SYNCHRONIZATION POLICIES AND MECHANISMS IN A CONTINUOUS MEDIA I/O SERVER

David P. Anderson
George Homsy

Report No. UCB/CSD 91/617
February 1991

Computer Science Division (EECS)
University of California
Berkeley, California 94720

SYNCHRONIZATION POLICIES AND MECHANISMS IN A CONTINUOUS MEDIA I/O SERVER

David P. Anderson
George Homsy

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

February 1, 1991

ABSTRACT

We are developing ACME, a server for input and output of continuous media (digital audio and video) at the user interface. ACME supports a range of applications, including telephony, production, and browsing. The server runs as a multithreaded user-level process on a UNIX system. It is controlled via an extended window server (X11/NeWS) and communicates continuous media data on standard network connections. In this environment there are many sources of possible timing and synchronization error: 1) nonuniform startup times of I/O devices and network connections; 2) rate mismatch among I/O devices; 3) starvation of output streams; 4) buffer overrun on input streams.

ACME provides an abstraction called a *logical time system* (LTS) that allows clients to express their synchronization requirements. Multiple CM data streams can be grouped into an LTS. Within an LTS, data with the same "timestamp" is displayed (for output) or collected (for input) at approximately the same real time. The skew tolerance, and the policy for recovering from errors, are client-specified. An LTS can be paused and resumed, and discrete I/O events (*e.g.*, movie subtitles) can be scheduled within an LTS. The LTS mechanism is sufficient to express the synchronization requirements of a wide range of applications.

1. INTRODUCTION

Audio and video are perceived as changing continuously over time. To distinguish them from other I/O media, we therefore call them *continuous media* (CM). Continuous media dominate mass-market information systems (television, telephone, music recording, *etc.*). and, with advances in conversion and compression hardware, are becoming viable as computer I/O media as well.

CM and computer systems do not coexist easily. In many cases, computer hardware (processors, buses, networks) cannot handle the data rates involved; even if the hardware is able, the software may be incapable of providing the needed real-time performance characteristics. For these reasons, many CM-related projects have adopted solutions in which CM data is handled in separate systems, either analog or digital in nature, that operate under computer control.

On the long run, however, it is likely that the above hardware and software problems *can* be solved, and that CM data can be handled by application programs in general-purposes computer systems. This approach, which we call *integrated digital CM* (IDCM), has the following properties:

- Workstations are equipped with CM I/O devices (cameras, microphones, speakers) and with hardware for conversion and compression of CM data to and from main memory.
- User-level processes, running in protected virtual address spaces, can read and write multiple streams of CM data.
- Communication networks and protocols allow processes on different hosts to communicate CM data in real time.
- File servers can store CM data together with other data, and their clients can read and write it in real time.

A typical application in an IDCM system is the *audio playback* program shown in Figure 1. This program reads audio (or audio/video) data from a file and plays it through a CM I/O server.

The goal of IDCM presents research and engineering problems at every level of software and hardware design. One such level is that of *CM I/O server*. Such a server offers network-transparent access to CM I/O hardware, and mediates between concurrent clients. It is analogous

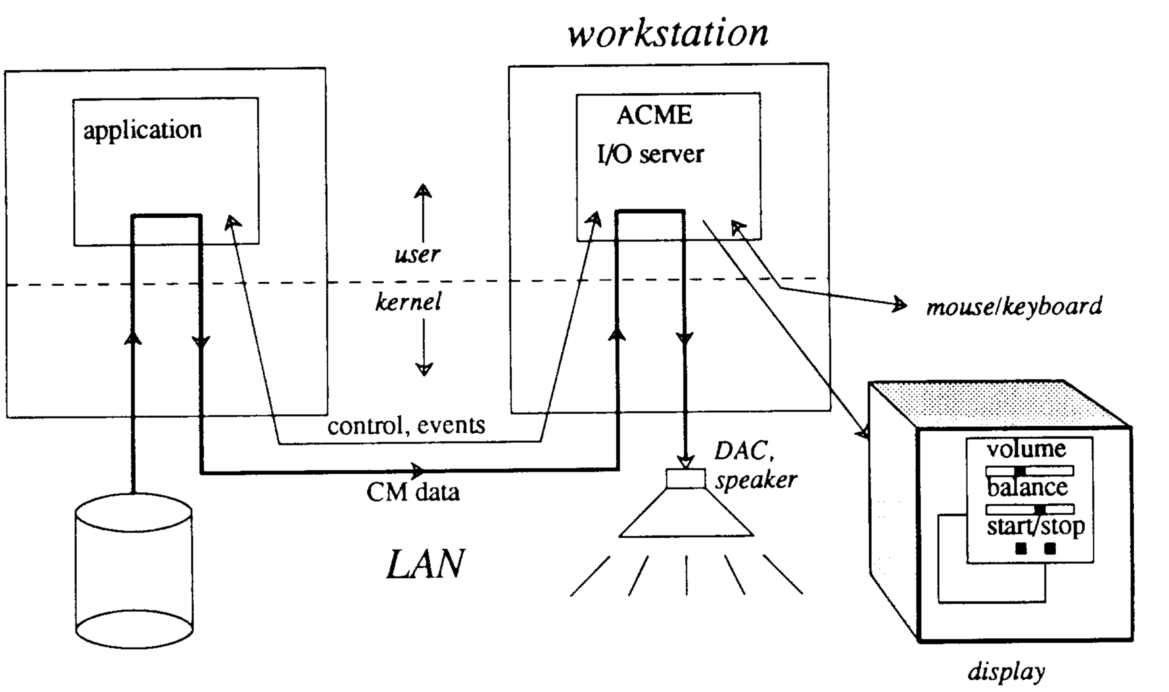


Figure 1: Audio playback is a basic integrated CM application. The application reads CM data from a file and sends it to the ACME server (CM data flow is shown as a bold line). The application provides a graphical interface for making selections and controlling the playback parameters.

to (and may in fact be combined with) a window system that provides access to, and contention resolution for, discrete I/O devices such as graphics display, keyboard, and mouse.

Applications that use CM often have *synchronization* requirements. They may require that input or output of multiple CM data streams start at given relative times and proceed in "lockstep", or that discrete events occur at given times relative to the input or output of a CM data stream. These conditions should hold in the face of clock rate mismatch, buffer overrun or starvation, and other anomalies. A CM I/O server should allow clients to express their synchronization requirements, so that the mechanisms used (which may be expensive) are applied only when needed.

This paper describes ACME, a CM I/O server under development at UC Berkeley [1]. ACME provides the abstraction of "logical CM I/O devices" that produce or consume data via network connections, and that can be grouped together. These basic features are described in Section 2. Section 3 describes ACME's synchronization mechanism, the *logical time system* (LTS).

Section 4 gives a discussion of a prototype implementation of LTSs. Section 5 discusses other work in the area.

2. BASIC FEATURES OF ACME

2.1. Basic Abstractions

A physical device (PDev) represents a CM I/O device. Each PDev has a direction (input or output), a medium (audio or video) and a spatial parameter (workstation-mounted, headset, *etc.*). A stereo audio device is viewed as a single PDev. A PDev may consume or produce data in any of a variety of formats, or *strand types*. A strand type includes the data representation (compression scheme, bits per sample) and sampling rate or spatial resolution. Clients may query the server for the set of PDevs, their attributes, and the strand types they support.

