# ABSTRACTIONS FOR CONTINUOUS MEDIA IN A NETWORK WINDOW SYSTEM

*David P. Anderson*
*Ramesh Govindan*
*George Homsy*

Computer Science Division, EECS Department
University of California at Berkeley
Berkeley, CA 94720

September 19, 1990

## ABSTRACT

ACME is a set of abstractions for the input and output of "continuous media" (audio and video) by a network window system. In ACME, continuous media (CM) data is handled by user-level software in both the client and server, and is conveyed on network connections. The ACME design has the goals of network transparency, device independence, support for concurrency, and management policy independence. The ACME abstractions include *strands* (streams of audio or video data), *ropes* (combinations of several strands), *logical time systems* (reference frames in which several strands or ropes can be played synchronously), and *logical devices* (representing microphones, speakers, video cameras, and video windows).

---

Sponsored by the California MICRO program, Digital Equipment Corporation, IBM Corporation, Olivetti S.p.A, Sun Microsystems, and the National Science Foundation.

## 1. INTRODUCTION

Future workstations will support *continuous media* (digital audio and/or video I/O) as well as *discrete media* (bitmap display, mouse and keyboard). Our projected "typical workstation" has a video camera attached to the workstation and aimed at the user, and perhaps an additional camera for scanning nearby objects. These cameras are interfaced by video digitizers. The workstation is able (perhaps using a video DSP coprocessor) to compress and decompress video data in real time to a data rate that can be accommodated by networks and file systems. displaying the result within the bitmap display. The workstation also has conversion hardware for audio input and output, using external speakers or headset, and a microphone or stereo microphone pair. The workstation also has audio DSP hardware capable of mixing and sample rate conversion.

The multimedia workstation is part of a distributed system in which the other components (networks and file servers) also have the capability of handling continuous media (audio and video) in digital form.

There is a need for a software framework that allows these capabilities to be fully exploited. We say that a system provides *integrated digital continuous media* (IDCM) if it has the following properties:

- Applications can be distributed, allowing continuous media data to be shipped in real time through digital communication networks.
- The software framework includes the essential elements of existing distributed environments: network-transparent window systems, network file access, naming and authentication, and so on.
- All system components can be shared concurrently by clients using continuous media: a user can run multiple applications, a file server can source or sink multiple streams of data, and the network can carry multiple streams.

This paper describes Abstractions for Continuous Media (ACME), a proposed set of extensions to a network transparent window system for handling CM. Such an extended window system (or "I/O server") is a component of a proposed IDCM software framework [1]. ACME is intended to be implemented as an extension to a standard network window system such as X or NeWS [15, 17]. The proposed software organization for such an "ACME server" consists of 1) a modified version of the base window system; 2) a "window system extension" that handles ACME-specific client requests and replies; 3) an ACME kernel that implements the ACME abstractions in a window-system-independent way (see Figure 1). The details of the ACME kernel and its interfaces to the WS and WSE are described in [8]. A discussion of the use of ACME to implement a variety of applications is given in [18].

The ACME abstractions are presented to clients via a set of request, reply, and event messages that extend the base window system protocol. In this document we are not concerned with the exact form of these messages, referring to them simply as "ACME requests", "ACME replies", and "ACME events".

The paper is organized as follows. Section 2 lists the goals of the ACME design, and Section 3 justifies including CM in a window system. Sections 4 and 5 describe the basic ACME abstractions and their semantics. Sections 6 and 7 describe the details of audio and video I/O respectively. Section 8 discusses related work, and Section 9 is the conclusion.

## 2. DESIGN GOALS OF ACME

We now enumerate and discuss some design goals for the ACME design. Some of these goals are similar to those of window systems; others are particular to CM.

**Compatibility with standard window systems.** ACME is intended to be implemented as an extension of an existing window system such as X11 or NeWS. This extension should be

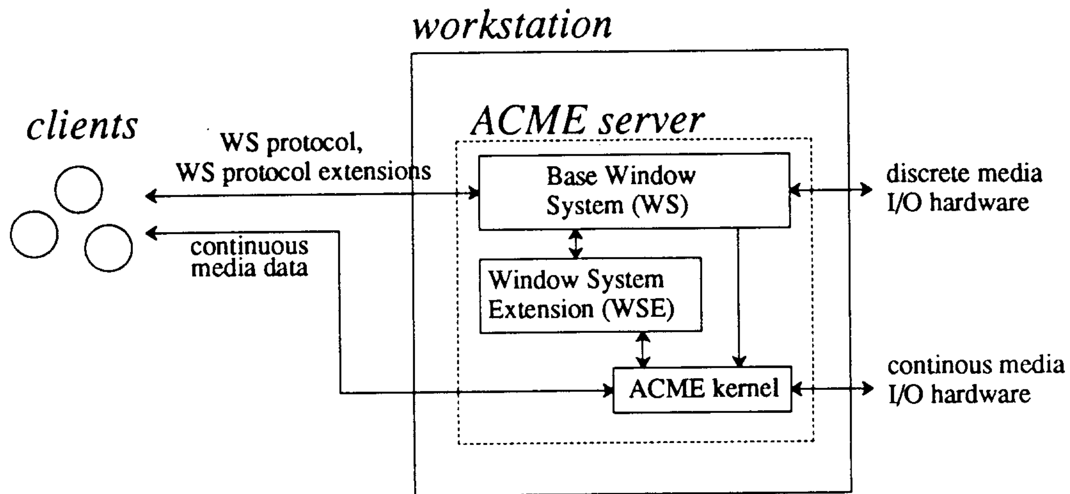


**Figure 1:** An ACME server consists of a base window system such as X11 or NeWS, a window system extension (WSE), and the ACME kernel.

backwards compatible, allowing existing applications to be used and to coexist with applications that use CM.

**Window system independence.** It must be possible to add ACME to a wide range of window systems, including at least NeWS and X11.

**Concurrency.** ACME should support multiple concurrent clients, and multiple I/O streams per client.

**Network transparency.** The source of CM data to be displayed, or the sink of input CM data, should not be constrained to be on the server machine. This can be accomplished by using network communication channels to convey CM data between client and server.

**Transport Protocol Independence.** The network connections used for CM data should not be restricted to a particular transport protocol.

**WS Protocol Independence.** CM data should not be required to be packaged as messages of the underlying window system. In many cases there is a "native" format for CM data (*e.g.*, DVI file or CD sample stream). It would be costly and nonproductive to repackage this data as window system messages, and would place an unnecessary requirement on sources and sinks of CM data.

**Data Representation Independence.** Because there are many different CM data representations (*i.e.*, encodings) in current use, the ACME server should not mandate a particular representation of CM data. Instead, an extensible range of representations should be defined.

**Support for Interleaving.** Temporally synchronized CM strands are often interleaved onto a single data stream for storage and/or transmission purposes. The ACME server should support I/O directly in these interleaved formats, so that the client does not have to independently separate or merge the strands.

**Synchronization.** There must be some method of allowing the client (or multiple clients) to display a number of CM data streams in a synchronized fashion. The streams must be able to be

started synchronously, and must be kept in lockstep by the ACME server.

