects or the API presented here. The code for this application is available in Appendix D. Section 3 showed that using CMT alone required 27 lines of code to just create a simple *playchain*.

The Stream File window, shown in Figure 10, allows the user to specify the name of a file to use when loading or saving the *Stream* description. When the user presses the “Load” button, the description from the specified file is loaded into a *Stream* object. The *Stream* object is then readied for play and the contents displayed in the Display window (Figure 11). Pressing the “Save” button

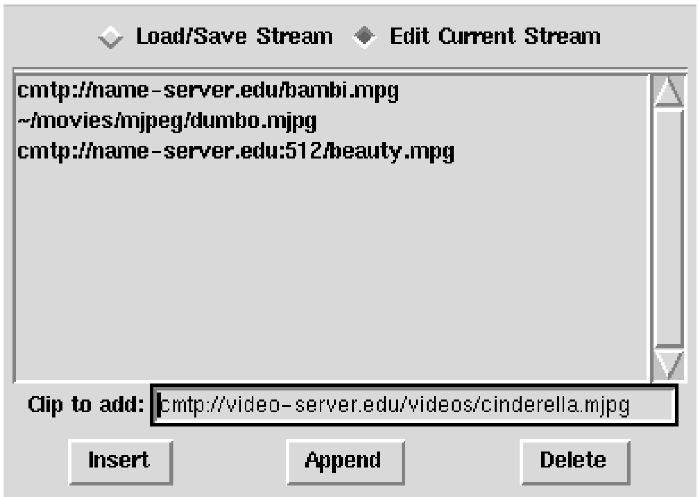


Figure 12: Stream Edit window

causes the editor to save the description of the current *Stream* into the specified file. The "Clear" button erases the file name field.

When the user presses the "Edit Current Stream" radiobutton in the Stream File window, as shown in Figure 12, the window displays a list of clips in the current stream. Using the "Clip to add" entry field, the user can type in the clips s/he wishes to add to the stream. The "Append" button adds the clip to the end of the list, while the "Insert" button puts the clip in front of the selected file. Clips are deleted by selecting one in the list and clicking on the "Delete" key. Pressing the "Load/Save Stream" radiobutton causes the window revert back to the Stream File window appearance.

The Display window allows the user to use VCR-like controls to view the contents of the video stream.

This application required 157 lines of code. It is very simple and lacks features needed for a non-linear video editor. However, it does show that the API allows an application programmer to spend little time and effort getting the continuous media portion of the application to work. Only 16% of the code manipulates CM objects. Our goal was to develop a high-level interface which would simplify CM application development. We believe this API succeeded.

Two applications have been written using the API: CMEdit and Ceedit. CMEdit [Bal96] is a non-linear CM editing application. Ceedit [Bac96] is a shot boundary editor which provides a mechanism for examining the output of shot detection algorithms.

6 Future Work

This section presents some issues raised by concurrent research, and suggests related problems we believe need to be addressed.

CMEdit was developed concurrently and raised many important issues addressed by the CMT Media Playback API. For example, whether to merge the *mediaPlayer* and *Stream* class abstractions into one class, whether *mediaPlayers* should be Tk widgets, and so forth. More discussion with application writers is needed to improve the design of the *mediaPlayers* and *Streams*. Will the players remain unchanged because most interesting applications will be user-interface related? Or will new algorithms for frame dropping and network feedback be added to CM objects, thus requiring a more flexible interface to the *mediaPlayer* objects? Only by developing more applications will it be possible to determine appropriate choices.

A continuous text CMT object is being developed that will add time sensitive text [Jaf]. It will be incorporated into the media player framework. However, it is unknown whether the *Stream* class will need to be changed to supplement this abstraction or whether a new class will be needed.

Currently, CMT does not have `videoSegment` or `audioSegment` objects that understand live sources, such as camera and microphone. Audio and video capture objects exist, but they cannot be used in a CMT script. After modifying the segment objects to understand live sources, existing CM applications will be able to play live sources interchangeably with stored sources. For example, a CM script can be constructed that specifies that one window shows 10 seconds from several cameras in some order. This script can be saved in the Berkeley VOD system [RBB95] like any other playback object.

The name resolution service needs to allow processes, for example a camera object, to register a name for themselves. By allowing applications and objects to dynamically add names to a name service, client applications can resolve a well-known URL to play the live sources, regardless of where the source originates. The resolution service should also incorporate new formats to allow processes to register multicast data streams.