A logical device (LDev) represents a "virtual" microphone, speaker, camera, or video window. Each LDev is associated with a fixed PDev and has a fixed strand type. An LDev may have type-dependent attributes. For example, audio output LDevs have a 2x2 "volume control matrix", while a video output LDev has a position and size on the display. Several LDevs can be associated with a single PDev. For an input PDev, data from the PDev is replicated to each LDev. For an output PDev, data from the LDevs is mixed together (overlaid or summed) and output to the PDev. LDevs may be *mapped* or *unmapped*. While an output LDev is unmapped, data arriving at the LDev is discarded; while an input LDev is unmapped, no data is produced from the LDev.

Multiple strands may be combined into an interleaved *rope* format (for example, the DVI AVSS file format). A *compound logical device* (CLDev) is a set of LDevs that source or sink a rope. A *CM connection* is a network connection that conveys a rope.

Finally, each CLDev is associated with a *logical time system* (LTS). An LTS may include several CLDevs. The semantics of LTSs are discussed in Section 3.

2.2. Basic Implementation

ACME video output LDevs are intended to appear as "windows" on a display shared with a window server such as X11 or NeWS. An ACME server must therefore communicate with the local window server to obtain location and visibility information. In addition, it is convenient for ACME to use the window server for event distribution, and control interfaces (see Figure 2).

We have developed a prototype version of ACME. It runs on the Sun Sparcstation, and is integrated with Sun's Openwindows window system. We have extended Openwindows so that both X11 and NeWS clients can issue ACME control operations and receive events. The ACME prototype provides access to the Sparcstation's 8-bit, 8 KHz audio I/O hardware, and provides compressed and uncompressed video output.

The ACME prototype is written in C++, and uses a set of basic classes that provide preemptive real-time lightweight processes and LWP-synchronous I/O. Execution in ACME is structured as a set of LWPs (see Figure 3). The process types are as follows.

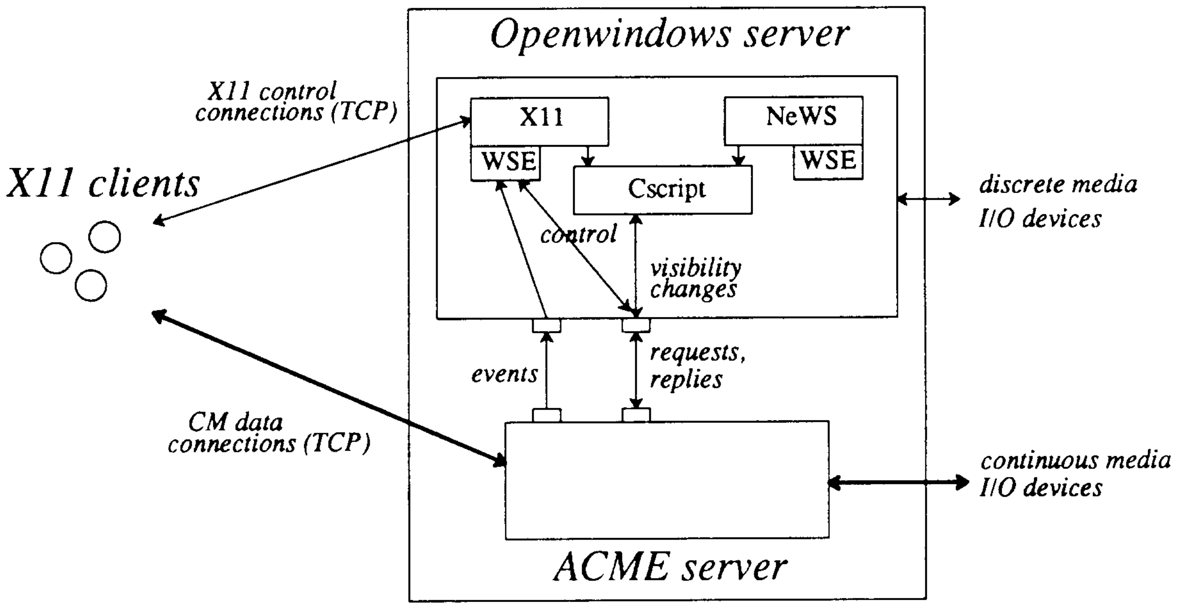

Openwindows server
X11 control connections (TCP)
X11
NeWS
WSE
WSE
Cscript
X11 clients
control
visibility changes
discrete media I/O devices
events
requests, replies
CM data connections (TCP)
continuous media I/O devices
ACME server

Figure 2: The prototype ACME server is controlled through a modified version of Sun's Openwindows server. The NeWS and X11 interpreters have extensions (labeled WSE) for ACME commands and events. The graphics portion of the server (Cscript) informs ACME of video window movement and visibility changes. CM data is not handled by Openwindows; it goes directly between clients and ACME.

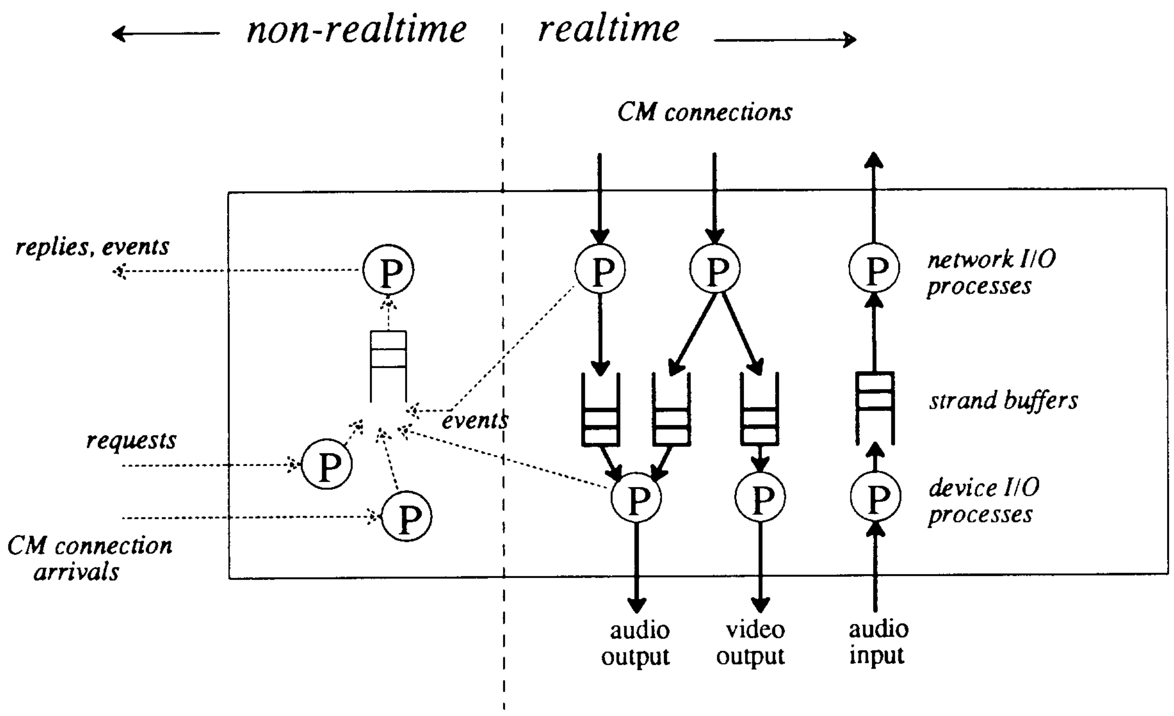


Figure 3: The ACME server is structured as a set of LWPs. Network and device I/O processes handle CM data (bold lines), while non-realtime processes handle control operations and event reporting (dotted lines).