**Management Functionality.** Modern window systems allow a *window manager* to supervise the layout and manipulation of windows. They provide mechanisms to support window managers, but do not dictate any particular management policy. These mechanisms should be extended to CM, allowing the window manager to control attributes of CM I/O (volume levels, input direction, video properties). For example, possible management policies for audio output include 1) play only the output of the application that is the input focus; 2) play all output but attenuate the output of occluded windows; 3) play only the output of the window containing the mouse cursor. The window manager should be able to implement any of these.

**Support for Resource Reservation.** CM data has associated performance (throughput, delay) requirements. To reliably satisfy such requirements, the overall system design may allow clients to *reserve* part of the capacity of the hardware resources (network, CPU, DSP, *etc.*) that handle CM data [3,6]. An ACME server would govern critical resources (CPU and DSP in the server machine). Earlier work in the ACME project has defined a *CM-resource model* for describing workload and delay, and for distributed negotiated resource reservation [2]. However, work in this area is at an early stage, and there is no standard method for resource reservation. ACME should, therefore, be able to participate in a resource reservation protocol, but should not be restricted to a particular protocol.

## 3. WHY HANDLE CONTINUOUS MEDIA IN A WINDOW SYSTEM?

Given the design goals described in the previous section, we believe that ACME should be implemented as part of a network window system, rather than in a separate server or library. A pragmatic argument for this position is that a separate server would have to duplicate many of the functions (connection management, multiplexing of client requests, event reporting) of a window system. The implementation of these functions in modern window systems such as X and NeWS is the result of considerable thought and effort.

Other reasons for handling CM in a window system are discussed in the following sections. In summary, we argue that the role of "window server" should be expanded to that of "I/O server". Such an IDCM I/O server provides graphical, textual, video and audio output capabilities and screen and speaker management functions to clients, and provides a method of distributing keyboard, mouse, microphone, and video camera input to interested clients. In this way, the client application that makes use of many different media is presented with a fairly uniform I/O interface across those different media.

### 3.1. Spatial Coordination of Discrete and Continuous Media

A CM application may generate concurrent and intermixed output of CM data, graphics and text. It must be possible to freely intermingle these media types on a single screen. Given display hardware with the ability to display[1] multiple streams of CM data concurrently, it would be possible to write a CM I/O server disjoint from the window system. However, in order to change dynamically the areas of the screen in which video is being displayed, and to support clipped video output to a tree-structured hierarchy of "video windows", the CM I/O server would have to duplicate much of the software already present in the window system.

Moreover, if graphics and text reside in the same framebuffer as the video output, the window system and the video display system software must interact to resolve screen area conflicts between overlapping video windows and graphics/text windows. If overlay capability is desired,

---

[1] By "display" we mean whatever method of presentation is appropriate for the medium. For audio this would be output to a speaker; for video, output to (an area of) the workstation screen.

the situation becomes even more complex: the two servers must exchange arbitrarily shaped screen areas (for "chroma-key" type overlays) or blendmaps (alpha-channel overlays for translucence). For efficiency, it is clear that at least shared memory is needed.

### 3.2. Temporal Coordination of Discrete and Continuous Media

Some applications must synchronize text and/or graphics output with video and/or audio output. For instance, consider an application that associates textual subtitles with given ranges of frame numbers in a movie, and displays the subtitles as the movie is viewed. This type of synchronization is best provided by an integrated I/O server which can accept requests of the type: "When CM stream X reaches time T, perform event R".

Synchronization of CM I/O with discrete I/O is also important for input. Consider an application that uses speech recognition to respond to spoken user requests. Such an application might want to use the keyboard modifier keys, the mouse position, or the mouse buttons to modify the meaning of the user request. For example, a window manager might respond to the verbal command "move" by allowing the user to move the window being pointed to by the mouse. The single-server approach simplifies event serialization and timestamping.

### 3.3. Integration of Management Functions

Standard window systems supply management functions such as window moving, resizing, mapping ("opening"), and unmapping ("closing"). In addition, they provide mechanisms for directing keyboard and mouse input to interested client(s). We believe that users and applications will want to manage video images in the same way as graphic windows. By providing integrated management functions for windows and video images, the complexity of both client and server software will be reduced.

Turning to input management, it may be desirable to implement CM input management policies that are based on the state of the window system. For instance, one simple management policy for audio input, paralleling a common keyboard input management policy, would be as follows: If the pointer is in a top level window, audio input is directed to the client which owns that window; otherwise, audio input is directed to the window manager. This is done most easily in the single-server approach.

### 3.4. Input/Output of Combined CM Streams

Audio and video data that are intended to be temporally synchronized are most often interleaved in a single byte stream. If audio and video output were handled by separate servers, then it would be necessary for the client to separate the audio from the video and send them on separate connections. It is simpler and more efficient to have the client send the interleaved stream to a single server. A similar argument applies to input.

## 4. BASIC ABSTRACTIONS

This section describes the basic ACME abstractions, which are summarized in the following table.

| Name | Brief Description | Created Dynamically |
| --- | --- | --- |
| CM Connection | Transport-level connection for CM data | Yes |
| Strand type | Encoding of single-track CM data on byte stream | No |
| Rope type | Interleaved encoding of multiple strands | No |
| PDev | Physical CM I/O device | No |
| LDev | Logical CM I/O device | Yes |
| CLDev | Set of LDevs as source or sink of rope | Yes |
| LTS | Time system for synchronizing I/O streams | Yes |

Instances of these abstract types have IDs that are globally unique (strand and rope types) or unique to a particular server. The mechanism for creating and manipulating instances is WS dependent. Instances may have *attributes* that can be read and, in most cases, written.

### 4.1. Continuous Media Connections

A *continuous media connection* is a transport-level connection that conveys CM data in a "native" (WS-independent) format. The CM connection establishment mechanism is part of the WS extension. This extension must at least passively accept connections; it may also provide operations for actively establishing connections.

For CM data, the features required of transport protocols often differ from those of discrete media. CM transport protocols need not be reliable if absolute encodings are used, since small errors will not, in general, affect the perceived quality of the data (this is discussed further below). Further, if the sending rate is limited, the network connection provides throughput and delay bounds, and the receive processing has performance guarantees, then CM transport protocols may not need to provide flow control, CM transport protocols must support at least sequenced delivery.

An ACME server can support any set of transport protocols (the current implementation supports only TCP). If more than one transport protocol is supported by an ACME server, clients must be able to determine whether each protocol is reliable, since this may determine what CM data representations can be used.

### 4.2. Strands and Ropes

A *strand* is a single "track" of CM data encoded on a byte stream (*e.g.*, a monaural or stereo audio stream, or a video track). A *strand format* is a specification of this encoding. Audio and video have separate sets of strand formats. Each set of strand formats can be divided into *classes* where each class corresponds to an algorithmically distinct encoding method. Classes may have parameters. A strand format is a combination of a class and a set of parameter values.

Several strands of CM data may be linked in a temporally "parallel" manner. That is, the strands are intended to be played in synchrony with each other, whether or not all are used simultaneously. Examples include:

- One video strand with one audio strand (*e.g.*, monaural "TV program" or "movie").
- More than one audio strand with one video strand (*e.g.*, stereo or bilingual program).
- Many audio strands (*e.g.*, 24 track audio studio master tape).
- Many audio and video strands (*e.g.*, audio/video postproduction clips).
- Many video strands (*e.g.*, different, simultaneous views of the same real world scene).