The API presented in this paper does not provide any means to send or receive multicast data streams. A major obstacle to this extension is the lack of name resolution for multicast sessions. Additionally, a multicast-receive object is conceptually different from the *mediaPlayers* presented in this paper. *MediaPlayers* create *playchains* from the media source to the output device (terminal or speaker), but multicast objects only create "half" of a *playchain* because they either send or receive data. Does it make sense to fold the functionality of a multicast receive object into the *mediaPlayer* class? Is there any benefit to doing so?

7 Conclusion

This paper introduced the CMT Media Playback API that manages playback of CM streams. The *Stream* and *mediaPlayer* classes were defined to simplify program development while at the same time providing enough flexibility to be useful in a variety of applications. Future media objects should strongly resemble those already existing CM objects. The similarities will allow *Stream* and *mediaPlayer* classes to be easily changed to incorporate new media objects.

So far, a non-linear video editor and a shot-boundary editor have been implemented using the API. Both applications focused on manipulating the *Stream* objects rather than *playchains*. It is encouraging to know the *Stream* abstraction is flexible enough to accommodate the two applications.

The tools described here were built using an object-oriented framework built on top of Tcl. The OTcl framework provided elegant code reuse, which should be applied to CMT as a whole.

References

[Ack94] Philipp Ackermann. Direct manipulation of temporal structures in a multimedia application framework. In *Multimedia 94*, San Francisco, CA, October 1994.

[App] Apple Corp., Apple Quicktime Conferencing. `http://qtc.quicktime.apple.com/qtc/qtc.faq.tech.html`.

[Bac96] David Bacher. GUI tools for the Berkeley Distributed Video-on-Demand Database. Master's thesis, University of California at Berkeley, 1996.

[Bal96] J. Eric Baldeschwieler. Editing extensions to the Berkeley Continuous Media Toolkit. Master's thesis, University of California at Berkeley, 1996.

[dMG93] V. de Mey and S Gibbs. A multimedia component kit. In *ACM Multimedia 93 Proceedings*, pages 291–300, Anaheim, CA, June 1993.

[Ins] InSoft Corp., OpenDVE Home Page. `http://www.insoft.com/products/OpenDVE/OpenDVE.html`.

[Jaf] Fiesal mahmood nawaz jaffer, personal conversations, 1996.

[LSR95] P. Liu, B.C. Smith, and L.A. Rowe. Tcl-DP name server. In *Proceedings of 9th International Conference on Distributed Computing Systems*, Toronto, Canada, July 1995.

[McL93] Michael J. McLennan. Object Oriented Programming in TCL. In *Proceedings of the Tcl/Tk Workshop 1993*, Berkeley, California, May 1993.

[MFY94] Robert F. Mines, Jerrold A. Friesen, and Christine L. Yang. Dave: A plug and play model for distributed multimedia application development. In *Multimedia 94*, San Francisco, CA, October 1994.

[MPR96] Ketan Mayer-Patel and Lawrence A. Rowe. Design and performance of the continuous media toolkit. `http://bmrc.berkeley.edu/projects/cmt/index.html`, 1996.

[MSa] Microsoft Corp., ActiveMovie streaming format - product information. `http://www.microsoft.com/advtech/activemovie/productinfo.htm`.

[MSb] Microsoft Corp., Interactive Media Technologies: Direct3D. `http://www.microsoft.com/imedia/direct3d/direct3d.htm`.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Co., 1994.

[RBB95] Lawrence A. Rowe, David A. Berger, and J. Eric Baldeschwieler. The Berkeley Distributed Video-on-Demand System. In T. Ishiguro, editor, *Multimedia Computing, Proc. 6th NEC Research Symposium*, 1995.

[SRY93] B.C. Smith, L.A. Rowe, and S. Yen. Tcl Distributed Programming. In *Proceedings Tcl 1993 Workshop*, Berkeley, CA, 1993.

[Tea95] D. Tennenhouse and et. al. The ViewStation: a software-intensive approach to media processing and distribution. *Multimedia Systems*, 3:104–15, 1995.

[WL95] David Wetherall and Christopher J. Lindblad. Extending Tcl for dynamic object-oriented programming. In *Proceedings of the Tcl/Tk Workshop 1995*, Toronto, Ontario, July 1995.