- **Device I/O processes** perform I/O to and from CM devices. They are awakened on each I/O completion. A device output process gathers data from the strand buffers of the associated LDevs, perhaps combines them (*e.g.*, by summing audio samples) and writes them to the device. A device input process reads data from the device and writes a copy to the strand buffer of each associated LDev.

- **Network I/O processes** perform I/O to and from CM connections. A network output process reads a block of data from its associated CM connection, splits it into strands, and writes each subblock to the strand buffer of the associated LDev. Before writing the data it may process it in some way; for example, in the ACME prototype it converts 8-bit mu-law audio samples to word-aligned 16-bit linear samples, scaled by a volume parameter.

- **Discrete event processes** handle control operations and events. The ACME prototype has three such processes. The *accepter process* accepts new CM connections, and the *reader process* reads request messages from the window server. Neither process directly takes action; instead they send a message to a *main process* that carries out the appropriate action, perhaps sending a request or event message to the window server. Network and device I/O processes may also send messages to the main process to signal an event to be reported to the client.

3. LOGICAL TIME SYSTEMS: SEMANTICS AND EXAMPLES

A typical ACME client inputs and/or outputs streams of digital audio/video data, and imposes some requirements on the timing, rate, ordering, and loss characteristics of this I/O. We collectively call these *synchronization requirements*. For example, an application that displays a movie might demand that the temporal "skew" between the audio and video output not exceed, say, 100 milliseconds. It might demand that locally-generated "subtitles" be displayed during particular intervals of the movie. These requirements may involve input as well as output: for example, an audio/video dubbing application would need to limit the skew between input and output.

In the ACME environment, several factors can contribute to violations of synchronization requirements. CM I/O devices may run at slightly different rates. Some devices may have a "pipeline" structure that delays input or output. Data on different network connections may not start simultaneously, and may be subject to jitter. Finally, contention for system resources (network bandwidth, CPU time, *etc.*) may cause an ACME server to starve on output, or suffer buffer overrun on input.

ACME's *logical time system* (LTS) abstraction allows clients to give the ACME server "instructions" for how to synchronize input and output, and how to deal with error conditions. In this section we describe the semantics of LTSs and give examples of their use.

3.1. Basic LTS Semantics

A CM data strand consists of "units" (video frames, audio samples). Each unit *U* has an associated nonnegative "timestamp". Timestamps can be implicit. For example, in an audio encoding with a uniform sampling rate, the timestamp of a given sample is determined by its sample number.

Recall from Section 2.1 that a CLDev is a group of one or more logical devices that receive from or send to a single network connection, and that each CLDev is associated with an LTS. ACME provides operations for creating LTSs, associating a CLDev with an LTS, and starting and stopping an LTS. Each CLDev C in an LTS has a nonnegative offset T_C. The *logical time* of a data unit is its timestamp plus T_C.

Very roughly, the ideal semantics of an LTS are as follows: *If two data units in the same LTS have the same logical time, they are displayed or collected at approximately the same real time.* In other words, data streams in an LTS are synchronized with each other (but not necessarily with the streams of other LTSs). An LTS can be viewed as a coordinate system for an interval of real time (see Figure 4); LTS time does not necessarily advance at the same rate as real time.

ACME provides the following primitives for discrete interactions with LTSs:

- A client can read the current value of an LTS.

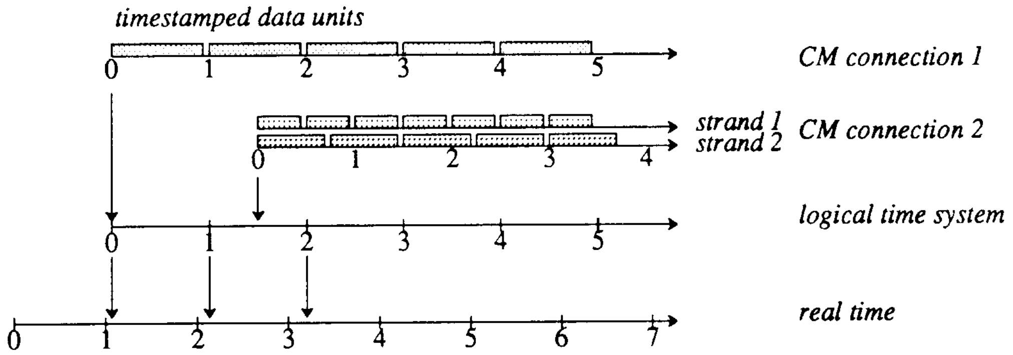


Figure 4: CM data streams to/from several network connections can be mapped, with different offsets, onto a *logical time system* (LTS), and are then synchronized relative to each other, but not necessarily relative to real time. In this example, an LTS includes two CM connections, one comprised of two strands. The LTS as a whole runs slightly slower than real time.

- A client can request that the ACME server send an event when an LTS reaches a certain point.
- A client can submit ACME or window system requests for deferred execution when a given value of an LTS is reached.

Finally, ACME allows an LTS to be paused (stopping the LTS and suspending I/O on its LDevs) and later resumed.

3.2. LTS Parameters

In order to enforce LTS semantics, ACME performs three types of synchronization: startup synchronization, rate matching, and starvation/overrun recovery. ACME provides client-specified parameters that determine how each type of synchronization is accomplished. We now discuss each type of synchronization in detail.

3.2.1. Startup Synchronization

To start an LTS correctly, or to add a new CLDev to a running LTS, two problems must be addressed:

(1) Output PDevs may have different latencies due to buffering or pipelining.

(2) Data for output CLDevs may begin arriving at different times, and may arrive with variable delay ("jitter"). To avoid the possibility of starvation in the future, the server must wait for sufficient data to arrive on all the CM connections.

Problem 1) is addressed by requiring all PDevs associated with an LTS to be ready to display, before starting the LTS. To address problem 2), each output CM connection C has a "start count" N_C. N_C bytes of data must be received on C before the associated LTS is started. If the network supports connections with bounded jitter, an appropriate value for N_C can be calculated from the jitter bound of the connection.

3.2.2. Rate Matching

Rate mismatch can occur when two PDevs in an LTS have different clocks. For instance, assume that a video output PDev has a nominal frame rate of 60 frames per second, an audio output PDev has a nominal sample rate of 8000 samples per second, and that the actual rates at which each PDevs displays data are determined by separate quartz crystal oscillators. If the ratio of the oscillator frequencies is not exactly 8000 / 60, then audio and video output will not stay in synchrony. To restore synchrony, several approaches are possible:

(1) Vary the display rate of one or more of the devices.

(2) Resample data to adjust the effective display rate. This approach has the advantage that, in the case where a PDev handles streams from different LTSs, the rate of each stream can be adjusted separately.