Strands associated in this manner are often stored and transmitted on a single byte stream; sections of the strands are interleaved onto this stream. This allows several parallel strands to coexist in a single file and to be transmitted over the same network connection. The sequential nature of the storage or transmission medium guarantees that the ordering of the interleaved

chunks is preserved over storage, retrieval, and transmission. The data rates of the individual strands are also matched over the long term. These properties simplify synchronization of the strands.

In ACME, such a combination of strands is called a *rope*. The method of combination is called a *rope format*. An example of a rope format is DVI AVSS format [11]. A rope format says nothing about the formats of the individual strands: A given rope format can be used with different types of strands, and conversely the same set of strands could be combined with different rope formats. Each ACME server can support any set of rope formats. The set of supported rope formats may be queried by clients.

Strand and rope formats can be partitioned into two classes: *error tolerant* and *error intolerant*. A error tolerant strand has the property that, if the strand is being displayed and data errors occur, the display will return to its proper value within finite time. An error intolerant strand is one that never recovers from data errors.

Error tolerant strand and rope formats have an *error intolerance period*: the maximum length of time following a data error during which the display is perceptibly different than what it would be had the error not occurred. For example, absolute encoded NTSC video has an error intolerance period of 1/30 of a second, and differentially encoded DVI video with absolutely encoded keyframes inserted every fifteenth frame has an error intolerance period of 1/2 second. Differentially encoded video with no keyframes except the first is error intolerant. An error intolerant strand can be thought of as having an infinite error intolerance period.

Some combinations of transport protocol and strand or rope format (*e.g.*, an unreliable transport protocol and an error intolerant strand format) are unacceptable to some clients. Clients must decide what data format(s) and transport protocols to use, based on their needs.

### 4.3. Physical Devices (PDevs)

A *physical CM device* (PDev) is a CM hardware device accessed via the ACME server. A PDev is either a video output device (video-capable display), a video input device (video camera), an audio output device (speaker), or audio input device (microphone). Each PDev has a *physical scope* indicating roughly what area of physical space is served by the PDev. The values for physical scope include the following.

| *value* | *example* |
|---|---|
| `personal` | a headset or head-mounted display |
| `workstation` | a camera mounted on the workstation |
| `room` | a wall-mounted loudspeaker |
| `mobile` | a hand-held camera |

When a stereo speaker pair or microphone pair is most easily accessed as a single entity (*i.e.*, if it sources or sinks an interleaved sample stream) then it is represented as a single PDev. If hardware supports it, the separate channels of a stereo PDev may also be available as monaural PDevs.

Client programs may query the ACME server for the devices supported and their characteristics. The query returns a list with one entry per PDev. The entry for a PDev includes the PDev ID and a list of strand formats that the PDev is capable of sourcing or sinking. In some cases the server will have to do work in software or DSP (*e.g.*, sample rate conversion) to handle a strand format. The list of strand formats for a given PDev is ordered by the amount of work needed for such conversion.

### 4.4. Logical Devices (LDevs)

A *logical continuous media device* (LDev) represents a virtual CM I/O device. There are several *classes* of LDevs: *players* and *listeners* for digital audio (see Section 6), and *VWins* and *VCams* for digital video (see Section 7). Players and listeners may be either stereo or monaural.

Multiple LDevs may be multiplexed onto a single PDev. The nature of this multiplexing depends on the LDev class. VWins are like windows; they spatially share a display. If more than one player is mapped to a physical speaker, the outputs are mixed (see Section 6) and played together. On input, if more than one LDev (VCam or listener) share a PDev, separate copies of the input (perhaps in different encodings) are sent on CM connections associated with each of the LDevs.

Each LDev has associated with it a set of attributes. LDev attributes are of two types: *generic* attributes (associated with every LDev) and *specific* attributes that depend on the class of the LDev. Specific attributes vary with the class of the LDev (see Sections 6 and 7). The generic attributes of an LDev are:

- *PDev*: The physical device to which the LDev is to be mapped. This attribute cannot be changed.
- *Strand format*: The representation of the data to be input from or output to the LDev. This must be a format supported by the PDev.

The LDevs of input type (VCam and Listener) have a *local echo device* attribute. This attribute, if non-null, is the ID of an output LDev with the same strand format. Data gathered from the input LDev is echoed directly to the local echo device, internally to the server. This feature can be used to provide ''side tone'' in a headset with microphone, local echo of video input, *etc*. The output LDev must not be a member of any CLDev (see Section 4.5).

The generic operations *create, destroy, map* and *unmap* can be performed on LDevs. *Map* and *unmap* are used to start and stop I/O to or from the corresponding PDev. They return the logical time at which the transition actually occurs (see Section 4.6). When an input LDev is not mapped, no data is sent to the corresponding CM connection or local echo LDev. When a player is not mapped, the output level returns slowly to zero, avoiding clicks. When a VWin is not mapped, its visible area continues to display the last complete frame; the window system may overwrite this area if desired. A *refresh* operation can be performed on unmapped VWins, causing the last displayed frame to be redrawn.

Some strand formats embed synchronization points in the data. For these formats, an event is sent to the client when such a point is encountered during playback of the strand data. The event includes the time at which the synchronization point was encountered (see Section 4.6), and optionally some event-specific data, which can be interpreted by the client.

### 4.5. Composite Logical Devices (CLDevs)

A composite LDev (CLDev) is an abstraction for the source or sink of data on a particular CM connection. Each CLDev has an ordered list of constituent LDevs, and is associated with a CM connection. If the CM connection carries a rope, each constituent LDev sources or sinks a strand of the rope. If the CM data is a strand, the list contains only a single LDev. There are two classes of CLDevs: *Input* and *Output*. If the type of the CLDev is *output*, data received on the CM connection is treated as output to be rendered on the constituent LDevs. Conversely, if the type of the CLDev is *input*, any output generated by the constituent LDevs will be combined into the appropriate rope format and sent over the connection attached to the CLDev.

The generic operations on CLDevs include *create, destroy, map, unmap,* and *clear*. The *create* operation takes the class of the CLDev and a list of constituent LDevs. The *destroy* operation deletes the CLDev; it unmaps the constituent LDevs but does not destroy them. The *map*

operation takes a bitmask specifying the map state of the constituent LDevs. I/O begins simultaneously for all mapped LDevs, and the logical time at which the map operation took effect (see Section 5) is returned. The *unmap* operation unmaps all constituent LDevs. The *clear* operation flushes all buffered data for an output CLDev. An LDev can only be part of a single CLDev at any time. An LDev that is part of an existing CLDev cannot be destroyed. The attributes of a CLDev include:

- *Rope format*: The format of data to or from the CLDev. Each LDev in the CLDev is associated with the corresponding strand in the rope.
- *CM connection*: The CM connection over which data of the rope is sent or received.
- *Logical time system*: Each CLDev is associated with a logical time system (see Section 4.6).

Like strand formats, rope formats may contain embedded synchronization points. An event is sent to the client when such a point is encountered during playback of the strand data.