A Tcl script not using the API

This appendix contains the Tcl code required to play the movie specified by `cmtp://name-server.edu:512/bambi.mpg` without using the API.

```
% create local objects
set frame [frame .vp]
set local_lts [lts ""]
set mpeg_play [mpegPlay ""]
set pkt_dest [pktDest ""]
set remote_cmt [cmt ""]

% open connection
$remote_cmt open -host video-server.edu 1490

% create remote objects
set pkt_src [$remote_cmt create pktSrc]
set remote_seg [$remote_cmt create mpegSegment]
set remote_lts [$remote_cmt create lts]

% connect LTSs
$local_lts addSlave $remote_lts

% configure network objects
% NOTE: the '@' tells the object to use the C function call
% corresponding to the given Tcl command
$pkt_src  configure -dest [$pkt_dest address]
$pkt_dest configure -outCmd @$mpeg_play.accept

% configure segment object
$remote_seg configure -outCmd @$pkt_src.accept -lts $remote_lts \
    -filename /video/bambi.mpg -clipStart 0 -clipEnd 100 \
    -logicalStart 0 -logicalEnd 100

% configure play object
$mpeg_play configure -lts $local_lts -device software -squeeze 0\
    -xpos -1 -ypos -1 -xid [winfo id $frame]
$mpeg_play configure -width [$remote_seg width] -height [$remote_seg height]

% ready objects
$mpeg_play ready;  $remote_seg ready

% configure and display window
$frame configure -width [$remote_seg width] -height [$remote_seg height]
pack $frame

% start video
$local_lts configure -speed 1
```

B Stream specification

This appendix contains two subsections. The first describes the interface to the *Stream* class as it pertains to application programmers who use the API. The second subsection presents details of the *Stream* class interface that may be useful for programmers wishing to extend or modify the *mediaPlayer* classes.

B.1 Stream API details

`Stream create` *stream type*

creates a Stream named *stream* of type *type*
type must be either `video` or `audio`

stream `destroy`

causes *stream* become undefined

stream `gettype`

returns the type specified upon creation

stream `getstart`

returns the logical start of the first clip

stream `getend`

returns the logical end of the last clip

stream `getnumclips`

returns the number of clips

stream `getclip` *clipNum*

returns clip number *clipNum* in the form:

ss se ls le URL format {} { host port file}

The first four elements of the clip are the timing information which is computed as described in (see `addclip`). The URL is the specification added with `addclip`. The `format` specifies the media type and format (e.g., `video/mpeg`). And the last element of the clip is a triple specifying the location of the file specified by the URL.

stream `getcliplist`

return a list of the clips in the stream
see `getclip` for clip format

stream `addclip url [time specifiers]`

add the clip specified by the URL to the cliplist
the optional time specifiers default to

option	mnemonic	default value
[`-ss` *time*]	source start	0
[`-se` *time*]	source end	length of segment
[`-ls` *time*]	logical start	logical end of last segment
[`-le` *time*]	logical end	ls + actual length of segment

stream `setcliplist` *resolved_cliplist*
set the cliplist to be *resolved_cliplist*
each clip in the *resolved_cliplist* must be fully specified

stream `deleteclip` *clipNum*
remove clip number *clipNum* from the cliplist

stream `shift` *start delta*
move all clips on or after *start* time by *delta* time units
if *delta* is positive, a clip that spans *start* will be split in two
if *delta* is negative, all clips existing entirely in the region [*start* + *delta*, *start*]
will be deleted, a clip that spans (*start* + *delta*) will be truncated

stream `clear` *start end*
truncate clips that span either *start* or *end*
remove all clips in the range [*start*, *end*]
start can be "start", *end* can be "end" (in which case they default to the logical
start of the first clip and the logical end of the last clip respectively)

stream `find` *time*
return a list of two values:
the index of the clip that is to be played at time *time*
and a boolean value specifying that a clip exists at time *time*

stream `snap` *time dist*
return the time of the closest clip boundy within *dist* of time
If there is none, then *time* is returned
dist is allowed to be negative (takes absolute value)

B.2 Stream low-level details

This subsection demonstrates *Stream* commands that may be useful to a programmer who wishes to extend or modify the API. Let us assume we have created stream `vid_stream` with two clips by evaluating the following:

```
Stream create vid_stream video
vid_stream addclip cmtp://name-server.edu/bambi.mpg
vid_stream addclip cmtp://name-server.edu/dumbo.mjpg
```

In order to help determine when a clip has been modified, the *Stream* object maintains a set of 32 flags for each clip. Typically an application will need only one flag per clip. The `allocflag` command returns an unused flag index. The application can now reference and modify that flag on any clip in the *Stream* using that index. For example, the following stores a newly allocated flag index for the stream vid_stream in the variable flagI.

```
set flagI [vid_stream allocflag]
```

On all clips in vid_stream, flag number $flagI has now been reserved for that application. All flags are initialized to 1, and when a clip is created or modified its flags are reset to 1.