(3) For an output LDev, skip small portions of data (to catch up to a faster running LDev), or pause output for small lengths of time (to be caught up to by a slower running LDev). Input LDevs can skip data to speed up, and can write "null" data to their stream to slow down.

In the presence of nonuniform PDev rates, there are several possible ways to determine the rate of advance of an LTS. The LTS advance can be driven by:

(1) The rate of a specified LDev, D_{master}. In this case, D_{master} always displays at its natural rate, while other LDevs in the LTS may require rate adjustment, skipping, or pausing.

(2) The rate of the slowest LDev. If the skip/pause approach is used, only pausing will occur in this case.

(3) The rate of the fastest LDev, If the skip/pause approach is used, only skipping will occur in this case.

(4) The rate of data arrival on a CM connection. In this case, output is sped up if a backlog of data builds up in the server, and is slowed down if starvation occurs repeatedly.

(5) An external timing source, such as a clock. Rate control in either direction may be needed for all LDevs in the LTS.

Our ACME prototype supports only skip/pause rate control, and supports only option (1) for LTS advance. However, a full-featured I/O server might provide a wider range of methods as a client-selectable option.

3.2.3. Starvation and Buffer Overrun

If data is consumed by an output PDev faster than it arrives on a CM connection, the corresponding strand buffer may become empty. We say the LDev *starves* in this case. The corresponding situation for input LDevs is when a strand buffer becomes full because the CM connection cannot transfer data as fast as the PDev collects data. This is called *buffer overrun*. The following discussion is phrased in terms of output only, but applies equally to LTSs that include a mixture of input and output LDevs.

If a CM connection is starved, a decision must be made whether to continue displaying data from connections that are not starved. If display is continued, then skew increases. Here, skew is defined as the maximum over all LDevs of the difference of the LDev's logical time and the current LTS value.

On the other hand, if a connection is starved and display is stopped immediately, then the entire LTS will suffer from "hiccups", simply because one connection is not performing as well as it should. Some applications require smoothness of display, whereas others require low skew; thus neither of these options is acceptable for all applications.

ACME allows the client to select either extreme, or an intermediate policy, as follows. Each LTS has a maximum allowable skew parameter, S_{max}. If one connection starves, the LTS continues advancing and display of the other connections' data continues, until the skew exceeds S_{max}. This gives the offending connection a chance to "catch up". That is, if a burst of data arrives, the server can either drop data or display data rapidly until it encounters a timestamp comparable to the LTS value, thus avoiding hiccups in the display of the other connections' data. On the other hand, if no burst of data arrives and the skew exceeds the S_{max}, the LTS is paused. In this case, all display pauses until the offending connection has caught up, at which point the LTS is restarted and synchronized display resumes.

Display can be continued without regard to synchronization by setting S_{max} to infinity. In this case, display is synchronized only in the absence of starvation. If starvation does occur, skew can get arbitrarily large. On the other hand, if S_{max} is zero, the LTS will pause immediately whenever any connection starves, thus keeping all display tightly synchronized, but possibly causing hiccuping.

3.3. Examples of LTS Usage

To summarize Section 3.2, an LTS has several client-selectable parameters: the start count N_C for each connection, the method of LTS advance, and the skew bound S_{max}. We now give descriptions of some example ACME applications, with suggestions for the likely settings of these parameters.

Audio/Video Playback. This application reads audio/video material from a file server, sends it on a single CM connection, and displays it on the user's ACME server. Smoothness of display is important here, but latency is not. N_C should therefore be set to a large value to reduce the chance of starvation. It is likely that smoothness of audio display if more important than smoothness of video display. Therefore, the advance of the LTS should be governed by the audio PDev. Video frames will be repeated or dropped as needed to maintain a smooth flow of audio output.

Audio Overdubbing. This application allows the user to dub an audio track onto an existing video track; this could be used, for instance, to "lip synch" music videos. The input and output CM connections should be placed in a single LTS. A moderately low value of S_{max} should be used to ensure synchronization; for example, if video output starves then audio input should be suspended soon thereafter. Also, the quality of the recorded audio data is presumably more important than the quality of video output. Skipping input data or pausing input will produce glitches in the recorded data. The audio input device should therefore control the rate of the LTS.

Multitrack Audio/Video Editing. Consider an all-digital A/V postproduction studio. This could be implemented as an ACME application that stores audio and video segments on one or more file servers. When previewing a multitrack segment, the application combines the output

CM connections into a single LTS. There are two main requirements. First, the time correspondence between video frames in different tracks, and between video and audio, must be preserved. Hence S_{max} should be small. Second, *all* data should be displayed without skipping, so that no video frame is omitted during preview. Hence the LTS advance should be determined by the slowest PDev.

SMPTE Synchronized Dubbing. Suppose that a multitrack analog recording exists, with SMPTE time code on one of its tracks, and that the SMPTE code can provide a source of hardware interrupts to the ACME server. Now suppose we want to dub an audio track being output by the ACME server onto this analog recording. The method of LTS advance in this case should be external synchronization slaved to the SMPTE time code, so that the ACME server output is synchronized with the analog recording.

Telephony. In its simplest form, this application conveys audio and perhaps video data between two workstations. One or two CM connections may be used in each direction. Low end-to-end delay is the main concern. The CM connections can be placed in separate LTSs. Start counts of zero should be used, and the LTSs should be driven by arrival rate (so that no buffer buildup occurs if the rate of a microphone at one end exceeds that of the speaker at the other end).

4. LTS IMPLEMENTATION

We now discuss the implementation of LTSs in the prototype ACME server. Several simplifying assumptions are made:

- Periodic I/O-completion interrupts are generated by all PDevs. For example, in the ACME prototype, a UNIX signal is generated for every 256 samples displayed by the audio output hardware, and for every 1024 samples collected by the audio input hardware.
- The delay in interrupt delivery (due, *e.g.*, to other running processes), and the time to service an interrupt, are negligible.
- Interrupts do not carry real-time "timestamps", and a precise real time clock is not available. Therefore, PDev rate mismatch can be detected only by the order of interrupts.

In the ACME implementation, each entity (LTS, LDev, CM connection) is represented by a C++ object. Each of these objects includes a *state* field. The possible states are shown in Figure 5.

4.1. Startup Synchronization

Some output PDevs impose a delay between the arrival of the first data in a stream and the first display. For example, a video decompression system may require that several frames be "in the pipeline" before display can start. The PDevs in an ACME server may have different latencies, and ACME must defer I/O on LTS startup until all PDevs are ready. I/O must also be deferred until each CM connection *C* has received N_C bytes of data (see Section 3.2.1).