### 4.6. Logical Time Systems

An LTS represents a temporal coordinate system in which the timing of I/O is expressed. An LTS is created by a client, and is *started* at some later time. The rate of progress of an LTS may be determined by the I/O devices, rather than by a realtime clock. Each CLDev is associated with exactly one LTS, and all timing information referring to a CLDev is expressed relative to the start of its LTS. The LTS provides a unified framework for the following functions:

**Output synchronization.** The strands of a rope are automatically synchronized by the ACME server. To synchronize output of a set of ropes, a client associates their CLDevs with a single LTS. The client notifies the ACME server that it is ready for the server to start the LTS, and begins sending data on the CM connections. When all CLDevs have received enough data to prevent starvation, and when all LDevs are ready to display data, the servers starts display of all the LDevs simultaneously. An event containing the start time (in real-time units) of the LTS is sent to the client when the LTS is started.

**Input synchronization.** An LTS can also be used to synchronize input. Input data sent on CLDevs in the same LTS are assigned equal timestamps for input gathered at the same moment. (Timestamps need not be explicit: The timestamp of a sample in an audio data stream, for example, can be inferred by the sample index.) When the client starts the LTS, data collection from the various input LDevs is started. An event containing the start time of the LTS is sent to the client when the LTS is started.

**Relative timing of discrete input events.** Clients (or the WSE) can query the current value of an LTS. A WSE could use this to generate input events with LTS time-tags. For example, a video postproduction system might allow the user to set "marks" in a video clip by clicking a mouse while it is playing. This feature would allow the application to easily obtain the time position of a mouse click within the video clip.

**Timed notification.** A client can request an event when an LTS reaches a certain value.

**Deferred requests.** A client can schedule ACME requests (or base window system requests) for deferred execution at some value of an LTS. The time synchronization of this type of future scheduling is tighter than that provided by the alarm clock, since no network delay is incurred.

The last two features might be used for a subtitling application: A video sequence is displayed by a CLDev, and the client sends WS requests to display the subtitles. The client reads all the subtitles and their timestamps from a file, and schedules them to be displayed at the appropriate future times. Suppose, however, that the material to be subtitled is very long, and that the client does not want to immediately read in all the subtitles and schedule them for display. The client could instead read in the first five minutes' worth of subtitles and schedule

them, and then request an alarm event four minutes hence. When the alarm event is received, the client would read and schedule the next five minutes of subtitles, schedule another alarm, *etc*.

### 4.7. Examples of Continuous Media I/O

In a simple scenario (see Figure 2), a client plays a monaural audio stream through a speaker on the ACME server. To do this, it first establishes a CM connection over which the audio data is to be sent. It creates a *player* LDev, associates a PDev with the LDev, and creates an LTS. It creates a CLDev with the appropriate strand, CM connection, and LTS attributes. The CLDev includes only the player LDev. The client starts the LTS, maps the CLDev, and begins sending data over the CM connection. The ACME server converts this data to sound and plays it through the speaker.

In a slightly more complex case (see Figure 3), CM data is shipped to the ACME server from a remote "third party", in this case a file server, which need not speak the WS protocol. This is done to eliminate data handling overhead in the client and to reduce overall delay. The client establishes a CM connection from the file server to the ACME server. The rest of the steps are as above. Finally, the client issues a command to the file server to start sending the data on

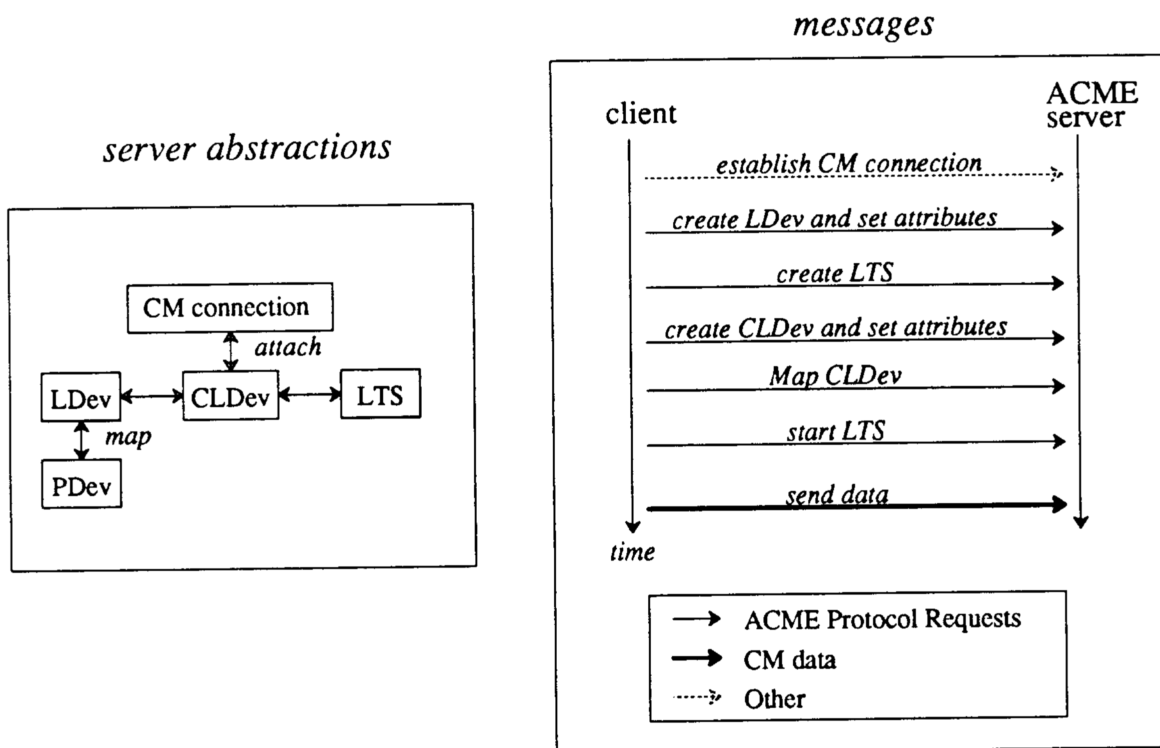


**Figure 2:** To output a stream of CM data, the client must 1) create a CM connection, a Logical Device (LDev), a Logical Time System (LTS), and a Compound Logical Device (CLDev); 2) map the LDev to a Physical Device (PDev); and 3) send data on the CM connection.

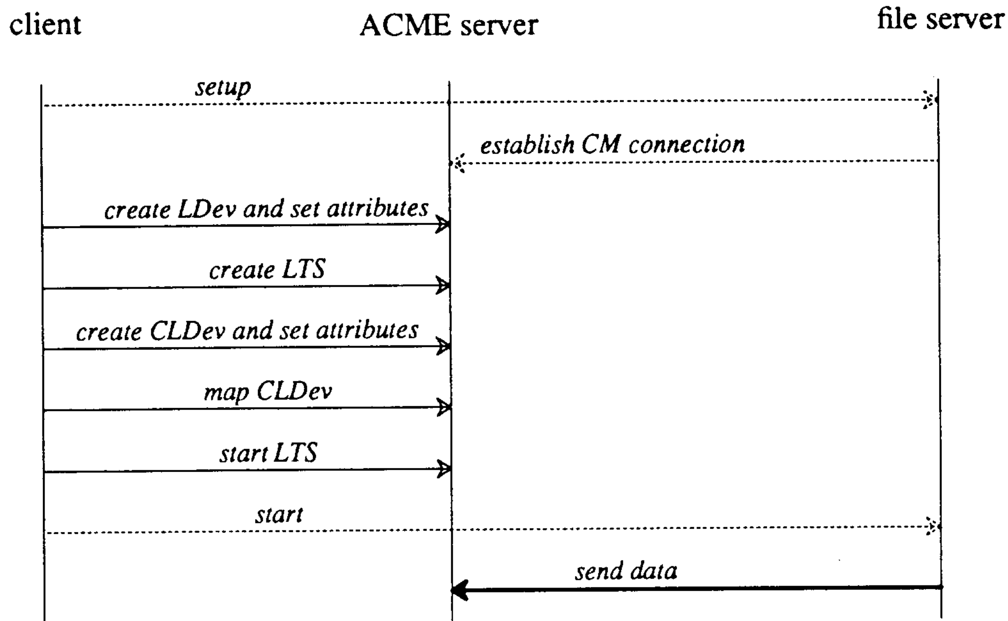


**Figure 3:** A client can arrange for CM data to be sent to an ACME server from a "third party", such as a file server, that need not speak the window system protocol.

the CM connection.