The command clipflag can be used to test and set the value of clip flags in a *Stream*.

```
vid_stream clipflag 0 $flagI     ⇒  1
vid_stream clipflag 0 $flagI 0   ⇒  0
vid_stream clipflag 0 $flagI     ⇒  0
vid_stream shift 0 10
vid_stream clipflag 0 $flagI     ⇒  1
```

The first line tests the value of $flagI on clip 0, the value is 1. The second line sets the value of $flagI on clip 0 to be 0. The third line tests that the flag is 0. The fourth line causes clip 0 to be modified, and the fifth line shows that the flag $flagI has in fact been reset to 1.

Application writers may also want to associate a piece of data with each of the clips in a *Stream*. The clipprop command allows an application to associate a value with a name for each clip. This command works just like clipflag, except that the value does not change when the clip is modified.

```
vid_stream clipprop 0 test_prop 3     ⇒  {test_prop 3}
vid_stream clipprop 0 test_prop       ⇒  3
vid_stream clipprop 0 test_prop2 12   ⇒  {test_prop2 12} {test_prop 3}
vid_stream clipprop 1 test_prop       ⇒  {}
```

The first line creates a property with the name test_prop and the associated value of 3. When assigning a value to a clip property, a list of all the property names and corresponding values is returned. The second line queries vid_stream for the value associated with test_prop and a 3 is returned. The third line demonstrates the result when a second property is added to the same clip. The fourth line shows that the properties are maintained on a per-clip basis.

Stream objects have a built in looping mechanism. The format of the foreach command is as follows:

stream foreach *start end sName index script*

The *start* and *end* are specify which clips start and end the loop. *sName* is a name of a variable that will be assigned the name of the *Stream* inside the script. *index* is a name of the loop variable. And *script* is the script to run.

The *Stream* class also provides callback hooks for applications. A hook is a list of scripts that will be executed when a specific event occurs. *Streams* have four hooks: addclip, deleteclip, copyclip, and destroystream. Hooks are manipulated with the following commands:

stream gethooks *hookName*
stream addhook *hookName script*
stream removehook *hookName script*

The `gethook` takes the name of the hook and returns a list of the scripts to be called. `addhook` adds `script` to the event specified by *hookName*. `removehook` removes the specified *script* from the event *hookName*.

The scripts in each hook are called with arguments. The `destroystream` hook is called with the name of the *Stream* when it has been given a `destroy` command. The `addclip` hook is called with the name of the *Stream* and the index of the clip that has been added. When an existing clip is split in two (by a `shift` command for example), the `copyclip` hook is called with the name of the *Stream* and the index of one of the copies.

The following code creates a *Stream* and adds a script that will print the arguments each time a hook is called.

```
Stream create foo video

proc sa args {puts "sa - '\$args'"}

foo addhook addclip {sa clipadd}
foo addhook deleteclip {sa deleteclip}
foo addhook copyclip {sa copyclip}
foo addhook destroystream {sa destroy}
```

C videoPlayer and audioPlayer specifications

This appendix contains the API specifications for the *mediaPlayers* `videoPlayer` and `audioPlayer`. Section C.1 describes the behavior of the `playerBase` class, from which the `videoPlayer` and `audioPlayer` both inherit most of their behavior. The attributes found only in the `videoPlayer` and `audioPlayer` classes are described in sections C.2 and C.3 respectively.

The classes are specified by listing the commands that each class supports as well as user configurable slots. Commands are uses as messages passed to the object, for example:

player command *arg1 arg2 arg3* ...

The arguments *arg1 arg2 arg3* ... are necessary only when specified.

The *mediaPlayer* objects have slots just like those of Tk widgets. The slots are set by using the `configure` command and slots are read by using the `cget` command, for example:

player configure -slot1 *value* *player* cget -slot1 $\Rightarrow$ *value*

C.1 playerBase

The `playerBase` class contains functionality common to both the `videoPlayer` and `audioPlayer` classes.