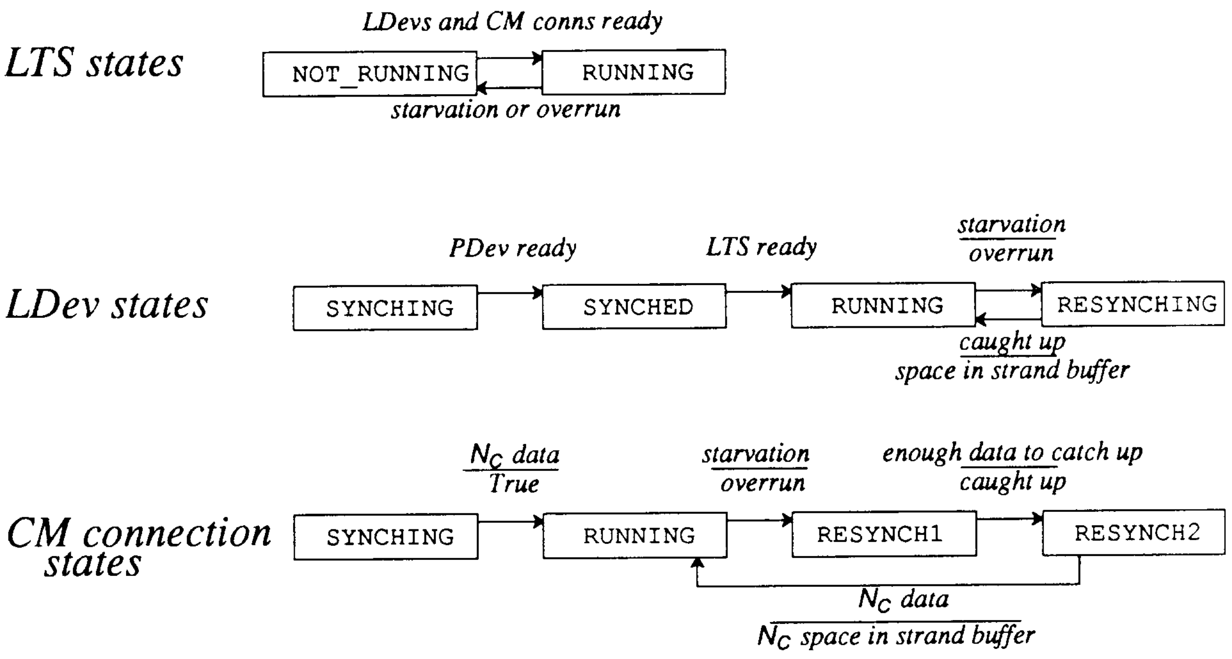


Figure 5: Each LTS, LDev, and CM connection exists in one of a finite set of states. The possible states and transitions are shown. Where two reasons for a transition are shown separated by a horizontal line, the upper condition applies to output LDevs or connections, and the lower condition applies to input LDevs or connections.

An LTS is initially `NOT_RUNNING` and its LDevs and CM connections are `SYNCHING`. An LTS object maintains a count of how many LDevs and CM connections are not yet ready, and exports a `ready()` operation to be called when one of these entities becomes ready. A network output thread begins reading data from its network connection; when N_C bytes of data have arrived it calls `ready()` and sets the connection to `RUNNING`. A network input thread immediately calls `ready()` and sets the connection to `RUNNING`.

A device I/O thread determines when its PDev is ready to display data for an LDev. When the condition is met, the device I/O thread performs a `ready()` operation on the LTS and changes the LDev state to `SYNCHED`.

When the LTS not-ready count goes to zero the LTS becomes `RUNNING` and all LDevs are set to the `RUNNING` state. These state changes are all performed by whatever thread performs the final `ready()` operation on the LTS. The device threads do not display data for an LDev until it is `RUNNING`. Thus, when the LTS starts and all LDevs are `RUNNING`, output commences on all LDevs during the next interrupt cycle for each PDev. Skew on startup is therefore bounded by the longest PDev interrupt period.

4.2. Rate Matching

The rate of the LDevs associated with a given LTS must be matched to the rate of the LTS (as determined by one of the methods discussed in Section 3.2.2). To illustrate algorithms for rate matching and starvation synchronization, we introduce the "stairstep diagram" (see Figure 6). Such a diagram shows the relation between real time and the timestamp of an LDev data stream being displayed. Figure 6a shows an example in which there is no rate adjustment. Ideally, the slope of the display time function is one. In general, it differs slightly from one. For example, suppose an audio output PDev has a nominal rate of 8192 samples per second, but that it takes only .99 seconds to output a block of 8192 samples. In this case, the horizontal edges of the boxes occur at 1, 2, 3 *etc.*, but the vertical edges of the boxes occur at .99, 1.98, 2.97, *etc.* The slope of the display time function is 1/.99.

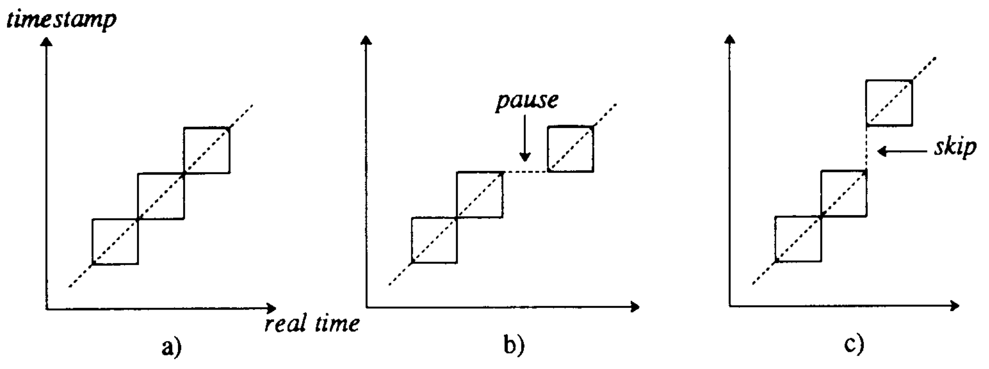


Figure 6: *Stairstep diagrams* show the relation between real time and the timestamp of an LDev data stream being displayed. Each box represents the display of one block of data. The left and right sides of a box show the real times at which PDev interrupts are received, and the bottom and top show the start and end timestamps of the data. The dotted line is the *display time function.*

Figure 6b shows the progress of an LDev with a pause occurring on one block; in Figure 6c, a skip occurs on one block. In both cases, the display time function is indicated by the dotted line. The *skew* between two LDevs at time t is defined as the difference between their display time functions at t.

4.2.1. The DDSP Algorithm for Rate Matching

We now describe and discuss an algorithm for rate matching, the *Device-Driven Skip/Pause* (DDSP) algorithm. The DDSP algorithm uses the following simplifying assumptions in addition to those stated earlier:

- LTS advance is driven by a particular LDev D_{master}.
- LDev rate adjustment is done by the skip/pause method.
- The algorithm must be "stateless" in the sense that the only information used to determine whether to display, skip, or pause is the timestamps of the start and end of the current data block for each LDev, and the current value of the LTS.

The stairstep diagram for D_{master} has no skips or pauses, and the current value of the LTS is defined to be the bottom edge of the diagram. On each interrupt of another LDev in the LTS (call it D_{slave}) the algorithm must decide whether to 1) display the next block of data for D_{slave}, 2) skip one or more blocks and display some subsequent block, or 3) pause, not displaying any subsequent block. In the case of an input LDev, the decision is whether to 1) move the next block of data into the strand buffer, 2) pad the strand buffer with some "dummy" data in order to rapidly advance the timestamps of the outgoing data stream (analogous to skipping), or 3) not move any data into the strand buffer in order to delay the timestamps of the outgoing stream (analogous to pausing).