In the example shown in Figure 4, the client wants to output a combination of continuous media streams, say video with stereo audio. This case is similar to the first example, except that the client creates three LDevs (two for audio, one for video), and the CLDev consists of all three LDevs. In the first example no rope format was specified (since only one LDev was used), but in this case the client must specify a rope format when creating the CLDev. When data is received over the CM connection, it is disassembled into its component strands, which are displayed on their corresponding output devices.

Figure 5 shows an example in which several strands, each coming from a different source, are played synchronously. This is done by creating CLDevs to handle the different strands , and associating each of the CLDevs with the a common LTS. The ACME server ensures that the strands are synchronized as explained above.

### 4.8. Resource Reservation

As mentioned in Section 2, it may be desirable for ACME to participate in a resource reservation protocol. Such a protocol would allow CM clients to reserve the resources (*i.e.*, network bandwidth, CPU time, DSP time, *etc.*) needed to perform their tasks. ACME servers may support one or more such protocols. If a reservation protocol is present, the client can interpose a resource reservation request between the creation request for a CLDev and the actual use of the CLDev. This resource reservation request simply informs the server that the client has set up whatever CLDevs it needs and that the server should now make appropriate resource reservations to meet the performance requirements of the LDevs and CLDevs. The exact form of this request

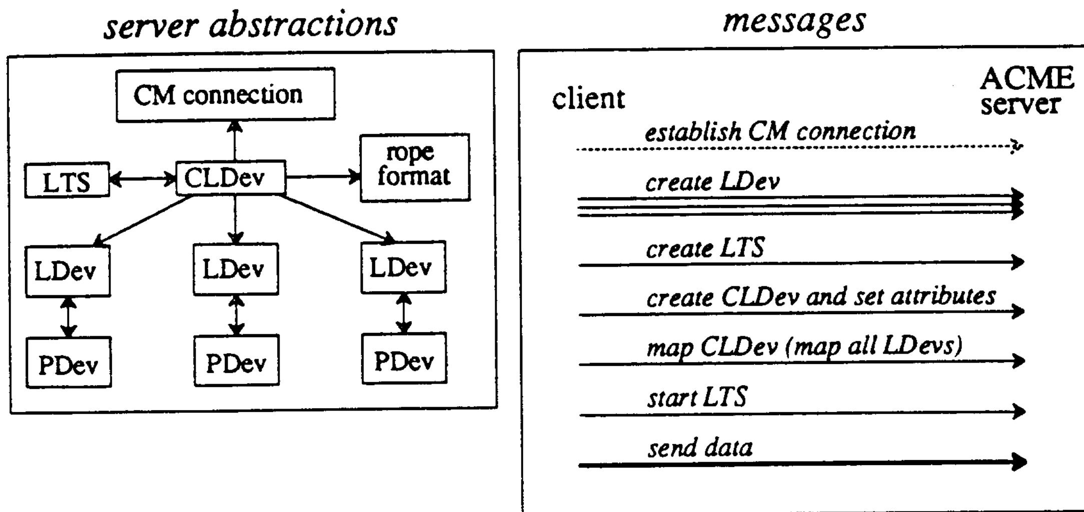


**Figure 4:** A "rope" interleaves several CM data streams. When a rope is sent to a Compound Logical Device (CLDev), it is separated into strands, which are played synchronously on several PDevs.

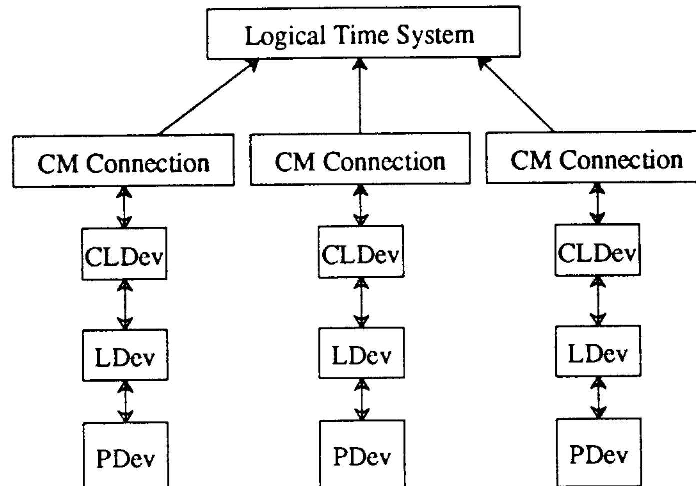


**Figure 5:** A Logical Time System (LTS) can be used to synchronize playback of multiple streams coming from different sources.

will vary with the resource reservation protocol being used.

As an example, the Session Reservation Protocol (SRP) [3] is a resource reservation protocol suitable for making these kinds of reservations. SRP is based on the CM-resource model [2]. In this model, each *resource* (CPU, DSP, network, *etc.*) is represented by a *resource manager*. The manager allows *sessions* to be established, each representing a reservation of part of the capacity of the resource. The session request specifies the workload of the session. Session requests may be granted or refused. End-to-end sessions (providing end-to-end throughput and delay guarantees) may be established using a two-phase resource reservation protocol.

To let ACME clients use SRP to reserve server resources, it suffices to introduce two new requests: One to establish input sessions and another to establish output sessions. To establish an output session the client sends a session ID and a CM connection ID. When the server receives a session reservation request, it reserves local resources based on the processing and device requirements of the LDevs associated with the CM connection. To establish an input session, the client sends a CM connection ID and a destination address. The server, upon receipt of this request, reserves local resources, allocates an SRP session ID, and initiates establishment of an end-to-end session to the remote host (the sink of the CM data).

## 5. TEMPORAL SEMANTICS OF CM INPUT AND OUTPUT

The semantics of CM I/O are necessarily more complex than those of discrete media. We must specify not only the order in which events occur, but also the times and rates at which they occur. Unlike discrete I/O, CM data may legitimately be discarded by the server, and the server semantics must specify when this may happen. In this section we give an informal description of the semantics of an ACME server. We assume that strands have a constant data rate. However, the semantics can be generalized to variable data rate strands, provided the maximum data rate is

bounded[2].

ACME output semantics have two main components: the *timing/discarding semantics* define the time when a given piece of CM data is displayed (if it displayed at all), and when it is discarded. The *mapping semantics* define the time intervals during which display actually occurs. There are corresponding semantics for input as well. The two components of the semantics are independent. The display time for a given piece of data is determined by the timing of the data arriving at the server, not by the map state of the LDev. This is important because the time to service a map or unmap request is nondeterministic. If map state affected display time, display time would also be nondeterministic, making synchronization impossible.