Commands:

`ready`
: creates CM objects required to play clips in the stream

`unready`
: sends an unready command to all segment and play objects

`destroy`
: deletes the player object
: causes the player object to destroy all CM objects it created

`speed` n
: configures the -speed slot of LTS that controls playback to be n

`value` n
: configures the -value slot of LTS that controls playback to be n

Slots:

`lts`
: contains the name of the lts that controls playback

`stream`
: contains the name of the *Stream* to play

`autoready`
: when non-zero stream automatically readies itself every time
the stream is modified

C.2 videoPlayer

The `videoPlayer` class adds one more slot and modifies the behavior of the `destroy` command.

Commands:

`destroy`
: if the user uses the frame created by default
this command destroys the frame used to display video

Slots:

`frame`
: frame in which to display video

C.3 audioPlayer

The `audioPlayer` adds new slots to the `playerBase` class.

Slots:

`device`
: specifies which output device to use
must be one of: "default," "sparc," or "af"

`gain`
: specifies output gain at which sound is to be played
must be a value between 0 and 1

D Example application code

This appendix contains the code for the application described and shown in section 5.

```
## utility function to read stream description into a newly
## created stream
proc CMApp_ReadStream {filename stream} {
    set result [CMApp_ResolveMovie [CMApp_ReadMovie $filename]]

    ## if already exist, destroy it
    if { [catch {$stream getcliplist}] == 0 } {
        $stream destroy
    }

    Stream create $stream [lindex [lindex $result 1] 0]

    $stream setcliplist [lrange [lindex $result 1] 1 end]
}

## utility function to write stream description to file
proc CMApp_WriteStream {stream filename} {
    CMApp_WriteMovie [list "" [lappend [$stream gettype] \
                                [$stream getcliplist]]] $filename
}


## setup load/save stream file window
proc setup_file_menu {win} {
    global streamFile

    frame $win

    frame $win.input
    pack $win.input -side top -expand 1 -fill x
    label $win.input.label -text "Stream file:"
    entry $win.input.entry -width 30 -relief sunken -bd 2 \
                           -textvariable streamFile
    pack $win.input.label -side left -pady 10
    pack $win.input.entry -side left -expand 1 -fill x

    frame $win.group
    pack $win.group -side bottom

    set win $win.group
    button $win.load -command "load_file" -text Load
    button $win.save -command "save_file" -text Save
    button $win.clear -command "set streamFile {}" -text Clear
```

```
    pack $win.load $win.save $win.clear -side left -pady 5 -padx 5
}

## load Stream description
proc load_file {} {
    global streamFile curP

    CMApp_ReadStream $streamFile curStream

    $curP config -stream curStream
    $curP ready
}

## save Stream description
proc save_file {} {
    global streamFile

    CMApp_WriteStream curStream $streamFile
}


## create VCR-like controls
proc vid_controls { win } {
    frame $win.f
    pack $win.f -side bottom -padx 2 -pady 2
    set win $win.f

    button $win.play -text Play -command {$glts config -speed .8}
    button $win.pause -text Pause -command {$glts config -speed 0}
    button $win.ff -text FastForward -command {$glts config -speed 2}
    button $win.rw -text Rewind -command {$glts config -speed -2}
    pack $win.play $win.pause $win.ff $win.rw -side left
}



## setup edit stream window
proc setup_edit_menu {win} {
    global sourceURL

    frame $win

    ## entering info frame
    frame $win.enter
    pack $win.enter -side bottom -expand 1 -fill x
    set WE $win.enter
```

```
    ## entering info buttons
    frame $WE.buts
    pack $WE.buts -side bottom -fill x -pady 5 -expand 1
    button $WE.buts.del -text Delete -command "delete_clip $win.clips"
    button $WE.buts.app -text Append -command "append_clip $win.clips"
    button $WE.buts.ins -text Insert -command "insert_clip $win.clips"
    pack $WE.buts.ins $WE.buts.app $WE.buts.del -side left -expand 1 -padx 20

    ## entering info label/entry
    frame $WE.data
    pack $WE.data -side top -expand 1 -fill x -padx 5