Given the above restrictions, this decision must be based solely on the top and bottom edges of the current block of D_{master}. The bottom edge is given by the current value of the LTS, and the top edge is the sum of the blocksize (expressed in units of time) of D_{master} and the LTS value. Hence an algorithm satisfying these restrictions defines a partition of the plane of the stair-step diagram into regions $R_{display}$, R_{skip} and R_{pause} within which the algorithm displays, skips and pauses respectively. These regions are rectilinear, and their vertical edges occur only at interrupt times for D_{master}.

The DDSP algorithm is as follows. Let the difference in initial timestamps of successive blocks of D_{slave} data (the height of the D_{slave} blocks) be denoted by P_{slave}, and that of D_{master} by P_{master}. Let T_{slave} and T_{master} denote the interrupt periods of D_{slave} and D_{master} respectively. Let X denote the rate of D_{slave} relative to D_{master}, *i.e.*, $(\frac{P_{slave}}{T_{slave}})/(\frac{P_{master}}{T_{master}})$. and let $\bar{X}$ denote an *a priori* upper bound on $\max(X, \frac{1}{X})$, as determined by hardware tolerances. Let $H = P_{slave}\bar{X}$. Extend each block of D_{master} by $H/2$ in both the upward and downward directions, and define $R_{display}$, R_{pause} and R_{skip} as shown in Figure 7.

4.2.2. Properties of the DDSP Algorithm

We now prove three properties of the DDSP algorithm, under the assumption that no buffer overrun or starvation occurs, and that PDev rates are constant. We say that the *operating point* of D_{slave} at time t is $<R,T>$ if the most recent interrupt was at real time R, and D_{slave} began displaying a data segment with timestamp T at that time.

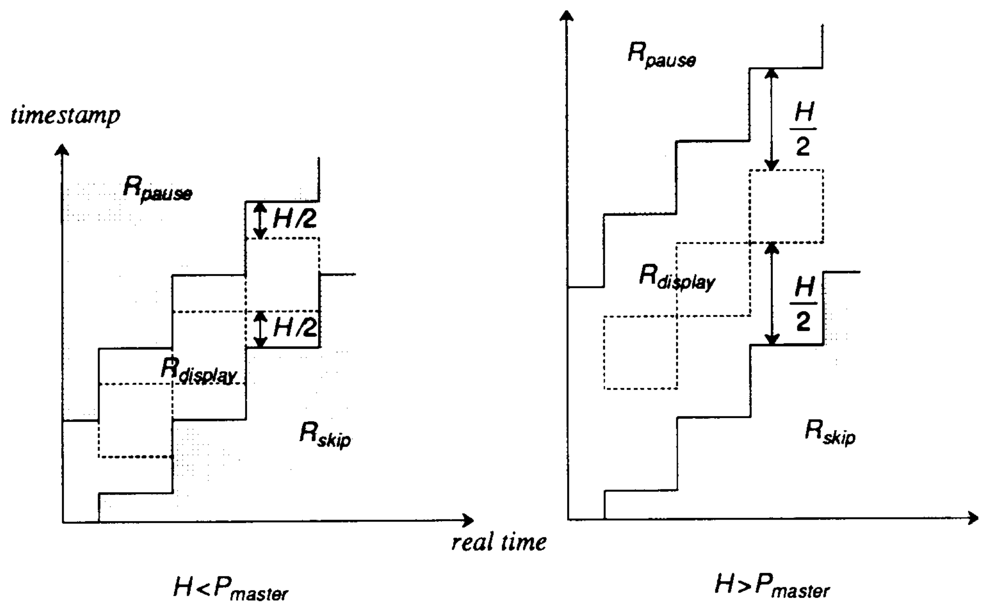


Figure 7: the plane partition representing the DDSP algorithm for a given slave LDev. The dotted lines are the stairstep diagram of the master LDev D_{master}. The shaded regions represent situations in which D_{slave} must skip or pause.

Property 1: Skips and pauses are not interspersed for any LDev. That is, if both skips and pauses occur, one strictly precedes the other.

The proof consists of two cases. In each case, we will construct a set of LDev "states", and enumerate the possible state transitions. It will be shown that each state diagram has no paths containing interspersed skips and pauses.

Case 1: $H < P_{master}$. Consider the five regions separated by dashed lines in Figure 8a. At least one D_{slave} interrupt occurs within the horizontal projection of each shaded triangle. In particular, an interrupt occurs within the horizontal projection of the first shaded triangle after the current operating point of D_{slave}. If the operating point of D_{slave} is in A when this interrupt occurs, it is either above a grey triangle (in which case D_{slave} will pause) or D_{slave} has drifted into B. Therefore if D_{slave} is in A it will pause or drift into B within T_{master}. Similarly, if D_{slave} is in E, it will skip or drift into D within T_{master}.

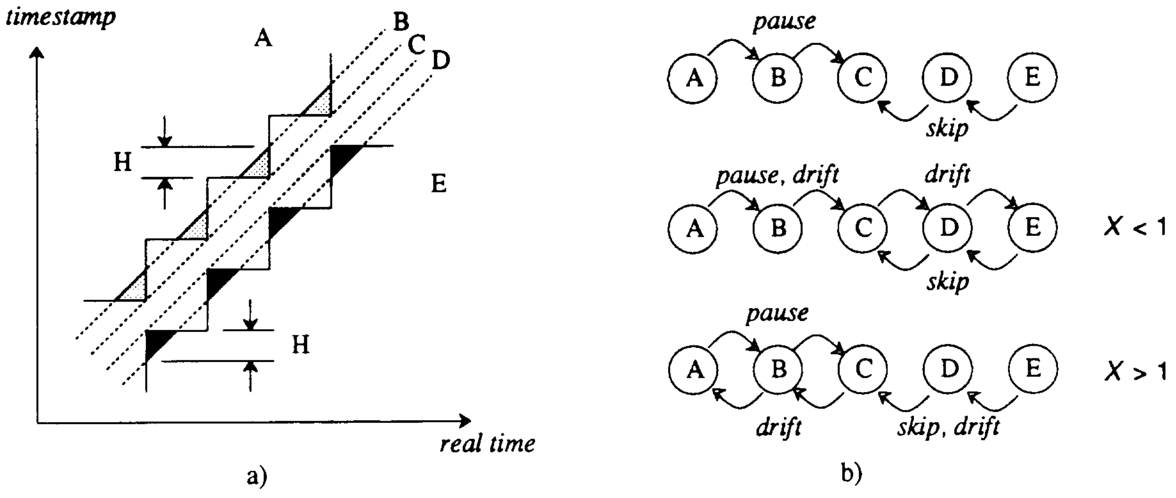
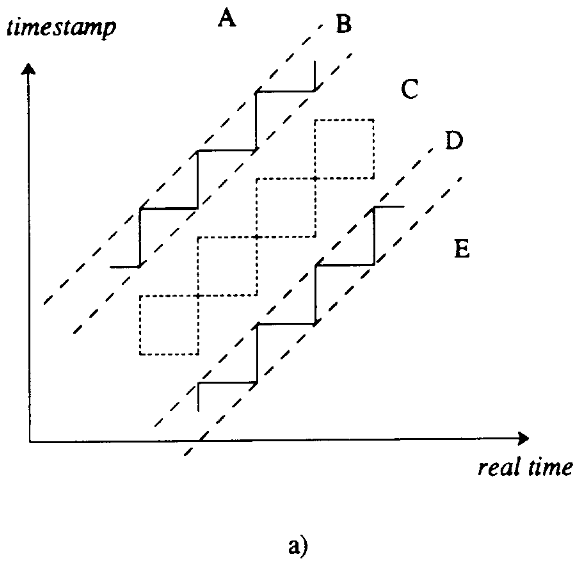


Figure 8: When $H < P_{master}$ the only transitions between the regions (A, B, C, D, E) shown in a) are those indicated in b). Thus skips and pauses are not interspersed.