We define a conceptual model of an output CLDev, a diagram of which is shown in Figure 6. (This model is not meant to reflect the implementation of a CLDev, but to illustrate its behavior.) Incoming data from a CM connection traverses a counter and flows into a rope buffer. Data from the rope buffer is split into constituent strands and sent to PDevs. Each PDev has a *display device*, a *bit bucket* (conceptually, a null display device), and an optional *private queue*. The private queue is needed in some cases because, for some differentially encoded strands with non-constant data rates, a certain number of messages must be accumulated before display may begin.

We define the mapping semantics as follows. The map state of a strand is determined by the map and unmap operations on the corresponding CLDev and LDev. These take place in the order requested, but with no timing guarantees. The mapped/unmapped state of each strand determines whether output is directed to the display device or to the bit bucket; it is completely independent of the flow of data on the connection and through the rope buffer.

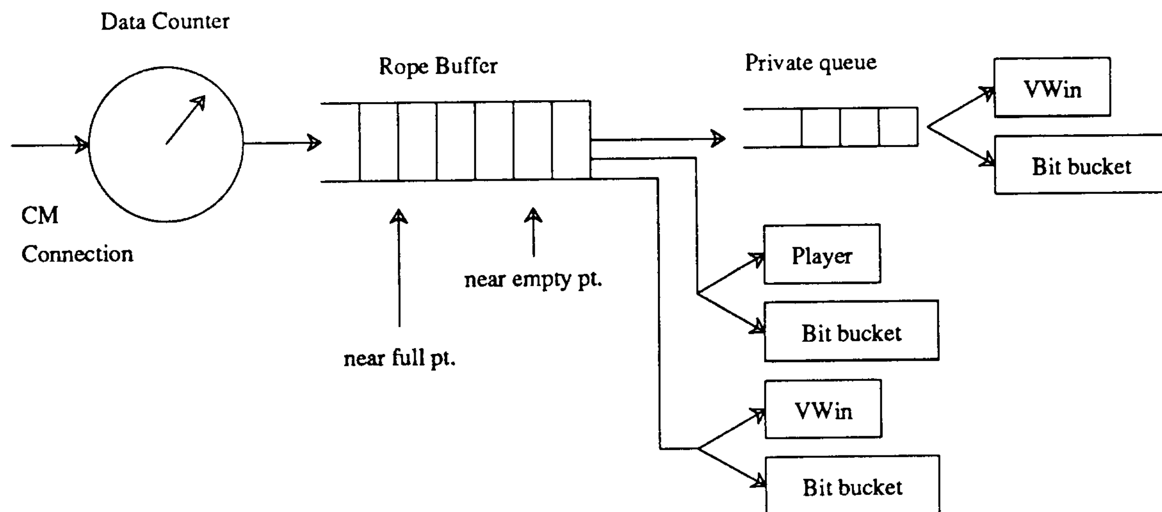


**Figure 6:** To define the semantics of output, an abstract model of the CLDev is used.

[2] In most cases, this can be done by substituting temporal size for physical size.

### 5.1. Flow Control and Data Discarding

Each CLDev has three attributes relating to the rope buffer, specified by the client when the CLDev is created:

- *Buffer Size*: The (conceptual) size of the rope buffer.
- *Near empty point*: If the amount of data in the buffer falls below the near empty point, the server sends the client a `CLDevStartData` event indicating the possibility of buffer underflow. For most applications, the client should set the near empty point so that it can ensure that the flow of data is started before the buffer actually underflows.
- *Near full point*: If the data in the buffer goes above the near full point, the server sends the client a `CLDevStopData` event indicating the possibility of buffer overflow. For most applications, the client should set the near full point low enough to ensure that the flow of data can be stopped before the buffer actually overflows.

If the rope buffer actually does overflow or underflow, an event is sent to the client informing it of the rope buffer status. For a given burstiness[3] and data rate, there is a minimum buffer size to ensure that packets are not lost due to buffer overrun. If the client-supplied buffer size is smaller than this limit, the server transparently pads the buffer to this size, and the near full point is adjusted upward accordingly.

#### 5.1.1. Timing Semantics

An output CLDev has two states: `Ready` and `NotReady`. A CLDev is `Ready` if and only if it is ready to immediately display synchronized output on all its LDevs. This is true when the following two conditions are met:

- Enough data has arrived on the CM connection to guarantee that the CLDev will not starve for data at some future time. The amount of data required, *display_start_amount*, depends on the parameters of the CM connection[4].
- All PDevs have done any initial processing required to fill their private queues.

A CLDev is initially in the `NotReady` state, the PDev private queues are empty, and the data counter is at zero. When data arrives on the CM connection, the data counter is incremented accordingly, data is placed in the rope buffer, and the PDevs start filling their private queues. As soon as `display_start_amount` of data has been counted and all PDevs associated with the CLDev have filled their private queues, the CLDev makes the transition to the `Ready` state.

Data is then displayed or discarded (depending on the map state of the LDevs) by the PDevs in earliest-first order at a rate determined by the characteristics of the PDevs. If the rope buffer overflows, data is discarded in earliest-first order. If the rope buffer underflows, device specific action takes place: For video output, the last frame displayed may be repeated; for audio output, zero-valued samples may be output.

The private data queues are non-shrinking: They are initially empty, and fill up as data arrives in the rope buffer. They do not shrink if the buffer underflows after the CLDev has reached the `Ready` state. This is so that output of all PDevs will remain synchronized when the rope buffer recovers from the underflow condition,

---

[3] We use the term loosely. The CM-resource model [2] has a *workahead limit* parameter that captures our intended meaning.

[4] The CM-resource model makes such non-starvation guarantees possible [2].

A transition from `Ready` state to `NotReady` state takes place if the client sends a request to *clear* the buffer associated with the CLDev. This request instantaneously resets the data counter to zero, discards all data in the rope buffer and the private data queues, and stops display on all PDevs.

The semantics for input CLDevs are simpler. Each input PDev has an optional fixed-length private queue. Conceptually, the private queue starts filling up when the server is started. An input CLDev transits from the `NotReady` to the `Ready` state when all the private queues of PDevs associated with the CLDev are full. When the CLDev is mapped and `Ready`, the individual strands are combined into a rope and placed in the rope buffer. The server simultaneously starts sending data from the rope buffer over the CM connection. If the CM connection is flow-controlled and the client does not read the CM data sent on the connection, the rope buffer may overflow. When the near-full point is reached, the server sends a `CLDevReadData` event. If the rope buffer overflows, data is discarded in earliest-first order. When the unmap request is received, the server stops sending data on the CM connection and stops reading data into the rope buffer. The near empty point attribute has no significance for input CLDevs.

### 5.2. Logical Time Systems Semantics

The timing semantics for output CLDevs can be extended to multiple output CLDevs mapped onto the same LTS by modifying the condition for the transition from `NotReady` to `Ready` for each CLDev as follows. All CLDevs transit to the `Ready` state only when all their data counters have counted their respective *display_start_amount* of data, and all their associated PDevs are ready to display. This ensures that all the CLDevs are synchronized initially.