    label $WE.data.label -text "Clip to add:"
    entry $WE.data.entry -width 30 -relief sunken -bd 2 \
                     -textvariable sourceURL
    pack $WE.data.label -side left
    pack $WE.data.entry -side left -expand 1 -fill x


    ## listbox to display clips in stream
    listbox $win.clips -relief sunken -borderwidth 2 -yscrollcommand \
                "$win.scroll set" -highlightthickness 0
    pack $win.clips -side left -fill x -expand 1
    scrollbar $win.scroll -command "$win.clips yview" -relief sunken \
                    -highlightthickness 0
    pack $win.scroll -side right -fill y
}


## delete clip from stream & listbox
proc delete_clip {lbox} {
    global curP

    ## adjust the Stream and Player
    set clipnum [$lbox curselection]

    if { $clipnum == {} } {
        return
    }

    set ls [lindex [curStream getclip $clipnum] 0]
    set le [lindex [curStream getclip $clipnum] 1]
    set neglen [expr $ls - $le]

    curStream deleteclip $clipnum
    curStream shift $le $neglen
    $curP ready
```

```
    ## update list
    update_edit_menu
}

## append clip to stream & listbox
proc append_clip {lbox} {
    global curP sourceURL

    ## adjust Stream and Player
    curStream addclip $sourceURL
    $curP ready

    ## update list
    update_edit_menu
}

## insert clip before the clip selected in the listbox
proc insert_clip {lbox} {
    global curP sourceURL

    ## adjust Stream and Player
    set clip [CMApp_ResolveClip [CMApp_ParseClipLine $sourceURL] 0]
    set clipnum [$lbox curselection]

    if {[curStream getnumclips] == 0} {
        curStream addclip $sourceURL
        update_edit_menu
        $curP ready
        return
    }

    if {$clipnum == {}} {
        set clipnum [expr [curStream getnumclips] - 1]
    }

    set oldclip [curStream getclip $clipnum]

    ## shift by length of new clip, and insert new clip
    curStream shift [lindex $oldclip 0] [expr [lindex $clip 1] - [lindex $clip 0]]
    curStream addclip $sourceURL -ls [lindex $oldclip 0]

    $curP ready
    update_edit_menu
}

## display list of clips in listbox
proc update_edit_menu {} {
```

```
    global editWin

    $editWin.clips delete 0 end

    foreach i [curStream getcliplist] {
        $editWin.clips insert end [lindex $i 4]
    }
}

## create two windows, one for display, one for load/edit/save stream
proc setup_gui_menu {win} {
    global which_menu editWin fileWin

    ## buttons for toggling stream manipulation
    frame $win.buts -relief groove
    pack $win.buts -padx 5 -pady 5 -expand 1
    radiobutton $win.buts.fileBut -text "Load/Save Stream" \
                                  -variable which_menu -value 1
    radiobutton $win.buts.editBut -text "Edit Current Stream" \
                                  -variable which_menu -value 0
    pack $win.buts.fileBut $win.buts.editBut -side left -expand 1

    bind $win.buts.fileBut <ButtonRelease-1> {update_gui_menu}
    bind $win.buts.editBut <ButtonRelease-1> {update_gui_menu}

    ## divider
    frame $win.seperate -relief sunken -height 3 -borderwidth 2
    pack $win.seperate -padx 5 -expand 1 -fill x

    ## set up the two windows
    set editWin $win.editWin
    set fileWin $win.fileWin
    setup_edit_menu $editWin
    setup_file_menu $fileWin
    pack $fileWin -side bottom -expand 1 -fill x -padx 5
}

## called to switch which frame is displayed
proc update_gui_menu {} {
    global which_menu editWin fileWin

    if { $which_menu } {
        pack forget $editWin
        pack $fileWin  -side bottom -expand 1 -fill x -padx 5
    } else {
        pack forget $fileWin
        pack $editWin -side bottom -expand 1 -fill x -padx 5
```

```
        update_edit_menu
    }
}


## set up playback window
toplevel .display;

## playback obj
videoMaster create display.curPlayer
pack .display.curPlayer;

## VCR controls
vid_controls .display

## shorter name
set curP display.curPlayer

## for use by VCR controls and upate_gui_menu
set glts [$curP cget -lts]
set which_menu 1


## global vars
set streamFile {}
set sourceURL {}
set editWin {}
set fileWin {}
Stream create curStream video

$curP config -stream curStream

## start the application
toplevel .gui
setup_gui_menu .gui
```