If D_{slave} is in B it will either display or pause, and in D it will either display or skip. In region C, it will always display. The possible transitions between these regions due to pausing or skipping are shown in the top diagram of Figure 8b. Region A cannot pause to C because the width of region B is at least T_{slave}. Similarly, region E cannot pause to C because the height of region D is at least P_{slave}. Region B cannot pause to D because the width of C is at least T_{slave}, and region D cannot skip to B because the height of C is at least P_{slave}. C never skips or pauses.

The only other way the operating point of D_{slave} can change regions is by "drifting". If $X < 1$, the transitions $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ can all take place, even when no skipping or pausing occurs (see the middle diagram of Figure 8b). Similarly, if $X > 1$, the transitions $A \leftarrow B \leftarrow C \leftarrow D \leftarrow E$ can all take place (see the bottom diagram of Figure 8b). Since no drift occurs when $X = 1$, the top diagram of Figure 8b suffices to illustrate this case. Clearly, in each of these three diagrams, there is no path which intersperses skips and pauses.

Case 2: $H > P_{master}$. The regions A, B, C, D, and E are defined as shown in Figure 9a. If D_{slave} is operating in A, it will immediately pause into B or C. B will either display or pause into C, C will always display, D will either display or skip into C, and E will immediately skip into C or D.



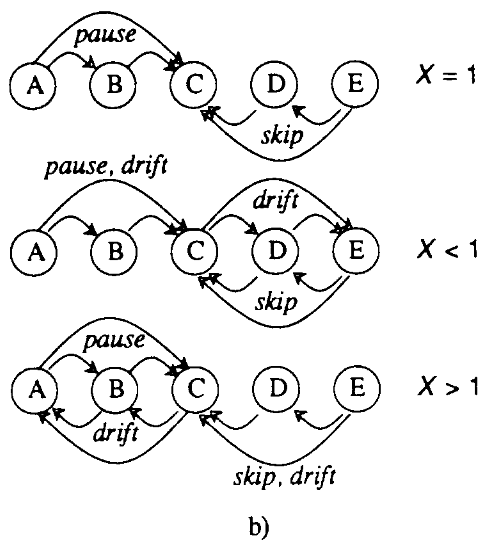


Figure 9: When $H > P_{master}$ the only transitions between the regions (A, B, C, D, E) shown in a) are those indicated in b). Thus skips and pauses are not interspersed.

The possible drift transitions for $X < 1$ are $A \to B \to C \to D \to E$ and $A \to C \to E$, since B and D are narrower than P_{slave}. The possible drift transitions for $X > 1$ are $A \leftarrow B \leftarrow C \leftarrow D \leftarrow E$ and $A \leftarrow C \leftarrow E$. These transitions are shown in Figure 9b. Again, inspection of the diagrams shows that skips and pauses cannot be interspersed.

Property 2: The skew between D_{slave} and D_{master} is bounded by $\frac{3H}{2} + (\bar{X}-1)\max(P_{master}, P_{slave})$.

$\bar{X}$ is typically very close to one, so this bound is close to $\frac{3H}{2}$. Again, the proof has two cases.

Case 1: $H < P_{master}$. First assume $X = 1$. If D_{slave} is operating in A, it will soon pause into B, since it must pause at least once every P_{master}. If D_{slave} is operating in E, it will soon skip into D. Therefore, D_{slave} will enter regions B, C, and D and not leave. The skew will not exceed $3H/2$, since the display time function of D_{master} is a line bisecting region C along its symmetry axis.

Next assume $X < 1$. If D_{slave} is operating in A, it will soon pause into B. It will also either pause or drift into C within finite time, since it would drift into C even if no pauses ever occurred.

Thus, D_{slave} will eventually operate within C, D, and E. E is a semi infinite region, which might cause problems for bounded skew. However, D_{slave} cannot penetrate far into E, since it can only drift into E, and it must return to D within P_{master} by skipping. The maximum possible drift into E, measured by timestamp, is $(1-X)P_{master}$. The skew in this case is therefore in the range $[-3H/2 - (1-X)P_{master}, H/2]$.

A similar argument applies if $X > 1$. The skew eventually remains within $[-H/2, 3H/2 + (X-1)P_{master}]$.

Case 2: $H > P_{master}$. The arguments are similar to Case 1. If $X = 1$, D_{slave} will eventually operate within regions B, C, and D, and the skew is bounded by $H/2 + P_{master}$. If $X < 1$, D_{slave} will eventually operate within regions C, D, and E. D_{slave} can only enter region E by drifting. Since it must immediately skip back to D or C on the next D_{slave} interrupt, the maximum drift into E, measured by timestamp, is $(1-X)P_{slave}$, so the skew is in the range $[-H/2 - P_{master} - (1-X)P_{slave}, H/2]$. Finally, if $X > 1$ then the skew eventually remains within $[-H/2, H/2 + P_{master} + (1-X)P_{slave}]$.

Property 3: If $X < 1$ **then the fraction of skipped blocks of** D_{slave} **approaches** $(1-X)/X$. **If** $X > 1$ **then the fraction of skipped interrupt periods approaches** $(X-1)/X$.

Suppose $X < 1$. Then, from Property 1, there is some time T_0 such that D_{slave} does not pause after T_0. Let $T_1 > T_0$, and let N denote the number of blocks skipped between T_0 and T_1. If D_{slave} denotes the change in skew between T_0 and T_1, $|D_{slave}| \leq 3H$ by Property 2, so we have

$$
\begin{aligned}
3H &\geq |D_{slave}| \\
&= |\frac{P_{master}}{T_{master}}(T_1-T_0) - \frac{P_{slave}}{T_{slave}}(T_1-T_0) - NP_{slave}| \\
&= |(1 - \frac{P_{slave}\,T_{master}}{T_{slave}\,P_{master}})(T_1-T_0) - \frac{NT_{master}}{P_{master}}P_{slave}| \\
&= |(1-X)(T_1-T_0) - \frac{NT_{master}}{P_{master}}P_{slave}| \\
&= |(1-X)(T_1-T_0) - NT_{slave}\frac{P_{slave}\,T_{master}}{T_{slave}\,P_{master}}|
\end{aligned}
$$

$$= |(1-X)(T_1-T_0) - NT_{slave}X|$$

Dividing by $X(T_1-T_0)$ gives

$$\left|\frac{1-X}{X} - \frac{NT_{slave}}{T_1-T_0}\right| \leq \frac{3H}{X(T_1-T_0)}$$

The quantity $\frac{NT_{slave}}{T_1-T_0}$ is precisely the fraction of skipped blocks, and this evidently approaches $\frac{1-X}{X}$ as T_1-T_0 becomes large. The proof for $X < 1$ is similar.