However, if one of the CLDevs receives a clear request, synchronization is lost. This is also true if at some point data is lost due to buffer overflow or underflow. In the current design, we do not attempt to define mechanisms for re-synchronizing such streams. Such mechanisms may be provided in the future in any of a number of ways. It may be possible to define more control operations on LTSs. In particular, we might allow a client to stop and restart an LTS. Thus, to resynchronize after clearing a buffer, the client would stop the LTS and then restart the LTS. Restarting the LTS would involve waiting for all CLDevs to become `Ready`, automatically synchronizing them.

The same effect could be achieved by changing the semantics of *clear*. For instance, when the buffer of one of the CLDevs is cleared, so is that of all other CLDevs that are mapped to the same LTS. Alternatively, if the buffer of one of the CLDevs is cleared, it does not return to the `Ready` state until all other CLDevs have been cleared also.

## 6. AUDIO FEATURES OF ACME

The set of audio strand types in our current design includes uncompressed encodings with constant sampling rate and constant bits per sample. Four parameters suffice to characterize audio strands in this class:

*sampling rate (samples per second)*
*bits per sample*
*number of channels (it is assumed that samples from different channels are interleaved)*
*encoding technique (PCM, μ-law, A-law, differential PCM, etc.)*

There are two classes of audio LDevs: *players* (abstract speakers) and *listeners* (abstract microphones). The attributes of an audio LDev include a 2x2 matrix representing volume control for players and recording level for listeners. The matrix encodes a linear transformation from stereo input to stereo output (see Figure 7). If input is monaural, it is duplicated into both input channels. If output is monaural, the two output channels are averaged. The identity matrix can be used as a default since it will work reasonably for any combination of stereo and mono.

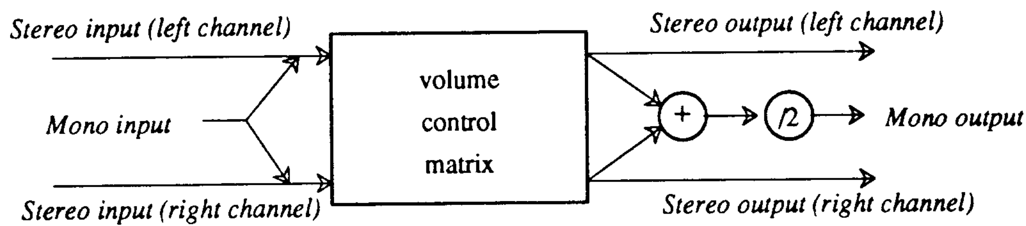


**Figure 7:** The attributes of an audio LDev include a 2x2 volume control matrix mapping stereo input to stereo output.

Clients can use other values to vary the sound-stage width and the perceived position. Attributes for other effects such as distancing, peaking, muffling and thinning [10] may be added in a later version of the design.

## 7. VIDEO FEATURES OF ACME

### 7.1. Video Strands

Video strand classes include px64K, MPEG, DVI SVM (special video mode, 9bpp), and DVI mono. Each class has an associated set of parameters. For example, DVI SVM is parameterized by image resolution and total average data rate.

The *native image* associated with an active video strand is the video image that could be decoded directly from the strand. The *native size* is the X and Y sizes, in pixels, of the native image.

### 7.2. Video Windows (VWins)

A VWin is a window that displays video from its associated CM connection instead of servicing graphics requests. A VWin has a size, position, and any other window attributes defined by the base window system. In addition, it has some VWin-specific attributes. These include the generic LDev attributes: *strand* and *PDev*, as well as the following attributes specific to VWins[5]:

- *Xoffset, Yoffset*: the coordinates of the pixel in the native video image that is mapped to the upper left corner of the VWin when it is fully viewable on the workstation screen.
- *Xmagnification, Ymagnification*: rational numbers representing the number of screen (columns, rows) associated with each native video (column, row). If either is not equal to one, the server must resample the native video image. The client can query the server for the allowed magnifications for each strand type.

For example, if a VWin has a size of 250x200 workstation pixels, an X magnification of 5/2, and a Y magnification of 4, then the size of the subrectangle of the native image which is viewable is 100x50 native image pixels, as shown in Figure 8.

[5] A more general set of transformations, such as affine or curved mappings, would be desirable for some applications.

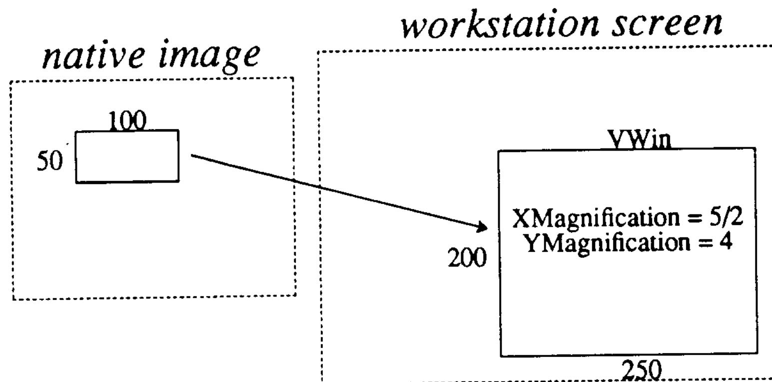


**Figure 8:** When a video stream is shown in a VWin, any subrectangle of the "native image" may actually be displayed, and it may be scaled independently in the X and Y directions.

In an alternative design, the client specifies the subrectangle of the native image that is to be viewable, and the server fits that subrectangle to the VWin. We avoided this approach because the resulting magnification factors might be impossible or inefficient for the server to provide.

The rectangle actually drawn to in a VWin is aligned with the upper left corner of the VWin. Let $W$ denote the X magnification times the native image X size. If $W$ exceeds the VWin X size, display is clipped to the VWin. If $W$ is less than the VWin X size, the right part of the VWin is undefined (see Figure Figure 9). Similarly for the Y direction.

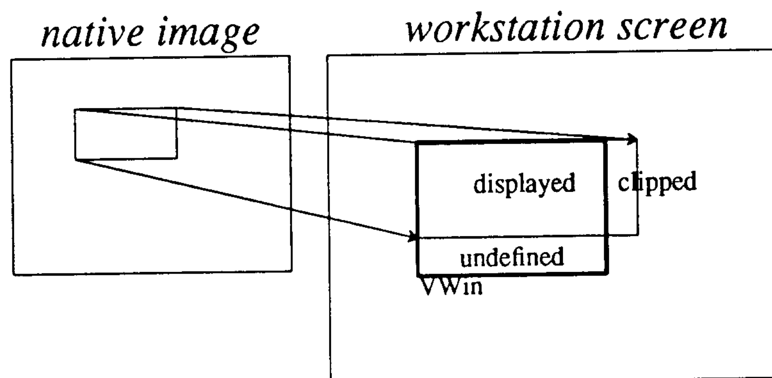


**Figure 9:** In this example, the magnified subset of the native image exceeds the VWin in the X direction, and is clipped. It is smaller than the VWin in the Y direction, so the bottom part of the VWin is undefined.

The values for a VWin's geometrical parameters may involve a negotiation between the client and a window manager. A possible scenario for the creation of a VWin is as follows.

- The client, knowing the native size of the strand and the allowable set of magnifications, gives the window manager a hint for the VWin size.
- The window manager chooses VWin size, perhaps using the client's hint and perhaps using a user "rubber-banding" interaction.
- Given the window manager's decision on VWin size, the client selects the X and Y magnifications and creates the VWin.