4.3. Buffer Over/Underrun

During normal operation of an LTS, it and all its LDevs and CM connections are in the RUNNING state. If an LDev drops behind the LTS due to buffer starvation or overrun, but the skew does not exceed S_{max}, display or collection proceeds according to the DDSP algorithm. LDevs can catch up to the LTS if the starvation/overrun condition is alleviated before S_{max} is exceeded, as shown in Figure 10a.

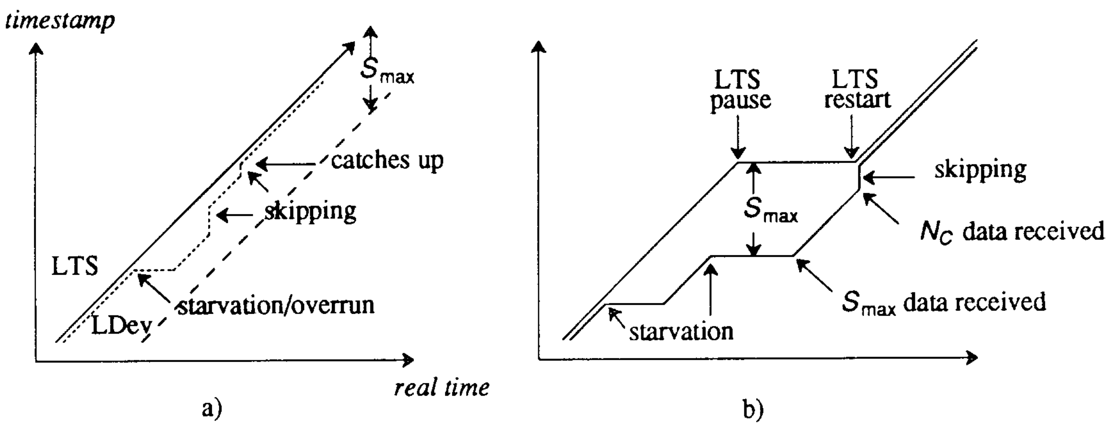


Figure 10: Two scenarios of *starvation* caused by late arrival of data on a CM connection. In a), more data arrives on the connection before it reaches the skew bound of the LTS; this data is displayed or skipped, and the CM connection catches up to the LTS. In b), the skew bound S_{max} is exceeded, and the LTS is stopped. It is restarted only when enough data is received for the offending LDev to catch up, plus N_C additional data bytes.

If the skew of an LDev exceeds S_{max}, this condition is detected by the device thread for the PDev of that LDev. In this case, the LTS state is set to NOT_RUNNING, the offending LDev state is set to RESYNCHING, and the state of the connection associated with the offending LDev is set to RESYNCH1 (see Figure 10b). As long as the LTS is NOT_RUNNING, its current value is not updated. Hence any other LDevs associated with this LTS effectively pause, waiting for the LTS to restart.

While the LTS is not running, the CM connection counts data until it has received or sent enough to bring the offending LDev up to the current value of the LTS. At this point, the connection is set to RESYNCH2, the LDev is set back to RUNNING, and display on the offending LDev recommences, approaching the current value of the LTS. In the output case, once the connection has counted the above mentioned amount of data, it resets its start counter and starts counting data again. Once it has received N_C bytes of additional data (satisfying the guard condition against future starvation) it sets its state to RUNNING, and starts the LTS. In the input case, instead of resetting the start counter, the connection simply waits until at least N_C bytes of free space exist in the buffer, sets its state to RUNNING, and restarts the LTS.

After the LTS is restarted, the state of all resources is RUNNING. If the offending LDev has not yet caught up to the LTS, it skips immediately to bring itself back into synchronization, and display continues normally.

4.4. Discrete Event Synchronization

To implement LTS value requests, alarm clock events, and deferred requests, each LTS maintains a current value field. This value is updated by the master PDev of the LTS. Requests for the current LTS value simply return the value of this field. A time-sorted list of typed discrete event descriptors is kept with the LTS. Each time the LTS value is updated, the new value is compared with the time field of the earliest discrete event descriptor. If the event time has been met or exceeded, the descriptor is processed appropriately: alarm clock events are delivered to the window server, and deferred requests are inserted in the ACME request queue.

5. 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 [6], 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 [2], Pandora [7], IMAL [5], and VEX [3]. 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) [8] 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.

Much of the work addressing synchronization of continuous media is at a higher level than that addressed by ACME. For example, Little and Ghafoor [4] describe a formalism based on Petri nets for describing concurrency and ordering requirements among sets of media streams. Steinmetz [9] describes a set of programming constructs for expressing requirements on bounds between "events" such as the start and end of media streams. ACME's LTS mechanism could be used to implement some of these higher-level specifications.

6. CONCLUSION

We have described ACME, a set of abstractions for continuous media I/O. The ACME interface parallels the interface offered by a network-transparent window system such as NeWS or X11, and can be implemented as an extension to such a window system. ACME offers its clients "virtual" microphones, speakers, video cameras and displays, which are multiplexed onto physical I/O devices. Clients communicate digital CM data through network connections.

ACME offers a synchronization mechanism called the *logical time system* (LTS) that allows clients to specify their synchronization requirements and preferences. Such a mechanism is needed for a variety of reasons: device rate mismatch, device pipelining, network connection throughput and delay variation, and indeterminacy of control operation timing.

We have sketched an implementation of LTSs, describing the techniques used to provide startup synchronization, rate matching, and handling of starvation and overrun conditions. Our prototype implementation provides only a particular type of LTS: that in which timing is determined by a single "master" device, and in which rate control is accomplished by skipping or pausing. It also makes some pessimistic assumptions: *e.g.*, that interrupts are not timestamped. Considerable future work remains to be done in exploring the implementation of other variants of LTSs, and in evaluating the utility of the LTS mechanism for real-world continuous media applications.

REFERENCES

1. D. P. Anderson, R. Govindan and G. Homsy, "Abstractions for Continuous Media in a Network Window System", *International Conference on Multimedia Information Systems*, Singapore, Jan 1991.
2. 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.
3. T. Brunhoff, *VEX: Video Extension to X, Version 5.5*, Tektronix, Inc., 1989.
4. T. D. C. Little and A. Ghafoor, "Synchronization and Storage Models for Multimedia Objects", *IEEE Journal on Selected Areas in Communications 8*, 3 (Apr. 1990), 413-427.
5. 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.
6. W. E. Mackay and G. Davenport, "Virtual Video Editing in Interactive Multimedia Applications", *Comm. of the ACM 32*, 7 (July 1989), 802-810. n. Example: video editor w/

time-line. Upcall draws time line, checks drift, and changes QOS if needed. Also proposes "synch variables"

7. C. Nicolau, "An Architecture for Real-Time Communication Systems", *IEEE JSAC on Multimedia Communications*, 1990.
8. G. D. Ripley, "DVI - A Digital Multimedia Technology", *Comm. of the ACM 32*, 7 (July 1989), 811-822.
9. R. Steinmetz, "Synchronization Properties in Multimedia Systems", *IEEE Journal on Selected Areas in Communications 8*, 3 (Apr. 1990), 401-412.