### 7.3. VWin Management and Modified Exposure Processing

Window systems typically provide window management requests to support window mapping, unmapping, resizing, moving, and restacking. In an ACME implementation, window system requests of this type work properly on VWins. Therefore, existing window managers do not have to be modified to distinguish between VWins and conventional windows.

Window systems also provide some method for repairing the contents of a window which has become damaged due to obscuration followed by exposure. Usually this is done by providing backing store for the window contents, or by sending an event to the client informing it of the exposed area. In the latter case, the client is expected to repaint the exposed areas. ACME does not modify the base window system's damage repair mechanism, except when a VWin has become damaged.

When a VWin has been damaged, the client is not notified. Instead, if all or part of a VWin is exposed, the server waits for one frame time and then checks to determine whether the next video frame is available. If so, the frame is processed normally, and the damage is repaired by default. If not, the server assumes the client is not currently sending video data to the VWin, and the server repairs the VWin contents from backing store.

### 7.4. Blending and Overlays

ACME provides rudimentary support for overlays via VWin clipmask manipulation. If the base window system or another extension has overlay semantics, these can and should be supported for VWins. However, ACME has no direct support for blending. Commonly available video coprocessors are not yet fast enough to make blending practical. It is possible that this functionality will be added to ACME in the future.

## 8. RELATED WORK

Although several projects have pursued the general goal of adding continuous media to a computer system, their assumptions and approaches differ from those of ACME. Some systems use analog storage and communication of CM data, with computer control of the analog devices. An example is Galatea [12], in which "visual workstations" are connected to a videodisc server by an analog network. The system provides user access to continuous media display functions in a distributed computing environment. In this case, however, the continuous media data is stored and transported in analog form over a separate network.

Several systems provide a server-based architecture for controlling "connections" between CM devices. Examples include the VOX Audio System [4], Pandora [13], IMAL [9], and VEX [5]. These connections may be digital, but in any case data is handled in an external framework, and cannot be accessed in real time by clients. Some of these systems also do not fully address issues of sharing and concurrent access to CM I/O devices, and synchronization of CM streams with discrete media.

Intel's Digital Video Interactive (DVI) [14] is a combination of 1) hardware for capture and display of digital audio and compressed video, and 2) an MS-DOS software environment for this

hardware. While DVI hardware is very flexible, the DVI software environment is limited. It provides a platform for standalone, CD-ROM based interactive multimedia applications. Concurrent applications, network communication of CM data, and application-independent management functions are not supported directly.

## 9. CONCLUSION

Abstractions for Continuous Media (ACME) is a set of abstractions for the input and output of digital continuous media (audio and video). ACME is intended for inclusion in a network window system such as X11 or NeWS. The ACME design fulfills several important goals. It can handle multiple data formats, including interleaved data streams. It supports synchronized output from separate streams. It contains provisions for a resource reservation protocol, used to allocate and schedule hardware resources to provide the performance guarantees needed by continuous media.

As of the writing of this report (August 1990), a prototype ACME kernel, supporting audio only, has been developed, and is being used for experiments. This ACME kernel is being integrated with NeWS and X11 window systems. An ACME server is only one component of a software system for continuous media. Other necessary components include the following:

- Facilities for mass storage of continuous-media data. It may be possible to use existing UNIX-type file systems. For more ambitious goals, such as large-scale distributed hypermedia, it may be necessary or desirable to start from scratch. Existing work in this area includes the Sun multimedia file server project [16] as well as various standards for CD-based storage [7].
- User-interface toolkits for continuous media. Applications will increase in complexity both in their internal structure (*e.g.*, because of the use of multiple concurrent processes to handle CM data streams) and in their interface (because of concurrency and mixed-mode interactions). Toolkits, interface specification languages, or other approaches will be useful in managing this complexity.
- Basic applications. These include editors for continuous media, extensions to mail and document preparation systems, and two-way conversation systems.

Ultimately, the uses of distributed CM systems will extend beyond the client/server paradigm. In particular, it will be necessary to support multi-user CM applications, such as n-way audio/video conferencing and collaboration on CM applications. This raises a host of issues and problems beyond those discussed here.

## ACKNOWLEDGEMENTS

Kyoji Umemura, Y.K. Hui, and Ralf Guido Herrtwich participated in the ACME project. Greg McLaughlin of Sun Microsystems provided valuable support and input.

## REFERENCES

1. D. P. Anderson, R. Govindan, G. Homsy and R. Wahbe, ''Integrated Digital Continuous Media: a Framework Based on Mach, X11, and TCP/IP'', Technical Report No. UCB/CSD 90/566, Mar. 1990.
2. D. Anderson, ''Meta-Scheduling for Distributed Continuous Media'', UCB Technical Report, Aug. 1990.
3. D. P. Anderson, R. G. Herrtwich and C. Schaefer, ''SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet'', Technical Report 90-006, International Computer Science Institute, Feb. 1990.
4. B. Arons, C. Binding, K. Lantz and C. Schmandt, ''The VOX Audio Server'', *Multimedia '89: 2nd IEEE COMSOC International Multimedia Communications Workshop*, Ottawa, Ontario, April 20-23, 1989.
5. T. Brunhoff, *VEX: Video Extension to X, Version 5.5*, Tektronix, Inc., 1989.
6. S. Casner, K. Seo, W. Edmond and C. Topolcic, ''N-Way Conferencing with Packet Video'', *Third International Workshop on Packet Video*, Morristown, NJ, March 22-23, 1990.
7. K. A. Frenkel, ''The Next Generation of Interactive Technologies'', *Comm. of the ACM 32*, 7 (July 1989), 872-881.
8. G. Homsy, R. Govindan and D. P. Anderson, ''Implementation Issues for a Network Audio/Video Server'', In preparation, July 1990.
9. L. F. Ludwig and D. F. Dunn, ''Laboratory for Emulation and Study of Integrated and Coordinated Media Communication'', *Proc. of ACM SIGCOMM 87*, Stowe, Vermont, Aug. 1987, 283-291.
10. L. F. Ludwig, N. Pincever and M. Cohen, ''Extending the Notion of a Window System to Audio'', *IEEE Computer 23*, 8 , 66-72.
11. A. C. Luther, *Digital Video in the PC Environment*, McGraw-Hill, 1989.
12. W. E. Mackay and G. Davenport, ''Virtual Video Editing in Interactive Multimedia Applications'', *Comm. of the ACM 32*, 7 (July 1989), 802-810.
13. C. Nicolau, ''An Architecture for Real-Time Communication Systems'', *IEEE JSAC on Multimedia Communications*, 1990.
14. G. D. Ripley, ''DVI - A Digital Multimedia Technology'', *Comm. of the ACM 32*, 7 (July 1989), 811-822.
15. R. W. Scheifler and J. Gettys, ''The X Window System'', *ACM Transactions on Graphics 5*, 2 (Apr. 1986), 79-109.
16. *Multi-Media File System Overview*, Sun Microsystems, Aug. 1989.
17. *NeWS Manual*, Sun Microsystems, Inc., March 1987.
18. K. Umemura, Y. K. Hui and D. Anderson, ''Applications of Distributed Continuous Media'', In preparation, July 1990.